ECAI 2010
H. Coelho et al. (Eds.)
IOS Press, 2010
© 2010 The authors and IOS Press. All rights reserved.
doi:10.3233/978-1-60750-606-5-635

Semantics for the *Jason* Variant of AgentSpeak (Plan Failure and some Internal Actions)

Rafael H. Bordini[1] and **Jomi Fred Hübner**[2]

Abstract. ***Jason*** is a platform for agent-based software development that is characterised both by being based on a programming language with formal semantics as well as having many language and platforms features that are very useful for practical programming, but not fully formalised. In this paper, we make significant progress in the direction of formalising the aspects of the variant of AgentSpeak that is interpreted by ***Jason*** that were not included in previous work on giving formal semantics to AgentSpeak. In particular, we give semantics to the plan failure handling mechanism which is unique to ***Jason***, and also for some of the predefined internal actions that can alter an agent's mental state. Such internal actions are essential for some aspects of BDI-based programming, such as checking or dropping current goals or intentions, and therefore need to be formally defined within the operational semantics of the language.

1 Introduction

The ***Jason*** platform for the development of multi-agent systems has become one of the most popular platforms based on agent-oriented programming in the Agents community. It has over 500 downloads a month on average (over the last couple of years), with some months being particularly busy — during March 2009 there were over a thousand downloads. ***Jason*** is also used in 15 different countries (that we are aware of) for teaching of under or post-graduate courses. It has been used in various academic projects too (see `http://jason.sf.net`).

In our opinion, the reasons for ***Jason***'s popularity are first because it is based on a simple, elegant, and intuitive programming language, and second because it has, on top of such a language, additional features that make programming easier and more efficient. There has been many additions in ***Jason*** to the original AgentSpeak language proposed by A.Rao [16]. Not just the language, but many features have been added to the platform over the years, often by request of the user base, which have led to the formal semantics lagging behind some of those extra features we have included in ***Jason*** to make programming more practical and pleasant. Although most of them were clearly described in [6], they have never been formally defined, a problem we address in this paper.

Besides hindering the precise understanding of the language, the lack of semantics is a problem more generally, for example for the work on formal verification of multi-agent systems reported in [4, 5] that could potentially be used for ***Jason*** too but, of course, only for the features formally defined; besides, the formal semantics can be used for proving properties of the agent language. It has been argued (e.g., at the ProMAS Technical Forum) that agent programming languages can be divided into those that have a very strong formal basis such as 2APL [8] and GOAL [11], and those that have had more industrial use such as Jadex [15] and JACK [20]. ***Jason*** has been somewhat a compromise between these two territories, and there is ongoing work aiming to approximate ***Jason*** to *both* territories. This paper in particular contributes to reducing the gap in ***Jason***'s formal basis.

We here concentrate on the semantics of some of the ***Jason*** features currently not formalised. ***Jason*** uses a peculiar plan failure handling mechanism, quite different from other agent languages as has been discussed in the literature, for example [18, 21]. The fact that it is different from the other approaches increases the need for it to be unambiguously defined. The approach in ***Jason*** is to use the `-!` and `-?` triggering events that were part of the original AgentSpeak syntax but never used in any formal or informal accounts of the semantics to allow users to write *contingency plans* [6] (even though it is possible that such events were meant to be used with a completely different meaning when AgentSpeak was conceived). Being different from other approaches and possibly using a syntax originally meant for a different purpose are good motivations to formalise it here, but there is another very good reason. The plan failure handling mechanism of ***Jason***, together with special internal actions also formalised here, was absolutely essential in allowing the work on using plan patterns for the programming of sophisticated notions of goals not originally available in AgentSpeak [13].

Another feature that often is unclear to ***Jason*** users is the different interpretation for *test goals* (i.e., events of type `+?`) in ***Jason*** as opposed to its original definition. In ***Jason***, if a normal test goal fails, we try generating an event that can trigger user-defined plans allowing complex queries or agents acting further so as to obtain the missing information. The difference to achievement goals then becomes rather subtle, so again it is important to put this into a more formal basis. However, this was already done in the formal semantics of AgentSpeak used in [19], from which we start in this paper.

