JASON
Java-based AgentSpeak interpreter used with saci for multi-agent distribution over the net

Christina Bergmann and Manoela Ilic
(chbergman@uos.de, milic@uos.de)

Master Program in Computational Logic
Departamento de Informatica,
Faculdade de Ciencias e Tecnologia
Universidade Nova de Lisboa,
2829-516 Caparica, Portugal

Abstract. In this survey paper we present Jason, an interpreter for the AgentSpeak programming language. Our aim is to investigate the Jason platform theoretically and to explore its practical possibilities in implementing multiagent systems. Furthermore we introduce some relevant background on multiagent systems concerning the BDI architecture and the logic-based semantics of AgentSpeak(L). Finally, we compare the Jason platform to Impact, another platform for multiagent development.

1 Introduction

In the last years agent oriented programming (AOP) has won more and more attention in software engineering. Especially in the attempt to solve complex problems of distributed domains, AOP has turned out to be successful tool. Although most of the recent developed AOP languages lack the rigid semantics as proposed in [1], they represent a remarkable approach toward a new programming paradigm. The main difference between AOP and object oriented programming (OOP) is the functionality: AOP should specifically be meant for agent design while OOP does not have this kind of specification. Furthermore AOP is supposed to offer a more abstract level of abstraction beyond the one of OOP. This is manifested in the idea that AOP uses pseudo-mental terminology which is suitable for describing rational agents. AOP still has to be developed in the theoretical sense as well as for practical usage. An approach like AgentSpeak which we will describe in detail in section 2 is a promising rudiment for this development. Together with the interpreter Jason, multiagent systems can be programmed in a manageable way. To understand the ideas of the AgentSpeak language we will now introduce some basics about BDI architecture and multiagent systems in general.

1.1 BDI Architecture

Most of the definitions given in the domain of agents and multiagent systems have still not been fully agreed upon by everyone. Most authors vary in their

basic assumption about what an agent is and what exactly its main features should be. The reason for this blurriness lies in the issue itself. Clear definitions of these basics do not seem to be neccesary because it is a certain kind of behavior that is desired to emerge from an agent and a multiagent systems. Nonetheless, the abstract desiredata that an agent should be rational does propose some main features an agent should provide. The most important one is autonomy followed by communication, goal-orientation, mobility and learning. An agent is mostly seen as an acting entity which is situated in an environment. To realize the desired properties of an agent, certain agent architectures have been proposed. One of them is the Belief-Desire-Intention (BDI) architecture which describes an agent in a mental terminology adapted to the folk psychological terms used in philosophy to capture intentionality and practical reasoning. This abstract way of defining an agent has lead to a wide range of implementation possibilities and even to an improved understanding of human societies. The most important characteristics of the BDI architecture is its representation of beliefs which stand for what the agent knows about the environment and itself, of desires which are the states of world that the agent wishes to bring about and of intentions which are the desires that the agent has actually committed to. Practical reasoning emerges from the decision what goals are going to be chosen (deliberation) and how these goals can be achieved (means-end reasoning). As described in [2] the main components of a possible BDI architecture can be summarized as follows:

- a set of beliefs Bel,
- a belief revision function

$$brf : \wp(Bel) \times P \rightarrow \wp(Bel) \tag{1}$$

  which determines a new set of beliefs given the current beliefs and a percept;
- an option generation function

$$options : \wp(Bel) \times \wp(Int)\wp \rightarrow Des) \tag{2}$$

  which determines desires (available options) given the current beliefs about its environment and its current intentions. This function is responsible for the means-end reasoning of the agent;
- a set of desires (Des) which are the current options;
- a filter function

$$filter : \wp(Bel) \times \wp(Des) \times \wp(Int) \rightarrow \wp(Int) \tag{3}$$

  which determines the intention given the current beliefs, desires and intentions. The filter function represents the deliberation process in practical reasoning;
- a set of intentions (Int);
- an action selection or execute function

$$execute : \wp(Int) \rightarrow A \tag{4}$$

  where A stands for a directly executable action.

