Plan Patterns for Declarative Goals in AgentSpeak

Jomi Fred Hübner
Univ. Regional de Blumenau
Depto. de Sist. e Computação
Blumenau, SC, Brazil
jomi@inf.furb.br

Rafael H. Bordini
University of Durham
Dept. of Computer Science
Durham, UK
r.bordini@durham.ac.uk

Michael Wooldridge
University of Liverpool
Dept. of Computer Science
Liverpool, UK
mjw@csc.liv.ac.uk

1. INTRODUCTION

In this paper, we consider the use of *declarative goals* in the AgentSpeak programming language, originally introduced in [6]. By a declarative goal, we mean a goal that *explicitly* represents a state of affairs to be achieved, in the sense that, if an agent has a goal $p(t_1, \ldots, t_n)$ then it expects to eventually believe $p(t_1, \ldots, t_n)$, and only then can the goal be considered achieved. Currently, although goals form a central component of AgentSpeak programming, they are only *implicit* in the plans defined by the agent programmer. For example, there is no explicit way of expressing that a goal should be maintained until a certain condition holds; such temporal goal structures are hidden within the plan structures themselves. While one possibility would be to extend the language to introduce an explicit construct for declarative goals and a goal base, we show that this is unnecessary. We introduce a number of *plan patterns*, corresponding to common types of explicit temporal/conditional (declarative) goal structures, and show how these can be mapped into AgentSpeak code. Thus, a programmer or designer can conceive of a goal at the declarative level, and this goal will be "translated" via these patterns into standard AgentSpeak code.

2. PLAN FAILURE

In order to present plan patterns for declarative goals, the *plan failure* handling mechanism implemented in ***Jason*** [2], and some specific *internal actions* used for dropping goals need to be presented.

We identify three cases of plan failure. The first cause of failure is a *lack of relevant or applicable plans*, which can be understood as the agent "not knowing how to do something". The second is where a test goal fails; that is, where the agent "expected" to believe something about the world, but that did not hold as expected. The third is when an action within a plan fails. Regardless of the reason for a plan failing, the interpreter generates a goal deletion event (represented by `-!`g) if the achievement of the goal g has failed. This event may be handled by a plan written to deal with such failure; e.g., the plan "`-!`g `: true <- !`g`.`" will attempt the achieve goal g again when a plan to achieve g has failed.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AAMAS'06 May 8–12 2006, Hakodate, Hokkaido, Japan.
Copyright 2006 ACM 1-59593-303-4/06/0005 ...$5.00.

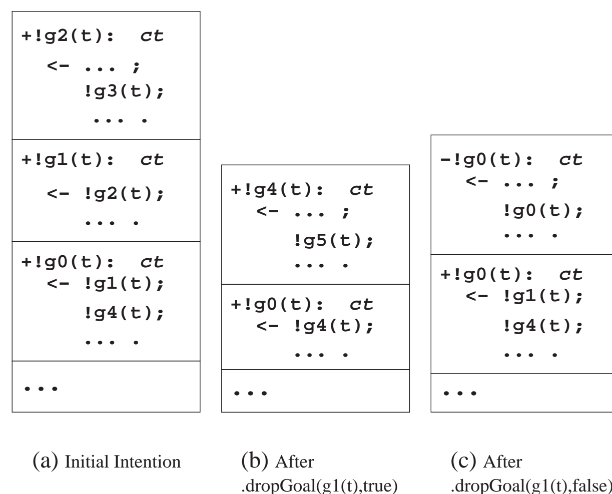

Figure 1: Internal Actions for Dropping Goals

In the next section, we also make use of the following predefined internal actions. The first, `.dropGoal(`g`,true)`, terminates g successfully, as if g were somehow achieved, while `.dropGoal(`g`,false)` terminates g with failure, meaning that g was impossible to achieve (in this case, a failure event for the plan that generated goal g is generated). It is perhaps easier to understand how these actions work with reference to Figure 1. The figure shows the consequence of each of these internal actions being executed (the plan where the internal action appeared is not shown; it is likely to be within another intention). Note that the state of the intention affected by the execution of one of these internal actions, as shown in the figure, is not the immediate resulting state (at the end of the reasoning cycle where the internal action was executed) but the most significant next state of the changed intention.

