# Programming Declarative Goals Using Plan Patterns

Jomi Hübner[1], Rafael H. Bordini[2], and Michael Wooldridge[3]

[1] University of Blumenau (Brazil)
jomi@inf.furb.br

[2] University of Durham (UK)
R.Bordini@durham.ac.uk

[3] University of Liverpool (UK)
mjw@csc.liv.ac.uk

**Abstract.** AgentSpeak is a well-known language for programming intelligent agents which captures the key features of reactive planning systems in a simple framework with an elegant formal semantics. However, the original language is too abstract to be used as a programming language for developing multi-agent system. In this paper, we address one of the features that are essential for a pragmatical agent programming language. We show how certain *patterns* of AgentSpeak plans can be used to define various types of declarative goals. In order to do so, we first define informally how plan failure is handled in the extended version of AgentSpeak available in ***Jason***, a Java-based interpreter; we also define special (internal) actions used for dropping intentions. We present a number of *plan patterns* which correspond to elaborate forms of declarative goals. Finally, we give examples of the use of such types of declarative goals and describe how they are implemented in ***Jason***.

## 1 Introduction

The AgentSpeak(L) language, introduced by Rao in 1996, provides a simple and elegant framework for intelligent action via the run-time interleaved selection and execution of plans. Since the original language was proposed, substantial progress has been made both on the theoretical foundations of the language (e.g., its formal semantics [6]), and on its use, via implementations of practical extensions of AgentSpeak [5]. However, one problem with the original AgentSpeak(L) language is that it lacks many of the features that might be expected by programmers in practical development. Our aim in this paper is to focus on the integration of one such features, namely the definition of declarative goals and the use of plan patters. Throughout the paper, we use AgentSpeak as a more general reference to AgentSpeak(L) and its extensions.

In this paper, we consider the use of *declarative goals* in AgentSpeak programming. 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 $p(t_1, \ldots, t_n)$, it expects to eventually believe $p(t_1, \ldots, t_n)$ (cf. [19]) and only then can the goal be considered achieved. Moreover, we are interested not only in goals representing states of affairs, but goals that may have complex temporal structures. 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 defined implicitly, within the plans themselves, and by *ad hoc* efforts on the part of programmers.

While one possibility would be to extend the language and its formal semantics to introduce an explicit notion of declarative goal (as done in other languages, e.g., [19, 7, 22]), we show that this is unnecessary. We introduce a number of *plan patterns*, corresponding to common types of explicit temporal (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 expanded, via these patterns, into standard AgentSpeak code. We then show how such goal patterns can be used in ***Jason***, a Java-based implementation of an extended version of AgentSpeak [4].

In order to present the plan patterns that can be used for defining certain types of declarative goals discussed in the literature, the *plan failure* handling mechanism implemented in ***Jason***, and some pre-defined *internal actions* used for dropping goals, need to be presented. Being able to handle plan failure is useful not only in the context of defining plan patterns that can represent complex declarative goals. In most practical scenarios, plan failure is not only possible, it is commonplace: a key component of rational action in humans is the ability to handle such failures. After presenting these features of ***Jason*** that are important in controlling the execution of plans, we can then show the plan patterns that define more complex types of goals than has been claimed to be possible in AgentSpeak [7]. We present (declarative) maintenance as well as achievement goals, and we present different forms of commitments towards goal achievement/maintenance (e.g., the well-known blind, single-minded, and open-minded forms of commitment [18]). Finally, we discuss ***Jason*** implementations of examples that appeared in the literature on declarative goals; the examples also help in showing why declarative goals with complex temporal structures are an essential feature in programming multi-agent systems.

## 2 Goals and Plans in AgentSpeak

In [17], Rao introduced the AgentSpeak(L) programming language. It is a natural extension of logic programming for the BDI agent architecture, and provides an elegant abstract framework for programming BDI agents. In this paper, we only give a very brief introduction to AgentSpeak; see e.g. [6] for more details.

An AgentSpeak agent is created by the specification of a set of initial beliefs and a set of plans. A *belief atom* is simply a first-order predicate in the usual notation, and belief atoms or their negations are *belief literals*. The initial beliefs define the state of the belief base at the moment the agent starts running; the belief base is simply a collection of ground belief atoms (or, in ***Jason***, literals).

AgentSpeak distinguishes two types of goals: *achievement goals* and *test goals*. Achievement goals are predicates (as for beliefs) prefixed with the '`!`' operator, while test goals are prefixed with the '`?`' operator. Achievement goals state that the agent wants to achieve a state of the world where the associated predicate is true. (In practice,

these lead to the execution of other plans.) A *test goal* states that the agent wants to test whether the associated predicate is a belief (i.e., whether it can be unified with one of the agent's beliefs).

Next, the notion of a *triggering event* is introduced. It is a very important concept in this language, as triggering events define which events may initiate the execution of plans; the idea of *event*, both internal and external, will be made clear below. There are two types of triggering events: those related to the *addition* ('+') and *deletion* ('−') of mental attitudes (beliefs or goals).