One of the most practical features of ***Jason*** programming is that it allows the use of *internal actions* anywhere where a predicate or an action appears in the context or body of plans (respectively). They can be used by programmers to have access to legacy code, but have had a variety of uses in ***Jason***'s interaction with other aspects of agent-based development (such as Environment infrastructures [17]) and ***Jason*** applications (e.g., [12]). An important point about internal actions is that they have been used to provide a number of functionalities in ***Jason***, often addressing limitations of the original language or providing efficient implementations of functionalities often required by programmers, in the form of *predefined internal actions*. These are internal actions available in a library that is distributed with ***Ja-***

[1] Federal University of Rio Grande do Sul, Brazil, email: R.Bordini@inf.ufrgs.br
[2] Federal University of Santa Catarina, Brazil, email: jomi@das.ufsc.br

son. That library of predefined internal actions includes some internal actions that – together with the other feature formalised in this paper, that of plan failure – are very important for advanced goal-based programming in ***Jason*** [13, 6] and very special in the sense that, unlike most other internal actions, they alter the mental state of an agent as reflected in the configuration of the transition system giving operational semantics to AgentSpeak. Given that the beginning the implementation of ***Jason*** started closely from that operational semantics, so it is absolutely essential to give formal semantics to any internal action that can access or alter such structure (i.e., the transition system configuration). Examples of actions that do so are those that check or drop current desires or intentions of the agent. These are the other main extensions of the semantics introduced in this paper.

The remainder of this paper is structured as follows. Section 2 gives a brief description of the AgentSpeak language, and Section 3 provides the preliminaries for the operational semantics. Section 4 then extends the existing semantics of AgentSpeak to account for: (i) the plan failure handling mechanism; (ii) ***Jason***'s action execution model whereby actions can fail (which is one of the reasons for plans failing thus requiring a failure handling mechanism) and intentions suspended while agent effectors execute the actions; and (iii) internal actions for checking and dropping desires and intentions as well as forcing success or failure of particular goals. Section 5 concludes the paper and discusses future work.

2 The AgentSpeak Language

The AgentSpeak programming language was introduced by Rao [16]. It can be understood as a natural extension of logic programming for the BDI agent architecture, and provides an elegant abstract framework for programming BDI agents. The BDI architecture is, in turn, perhaps one of the best known approaches to the implementation of rational practical reasoning agents.

The syntax of an AgentSpeak agent program ag is defined by the grammar in Figure 1. In AgentSpeak, an agent program is simply given by a set bs of beliefs and a set ps of plans. The beliefs bs define the initial state of the agent's belief base (i.e., the state of the belief base when the agent starts running), and the plans ps form the agent's plan library. The atomic formulæ at of the language are predicates, where $\mathtt{P}$ is a predicate symbol and $t_1, \ldots, t_n$ are standard terms of first-order logic. A *belief* is an atomic formula at with no variables; we use b as a meta-variable for beliefs.

$$
\begin{array}{lcllllllll}
ag & ::= & bs \;\; ps & & & & & & & \\
bs & ::= & b_1 \ldots b_n & & & & & & & (n \geq 0) \\
ps & ::= & p_1 \ldots p_n & & & & & & & (n \geq 1) \\
p & ::= & te : ct \leftarrow h & & & & & & & \\
te & ::= & +at & \mid & -at & \mid & +g & \mid & -g & \\
ct & ::= & ct_1 & \mid & \mathsf{T} & & & & & \\
ct_1 & ::= & at & \mid & \neg at & \mid & ct_1 \wedge ct_1 & \mid & & \\
h & ::= & h_1 ; \mathsf{T} & \mid & \mathsf{T} & & & & & \\
h_1 & ::= & a & \mid & g & \mid & u & \mid & h_1 ; h_1 & \\
at & ::= & \mathtt{P}(t_1, \ldots, t_n) & & & & & & & (n \geq 0) \\
a & ::= & \mathtt{A}(t_1, \ldots, t_n) & & & & & & & (n \geq 0) \\
g & ::= & !at & \mid & ?at & & & & & \\
u & ::= & +b & \mid & -at & & & & &
\end{array}
$$

Figure 1. Syntax of AgentSpeak

A plan in AgentSpeak is given by p above, where te is the *triggering event*, ct is the plan's *context*, and h is sequence of actions, goals, or belief updates (which should be thought of as "mental notes" created by the agent itself). We refer to $te : ct$ as the *head* of the plan, and h is its *body*. The set of plans of an agent is given by ps. Each plan has as part of its head a formula ct that specifies the conditions under which the plan can be chosen for execution.

A triggering event te can be the addition or the deletion of a belief from an agent's belief base (denoted $+at$ and $-at$, respectively), or the addition or the deletion of a goal ($+g$ and $-g$, respectively); the occurrence of such events might trigger the execution of plans with matching te part. For plan bodies, we assume the agent has at its disposal a set of *actions* and we use a as a meta-variable ranging over them. Actions are written using the same notation as predicates, except that an action symbol $\mathtt{A}$ is used instead of a predicate symbol. Goals g can be either *achievement goals* ($!at$) or *test goals* ($?at$). Finally, $+b$ and $-at$ (in the body of a plan) represent operations for updating (u) the belief base by, respectively, adding or removing *internal* beliefs; recall that an atomic formula must be ground if it is to be added to the belief base (this is not quite true in ***Jason*** but we will not deal with this extension in this paper).