The agent's action function represents the decision process:

$$action : P \rightarrow A. \tag{5}$$

The main problem in the design of a concrete agent using a BDI architecture is the 'commitment' process: should the agent be bold and overcommit to a certain goal or should it rather be cautious and try to reconsider its intentions for they might not be achievable any more? The answer to this question is highly dependent on the environment which can be a rapidly changing one, where a cautious agent would be preferred, or a less dynamic one, where a bold agent would score better. In the next section we will introduce Jason and AgentSpeak(L), an agent-oriented programming language that implements the BDI architecture and offers flexible modeling possibilities for autonomous agents.

1.2 Jason

Jason is language interpreter written in Java. The language interpreted is an extended version of AgentSpeak, one of the agent-oriented programming languages. AgentSpeak implements the BDI architecture of reactive planning systems that are comprised of beliefs and plans. Jason, which was developed as an open source project mainly by Rafael Bordini and Jomi Hübner, implements the operational semantics of AgentSpeak and makes it possible to distribute agents over the net hence developing multiagent systems using the Simple Agent Communication Infrastructure (SACI). The interpreter offers some additional features which we will introduce after the following section about AgentSpeak.

2 AgentSpeak

Implementing an agent or a multiagent system independent from the choice of architecture might be possible in various programming languages. Especially object-oriented programming provide some attractive features. So why designing an agent-oriented programming language? As mentioned before, there are certain desiredata for an agent's capabilities and if there would be a programming language that naturally captures these characteristics (also described in [3]), it could become by far more simpler to implement agents. Besides some restrictions on the abstract BDI architecture, AgentSpeak realizes this and provides a way to implement autonomous agents and systems of distributed agents.

2.1 The AgentSpeak Language

The core of the AgentSpeak language is the representation of beliefs and plans. The agent's reasoning which leads to an action given a percept, is interconnected with the agent's databases of plans and beliefs. The basic concepts of AgentSpeak can be summarized in the following points ([4] [5] [6]) :
Let b and g be predicate symbols, $\mathbf{t}$ a term $(t_1, ..., t_n)$ and a an action symbol, then we can define

- a set of beliefs in first-order logic form (belief base, where b($\mathbf{t}$) and $\neg b(\mathbf{t})$ is a belief literal),
- a set of plans (plan library),
- achievement goals (prefix operator '!': !g($\mathbf{t}$)),
- test goals (prefix operator '?': ?g($\mathbf{t}$)),
- triggering events (considering the beliefs, the achievement and test goals, +b($\mathbf{t}$), –b($\mathbf{t}$)+!g($\mathbf{t}$), +?g($\mathbf{t}$), –!g($\mathbf{t}$), –?g($\mathbf{t}$) are triggering events),
- addition(+) and deletion(–) of beliefs,
- actions (a($\mathbf{t}$); here t stands for first-order terms),

A plan is build up from actions that are represented in first-order predicate symbols. A triggering event e together with a context of beliefs and goals form a plan:

$$e : b_1 \wedge ... \wedge b_m \leftarrow h_1; ...; h_n, \tag{6}$$

where b_1 $\wedge$...$\wedge$b$_m$ is the context, b_1,...,b_m are belief literals and h_1,...,h_n are goals. Everything left from the arrow is called the head of the plan while the other side is the body. An empty body is replaced by the expression *true*. If the belief context is a logical consequence of the agent's present beliefs then the plan is applicable. Achievement goals are predicates which the agent seeks to make true, e.g. where g(t) is a true belief, while test goals are predicates that already exist in the agent's set of beliefs or not. The agent tests if this goal is actually in his belief base. If the environment changes or a new goal emerges, the agent has to react internally to these alterations. Triggering events represent such changes and they are responsible for the addition and deletion of beliefs or goals.
Now, that we have the most important notions and an informal syntax description, it is worth looking at an example to understand the usage. The example agent is a little robot and it has the simple task to pick up socks from the ground and to put them into the cloth basket, avoiding other obstacles in the room. If the robot detects a sock it should react to this, pick it up and bring it to the cloth basket. This can be represented by the following plan (Plan A):

```
+loc(sock,X): loc(me,X) &
              loc(basket,Y)
              <- grab(sock);
                 !loc(me,Y);
                 put(sock).
```