Plans refer to the *basic actions* that an agent is able to perform on its environment. Such actions are also defined as first-order predicates, but with special predicate symbols (called *action symbols*) used to distinguish them. The actual syntax of AgentSpeak programs is based on the definition of plans, as follows. 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 $e$ `:` $b_1$ `&` ... `&` $b_m \leftarrow h_1$ `;` ... `;` $h_n$`.` is a *plan*.

An AgentSpeak(L) plan has a *head* (the expression to the left of the arrow), which is formed from a triggering event (denoting the purpose for that plan), and a conjunction of belief literals representing a *context* (separated from the triggering event by ':'). The conjunction of literals in the context must be satisfied if the plan is to be executed (the context must be a logical consequence of that agent's current beliefs). A plan also has a *body*, which is a sequence of basic actions or (sub)goals that the agent has to achieve (or test) when the plan is triggered.

Besides the belief base and the plan library, the AgentSpeak interpreter also manages a set of *events* and a set of *intentions*, and its functioning requires three *selection functions*. The event selection function selects a single event from the set of events; another selection function selects an "option" (i.e., an applicable plan) from a set of applicable plans; and a third selection function selects one particular intention from the set of intentions. The selection functions are supposed to be agent-specific, in the sense that they should make selections based on an agent's characteristics in an application-specific way. An event has the form $\langle te, i \rangle$, where $te$ is a plan triggering event (as in the plan syntax described above) and $i$ is that intention that generated the event or $\mathsf{T}$ for external events.

*Intentions* are particular courses of actions to which an agent has committed in order to handle certain events. Each intention is a stack of partially instantiated plans. Events, which may start the execution of plans that have relevant triggering events, can be *external*, when originating from perception of the agent's environment (i.e., addition and deletion of beliefs based on perception are external events); or *internal*, when generated from the agent's own execution of a plan (i.e., a subgoal in a plan generates an event of type "addition of achievement goal"). In the latter case, the event is accompanied with the intention which generated it (as the plan chosen for that event will be pushed on top of that intention). External events create new intentions, representing separate focuses of attention for the agent's acting within the environment.

## 3 Plan Failure

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". This happens either because the agent simply does not have the know-how (in case it has no relevant plans) — this could happen through simple omission (the programmer did not provide any appropriate plans) — or because all known ways of achieving the goal cannot currently be used (there are known plans but whose contexts do not match the agent's current beliefs). The second is where a test goal fails; that is, where the agent "expected" to believe in a certain condition of the world, but in fact the condition did not hold. The third is where an internal action ("native method"), or a basic action (the effectors within the agent architecture are assumed to provide feedback to the interpreter stating whether the requested action was executed or not), fails.

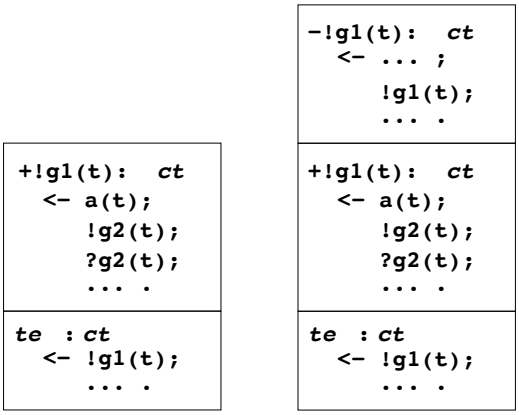
Regardless of the reason for a plan failing, the interpreter generates a goal deletion event (i.e., an event for "$-!g$") if the corresponding goal achievement ($+!g$) has failed. This paper introduces for the first time an (informal) semantics for the notion of goal deletion as used in ***Jason***. In the original definition, Rao syntactically defined the possibility of goal deletions as triggering events for plans (i.e., triggering event with `-!` and `-?` prefixes), but did not discuss what they meant. Neither was goal deletion discussed in further attempts to formalise AgentSpeak or its ancestor dMars [12, 11]. Our own choice was to use this as some kind of plan failure handling mechanism[4], as discussed below (even though this was probably not what they originally were intended for).

The idea is that a plan for a goal deletion is a "clean-up" plan, executed prior to (possibly) "backtracking" (i.e., attempting another plan to achieve the goal for which a plan failed). One of the things programmers might want to do within the goal deletion plan is to attempt again to achieve the goal for which the plan failed. In contrast to conventional logic programming languages, during the course of executing plans for subgoals, AgentSpeak programs generate a sequence of actions that the agent performs on the external environment so as to change it, the effects of which cannot be undone by simply backtracking (i.e., it may require further action in order to do so). Therefore, in certain circumstances one would expect the agent to have to "undo" the effects of certain actions before attempting some alternative courses of action to achieve that goal, and this is precisely the practical use of plans with goal deletions as triggering events.

It is important to observe that omitting possible goal deletion plans for existing goal additions implicitly denotes that such goal should never be backtracked, i.e., no alternative plan for it should be attempted in case one fails. To specify that backtracking should always be attempted (e.g., until special internal actions in the plan explicitly cause the intention to be dropped), all the programmer has to do is to specify a goal deletion plan (for a given goal $g$ addition) with empty context and the same goal in the body, as in "`-!`$g$`: true` $\leftarrow$ `!`$g$`.`".