3 Operational Semantics of AgentSpeak

In this section, we present part of the preliminaries of the operational semantics of AgentSpeak as given in [7, 19], as we use a similar basis in the new semantic rules given in the next section. The reader unfamiliar with AgentSpeak might benefit from looking at the existing semantic rules, which are assumed to be exactly as in [19]. Unfortunately, it would not be feasible to reproduce them all in a paper of this length, but it should be noted that the rules presented in this paper either replace or should be added to the existing ones so that together they give semantics to the ***Jason*** variant of AgentSpeak.

The operational semantics of AgentSpeak is given by a set of rules that define a transition relation between configurations $\langle ag, C, T, s\rangle$ where:

- An agent program ag is, as defined above, a set of beliefs and a set of plans.
- An agent's circumstance C is a tuple $\langle I, E, A\rangle$ where:
  - I is a set of *intentions* $\{i, i', \ldots\}$. Each intention i is a stack of partially instantiated plans.
  - E is a set of *events* $\{\langle te, i\rangle, \langle te', i'\rangle, \ldots\}$. Each event is a pair $\langle te, i\rangle$, where te is a triggering event and i is an intention (a stack of plans in case of an internal event or $\top$ representing an external event, i.e., one originating from belief update following environment sensing or from agent communication).
  - A is a set of *actions* to be performed in the environment. An action expression included in this set tells other architecture components to actually perform the respective action on the environment, thus changing it. In the extension presented here, intentions can be suspended while an action is being executed; so elements of A are in fact pairs $\langle a, i\rangle$ where a is an action and i is an intention. This is similar to intentions being suspended in original AgentSpeak when associated with events in E, because a plan needs to be found for achieving a subgoal in the topmost plan body.
- It facilitates giving the semantics if we use a structure that keeps track of temporary information required in subsequent stages

within a single reasoning cycle. T is the tuple $\langle R, Ap, \iota, \varepsilon, \rho\rangle$ with such temporary information; the components are:

- R for the set of *relevant plans* (for the event being handled).
- Ap for the set of *applicable plans* (the relevant plans whose context are true).
- ι, ε, and ρ keep record of a particular intention, event, and applicable plan (respectively) being considered along the execution of an agent's reasoning cycle.

- The current step s within an agent's reasoning cycle is symbolically annotated by $s \in \{\mathsf{SelEv}, \mathsf{RelPl}, \mathsf{ApplPl}, \mathsf{SelAppl}, \mathsf{AddIM}, \mathsf{SelInt}, \mathsf{ExecInt}, \mathsf{ProcAct}, \mathsf{ClrInt}\}$, which stands for: selecting an event from the set of events, retrieving all relevant plans, checking which of those are applicable, selecting one particular applicable plan (the intended means), adding the new intended means to the set of intentions, selecting an intention, executing the selected intention; the last one means clearing an intention or intended means that may have finished in the previous step, while ProcActis a new step in the reasoning cycle introduced later in this paper.

In order to keep the semantic rules succinct, we adopt the following notation:

- If C is an AgentSpeak agent circumstance, we write C_E to make reference to component E of C. Similarly for all the other components of a configuration. In the *where* part of rules, any component not explicitly stated to change is assumed to remain unchanged in that transition.
- We write $i[p]$ to denote the intention that has plan p on top of intention i.

We define some auxiliary syntactic functions to be used in the semantics. If p is a plan of the form $te : ct \leftarrow h$, we define $\mathsf{Ctxt}(p) = ct$. That is, this projection function returns the context of a given plan. We also define an auxiliary function that determines the set of applicable plans for a certain event. Given a set of relevant plans R and the beliefs bs of an agent, the set of applicable plans $\mathsf{AppPlans}(bs, R)$ is defined as follows:

$$\mathsf{AppPlans}(bs, R) = \{(p, \theta' \circ \theta) \mid (p, \theta) \in R \ \ \textit{and} \ \ \theta' \textit{ is such that } bs \models \mathsf{Ctxt}(p)\theta\theta'\}.$$

4 Formal Semantics for the *Jason* Variant of AgentSpeak

In this section, we show the semantic rules that were changed or added to the semantics that appeared in [19], in order to account for plan failure and other important features available in the variant of AgentSpeak interpreter by ***Jason***.