Every time the robot finds itself in the position of a sock, Plan A gets triggered by the triggering event

```
+loc(sock,X).
```

The belief context in this case is

```
loc(me,X) & loc(basket,Y)
```

and the goals to be tested, or like here, to be achieved are

```
grab(sock); !loc(me,Y); put(sock)
```

which means that the robot should grab the sock, reach the location of the basket and put the sock into it. The body of a plan is desired to be made true. Until it is not, the robot has to try to bring up the state which makes it true.
A complete agent can be described as an 8-tupel $\langle E, B, P, I, A, S_E, S_O, S_I \rangle$ consisting of the following components: a set of events, a set of beliefs, a set of plans, a set of intentions, a set of executable actions, and three selection functions for events, applicable plans and intentions.
When an AgentSpeak program with all the necessary components of beliefs and plans is interpreted by Jason, a cycle is repeated which includes the following steps (see figure 1 in appendix):

1. The list of events is updated in every interpretation cycle. Perceptions and executions of intentions are responsible for that. A change in the agent's beliefs or a perception are an update causes for beliefs. Updating beliefs causes the insertion of an event in the set of events.
2. An event is selected by S_E.
3. Plans that are triggered by the selected event and that are executable in the current state are being retrieved. That means that the heads of plans are tried to be unified with that event (relevant plans). Plans are executable in the current state if the context of the plans can be unified with the agent's present beliefs (applicable plans).
4. Now, an applicable plan is selected by the function S_O and an instance of that plan is created (intended means). Depending on the type of event, internal or external, the plan is either pushed on a present intention or a new intention is created, respectively.
5. S_I selects an intention which has a plan on top. The beginning of the plan indicates what kind of execution is done. If the first formula in the body indicates an action then this action is executed. Otherwise, if it is a goal, then a corresponding event is added to the set of events which still have to be processed. If the goal is a test goal then a check is performed.
6. If this top plan is complete, the next plan is considered. The intention can be removed from the set of intentions if it is empty which means that the intention has succeeded.

A very detailed description of the interpretation cycle is given in [7] and [4].

2.2 Jason and the Operational Semantics of AgentSpeak

In this subsection we will give a summery of the formal semantics of AgentSpeak. For a full elaboration on this topic see [6].
The semantics of AgentSpeak is given in operational semantics form which is a set of rules that define the transitions between the configurations of the agent [4]. The configuration of an AgentSpeak agent is given as follows:

$$\langle ag, C, M, T, s \rangle \tag{7}$$

where ag is the program itself which is given by the set of beliefs and the plans. C denotes the circumstance of the agent which is again a 3-tupel $\langle I, E, A\rangle$ entailing a set of intentions, a set of events and a set of actions. M is as well a 3-tupel $\langle In, Out, SI\rangle$ called the communication component that consists of the mail inbox, the mail outbox and the intention monitor that captures the intentions that were suspended because of received messages. Messages have the form $\langle mid, id, ilf, cnt\rangle$ which entails the information of a message identifier, the sender id, the illocutionary force of the message and the content. T is the temporary information component $\langle R, Ap, \iota, \epsilon, \rho\rangle$ with the set of relevant plans, the set of applicable plans and record keepers ι, ϵ, ρ which during the execution monitor a certain intention, event and applicable plan. The last element s of the tupel indicates the present deliberation step of the reasoning cycle of the agent. It is an element of one of the 9 'meta-actions' partly mentioned in the interpretation cycle in the last subsection:

- processing a message from the inbox (ProcMsg),
- selecting an event from the set of events (SelEv),
- retrieving all relevant plans (RelPl),
- checking which of these plans are applicable (ApplPl),
- selecting one applicable plan which is the intended means (SelAppl),
- adding the new intended means to the set of intentions (AddIM),
- selecting an intention (SelInt),
- executing this intention (ExecInt),
- clearing the intention or intended means that got completed in the last step (ClrInt).