[4] The notation $-!g$, i.e., "goal deletion" also makes sense for such plan failure mechanism; if a plan fails there is a possibility that the agent may need to drop the goal altogether, so it is to handle such event (of the possible need to drop a goal) that plans of the form $-!g : \ldots$ are written.

When a failure happens, the whole intention is dropped if the triggering event of the plan being executed was neither an achievement nor a test goal *addition*: only these can be attempted to recover from failure using the goal deletion construct (one cannot have a goal deletion event posted for a failure in a goal deletion plan). In any other circumstance, a failed plan means that the whole intention cannot be achieved. If a plan for a goal addition (`+!`$g$) fails, the intention $i$ where that plan appears is suspended, and the respective goal deletion event ($\langle -!g, i\rangle$) is included in the set of events. Eventually, this might lead to the goal addition being attempted again as part of the plan to handle the `-!`$g$ event. When the plan for `-!`$g$ finishes not only itself but also the failed `+!`$g$ plan below it[5] are removed from the intention. As it will be clear later, it is a programmer's decision to attempt the goal again or not, or even to drop the whole intention (possibly with special internal action constructs, whose informal semantics is given below), depending on the circumstances. What happens when a plan fails is shown in Figure 1.




(a) An Intention before Plan Failure

(b) That Intention after Plan Failure

**Fig. 1.** Plan Failure.

In the circumstance described in Figure 1(a) above, suppose $a(t)$ fails, or otherwise after that action succeeds an event for $+!g2(t)$ was created but there were no applicable plans to handle the event, or $?g2(t)$ is not is the belief base, nor there are applicable plans to handle a $+?g2(t)$ event. In any of those cases, the intention is suspended and an event for $-!g1(t)$ is generated. Assuming the programmer included a plan for $-!g1(t)$, and the plan is applicable at the time the event is selected, the intention will eventually look as in Figure 1(b). Otherwise the original goal addition event is re-posted or the whole intention dropped, depending on a setting of the ***Jason*** interpreter that is configurable by programmers. (See [1] for an overview of how various BDI systems deal with the problem of there being no applicable plans.)

The reason why not providing goal deletion plans in case a goal is not to be backtracked works is because an event (with the whole suspended intention within it) is discarded in case there are no relevant plans for a generated goal deletion. In general, the lack of relevant plans for an event indicates that the perceived event is not significant for the agent in question, so they are simply ignored. An alternative approach for handling the lack of relevant plans is described in [2], where it is assumed that in some cases, explicitly specified by the programmer, the agent will want to ask other agents how to

[5] The failed plan is left in the intention, for example, so that programmers could check which plan failed (e.g., by means of ***Jason*** internal actions).

handle such events. The mechanism for plan exchange between AgentSpeak agents presented in [2] allows the programmer to specify which triggering events should generate attempts to retrieve external plans, which plans an agent agrees to share with others, what to do once the plan has been used for handling that particular event instance, and so on.

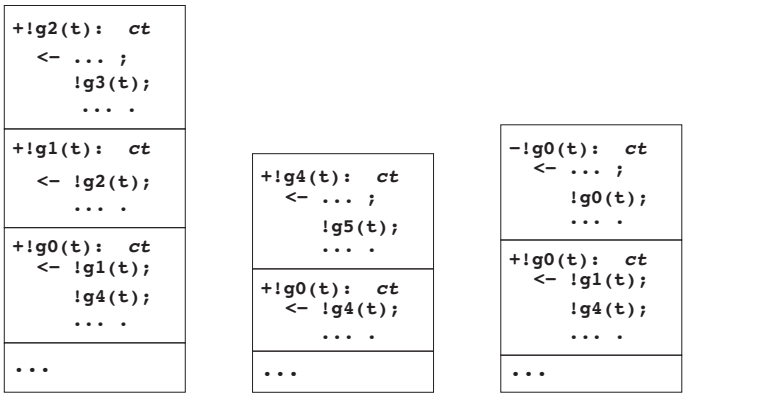
In the next section, besides the plan failure handling mechanism, we also make use of a particular standard internal action. Standard internal actions, unlike user-defined internal actions, are those available with the ***Jason*** distribution; they are denoted by an action name starting with symbol '`.`'. Some of these pre-defined internal actions manipulate the structure used in giving semantics to the AgentSpeak interpreter. For that reason, they need to be precisely defined. As the focus here is on the use of patterns for defining declarative goals, we will give only informal semantics to the internal action we refer to in the next section.