4.1 Changed Rules for Plan Failure

Rule **Appl$_2$** is the rule for the case where there are no applicable plans. It previously simply discarded the whole intention. Now, it is as formalised and explained below.

$$\frac{\mathsf{AppPlans}(ag_{bs}, T_R) = \{\} \qquad T_\varepsilon = \langle te, i\rangle}{\langle ag, C, T, \mathsf{ApplPl}\rangle \longrightarrow \langle ag, C', T, \mathsf{SelInt}\rangle} \quad (\textbf{Appl}_2)$$

where:

$$C'_E = \begin{cases} C_E \cup \{\langle -\%at, i\rangle\} & \text{if } te = +\%at \\ & \text{with } \% \in \{!, ?\} \\ C_E \cup \{T_\varepsilon\} & \text{otherwise} \end{cases}$$

The first part in the "where" clause generates a goal deletion if the corresponding goal addition fails. Goal deletion events happen whenever a plan fails, not only because there are no applicable plans for a goal addition, but also when an action fails. This paper introduces for the first time a semantics to the notion of goal deletion. In the original definition, Rao syntactically defined the possibility of goal deletions as triggering events for plans, but did not discuss what they meant. Neither has this been discussed in further attempts to formalise AgentSpeak [10], and the notion does not seem to exist in dMars [9] (an implemented PRS-style BDI system that inspired the creation of AgentSpeak). The choice in ***Jason*** was to use this as some kind of plan failure handling mechanism, as formalised here.

Intuitively, the idea is that a goal deletion is a "clean-up" plan that is executed prior to "backtracking" a previous goal (i.e., attempting another plan to achieve the goal for which a plan failed). One of the things the programmer might want to do within such 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 environment so as to change it. Therefore, in certain circumstances, one would expect the agent to have to act further (in ways specific to its environment) so as to reverse the effects of its previous actions before attempting some alternative course of actions to achieve that goal, and this is precisely the practical utility 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. It is equally easy to specify that backtracking should always be attempted. It should be noted that we are using the term "backtracking" for this, but it is not exactly the same as backtracking in traditional logic programming.

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. So, for goal deletions (both achieve and test), the lack of relevant plans is used to denote that the programmer did not wish for the corresponding goal addition to be backtracked, i.e., to be attempted again, thus the whole intention needs to be dropped, as its topmost goal can no longer be achieved. For normal events, an approach for what to do in case there are no relevant plans for an event is described in [2]. It assumes that in some cases, explicitly specified by the programmer, the agent will want to ask other agents for recipes of what to do to 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. See [1] for an overview of how various BDI systems deal with the problem of there being no applicable plans.

4.2 Action Execution with Plan Failure

AgentSpeak is a language used to define the high-level (practical) reasoning component that goes as part of an overall agent architecture. The relation of the other parts of the overall agent architecture (such as sensors and actuators) to the AgentSpeak interpreter is not essential in giving semantics to the *abstract* language. It suffices to note that belief update from perception of the environment inserts (external) events in the set C_E (which is then used in the AgentSpeak interpretation cycle), whilst the effectors simply execute every action that is included by the reasoner in the set C_A. However, because actions can fail, and intentions need to be suspended until the result of action execution attempts (through effectors) is known, we here need to give some (partial, still somewhat abstract) account for this.

One of the main reasons for a plan to fail is when an attempt to execute a basic action in the environment fails (another situation discussed above was related to the non-existence of relevant or applicable plans). We give below an extra rule to be specific about the result of the attempt to execute an action in the environment (or executing an internal action, for that matter).

$$\frac{\langle a, i\rangle \in C_A \qquad \mathsf{execute}(a) = e}{\langle ag, C, T, \mathsf{ProcAct}\rangle \longrightarrow \langle ag, C', T, \mathsf{ProcAct}\rangle} \qquad \textbf{(ExecAct)}$$

where:

$C'_A = C_A \setminus \{\langle a, i\rangle\}$

$C'_I = C_I \cup \{i'[te : ct \leftarrow h]\}$, if e

$C'_E = C_E \cup \{\langle -\%at, i\rangle\}$, if $\neg e \wedge (te = +\%at)$

with $i = i'[te : ct \leftarrow a; h]$ and $\% \in \{!, ?\}$.

$$\frac{C_A = \{\} \vee (\neg\exists \langle a, i\rangle \in C_A \,.\, \mathsf{execute}(a) = e)}{\langle ag, C, T, \mathsf{ProcAct}\rangle \longrightarrow \langle ag, C, T, \mathsf{ClrInt}\rangle} \qquad \textbf{(ExecDone)}$$

We use execute() to refer to the architectural component of an agent that takes care of the actual execution of actions. It is a Boolean function, denoting the fact that actions may fail or succeed. Recall that internal actions are run internally by the agent (it makes reference to any computation the agent is capable of performing internally, normally written in other programming languages, similarly to native methods in Java), whereas the basic actions are those that activate the agent effectors, which possibly change the environment in which the agent is situated. For example, if a robot's effectors attempt to move the robot but its way is blocked, that action *fails*. All internal actions also have a Boolean value. In particular, $\mathsf{execute}(a) = e$ in the premise of the rule above means that the execution of a has finished (e.g., the environment gave feedback on the execution of an action for a suspended intention) and e is either *true* or *false* depending on whether the action succeeded or failed.