3. DECLARATIVE GOAL PATTERNS

Although goals form a central component of the AgentSpeak conceptual framework, the language itself does not provide any explicit constructs for handling goals with complex temporal structures. For example, a system designer and programmer will often think in terms of goals such as "maintain P until Q becomes true", or "prevent P from becoming true". Creating AgentSpeak code to realise such complex goals has, to date, been largely an ad hoc process, dependent upon the experience of the programmer. Below, we define a number of *plan patterns* for declarative goals — that is, complex combinations of plan structures which frequently occur in

actual scenarios — and to show how these can be realised in terms of AgentSpeak code. As we shall see, such patterns can be used to implement, in a systematic way, certain types of declarative goals and the types of commitments they represent, as discussed e.g. by Cohen and Levesque [4].

As an initial motivational example for declarative goals, consider a robot agent with the goal of being at some location (represented by the predicate `l(X,Y)`) and the following plan to achieve this goal, provided the battery charge is greater than 20% (the predicate `bc/1` stands for "battery charge", and `go` identifies an action that the robot is able to perform in the environment):

```
+!l(X,Y) : bc(B) & B > 0.2 <- go(X,Y).
```

Sometimes, using an AgentSpeak plan as a procedure can be a quite useful programming tool. Thus, in a way, it is important that the AgentSpeak interpreter does not enforce any declarative semantics to its only (syntactically defined) goal construct. However, in the plan above, `l(X,Y)` is clearly meant as a *declarative goal*; that is, the programmer expects the robot to believe `l(X,Y)` (by perceiving the environment), if the plan executes to completion. If this does not happen because, e.g., the environment is dynamic or non-deterministic, the goal cannot be considered achieved.

This is where our plan patterns for declarative goals help to express this type of situation without adding new syntactic/semantic constructs to AgentSpeak. We can easily transform the above procedural goal into a declarative goal by adding a corresponding *test goal* at the end of the plan's body, as follows:

```
+!l(X,Y) : bc(B) & B > 0.2
   <- go(X,Y);
      ?l(X,Y).
```

This plan only succeeds if the goal is actually achieved; if the given (procedural) plan executes to completion (without failing) but the goal happens not to be achieved, the test goal at the end will fail.

This solution forms a *plan pattern*, and other patterns can be applied to solve other similar problems. Our approach to the inclusion of declarative goals in the AgentSpeak programming language is inspired by the successful adoption of design patterns in object-oriented design. To represent such patterns for AgentSpeak, we will define skeleton programs with meta variables. For example, the pattern for a simple declarative goal, as the one used in the robot's location goal, is defined by the following AgentSpeak plan:

```
+!g : c <- p; ?g.
```

Here, g is a meta variable that represents the declarative goal, c is a meta variable that represents the context expression stating in which circumstances the plan is applicable, and p represents the procedural part of the plan body (i.e., a course of actions to achieve g). Note that, with the introduction of the final test goal, this plan to achieve g finishes successfully only if the agent believes g after the execution of plan body p.

To simplify the use of the patterns, we introduce *pattern rules*, which rewrite a set of AgentSpeak plans into a new AgentSpeak program according to a given pattern. The following pattern rule, called **DG** (Declarative Goal), shows how procedural goals are transformed into declarative goals. The pattern rule name is followed by the parameters which need to be provided by the programmer, besides the actual code (i.e., sequence of plans) on which the pattern will be applied.

```
+!g : c1 <- p1.
+!g : c2 <- p2.
...
+!g : cn <- pn.
------------------------------------------ DG_g  (n ≥ 1)
+!g : g <- true.
+!g : c1 <- p1; ?g.
+!g : c2 <- p2; ?g.
...
+!g : cn <- pn; ?g.
+g : true <- .dropGoal(g, true).
```

Essentially, this rule adds `?g` to the end of every plan in the plan library that has `+!g` as triggering event, and creates two extra plans. The first additional plan checks whether goal g has already been achieved — in such case, there is nothing else to do. The second additional plan is triggered when the agent perceives that g has been achieved while it is executing any of the courses of action p_i ($1 \leq i \leq n$) which aim at achieving g; if that occurs, the plan being executed in order to achieve g can be immediately terminated (with success). The internal action `.dropGoal(g, true)` can be used for this.