The particular internal action used in this paper is `.dropGoal(`$g$`,true)`. Any intention that has the goal $g$ in the triggering event of any of its plans will be changed as follows. The plan with triggering event `+!`$g$ is removed and the plan below that in the stack of plans forming that intention carries on being executed at the point after goal $g$ appeared. Goal $g$, as it appears in the `.dropGoal` internal action is used to further instantiate the plan where the goal that was terminated early appears. With `.dropGoal(`$g$`,false)`, the plan for `+!`$g$ is also removed, but an event for the deletion of the goal whose plan body required $g$ is generated: this informally means that there is no way of achieving $g$ so the plan requiring $g$ to be achieved must fail. That is, `.dropGoal(`$g$`,true)` is used when the agent realises the goal has already been achieved so whatever plan was being executed to achieve that goal does not need to be executed any longer. On the other hand, `.dropGoal(`$g$`,false)` is used when the agent realises that the goal has become impossible to achieve, hence the need to fail the plan that required $g$ being achieved as one of its subgoals.

It is perhaps easier to understand how these actions work with reference to Figure 2. 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.

## 4 Declarative Goal Patterns

Although goals form a central component of the AgentSpeak conceptual framework, it is important to note that the language itself does not provide any explicit constructs for handling goals with complex temporal structure. 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. Our aim in this section is firstly to define a number of declarative goal structures, and secondly to show how these can be realised in terms




(a) Initial Intention

(b) After `.dropGoal(g1(t),true)`

(c) After `.dropGoal(g1(t),false)`

**Fig. 2.** Standard Internal Actions for Dropping Goals.

of *patterns* of AgentSpeak plans — that is, complex combinations of plan structures which are often useful in actual scenarios. As we shall see, such patterns can be used to implement, in a systematic way, not only complex types of declarative goals, but also the types of commitments they represent, as discussed for example by Cohen and Levesque [8].

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:

`+!l(X,Y): bc(B) & B` $> 0.2 \leftarrow$ `go(X,Y).`

where the predicate `bc/1` stands for "battery charge", and `go` identifies an action that the robot is able to perform in the environment.

At times, 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 it fails because, say, the environment is dynamic, the goal cannot be considered achieved and, normally, should be attempted again.

This type of situation is commonplace in multi-agent system, and this is why it is important to be able to define declarative goals in agent-oriented programming. However, this can be done without the need to change the language and/or its semantics. As similarly pointed out by van Riemsdijk *et al*. [19], 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 (believed to be ) achieved; if the given (procedural) plan executes to completion (i.e., without failing) but the goal happens not to be achieved, the test goal at the end will fail. In this way, we have taken a simple *procedural* goal and transformed it into a *declarative* goal – the goal to achieve some state of affairs.

This solution forms a plan pattern, which can be applied to solve other similar problems which, as we mention above, are commonplace in agent programming. Thus, our approach to include declarative goals in AgentSpeak programming is inspired by the successful adoption of design patterns in object oriented design [13]. To represent such patterns for AgentSpeak, we shall make use of skeleton programs with meta variables. For example, the general form of an AgentSpeak plan for a simple declarative goal, as the one used in the robot's location goal above, is as follows:

```
+!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 action 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 also define pattern rules which rewrite a set of AgentSpeak plans into a new AgentSpeak program according to a given pattern.[6] The following pattern rule, called **DG** (Declarative Goal), is used to transform procedural goals 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., a set of plans) on which the pattern will be applied.

```
+!g: c1 ← p1.
+!g: c2 ← p2.
...
+!g: cn ← pn.
```

$$\text{---------- } \mathbf{DG}_g \quad (n \geq 1)$$