In the rules above, note that various actions might be ready to be handled (e.g., feedback from various previous requests were all received from the environment in the last reasoning cycle). All are dealt with by repeated applications of Rule **ExecAct**; when no further action execution feedback has become available by the time the reasoning cycle reaches the step where these rules are evaluated, Rule **ExecDone** applies instead. To handle action execution in this way, we add a new step in the semantics called ProcAct (as can be seen in the rules above); accordingly, all rules of the original semantics that have ClrInt as the next step of the reasoning cycle are changed so that they first move to ProcAct which then proceeds to step ClrInt.

Note that the whole intention is dropped if the triggering event of the plan being executed was neither type of 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 action, and consequently a failed plan, means that the whole intention must be dropped.

4.3 Semantics of Special Internal Actions

Before we present the new rules for the special internal actions, we need definitions for what it means for an AgentSpeak agent to desire or to intend a particular state of affairs. Fortunately, part of this work was already done in [7], and used in the work on model checking for AgentSpeak too [5]. The definitions below are adapted from that work.

We first define an auxiliary function $agls : \mathcal{I} \rightarrow \mathcal{P}(\Phi)$, where $\mathcal{I}$ is the domain of all individual intentions and Φ is the domain of all atomic formulæ. Recall that an intention is a stack of partially instantiated plans, so the definition of $\mathcal{I}$ is as follows. The empty intention (or true intention) is denoted by T, and $\mathsf{T} \in \mathcal{I}$. If p is a plan and $i \in \mathcal{I}$, then also $i[p] \in \mathcal{I}$. The $agls$ function below takes an intention and returns all achievement goals in the triggering event part of the plans in it:

$$\begin{array}{lcl} agls(\mathsf{T}) & = & \{\} \\ agls(i[p]) & = & \begin{cases} \{at\} \cup agls(i) & \text{if } p = +!at \;:\; ct \leftarrow h \\ agls(i) & \text{otherwise} \end{cases} \end{array}$$

Because of changes we have made in this paper regarding plan failure and action execution, we had to change the definition as given in previous work so that suspended intentions are checked also in the C_A component, as intentions might be suspended there waiting for environment feedback.

Definition 1 [Intentions] *We say an AgentSpeak agent ag intends φ in circumstance C if, and only if, it has φ as an achievement goal that currently appears in its set of intentions C_I, or φ is an achievement goal that appears in the (suspended) intentions associated with events in C_E or actions in C_A. For an agent ag and circumstance C, we have:*

$$\begin{array}{rcl} \mathrm{Int}_{\langle ag,C\rangle}(\varphi) & \equiv & \varphi \in \bigcup_{i \in C_I} agls(i) \;\vee\; \varphi \in \bigcup_{\langle te,i\rangle \in C_E} agls(i) \\ & & \vee\; \varphi \in \bigcup_{\langle a,i\rangle \in C_A} agls(i) \end{array}$$

Note that we only consider triggering events that have the form of additions of achievement goals. The atomic formulæ at within those triggering events are the formulæ that represent (symbolically) properties of the states of the world that the agent is trying to achieve (i.e., the intended states). Importantly, we need to consider also the *suspended* intentions. Suspended intentions are precisely those that appear in the set of events or, with the latest extensions formalised in this paper, also in the set of actions to be executed.

Definition 2 [Desires] *We say an AgentSpeak agent ag desires φ in circumstance C if, and only if, φ is an achievement goal in C's set of events C_E (associated with any intention i), or φ is a current intention of the agent; more formally:*

$$\mathrm{Des}_{\langle ag,C\rangle}(\varphi) \;\equiv\; \langle +!\varphi, i\rangle \in C_E \;\vee\; \mathrm{Int}_{\langle ag,C\rangle}(\varphi)$$

It was argued in [7] that the *desire* modality in an AgentSpeak agent is best represented by additions of achievement goals presently as events in the set of events, as well as its present intentions. The goal additions in events represent desires strictly, i.e., before they become intentions (which will happen when the agent commits to a particular course of action to achieve that goal). Given that semantics of desire, this definition does not need changing with the extensions defined in this paper.

One of the important additions to BDI programming in ***Jason*** through some special predefined internal actions is the ability to check whether the agent currently desires (`.desire(`G`)`) or intends (`.intend(`G`)`) to achieve a particular goal G. Note that internal actions can also be used in place of predicates in plan contexts, so these two special internal actions give the ability for programmers to decide on appropriate courses of actions not only based on current beliefs of the agents (as already possible in original AgentSpeak), but also on its current desires and intentions. Normal predicates in plan contexts have a truth value which depends on whether they can be unified against the belief base or not; similarly, internal actions are always Boolean functions too.