In this pattern, when one of the plans to achieve g fails, the agent gives up achieving the goal altogether. However, it could be the case that for such a goal, the agent should try another plan to achieve it, similarly to the "backtracking" plan selection mechanism available in platforms such as JACK [9] and 3APL [5]. The *Backtracking Declarative Goal* rule (**BDG**), defines this pattern based on a set of AgentSpeak plans $\mathcal{P}$ transformed by the **DG** pattern (each plan in $\mathcal{P}$ is of the form `+!g : c <- p`):

```
P
------------------------------ BDG_g
DG_g(P)
-!g : true <- !g.
```

The last plan of the pattern catches a failure event, caused when a plan from $\mathcal{P}$ fails, and then tries to achieve that same goal g again. Notice that it is possible that the same (failed) plan is selected again, causing a loop if the plan contexts have not been carefully programmed (and the environment does not change).

In the pattern above, despite the various alternative plans, the agent can still end up dropping the intention with the goal g unachieved if none of those plans happen to be applicable. Conversely, with a *blind commitment goal*, the agent can drop the goal only when it is achieved:

```
P
------------------------------ BCG_g
BDG(P)
+!g : true <- !g.
```

The **BCG** pattern is based on the **BDG** pattern and extends it by adding a plan that ensures that the agent will still pursue goal g even when all known plans to achieve g are not applicable.

For most applications, **BCG**-style fanatical commitment is too strong. For example, if a robot has the goal to be at some location, it is reasonable that it can drop this goal in case its battery charge is getting very low; in other words, the agent has realised that it has become impossible to achieve the goal, so it is useless to keep trying. This is very similar to the idea of a persistent goal in the work of Cohen and Levesque: a persistent goal is a goal that is maintained as long as it is believed not achieved, but still believed possible [4]. The drop condition f (e.g., "low battery charge") is

used in the Single-Minded Commitment (**SMC**) pattern to allow the agent to drop a goal if it becomes impossible:

$$\frac{\mathcal{P}}{\mathbf{BCG}_{g,BDG}(\mathcal{P})} \ \mathbf{SMC}_{g,f}$$

```
+f : true <- .dropGoal(g, false).
```

This pattern extends the **BCG** pattern by adding the drop condition represented by the literal f in the last plan. If the agent comes to believe f, it can drop goal g, signalling failure. This effectively means that the plan which generated goal g must itself fail (as g is now impossible to achieve). However, there might be an alternative for that plan which does not depend on g, so failure handling for that plan may take care of such situation.

As we have a failure drop condition for a goal, we can also have a success drop condition, e.g., because the motivation to achieve the goal has ceased to exist. Suppose a robot has the goal of going to the fridge because its owner has asked it to fetch a beer; if the robot realises that its owner does not want a beer anymore, it should drop the goal [4]. The belief "my owner wants a beer" is the *motivation* m for the goal.

Of course we can combine the drop condition and motivation to create a goal which can be dropped if it has been achieved, has become impossible to achieve, or the motivation to achieve the goal no longer exists. This *Open-Minded Commitment* (**OMC**) pattern is defined as follows:

$$\frac{\mathcal{P}}{\mathbf{BCG}_{g,BDG}(\mathcal{P})} \ \mathbf{OMC}_{g,f,m}$$

```
+f : true <- .dropGoal(g, false).
-m : true <- .dropGoal(g, true).
```

For example, a drop condition could be "no beer at location (X,Y)" (denoted below by `~b(X,Y)`), and the motivational condition could be "my owner wants a beer" (denoted below by `wb`). Consider the initial plan below, which is intended to achieve goal `l(X,Y)`:

```
+!l(X,Y) : bc(B) & B > 0.2 <- go(X,Y).
```

