Composing High-Level Plans for Declarative Agent Programming

Felipe Meneguzzi and Michael Luck

Department of Computer Science
King's College London
felipe.meneguzzi@kcl.ac.uk,
michael.luck@kcl.ac.uk

Abstract. Research on practical models of autonomous agents has largely focused on a *procedural* view of goal achievement. This allows for efficient implementations, but prevents an agent from reasoning about alternative courses of action for the achievement of its design objectives. In this paper we show how a procedural agent model can be modified to allow an agent to compose existing plans into new ones at runtime to achieve desired world states. This new agent model can be used to implement a declarative goals interpreter, since it allows designers to specify *only* the desired world states in addition to an agent's basic capabilities, enhancing the agent's ability to deal with failures. Moreover our approach allows the new plans to be included in the plan library, effectively enabling the agent to improve its runtime performance over time.

1 Introduction

The notion of autonomous intelligent agents has become increasingly relevant in recent years both in relation to numerous real applications and in drawing together different artificial intelligence techniques. Perhaps the best known and most used family of agent architectures is that based around the notions of beliefs, desires and intentions, which is exemplified by such systems as PRS[1], dMARS[2] and AgentSpeak [3]. For reasons of efficiency and real-time operation, these architectures have been based around the inclusion of a plan library consisting of predefined *encapsulated procedures*, or *plans*, coupled with information about the context in which to use them [3]. However, designing agents in this way severely limits an agent's runtime flexibility, as the agent depends entirely on the designer's previous definition of all possible courses of action associated with proper contextual information to allow the agent to adopt the right plans in the right situations.

Typically, agent interpreters select plans using more or less elaborate algorithms, but these seldom have any knowledge of the contents of the plans, so that plan selection is ultimately achieved using fixed rules, with an agent adopting *black box* plans based solely on the contextual information that accompanies them. Alternatively, some agent interpreters allow for plan modification rules to allow plans to be modified to suit the current situation [4], but this approach still relies on a designer establishing a set of rules that considers all potentially necessary modifications for the agent to achieve its goals. The problem here is that for some domains, an agent description must either be extremely extensive (requiring a designer to foresee every possible situation the agent might find itself in), or will leave the agent unable to respond under certain conditions.

This *procedural* response to goal achievement has been favoured to enable the construction of practical systems that are usable in real-world applications. However, it also causes difficulties in cases of failure. When a procedural agent selects a plan to achieve a given goal it is possible that the selected plan may fail, in which case the agent typically concludes that the goal has also failed, regardless of whether other plans to achieve the same goal might have been successful. By neglecting the *declarative* aspect of goals in not considering the construction of plans on-the-fly, agents lose the ability to reason about alternative means of achieving a goal, making it possible for poor plan selection to lead to an otherwise avoidable failure.

In this paper we describe how a procedural agent model can be modified to allow an agent to build new plans at runtime by chaining existing fine-grained plans from a plan library into high-level plans. We demonstrate the applicability of this approach through a modification to the AgentSpeak architecture, allowing for a combination of declarative and procedural aspects. This modification requires no change to the plan language, allowing designers to specify predefined procedures for known tasks under ideal circumstances, but also allowing the agent to form new plans when unforeseen situations arise. Though we demonstrate this technique for AgentSpeak, it can be easily applied to other agent architectures with an underlying procedural approach to reasoning, such as JADEX or the basic 3APL [5]. The key contribution is a method to augment an agent's runtime flexibility, allowing it to add to its plan library to respond to new situations without the need for the designer to specify all possible combinations of low-level operators in advance.

The paper is organised as follows: in Section 2 we briefly review relevant aspects of AgentSpeak, in order to introduce the planning capability in Section 3; in Section 4 a classic example is provided to contrast our approach to that of traditional AgentSpeak; in Section 5 we compare our work with similar or complementary approaches that also aim to improve agent autonomy; finally, in Section 6 a summary of contributions is provided along with further work that can be carried out to improve our system.

2 AgentSpeak

AgentSpeak [3] is an agent language that allows a designer to specify a set of procedural plans which are then selected by an interpreter to achieve the agent's design goals. It evolved from a series of procedural agent languages originally developed by Rao and Georgeff [6]. In AgentSpeak an agent is defined by a set of beliefs and a set of plans, with each plan encoding a procedure that is assumed to bring about a desired state of affairs, as well as the context in which a plan is relevant. Goals in AgentSpeak are implicit, and plans intended to fulfil them are invoked whenever some triggering condition is met in a certain context, presumably the moment at which this implicit goal becomes relevant.