```
+!g: g ← true.
+!g: c1 ← p1; ?g.
+!g: c2 ← p2; ?g.
...
+!g: cn ← pn; ?g.
+g: true ← .dropGoal(g, true).
```

Essentially, this rule adds `?`$g$ at the end of each plan in the given set of plans which has `+!`$g$ as trigger event, and creates two extra plans (the first and the last plans above). The

[6] Note that some of the patterns presented in this paper require the atomic execution of certain plans, but we avoid including this in the patterns for clarity of presentation; this feature is available in ***Jason*** through a simple plan annotation.

first plan checks whether the goal $g$ has already been achieved — in such case, there is nothing else to do. That last 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 $g$; in this circumstance, the plan being executed in order to achieve $g$ can be immediately terminated. The internal action `.dropGoal(`$g$`, true)` terminates such plan with success (as explained in Section 3).

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 goal, the agent should try another plan to achieve it, as in the "backtracking" plan selection mechanism available in platforms such as JACK [21,14] and 3APL [10,9]. In those mechanisms, usually only when all available plans have been tried in turn and failed is the goal abandoned with failure, or left to be attempted again later on. The following rule, called **BDG** (Backtracking Declarative Goal), defines this pattern based on a set of conventional AgentSpeak plans $\mathcal{P}$ transformed by the **DG** pattern (each plan in $\mathcal{P}$ is of the form `+!`$g$`:` $c$ $\leftarrow$ $p$):

$$\frac{\mathcal{P}}{\textbf{DG}_g(\mathcal{P})} \; \textbf{BDG}_g$$

`-!`$g$`: true` $\leftarrow$ `!`$g$`.`

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 plan is selected and fails again, causing a loop if the plan contexts have not been carefully programmed. Thus the programmer would need to specify the plan contexts in such a way that a plan is only applicable if it has a chance of succeeding regardless of it having been tried already (recently).

Instead of worrying about defining contexts in such more general way, in some cases it may be useful for the programmer to apply the following pattern, called **EBDG** (Exclusive BDG), which ensures that none of the given plans will be attempted twice before the goal is achieved:

`+!`$g$`:` $c_1$ $\leftarrow$ $b_1$`.`
`+!`$g$`:` $c_2$ $\leftarrow$ $b_2$`.`
...
`+!`$g$`:` $c_n$ $\leftarrow$ $b_n$`.`

$$\frac{}{\textbf{EBDG}_g}$$

`+!`$g$`:` $g$ $\leftarrow$ `true.`
`+!`$g$`: not p1(`$g$`) &` $c_1$ $\leftarrow$ `+p1(`$g$`);` $b_1$`.`
`+!`$g$`: not p2(`$g$`) &` $c_2$ $\leftarrow$ `+p2(`$g$`);` $b_2$`.`
...
`+!`$g$`: not p`$n$`(`$g$`) &` $c_n$ $\leftarrow$ `+p`$n$`(`$g$`);` $b_n$`.`
`-!`$g$`: true` $\leftarrow$ `!`$g$`.`
`+`$g$`: true` $\leftarrow$ `-p1(`$g$`); -p2(`$g$`);` ... `.dropGoal(`$g$`, true).`

In this pattern, each plan, when selected for execution, initially adds a belief `p`$i$`(`$g$`)`; the goal $g$ is used as an argument to `p` so as to avoid interference among applications of the

pattern for different goals. The belief is used as part of the plan contexts (note the use of `not p`$i$ in the contexts of the plans in the pattern above) to state the plan should not be applicable in a second attempt (of that same plan within a single adoption of goal $g$ for that agent).

In the pattern above, despite the various alternative plans, the agent can still end up dropping the intention with the goal $g$ unachieved, if all those plans become non-applicable. Conversely, in a *blind commitment goal* the agent can drop the goal only when it is achieved. This type of commitment toward the achievement of a declarative goal can thus be understood as *fanatical commitment* [18]. The **BCG**$_{g,F}$ pattern below defines this type of commitment:

$$\frac{\mathcal{P}}{\boldsymbol{F}(\mathcal{P})}\ \textbf{BCG}_{g,F}$$

`+!`$g$`: true` $\leftarrow$ `!`$g$`.`

This pattern is based on another pattern rule (represented by the variable $\boldsymbol{F}$); $\boldsymbol{F}$ is often **BDG**, although the programmer can chose another pattern (e.g., **EBDG** if a plan should not be attempted twice). Finally, the last plan keeps the agent pursuing the goal even in case there is no applicable plan. It is assumed that the selection of plans is based on the order that the plans appear in the program and all events have equal chance of being chosen as the event to be handled in a reasoning cycle.

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 attempting it. 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 [8]. In [22] and [7], the "impossibility" condition is called a "drop condition". 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:

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

`+`$f$`: true` $\leftarrow$ `.dropGoal(`$g$`, false).`

This pattern extends the **BCG** pattern 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 (refer to the semantics of the internal action `.dropGoal` in section 3). This effectively means that the plan in the intention where $g$ appeared, which depended on $g$ to carry on execution, must itself fail (as $g$ is now impossible to achieve). However, there might be an alternative for that other plan which does not depend on $g$, so that plan's failure handling may take care of such situation.

As we have a failure drop condition for a goal, we can also have a successful 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 from there; then, if the robot realises that its owner does not want a beer anymore, it should drop the goal [8]. The belief "my owner wants a beer" is the *motivation* $m$ for the goal. The following pattern, called Relativised Commitment Goal (**RCG**) defines a goal that is relative to a motivation condition: the goal can be dropped with success if the agent looses the motivation for it.

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

$$-m\texttt{: true} \leftarrow \texttt{.dropGoal(}g\texttt{, true).}$$

Note that, in the particular combination of **RCG** and **BCG** above, if the attempt to achieve $g$ ever terminates, it will always terminate with success, since the goal will be dropped only if either the agent believes it has been achieved achieved (by **BCG**) or $m$ is removed from belief base.

Of course we can combine the last two patterns above to create a goal which can be dropped if it has been achieved, has become impossible to achieve, or the motivation to achieve it no longer exists (representing an open-minded commitment). The Open-Minded Commitment pattern (**OMC**) defines this type of goal:

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

$$+f\texttt{: true} \leftarrow \texttt{.dropGoal(}g\texttt{, false).}$$
$$-m\texttt{: true} \leftarrow \texttt{.dropGoal(}g\texttt{, true).}$$

For example, a drop condition could be "no beer at location ($X$,$Y$)" (denoted below by $\neg$ `b(X,Y)`), and the motivation condition could be "my owner wants a beer" (denoted below by `wb`). Consider the initial plan below with representing the single known course of action to achieve goal `l(X,Y)`:

$$\texttt{+!l(X,Y): bc(}B\texttt{) \& B} > 0.2 \leftarrow \texttt{go(X,Y).}$$

When the pattern $\mathbf{OMC}_{l(X,Y),\neg b(X,Y),wb}$ 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).
```