The semantics of these internal actions in plan contexts is similar to being able to unify a predicate against the belief base or not, except that desires and intentions are checked instead. More specifically, to give semantics to `.desire` it suffices to say that $\mathsf{execute}(\texttt{.desire(}G\texttt{)}) = \mathit{true}$ if, and only if, $\mathrm{Des}_{\langle ag,C\rangle}(G\theta)$ for some m.g.u. θ that is then used to further instantiate variable in the substitutions that will make the plan applicable, and similarly for `.intend(`G`)` and $\mathrm{Int}_{\langle ag,C\rangle}(G\theta)$. Note therefore that θ is a m.g.u. unifying goal G in the internal action with one of the goals currently desired/intended by the agent; this is also the case in the rules below, unless otherwise stated. Finally, it should be noted that when these actions appear in a plan *body* rather than context (although less common), they are treated by specific semantic rules, as for the other special internal actions formalised below.

We now give semantics to other special internal actions by providing one rule for each them, and all these rules should be interpreted as exceptions to Rule **ExecAct** above. That is, these are tried first, and **ExecAct** is only used if none of these apply. The rule mentioned above for checking whether the agent has a particular desire in a plan body is shown below; the rule for intention is very similar, as in the case of they appearing in the plan context, as discussed above.

$$\frac{\langle a,i\rangle \in C_A \qquad a = \texttt{.desire}(G)}{\langle ag,C,T,\mathit{ProcAct}\rangle \longrightarrow \langle ag,C',T,\mathit{ProcAct}\rangle} \quad \textbf{(ExDes)}$$

where:

$$\begin{array}{rcl} C'_A & = & C_A \setminus \{\langle a,i\rangle\} \\ C'_I & = & C_I \cup \{(i'[te:ct \leftarrow h])\theta\}, \\ & & \text{if } \mathrm{Des}_{\langle ag,C\rangle}(G\theta) \\ C'_E & = & C_E \cup \{\langle -\%at, i'\rangle\}, \\ & & \text{if } \neg\mathrm{Des}_{\langle ag,C\rangle}(G\theta) \wedge (te = +\%at) \end{array}$$

with θ a m.g.u., $i = i'[te:ct \leftarrow a;h]$, and $\% \in \{!,?\}$.

We now turn to the rule for the action that drops desires matching a particular goal given as parameter. In ***Jason***, the semantics of `.drop_desire(`G`)` is that intentions that have an intended means to achieve a goal that unifies with G, or desires to achieve G, are simply removed, with no plan failure events being generated.

$$\frac{\langle a,i\rangle \in C_A \qquad a = \texttt{.drop\_desire}(G)}{\langle ag,C,T,\mathsf{ProcAct}\rangle \longrightarrow \langle ag,C',T,\mathsf{ProcAct}\rangle} \quad \textbf{(ExDropDes)}$$

where:

$$\begin{array}{rl} C'_A = & (C_A \setminus \{\langle a,i\rangle\}) \setminus \{\langle b,j\rangle \mid \\ & \quad G\theta \in agls(j)\} \\ C'_I = & (C_I \cup \{(i'[te:ct \leftarrow h])\theta\}) \setminus \{j \in C_I \mid \\ & \quad G\theta \in agls(j)\} \\ C'_E = & (C_E \setminus \{\langle +!G\theta, k\rangle \in C_E\}) \setminus \{\langle e,j\rangle \mid \\ & \quad G\theta \in agls(j)\} \end{array}$$

with θ a m.g.u. and $i = i'[te:ct \leftarrow a;h]$.

In the rule above, we remove intentions that have a goal unifying with G whether they appear in the set of intentions or are suspended in C_E or C_A. Note that in `.drop_desire` we also need to remove the matching events in C_E for which no plan has been found yet (these are current desires but not yet intentions). For `.drop_intention`, we need to check only the active and suspended intentions.

$$\frac{\langle a,i\rangle \in C_A \qquad a = \texttt{.drop\_intention}(G)}{\langle ag,C,T,\mathsf{ProcAct}\rangle \longrightarrow \langle ag,C',T,\mathsf{ProcAct}\rangle} \quad \textbf{(ExDropInt)}$$

where:

$$\begin{array}{rl} C'_A = & (C_A \setminus \{\langle a,i\rangle\}) \setminus \{\langle b,j\rangle \mid \\ & \quad G\theta \in agls(j)\} \\ C'_I = & (C_I \cup \{(i'[te:ct \leftarrow h])\theta\}) \setminus \{j \in C_I \mid \\ & \quad G\theta \in agls(j)\} \\ C'_E = & C_E \setminus \{\langle e,j\rangle \mid G\theta \in agls(j)\} \end{array}$$

with θ a m.g.u. and $i = i'[te:ct \leftarrow a;h]$.