The control cycle of an AgentSpeak interpreter is driven by events on data structures, including the addition or deletion of goals and beliefs. 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 that 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 condition that specifies when the plan is *applicable* 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 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 of a given goal fails, the default behaviour of an AgentSpeak 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 1,[1] and strongly couples plan execution to goal achievement.

The control cycle of Figure 1 allows for situations in which the poor selection of a plan leads to the failure of a goal that would otherwise be achievable through a different plan in the plan library. While such limitations can be mitigated through meta-level [8] constructs that allow goal addition events to cause the execution of applicable plans in sequence, and the goal to fail only when *all* plans fail, AgentSpeak still regards goal achievement as an implicit side-effect of a plan being executed successfully.

3 Planning in an AgentSpeak Interpreter

In response to these limitations, we have created an extension of AgentSpeak that allows an agent to explicitly specify the world-state that should be achieved by the agent. In order to transform the world to meet the desired state, our extension uses a propositional planner to form high-level plans through the composition of plans already present in the agent's plan library. This propositional planner is invoked by the agent through a regular AgentSpeak action, and therefore requires no change in the language definition. The only assumption we make is the existence of plans that abide by certain restrictions in order to be able to compose higher-level plans, taking advantage of planning capabilities introduced in the interpreter.

[1] For a full description of AgentSpeak, refer to d'Inverno *et al.* [7].

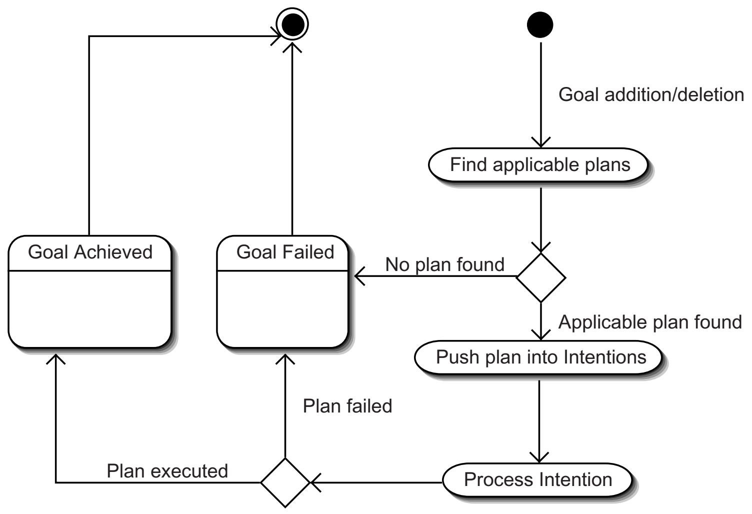

Fig. 1. AgentSpeak control cycle

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 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 plan and added to the intention structure for execution. Here, we liken the low-level procedural plans of AgentSpeak to STRIPS operators, connecting the agent interpreter to the planner by converting one formalism into the other and *vice versa*. We have chosen to use STRIPS as the planning language in this paper for simplicity reasons, and this approach would not lose applicability if one was to use PDDL [9] (or another language) as the planning language.

3.1 The Planning Action

In order to describe the connection of the planning component with AgentSpeak, we need to review the main constructs of this agent language. As we have seen, an AgentSpeak interpreter is driven by events on the agent's data structures that may trigger the adoption of plans. Additions and deletions of goals and beliefs 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 would be represented by $+!g$. Belief additions and deletions arise as the agent perceives the environment, and are therefore outside its control, while goal additions and deletions only arise as part of the execution of an agent's plans.

In our approach, in addition to the traditional way of encoding goals for an AgentSpeak agent implicitly as triggering events consisting of achievement goals

Table 1. Planner invocation plan

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

($!goal$), we allow 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 consists of a conjunction of beliefs the agent wishes to be true simultaneously at a given point in time. The execution of the planner component is triggered by an event $+goal\_conj([b_1, \ldots, b_n])$ as shown in Table 1.

Now, the key to our approach to planning in AgentSpeak is the introduction of 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 component, 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.