Another important type of goal in agent-based systems are *maintenance goals*: the agent needs to ensure that the state of the world will always be such that $g$ holds. Whenever the agent realises that $g$ is no longer in its belief base (i.e., believed to be true), it attempts to bring about $g$ again by having the respective declarative (achievement) goal. The pattern rule that defines a Maintenance Goal (**MG**) is as follows:

$\mathcal{P}$

——————————— $\mathbf{MG}_{g,F}$

$g$.
-$g$: true ← !$g$.
$\boldsymbol{F}(\mathcal{P})$

The first line of the pattern states that, initially (when the agent starts running) it will assume that $g$ is true. (As soon as the interpreter obtains perception of the environment for the first time, the agent might already realise that such assumption was wrong.) The first plan is triggered when $g$ is removed from the belief base, e.g. because $g$ has not been perceived in the environment in a given reasoning cycle, and thus the maintenance goal $g$ is no longer achieved. This plan then creates a declarative goal to achieve $g$. The type of commitment to achieving $g$ if it happens not to be true is defined by $\boldsymbol{F}$, which would normally be **BCG** given that the goal should not be dropped in any circumstances unless it is has been achieved again. (Realistically, plans for the agent to attempt proactively to prevent this from even happening would also be required, but the pattern is useful to make sure the agent will act appropriately in case things go wrong.)

Another useful pattern is a Sequenced Goal Adoption (**SGA**). This pattern should be used when various instances of a goal should not be adopted concurrently (e.g., a robot that needs to clean two different places). To solve this problem, the **SGA** pattern adopts the first occurrence of the goal and records the remaining occurrences as pending goals by adding them as special beliefs. As one such goal occurrence is achieved, if any other occurrence is pending, it gets activated.

——————————————————— $\mathbf{SGA}_{t,c,g}$

$t$: not fl(_) & $c$ ← !fg($g$).
$t$: fl(_) & $c$ ← +fl($g$).
+!fg($g$): true ← +fl($g$); !$g$; -fl($g$).
-!fg($g$): true ← -fl($g$).
-fl(_): fl($g$) ← !fg($g$).

In this pattern, $t$ is the trigger leading to the adoption of a goal $g$; $c$ is the context for the goal adoption; `fl(`$g$`)` is the flag to control whether the goal $g$ is already active; and `fg(`$g$`)` is a procedural goal that guarantees that `fl` will be added to the belief base to record the fact that some occurrence of the goal has already been adopted, then adopts the goal !$g$, as well as it guarantees that `fl` will be eventually removed whether !$g$ succeeds or not. The first plan is selected when $g$ is not being pursued; it simply calls the `fg` goal. The second plan is used if some other instance of that goal has already been adopted. All it does is to remember that this goal $g$ was not immediately adopted by adding `fl(`$g$`)` to the belief base. The last plan makes sure that whenever a goal adoption instance is finished (denoted by the removal of a `fl` belief), if there are any pending goal instances to be adopted, they will be activated through the `fg` call.

## 5 Using Patterns in *Jason*

***Jason*** is an interpreter for an extended version of AgentSpeak(L) and is available *Open Source* under GNU LGPL at `http://jason.sourceforge.net` [4]. It imple-

ments the operational semantics of AgentSpeak(L) as given in [6]. It also implements the plan failure mechanism and the pre-defined internal action[7] used in the patterns described in Section 4. Since these features are enough for programming declarative goals, ***Jason*** already supports them. However, it would be clearly not acceptable if the programmer had to apply the patterns by hand.

To simplify the programming of sophisticated goals by the use of patterns, we extend the language interpreted by ***Jason*** to include pre-processing directives. The syntax for pattern directives is:

```
directive ::=
  "{" "begin" <pattern-name>"("<parameters>")" "}"
      <agent-speak-program>
  "{" "end" "}"
```

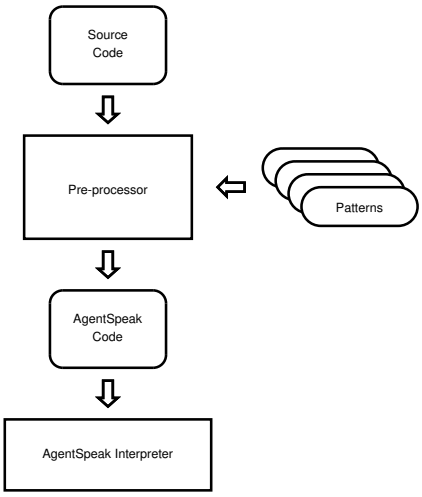



**Fig. 3.** ***Jason*** Pre-Processing and Patterns.

We have implemented a pre-processor for ***Jason*** which also handles patterns as illustrated in Figure 3. Each pattern is implemented in a Java class that receives an AgentSpeak program and returns another program, transformed as defined by the respective pattern. This implementation allows us, and even users, to make new patterns available in a straightforward manner. One simply has to create a new Java class for the new pattern and register this class with the pre-processor[8].