There are two more internal actions that change the mental state of the agent and are also important, for example because they were essential in the work on plan patterns [13] and only informally presented there. The internal action `succeed_goal` is used to drop goal instances in such a way as if plans to achieve those goal instances had already been executed successfully, whereas `fail_goal` is used to force the plans to achieve instances of the goal to immediately fail, activating the plan failure handling mechanism.

In the next two rules, we use some extra notation to help present them. First, with some abuse of notation, we use $\forall i \in C$ as a universal quantification over the finite set $\{i \mid \langle e,i\rangle \in C_E \vee \langle a,i\rangle \in C_A \vee i \in C_I\}$, that is, all the intentions in the set of intentions but also all the suspended intentions in C_E, and with the extensions presented here, suspended intentions can appear in C_A too. Similarly, $i \notin C'$ means that the components where intention i appeared in C are changed so that i is removed from there in C'. Also, we use $C[i \setminus j]$ to denote the statement that j replaces the occurrence of intention i in the previous state of configuration C. Variables i, j, and k all range over intentions.

$$\frac{\langle a,i\rangle \in C_A \qquad a = \texttt{.succeed\_goal}(G)}{\langle ag,C,T,\mathsf{ProcAct}\rangle \longrightarrow \langle ag,C',T,\mathsf{ProcAct}\rangle} \quad \textbf{(ExSuccGl)}$$

where:

$$\begin{array}{rcl} C'_A & = & C_A \setminus \{\langle a,i\rangle\} \\ C'_I & = & C_I \cup \{(i'[te:ct \leftarrow h])\theta\} \end{array}$$

with $i = i'[te:ct \leftarrow a;h]$

and:

$$\forall j \in C \,.\, G\theta \in agls(j) \rightarrow C'[j \setminus k]$$

with

$$\begin{array}{rcl} j & = & j'[te:ct \leftarrow !G;h]j'' \\ k & = & j'[te:ct \leftarrow h] \end{array}$$

and θ a m.g.u. of all matched goals.

Note how the top of the intention above a plan trying to achieve G is lost: the idea is that all the work being done to achieve G is no longer necessary (i.e., it has been successfully achieved already). The last new rule we present here is used to flag that instances of a goal can no longer be achieved, so the plan failure mechanism needs to be activated.

$$\frac{\langle a,i\rangle \in C_A \qquad a = \texttt{.fail\_goal}(G)}{\langle ag, C, T, \mathsf{ProcAct}\rangle \longrightarrow \langle ag, C', T, \mathsf{ProcAct}\rangle} \quad \textbf{(ExFailGl)}$$

where:

$$\begin{array}{rcl} C'_A & = & C_A \setminus \{\langle a,i\rangle\} \\ C'_I & = & C_I \cup \{i'[te : ct \leftarrow h]\theta\} \end{array}$$

with $i = i'[te : ct \leftarrow a; h]$

and:

$$\forall j \in C \,.\, G\theta \in agls(j) \rightarrow \langle -\%at, j\rangle \in C'_E$$

provided $j = j'[+\%at : ct \leftarrow !G; h]j''$

$$\forall j \in C \,.\, G\theta \in agls(j) \rightarrow j \notin C'$$

provided $j = j'[te : ct \leftarrow !G; h]j'', \quad (te \neq +\%at)$

with θ a m.g.u. of all matched goals, and $\% \in \{!, ?\}$.

Note how the failed plan and those to achieve subgoals thereof are kept below the contingency plan in the intention stack; this can provide useful information for programmers (where again internal actions could be used to access the state of the intention).

We do not present here the new **ClrInt** rules (similar to the ones in the semantics reported elsewhere) which take care of removing finished contingency plans and the failed plans that were kept in the intention above the failed goal that triggered the respective goal deletion.

5 Conclusions and future work

In this paper we have extended the semantics of AgentSpeak to account for some features of the variant of AgentSpeak interpreted by ***Jason*** [6]. We addressed plan failure (including action execution failure and intention suspension while actions execute) and some important internal actions that access or alter the configuration of the transition system giving semantics to AgentSpeak used in the ***Jason*** interpreter (i.e., they alter the agent's mental state). The semantics formalised so far also includes the particular semantics in ***Jason*** for plans triggered by test goals; that appeared in the semantics in [19] where another important extension was presented, that of speech-act based communication.

Having formal semantics is important for the precise understanding of the language, for work on formal verification [4, 5], but also for proving properties of the language. For example, with the semantics given here, we could easily prove that action `.desire(G)` will fail if it is executed in the next immediate reasoning cycle after an instance of `.drop_desire(G)` has been executed.