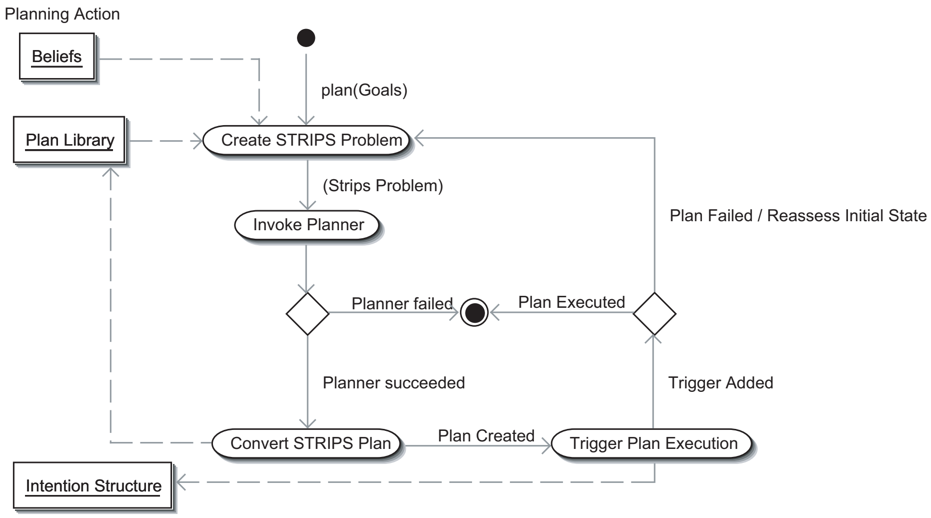

Fig. 2. Operation of the planning action

As illustrated in Figure 2, the internal action to plan takes as an argument the desired world-state, and uses this, along with the current belief database and the plan library, to generate a STRIPS [10] planning problem. This action then invokes a planning algorithm; if a plan is found, the planning action succeeds, otherwise the planning action fails. If the action successfully yields a plan, it converts the resulting STRIPS plan into a new AgentSpeak plan to be added to the plan library, and immediately triggers the adoption of the new plan. If the

Table 2. Movement plans

$$
\begin{aligned}
+!move\_to(A,B) \;&: available(car)\\
&\leftarrow get(car);\\
&\quad drive(A,B).\\[1em]
+!move\_to(A,B) \;&: \neg available(car)\\
&\leftarrow walk(A,B).
\end{aligned}
$$

newly created plan fails, the planner may then be invoked again to try and find another plan to achieve the desired state of affairs, taking into consideration any changes in the agent beliefs.

It is important to 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, and if no such plan is found to be applicable, the plan containing the planning action is invoked as the last remaining option.

3.2 Chaining Plans into Higher-Level Plans

The design of a traditional AgentSpeak plan library follows a similar approach to programming in procedural languages, where a designer typically defines fine-grained actions to be the building blocks of more complex operations. These building blocks are then assembled into higher-level procedures to accomplish the main goals of a system. Analogously, an AgentSpeak designer traditionally creates fine-grained *plans* to be the building blocks of more complex operations, typically defining more than one plan to satisfy the same goal (*i.e.* sharing the same trigger condition), while specifying the situations in which it is applicable through the context part of each plan. Here, we are likening STRIPS actions to low-level AgentSpeak *plans*, since the effects of primitive AgentSpeak actions are not explicitly defined in an agent description. For example, an agent that has to move around in a city could know many ways of going from one place to another depending on which vehicle is available to it, such as by walking or driving a car, as shown in Table 2.

Modelling STRIPS operators to be supplied to a planning algorithm is similar to the definition of these building-block procedures. In both cases, it is important that operators to be used sequentially *fit*. That is, the results from applying one operator should be compatible with the application of the possible subsequent operators, matching the effects of one operator to the preconditions of the next operator.

Once the building-block procedures are defined, higher-level operations must be defined to fulfil the broader goals of a system by combining these building blocks. In a traditional AgentSpeak plan library, higher-level plans to achieve broader goals contain a series of goals to be achieved by the lower-level operations. This construction of higher-level plans that make use of lower-level ones is analogous to the planning performed by a propositional planning system. By doing the *planning themselves*, *designers* must cope with every foreseeable situation the agent might find itself in, and generate higher-level plans combining lower-level tasks accordingly. Moreover, the designer must make sure that the subplans being used do not lead to conflicting situations. This is precisely the responsibility we intend to delegate to a STRIPS planner.