In the remainder of this section, we will illustrate how the ***Jason*** pre-processing directives for the use of patterns can be used to program a cleaning robot for the scenario described in [7] (where the robot was implemented using Jadex [15, 16]). The first goal of the robot is to maintain its battery charged: this is clearly a maintenance goal (**MG**). The agent should pursue this goal when its battery level goes below 20% and should remain pursuing it until the battery is completely charged. In the program below, based on the perception of the battery level, the belief `battery_charged`, which indicates that the goal is satisfied, is either removed or added to the belief base, signalling whether the corresponding achievement goal must be activated or not.

[7] The internal action used here is not yet available in the latest public release of ***Jason***, but will be available in the next release.

[8] Note that this too will only be available in the next release of ***Jason***

```
+battery_level(B): B < 0.2 ← -battery_charged.
+battery_level(B): B = 1.0 ← +battery_charged.

{ begin mg("battery_charged", bcg("battery_charged")) }
   +!battery_charged : not l(power_supply)
          go(power_supply).
   +!battery_charged: l(power_supply) ← plug_in.
{ end }
```

The first plan of the pattern for the `battery_charged` goal moves the agent to the place where there is a power supply, if it is not already there (according to its `l(power_supply)` belief). Otherwise, the second plan will plug the robot to the power supply. The `plug_in` action will charge the battery and thus change the robot's state that is perceived back through `battery_level(B)` percepts (which generate `+battery_level(B)` events).

The second goal the robot might adopt is to patrol the museum at night. This goal is therefore activated when the agent perceives sunset (represented by the event `+night`). Whenever activated, the goal can be dropped only if the agent perceives dawn (represented by the event `-night`). The following program defines `patrol` as this kind of goal using a **RCG** pattern with `night` as the motivation:

```
+night: true ← !patrol.

{ begin rcg("patrol", "night") }
   +!patrol: battery_charged ← wander.
{ end }
```

The agent will never have the belief `patrol` in its belief base, since no plan or perception of the environment will add this particular belief. The goal is, in some sense, deliberately unachievable, while RCG maintains the agent committed to the goal nevertheless. However, it is considered as achieved (finished with success) when the motivation condition is removed from the belief base. Note that the context for the `!patrol` plan is that the battery is charged, therefore while the maintenance goal `battery_charged` is active, the robot does not wander, but it resumes wandering as soon the battery becomes charged again. We are thus using this belief to create an *interference* between goals (i.e., charging the battery precludes patrolling).

The last goal the robot might adopt is to clean the museum during the day whenever it perceives waste around. Since the robot can perceive various different pieces of waste around, it would accordingly generate several concurrent instances of this goal. However these goals are mutually exclusive: they cannot be achieved simultaneously; trying to go in two different directions must be avoided, and expressing this at the declarative level avoids too much work on implementing application-specific intention selection functions (in the context of AgentSpeak). It is indeed another kind of interference between different goals. The **SGA** pattern is used in the program below to ensure that only one `clean` goal instance is being pursued at a moment in time. The event that triggers this goal is `+waste(X,Y)` (some waste being perceived at location X,Y), and the context is `not night`:

```
{ begin sga("+waste(X,Y)", "not night", "clean(X,Y)")}
{ end }

{ begin omc("clean(X,Y)", "night", "waste(X,Y)")}
   +!clean(X,Y): l(X,Y) ← pick; go(bin); drop.
   +!clean(X,Y): not l(X,Y) ← go(X,Y).
{ end }
+battery_charged: true ← .suspend(clean(X,Y)).
-battery_charged: true ← .resume(clean(X,Y)).
```

In the program above, an open-minded commitment pattern (**OMC**) is used to create the `clean(X,Y)` goal with `night` as the failure condition (at sunset, the goal should be abandoned with failure) and `waste(X,Y)` as the motivation (if the agent came to believe that there is no longer waste at that location, the goal could be dropped with success). The last two plans are used to suspend and resume the goal when the `battery_charge` goal is active. Of course we could add `battery_charge` in the context of the plans (as we did in the `patrol` goal); however, using the `.suspend` internal action is more efficient because the goal becomes actually suspended (until resumed with the respective `.resume` internal action) rather than being continuously attempted without any applicable plans.

## 6 Conclusions

In this paper we have shown that sophisticated types of goals discussed in the agents literature can be implemented in the AgentSpeak language with only the extensions (and extensibility mechanisms) available in ***Jason***. In fact, this is done by combining AgentSpeak plans, forming certain patterns, for each type of goal and commitment towards goals that agents may have. Therefore, 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 "plan patterns" to support the use of declarative goals with complex temporal structures in AgentSpeak programming.