When the pattern $\mathbf{OMC}_{l(X,Y),\sim b(X,Y),wb}$ is applied to the plan above, we get the following program:

```
+!l(X,Y) : l(X,Y) <- true.
+!l(X,Y): bc(B) & B > 0.2
   ← go(X,Y); ?l(X,Y).
+!l(X,Y) : true <- !l(X,Y).
-!l(X,Y) : true <- !l(X,Y).
+~b(X,Y) : true <- .dropGoal(l(X,Y), false).
-wb : true <- .dropGoal(l(X,Y), true).
```

The approach of defining goals with complex temporal structures by means of patterns can also be used to specify more complex declarative goals, such as, for example, maintenance goals (the agent needs to ensure that the state of the world will always be such that g holds) and sequenced goals (i.e., when various instances of a goal should not be adopted concurrently).

4. CONCLUSIONS

In this paper we have showed that sophisticated types of goal structures can be implemented in the AgentSpeak language, with only the extensions (and extensibility mechanisms) available in ***Jason***. This is done by combining certain types of AgentSpeak plans, forming certain patterns for each type of goal and commitment towards goals that agents might have. Our approach is to take advantage of the simplicity of the AgentSpeak language, using only its well-known support for procedural goals plus the idea of patterns to support the use of declarative goals in AgentSpeak programming.

Despite the use of pre-defined internal actions such as `.dropGoal` (which had already been conceived for use with procedural goals in ***Jason***), our approach requires neither syntactical nor semantical changes in the language (as done, for example, in [10, 3]), nor the usual addition of a goal base (see e.g. [7]). Van Riemsdijk *et al.* [8] also pointed out that it is possible to build declarative goals using the procedural goals available in 3APL. Their idea correspond to our **BDG** pattern. In this paper, we have defined various other types of declarative goals and represented them as *patterns* of AgentSpeak plans. Another advantage of our approach is that, as complex types of goals are mapped down to plain AgentSpeak code using patterns, programmers can change the patterns to suit their needs, or indeed create new patterns easily.

Acknowledgements

Rafael Bordini gratefully acknowledges the support of The Nuffield Foundation (grant number NAL/01065/G).

5. REFERENCES

[1] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms, and Applications*. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer, 2005.

[2] R. H. Bordini, J. F. Hübner, and R. Vieira. ***Jason*** and the Golden Fleece of agent-oriented programming. In Bordini et al. [1], chapter 1.

[3] L. Braubach, A. Pokahr, W. Lamersdorf, and D. Moldt. Goal representation for BDI agent systems. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Second Int. Workshop on Programming Multiagent Systems: Languages and Tools (ProMAS 2004)*, pages 9–20, 2004.

[4] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(3):213–261, 1990.

[5] M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Programming multi-agent systems in 3APL. In Bordini et al. [1], chapter 2.

[6] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. Perram, editors, *Proc. of MAAMAW'96, 22–25 January, Eindhoven, The Netherlands*, number 1038 in LNAI, pages 42–55, London, 1996. Springer-Verlag.

[7] B. van Riemsdijk, M. Dastani, and J.-J. C. Meyer. Semantics of declarative goals in agent programming. In F. Dignum *et al.*, editors, *Proc. of AAMAS 2005*, pages 133–140. ACM, 2005.

[8] M. B. van Riemsdijk, M. Dastani, and J.-J. C. Meyer. Subgoal semantics in agent programming. In C. Bento, A. Cardoso, and G. Dias, editors, *Proc. of EPIA 2005, Covilhã, Portugal, December 5-8, 2005*, volume 3808 of *LNCS*, pages 548–559, 2005.

[9] M. Winikoff. Jack intelligent agents: An industrial strength platform. In Bordini et al. [1], chapter 7.

[10] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and procedural goals in intelligent agent systems. In *Proc. of 8th Int. Conf. on Principles of Knowledge Representation and Reasoning*, 2002.