Plans resulting from propositional planning can then be converted into sequences of AgentSpeak achievement goals to comprise the body of new plans available within an agent's plan library. In this approach, an agent can still have high-level plans pre-defined by the designer, so that routine tasks can be handled exactly as intended. At the same time, if an unforseen situation presents itself to the agent, it has the flexibility of finding novel ways to solve problems, while augmenting the agent's plan library in the process.

Clearly, lower-level plans defined by the designer can (and often will) include the invocation of *atomic actions* intended to generate some effect on the environment. Since the effects of these actions are not usually explicitly specified in AgentSpeak (another example of reasoning delegated to the designer), an agent cannot reason about the consequences of these actions. When designing agents using our model, we expect designers to explicitly define the consequences of executing a given AgentSpeak plan in terms of belief additions and deletions in the plan body as well as atomic action invocations. The conversion process can then ignore atomic action invocations when generating a STRIPS specification.

3.3 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 planner 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.

Plans in AgentSpeak 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. 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, let us consider a triggering plan for accomplishing `!move(A,B)` corresponding to a movement from A to B, where:

- e is `!move(A,B)`;
- `at(A) & not at(B)` are belief literals; and
- `-at(A); +at(B).` is the plan body, containing information about belief additions and deletions.

The plan is then as follows:

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

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

In the 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 3, in which we convert the AgentSpeak 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.

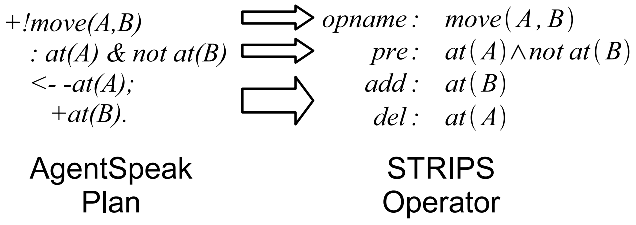

Fig. 3. Correspondence between an AgentSpeak plan and a STRIPS operator

A relationship between these two definitions is not hard to establish, and we define the following algorithm for converting AgentSpeak plans into STRIPS operators. Let e be a triggering event, $b_1 \& \ldots \& b_m$ a conjunction of belief literals representing a plan's context, and $a_1, \ldots, a_n$ 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 accomplished as follows:

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

In Section 3.1 we 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.

3.4 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 this 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. We define a new AgentSpeak plan in Table 3, where $goal\_conj(Goals)$ is the event that initially caused the planner to be invoked.

Table 3. AgentSpeak plan generated from a STRIPS plan

$$
\begin{aligned}
&+goal\_conj\ (Goals) : true\\
&\qquad\qquad \leftarrow !op_1; \ldots ; !op_n.
\end{aligned}
$$

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. Plans generated in this fashion are admittedly simple, since the development of a complete process of plan generalisation is not a trivial matter since, for instance, it involves solving the issue of deriving the context condition adequately. An extremely simple solution for this problem uses the entire belief base of the agent as context for that plan, but this solution includes a great number of beliefs that are probably irrelevant to the goal at hand, severely limiting this plan's future applicability.

Another solution involves replicating the preconditions of the first operator for the new plan, but this could also lead the agent to fail to execute the plan later on. We have developed an algorithm to derive a minimal set of preconditions, which we omit here due to space constraints, showing instead the simple solution of using a constantly true context. Another possible refinement to the conversion of a STRIPS plan into an AgentSpeak plan is to allow the same generated plan to be reused to handle side-effects of the set of goals that led to its generation. For example, a plan for a conjunction of goals g can be used to achieve any subset g' of g.

In the ensuing execution of the generated plan, the fact that multiple concurrent plans might be stacked in an agent's intentions structure must also be addressed. There are multiple ways of addressing this issue, namely:

1. delegate the analysis and resolution of conflicting interaction between plans to the designer;
2. implement provisions to ensure that the plans used by the planner process are executed atomically;
3. drop the entire intention structure before plan adoption, invoking some forward recovery plan, and prevent new intentions from being adopted during plan execution; and
4. analyse the current intention structure and prospective plan steps during planning to ensure they do not interfere with each other.