Besides the use of internal actions such as `.dropGoal` (that are available in ***Jason*** for general use, independently of this proposal for declarative goals), our proposal does not require either: ($i$) syntactical or semantical changes in the language (as done, for example, in [22,7]); nor ($ii$) the definition of a goal base (cf. [19]) which is also usual in other approaches. Van Riemsdijk *et at*. [20] also pointed out that declarative goals can be built based on the procedural goals available in 3APL, by simply checking if the corresponding belief is true at the end of the plan execution. What they proposed in that paper corresponds to our **BDG** pattern. In this work, we further define various other types of declarative goals, represented them as *patterns* of AgentSpeak programs, and presented an implementation in ***Jason*** (using a pre-processor) that facilitates this approach for declarative goals. Another advantage of our approach is that, as complex types of goals are mapped to plain AgentSpeak using patterns, programmers can change the patterns to fit their own requirements, or indeed create new patterns easily.

In future work we intend to formalise our approach based on the existing operational semantics and to verify some properties of the programs generated by the patterns, including a comparison with approaches that use a goal base to have declarative goals. An example of an issues that might be of particular interest in such comparison is how the use of plan patters will affect other aspects of agent-based development such as debugging. In the future, we also plan to support conjunctive goals such as $p \wedge q$ (where both $p$ and $q$ should be satisfied at the same time, as done in [19]), possibly through the use of plan patterns as well. Furthermore, we plan to investigate other patterns that may useful in the practical development of large-scale multi-agent systems.

## Acknowledgements

Many thanks to A.C. Rocha Costa for discussions on maintenance goals in AgentSpeak. Anonymous reviewers for this paper have made detailed comments which helped improve the paper. Rafael Bordini gratefully acknowledges the support of The Nuffield Foundation (grant number NAL/01065/G).

## References

1. D. Ancona and V. Mascardi. Coo-BDI: Extending the BDI model with cooperativity. In J. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *Declarative Agent Languages and Technologies, Proc. of the First Int. Workshop (DALT-03), held with AAMAS-03, 15 July, 2003, Melbourne, Australia*, number 2990 in LNAI, pages 109–134, Berlin, 2004. Springer-Verlag.
2. D. Ancona, V. Mascardi, J. F. Hübner, and R. H. Bordini. Coo-AgentSpeak: Cooperation in AgentSpeak through plan exchange. In N. R. Jennings, C. Sierra, L. Sonenberg, and M. Tambe, editors, *Proc. of the Third Int. Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2004), New York, NY, 19–23 July*, pages 698–705, New York, NY, 2004. ACM Press.
3. 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.
4. R. H. Bordini, J. F. Hübner, et al. ***Jason**: A Java-based AgentSpeak interpreter used with saci for multi-agent distribution over the net*, manual, release version 0.7 edition, Aug. 2005. `http://jason.sourceforge.net/`.
5. R. H. Bordini, J. F. Hübner, and R. Vieira. ***Jason*** and the Golden Fleece of agent-oriented programming. In Bordini et al. [3], chapter 1.
6. R. H. Bordini and Á. F. Moreira. Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L). *Annals of Mathematics and Artificial Intelligence*, 42(1–3):197–226, Sept. 2004. Special Issue on Computational Logic in Multi-Agent Systems.
7. 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.
8. P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(3):213–261, 1990.

9. M. Dastani, B. van Riemsdijk, F. Dignum, and J. Meyer. A programming language for cognitive agents: Goal directed 3APL. In *Proc. of the First Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS03)*, volume 3067 of *LNAI*, pages 111–130, Berlin, 2004. Springer.
10. M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Programming multi-agent systems in 3APL. In Bordini et al. [3], chapter 2.
11. M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In M. P. Singh, A. S. Rao, and M. Wooldridge, editors, *Intelligent Agents IV—Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL-97), Providence, RI, 24–26 July, 1997*, number 1365 in LNAI, pages 155–176. Springer-Verlag, Berlin, 1998.
12. M. d'Inverno and M. Luck. Engineering AgentSpeak(L): A formal computational model. *Journal of Logic and Computation*, 8(3):1–27, 1998.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
14. N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents™ — summary of an agent infrastructure. In *Proceedings of Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, held with the Fifth International Conference on Autonomous Agents (Agents 2001), 28 May – 1 June, Montreal, Canada*, 2001.
15. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In Bordini et al. [3], chapter 6.
16. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In Bordini et al. [3], chapter 6, pages 149–174.
17. 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 the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), 22–25 January, Eindhoven, The Netherlands*, number 1038 in LNAI, pages 42–55, London, 1996. Springer-Verlag.
18. A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.
19. B. van Riemsdijk, M. Dastani, and J.-J. C. Meyer. Semantics of declarative goals in agent programming. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, and M. Wooldridge, editors, *Proceedings of the 4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 133–140. ACM, 2005.
20. 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, *Proceedings of the 12th Portuguese Conference on Artificial Intelligence, EPIA 2005, Covilhã, Portugal, December 5-8, 2005*, volume 3808 of *LNCS*, pages 548–559, 2005.
21. M. Winikoff. JACK™ intelligent agents: An industrial strength platform. In Bordini et al. [3], chapter 7, pages 175–193.
22. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and procedural goals in intelligent agent systems. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning*, 2002.