Future work includes giving semantics to other extensions in ***Jason*** that are still not formalised, such as prolog-like rules in the belief base, the rather sophisticated use of higher-order variables in ***Jason*** [6], other internal actions, etc. Furthermore, it is still to be done the combination of the various extensions of the semantics that appeared in separate work (e.g., the one on communication mentioned above) and giving semantics to the complete agent architecture in which an AgentSpeak interpreter sits. A more ambitious work would be to give semantics to the interaction of ***Jason*** with platforms for other levels of a multi-agent system; in particular, there has been interesting recent work on combining $\mathcal{M}\text{OISE}^+$ [14] and CArtAgO [17] with ***Jason*** which is likely to be very useful in practice and could benefit from formal semantics.

Acknowledgements

The authors are grateful for the supported given by CNPq, grants 307924/2009-2, 307350/2009-6, and 478780/2009-5.

REFERENCES

[1] Davide Ancona and Viviana Mascardi, 'Coo-BDI: Extending the BDI model with cooperativity', in *DALT-03*, LNAI 2990, pp. 109–134, (2004). Springer-Verlag.

[2] Davide Ancona, Viviana Mascardi, Jomi F. Hübner, and Rafael H. Bordini, 'Coo-AgentSpeak: Cooperation in AgentSpeak through plan exchange', in *AAMAS-2004*, pp. 698–705, (2004). ACM Press.

[3] *Multi-Agent Programming: Languages, Platforms and Applications*, eds., Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, Springer-Verlag, 2005.

[4] Rafael H. Bordini, Louise A. Dennis, Berndt Farwer, and Michael Fisher, 'Automated verification of multi-agent programs', in *23rd ASE2008*, pp. 69–78. IEEE, (2008).

[5] Rafael H. Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge, 'Verifying Multi-Agent Programs by Model Checking', *Journal of Autonomous Agents and Multi-Agent Systems*, **12**(2), 239–256, (2006).

[6] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using **Jason***, John Wiley & Sons, 2007.

[7] Rafael H. Bordini and Álvaro 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, (September 2004).

[8] Mehdi Dastani, '2APL: a practical agent programming language', *Autonomous Agents and Multi-Agent Systems*, **16**(3), 214–248, (2008).

[9] Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge, 'A formal specification of dMARS', in *ATAL-97*, LNAI 1365, 155–176, Springer-Verlag, (1998).

[10] Mark d'Inverno and Michael Luck, 'Engineering AgentSpeak(L): A formal computational model', *Journal of Logic and Computation*, **8**(3), 1–27, (1998).

[11] Koen V. Hindriks, 'Modules as policy-based intentions: Modular agent programming in goal', in *ProMAS 2007*, LNCS 4908, pp. 156–171. Springer, (2008).

[12] Jomi F. Hübner, Rafael H. Bordini, and Gauthier Picard, 'Using *jason* to develop a team of cowboys', in *ProMAS-2008*. (Agent Contest Paper).

[13] Jomi Fred Hübner, Rafael H. Bordini, and Michael Wooldridge, 'Programming declarative goals using plan patterns', in *DALT 2006)*, LNCS 4327, pp. 123–140. Springer, (2007).

[14] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier, 'Developing organised multiagent systems using Moise', *IJAOSE*, **1**(3/4), 370–395, (2007).

[15] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf, 'Jadex: A BDI reasoning engine', In Bordini et al. [3], chapter 6, 149–174.

[16] Anand S. Rao, 'AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language', in *MAAMAW'96*, LNAI 1038, pp. 42–55, (1996). Springer-Verlag.

[17] Alessandro Ricci, Michele Piunti, L. Daghan Acay, Rafael H. Bordini, Jomi F. Hübner, and Mehdi Dastani, 'Integrating heterogeneous agent-programming platforms within artifact-based environments', in *AAMAS 2008*, pp. 225–232. IFAAMAS, (2008).

[18] John Thangarajah, James Harland, David Morley, and Neil Yorke-Smith, 'Aborting tasks in BDI agents', in *AAMAS'07*, pp. 1–8, (2007). ACM.

[19] Renata Vieira, Alvaro Moreira, Michael Wooldridge, and Rafael H. Bordini, 'On the formal semantics of speech-act based communication in an agent-oriented programming language', *Journal of Artificial Intelligence Research*, **29**, 221–267, (June 2007).

[20] Michael Winikoff, 'JACK$^{\text{TM}}$ intelligent agents: An industrial strength platform', In Bordini et al. [3], chapter 7, 175–193.

[21] Michael Winikoff and Stephen Cranefield, 'On the testability of BDI agent systems', Discussion Paper 2008/03, Department of Information Science, University of Otago, Dunedin, Otago, New Zealand, (2008). Unpublished.