The first way to resolve concurrency problems, by delegating resolution to the designer, is the traditional solution in an AgentSpeak context, but it is clearly not acceptable, since the main goal of our extension is to diminish the amount of designer tasks. On the other hand, the last alternative, avoiding plan interference, involves the introduction of a complex analysis procedure to solve a very limited number of potential conflicts. In the third option, an agent drops its intentions to prevent concurrently executing plans from interfering with the new plan, which was created without regard for the current intention structure. This alternative requires the existence of forward recovery *abort plans*, such as those described by Thangarajah *et al.* [11].

For our experiments we considered the second and third ways of dealing with concurrency problems and, in the prototype described in Section 4, we opted to enable the agent to execute dynamically generated plans atomically (by preventing other intentions from being selected from the stack while a dynamic plan is being executed).

3.5 Coping with Failure

The possibility of generating new plans at runtime can also be used as an alternative when plans previously selected from the plan library have failed to achieve a certain goal. Constructs for handling these failures are available in Jason [12] and CANPLAN [11], and consist of associating an *abort* plan to be executed when the plan selected to handle an event or goal fails. In Jason this construct is expressed as a goal deletion ($-!g$). For example, in our system, when a newly generated plan to achieve $!goal\_conj([dg_1, \ldots, dg_N])$ fails, we can attempt to invoke the planner again to find an alternative plan to achieve these declarative goals by including the plan shown in Table 4.

In addition, the application of our planning approach would be beneficial for agents that use a more flexible commitment strategy, such as in the case of CANPLAN2 [13]. In this architecture, multiple plan-library plans are attempted in sequence, until either the agent interpreter concludes that the goal is impossible or all *known* plans have failed. In these situations, an external planner can be

Table 4. Using the planner action to recover from plan failure

$-!goal\_conj([dg_1, \ldots, dg_N]) : true \leftarrow plan([dg_1, \ldots, dg_N]).$

invoked to try to generate new plans until it also finds that the desired goals are impossible.

4 Experiments and Results

We have implemented the planning action described in Section 3 using Jason [14], which is an open-source Java implementation of AgentSpeak that includes a number of extensions, such as facilities for communication and distribution. In addition to providing an interpreter for the agent language, Jason has an object-oriented API for the development of *actions* available to the agents being developed. Since planning is to be performed as part of a regular AgentSpeak plan, the planning action encapsulates the conversion process of Section 3.3 using Jason's *internal actions*.

This implementation was used in a number of toy problems, such as the Blocks world used with the original STRIPS planner [10], as well as some examples from the AgentSpeak literature [3]. Solutions for these problems were created using both a procedural approach characteristic of traditional AgentSpeak agents, and a declarative one, in which high-level plans are omitted and left to be derived by the planning system. This switch in the method for describing agents results in a reduction of the plan description size, as it is no longer necessary to enumerate relevant combinations of lower-level plans for the agent to be able to react to different situations.

In terms of complexity the most computationally demanding part of our architecture is the planning process, which can vary significantly depending on the specific planner being used. The complexity of solving propositional planning problems depends on the number of pre-conditions and post-conditions of the operators in a certain domain [15], varying from polynomial to NP-complete and PSPACE-complete complexity. On the other hand, the conversion process into STRIPS is clearly very simple, having linear complexity on the number of pre-conditions and post-conditions of the operators being converted. The same linear complexity applies to the conversion from a STRIPS plan into an AgentSpeak plan.

Rao [3] uses a simple example agent to describe the derivations performed by an AgentSpeak interpreter. This agent detects when waste appears in a particular road lane, and disposes of it in a waste bin. The original plan library for the agent is as follows:

```
% Plan 1
+location(waste, X)
              : location(robot,X) &
                location(bin,Y)
              <- pick(waste);
                 !location(robot,Y);
                 drop(waste).
% Plan 2
+!location(robot, X)
              : location(robot,X)
              <- true.
% Plan 3
+!location(robot, X)
              : location(robot,Y) &
                not X = Y &
                adjacent(Y,Z)&
                not location(car,Z)
              <- move(Y, Z);
                 !location(robot, X).
```

Using Plan 1, whenever an agent detects waste in its current position, the agent will pick up the waste, move to the location of the waste bin and drop it. In this plan library, the agent's movement is achieved by an internal action, `move(Y,Z)`, and the agent has no way of explicitly reasoning about it. Moreover, if an agent has to perform multiple moves, recursive instantiations of Plan 3 in this library are stacked in the agent's intention structure, until the recursion stop condition is reached in Plan 2.

In order to be able to call a planner we need to modify the portion of the plan library responsible for the agent's movement (*i.e.* the last two plans) into a declarative description yielding the following plan library:

```
+location(waste, X)
       : location(robot, X) &
         location(bin, Y)
       <- pick(waste);
          +goal_conj([location(robot,Y)]);
          drop(waste).

+!move(X,Y)
       : location(robot,X) &
         not X = Y &
         not location(car,Y) &
              adjacent(X,Y)
       <- -location(robot,X);
          +location(robot,Y);
          move(X,Y).
```

The new plan library includes a description of the preconditions and effects of the `move(X,Y)` action. This is the action that is to be handled by the planning process, and the agent derives the sequence of movements required to reach the waste bin by *desiring* to be in the position of the bin. In order to specify this desire, the plan to dispose of the waste includes a step to add the desire `+goal_conj([location(robot,Y)])`, which causes the planner to be invoked. Here, the atomic action to `move(X,Y)` is also included in the plan specification so that when `!move(X,Y)` is invoked, the agent not only updates its beliefs about the movement, but actually moves in the environment. Unlike the original plan library, however, the agent can plan its movements before starting to execute them, and will only start carrying out these actions if it has found the entire sequence of movements required to reach the desired location.

5 Related Work

Work on the declarative nature of goals as a means to achieve greater autonomy for an agent is being pursued by a number of researchers. Here we consider the approaches to declarative goals currently being investigated, namely those of Hübner *et al.* (Jason) [16], van Riemsdijk *et al.* [17] and Meneguzzi *et al.* [18]. There are multiple interpretations as to the requirements and properties of declarative goals for an agent interpreter, and while some models consist of an agent that performs planning from first principles whenever a goal is selected, others argue that the only crucial aspect of an architecture that handles declarative goals is the specification of target world states that can be reached using the traditional procedural approach.

5.1 Jason

A notion of declarative goals for AgentSpeak that takes advantage of the context part of the plans (representing the moment an implicit goal becomes relevant) was defined by Hübner *et al.* [16], and implemented in Jason [14]. More specifically, plans that share the same triggering condition refer to the achievement of the same goal, so that a goal can only be considered impossible for a given agent if all plans with the same triggering condition have been attempted and failed. In this extended AgentSpeak interpreter, these plans are modified so that the last action of every plan consists of testing for the fulfilment of the declared goal, and then the plans are grouped and executed in sequence until one finishes successfully. A plan only succeeds if at the end of its execution an agent can verify that its intended goal has been achieved. This approach retains the explicitly procedural approach to agent operation (a pre-compiled plan library describing sequences of steps that the agent can perform to accomplish its goals), only adding a more robust layer for handling plan-failure.

5.2 X-BDI

X-BDI [19] was the first agent model that includes a recognisably declarative goal semantics. An X-BDI agent is defined by a set of beliefs, a set of desires, and a set of operators that manipulate the world. The agent refines the set of desires through various constraints on the viability of each desire until it generates a set containing the highest priority desires that are possible and mutually consistent. During this process the agent selects the operators that will be applied to the world in order to fulfil the selected desires in a process that is analogous to planning. The key aspect of X-BDI is that desires express *world-states* rather than triggers for the execution of pre-defined plans, leaving the composition of plans from world-changing operators to the agent interpreter.

5.3 Formalisations of Declarative Goals

Several researchers have worked on a family of declarative agent languages and investigated possible semantics for these languages [20,17]. All of these languages have in common the notion that an agent is defined in terms of beliefs, goals and capabilities, which are interpreted in such a way as to select and apply capabilities in order to fulfil an agent's goals. These approaches have evolved from GOAL [20] into a declarative semantics very similar to that of X-BDI [19], in which an agent's desires express *world-states* which must be achieved by the agent selection and application of capabilities.

5.4 Discussion

In addition to the models described in this section, variations of the way an agent interpreter handles declarative goals have also been described. These approaches advocate the use of fast propositional planners to verify the existence of a sequence of actions that fulfil a declarative goal [18]. The planning process in this setting allows the consideration of the entire set of available operators to create new plans, providing a degree of flexibility to the agent's behaviour. Our research has not dealt with multi-agent issues so far, but the approach taken by Coo-BDI [21] to share plans between agents might provide an interesting extension to our architecture. The exchange of new plans might offset the sometimes significant time needed to create plans from scratch by allowing agents to request the help of other planning-capable agents.

The approaches in Sections 5.1 and 5.3 deal with important aspects of declarative goals in agent systems, such as the verification of accomplishment and logical properties of such systems. However, support for declarative goals in Jason still requires a designer to specify high-level plans, while the formalisms described by van Riemsdijk lack any analysis of the practicality of their implementation. Though X-BDI implements a truly declarative agent specification language, the language is very far from mainstream acceptance, and the underlying logic system used in X-BDI suffers from a stream of efficiency problems.

6 Concluding Remarks

In this paper we have demonstrated how the addition of a planning component can augment the capabilities of a plan library-based agent. In order to exploit the planning capability, the agent uses a special planning action to create high-level plans by composing specially designed plans within an agent's plan library. This assumes no modification in the AgentSpeak language, and allows an agent to be defined so that *built-in* plans can still be defined for common tasks, while allowing for a degree of flexibility for the agent to act in unforseen situations. Our system can also be viewed as a way to extend the declarative goal semantics proposed by Hübner *et al.* [16], in that it allows an agent designer to specify only desired world-states and basic capabilities, relying on the planning component to form plans at runtime. Even though the idea of translating BDI states into STRIPS problems is not new [18], our idea of an encapsulated planning action allows the usage of any other planning formalism sufficiently compatible with the BDI model.

Recent approaches to the programming of agents based on declarative goals rely on mechanisms of plan selection and verification. However, we argue that a declarative model of agent programming must include not only constructs for verifying the accomplishment of an explicit world-state (which is an important capability in any declarative agent), but also a way in which an agent designer can specify *only* the world states the agent has to achieve and the description of atomic operators allowing an underlying *engine* to derive plans at runtime. In this paper we argue that propositional planning can provide one such engine, drawing on agent descriptions that include atomic actions and desired states, and leaving the derivation of actual plans for the agent at runtime.

The addition of a planning component to a BDI agent model has been recently revisited by other researchers, especially by Sardiña *et al.* [22] and Walczak *et al.* [23]. The former describes a BDI programming language that incorporates Hierarchical Task Networks (HTN) planning by exploring the similarities between these two formalisms, but this approach fails to address the fact that designers must specify rules for HTN planning in the same way in which they would decompose multiple plans in a traditional BDI agent. The latter approach is based on a specially adapted planner to support the agent, preventing the model from taking advantage of novel approaches to planning.

The prototype implemented for the evaluation of the extensions described in this paper has been empirically tested for a number of small problems, but, further testing and refinement of this prototype is still required, for instance, to evaluate how interactions between the addition of new plans will affect the existing plan library. The system can also be improved in a number of ways in order to better exploit the underlying planner component. For example, the effort spent on planning can be moderated by a quantitative model of control, so that an agent can decide to spend a set amount of computational effort into the planning process before it concludes the goal is not worth pursuing. This could be implemented by changing the definition of $goal\_conj(Goals)$ to include a representation of motivational model $goal\_conj(Goals, Motivation)$, which can be used to tune the planner and set hard limits to the amount of planning effort devoted to achieving that specific desire.

As indicated above, the key contribution of this paper is a technique that allows procedural agent architectures to use state-space (and hence, declarative) planners to augment flexibility at runtime, thus leveraging advances in planning algorithms. It is important to point out that previous efforts exploring the use of HTN planning do not change the essential procedural mode of reasoning of the corresponding agent architectures, as argued by Sardiña *et al.* [22]. State-space planners operate on a declarative description of the desired goal state, and our conversion process effectively allows a designer to use an AgentSpeak-like language in a declarative way, something which previous planning architectures do not allow. Finally, we are currently working on addressing some of the limitations we have identified regarding the generation and execution of concurrent plans for multiagent scenarios, considering the use of external imported plans such as in Coo-AgentSpeak [24].

Acknowledgments. The first author is supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) of the Brazilian Ministry of Education. We would like to thank Rafael Bordini and Jomi Hübner for their support regarding the programming of AgentSpeak agents in their Jason implementation, as well as the discussion of many issues regarding planning and declarative goals.

References

1. Ingrand, F.F., Georgeff, M.P., Rao, A.S.: An architecture for real-time reasoning and system control. IEEE Expert, Knowledge-Based Diagnosis in Process Engineering 7(6), 33–44 (1992)
2. 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), 5–53 (2004)
3. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Perram, J., Van de Velde, W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996)
4. van Riemsdijk, B., van der Hoek, W., Meyer, J.J.C.: Agent programming in dribble: from beliefs to goals using plans. In: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 393–400. ACM Press, New York (2003)
5. Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E.: Multi-Agent Programming: Languages, Platforms and Applications. In: Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 15, Springer, Heidelberg (2005)
6. Rao, A.S., Georgeff, M.P.: BDI-agents: from theory to practice. In: Proceedings of the First International Conference on Multiagent Systems, San Francisco, pp. 312–319 (1995)
7. d'Inverno, M., Luck, M.: Engineering AgentSpeak(L): A formal computational model. Journal of Logic and Computation 8(3), 233–260 (1998)
8. Georgeff, M.P., Ingrand, F.F.: Monitoring and control of spacecraft systems using procedural reasoning. In: Proceedings of the Space Operations and Robotics Workshop, Houston, USA (1989)

9. Fox, M., Long, D.: PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. Journal of Artificial Intelligence Research 20, 61–124 (2003)
10. Fikes, R., Nilsson, N.: STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 2(3-4), 189–208 (1971)
11. Thangarajah, J., Harland, J., Morley, D., Yorke-Smith, N.: Aborting tasks in BDI agents. In: Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 8–15 (2007)
12. Bordini, R.H., Hübner, J.F.: Bdi agent programming in agentspeak using jason. In: Toni, F., Torroni, P. (eds.) Computational Logic in Multi-Agent Systems. LNCS (LNAI), vol. 3900, pp. 143–164. Springer, Heidelberg (2006)
13. Sardina, S., Padgham, L.: Goals in the context of BDI plan failure and planning. In: Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 16–23 (2007)
14. 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, pp. 3–37. Springer, Heidelberg (2005)
15. Bylander, T.: The computational complexity of propositional STRIPS planning. Artificial Intelligence 69(1-2), 165–204 (1994)
16. Hübner, J.F., Bordini, R.H., Wooldridge, M.: Programming declarative goals using plan patterns. In: Baldoni, M., Endriss, U. (eds.) DALT 2006. LNCS (LNAI), vol. 4327, pp. 123–140. Springer, Heidelberg (2006)
17. van Riemsdijk, M.B., Dastani, M., Meyer, J.J.C.: Semantics of declarative goals in agent programming. In: Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, Utrecht, The Netherlands, pp. 133–140. ACM Press, New York (2005)
18. Meneguzzi, F.R., Zorzo, A.F., Móra, M.D.C.: Propositional planning in BDI agents. In: Proceedings of the 2004 ACM Symposium on Applied Computing, Nicosia, Cyprus, pp. 58–63. ACM Press, New York (2004)
19. Móra, M.d.C., Lopes, J.G.P., Vicari, R.M., Coelho, H.: BDI models and systems: Bridging the gap. In: Rao, A.S., Singh, M.P., Müller, J.P. (eds.) ATAL 1998. LNCS (LNAI), vol. 1555, pp. 11–27. Springer, Heidelberg (1999)
20. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.J.C.: Agent programming with declarative goals. In: Castelfranchi, C., Lespérance, Y. (eds.) ATAL 2000. LNCS (LNAI), vol. 1986, pp. 228–243. Springer, Heidelberg (2001)
21. Ancona, D., Mascardi, V.: Coo-BDI: Extending the BDI Model with Cooperativity. In: Leite, J.A., Omicini, A., Sterling, L., Torroni, P. (eds.) DALT 2003. LNCS (LNAI), vol. 2990, pp. 109–134. Springer, Heidelberg (2004)
22. 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, pp. 1001–1008. ACM Press, New York (2006)
23. Walczak, A., Braubach, L., Pokahr, A., Lamersdorf, W.: Augmenting BDI Agents with Deliberative Planning Techniques. In: Programming Multi-Agent Systems, 4th International Workshop. LNCS, vol. 4411, pp. 113–127 (2006)
24. Ancona, D., Mascardi, V., Hübner, J.F., Bordini, R.H.: Coo-agentspeak: Cooperation in agentspeak through plan exchange. In: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 696–705 (2004)