At the beginning of a Jason cycle the initial configuration is $\langle ag, C, M, T, ProcMsg\rangle$ where C, M, and T are all empty. The following rules are applied:

1. **Event Selection**: A function like S_E is used to select an event which is then removed from the set of events E and assigned to ϵ ($SelEv_1$). If there is no event to handle ($SelEv_2$), the rule jumps to the Execution rule.
2. **Check Relevant Plans**: The R component in T is initialised with all its relevant plans (Rel_1) or the event is discarded if there are no relevant plans (Rel_2).
3. **Check Applicable Plans**: The Ap component of T gets initialised with the set of applicable plans ($Appl_1$) or discarded if there are no ($Appl_2$).
4. **Applicable Plan Selection**: An applicable plan from Ap is chosen by a function like S_O and assigned to the ρ component of T ($SelAppl$).
5. **Update**: When an applicable plan was chosen the set of intentions can be updated. According weather an event is external or internal, a new intention is added ($ExtEv$) or the relevant plan is put on top of intention related to the event ($IntEv$).
6. **Intention Selection**: S_I is used to select an intention ($IntSel_1$ and $IntSel_2$ if the set is empty).
7. **Execution**: A plan body is executed where the plan is on top of the selected intention. A new internal event is registered in the set of events

($Achieve, andTest_1 andTest2 forTestGoals$) or a new event is added to the set of events ($AddBel$) or deleted from it ($DelBel$), which functions as a belief update.

8. **Remove Finished Intentions**: Intentions are removed from the set of intentions if they are finished that means empty ($ClrInt_1$) or the leftovers of the plan that was put on the top of the intention are removed from the intention ($ClrInt_2$).

The formal notation of the rules like described in [4] can be found in the appendix of this paper.

3 Features of Jason

Jason offers additional features in the usage of AgentSpeak. It accepts a certain BNF grammar of the AgentSpeak syntax which is given in [5] on page 9. An important feature of Jason is strong negation. This allows a closed world assumption (not using strong negation anywhere) as well as an open world assumption. Moreover literals can be annotated which is useful in selecting plans and in communication. Plan labels can be annotated as well, e.g.

```
@Label[chanceSuccess(0.7), usualPayoff(0.9), OtherProperty]
+b(X) : c(t) <- a(X).
```

entails a label that indicates some information of how useful the plan might be, where the annotations are represented in a list of ground terms. Some standard annotations are already given by Jason. Handling plan failure is another feature of Jason which still has to be formally described. The programmer can give instructions on how to proceed when a plan failure has been detected, otherwise Jason answers with a warning. Furthermore, Jason has a standard library of internal actions which are actions that are different from the actions that cause an effect on the agent's environment. These internal actions can be programmed or taken from the standard library of Jason.
One of the most important features of Jason is the simple configuration of multiagent systems (MAS). The components of a multiagent system are its environment and the agents in it. The configuration file contains all the crucial information. It specifies the kind of infrastructure that is used as well as the society of agents, e.g.:

```
MAS AgentSystem {
architecture: Saci
environment: EnvironmentName
agents: ag1; ag2; ag3; ag4
}
```

Where each agent in a MAS runs an own interpreter of AgentSpeak. An environment description is given by the programmer in a Java class which is also

supported in its construction by Jason. The architecture of a MAS can be centralised or distributed (SACI) which applies if the agents are being run on different computers. All functions given initially by Jason can be defined and modified by the programmer itself. There are various customizable functions e.g. for section and trust as well as the possibility to define the overall agent architecture by modifying perception functions, the kind of inter-agent communication, how actions should be performed and how the belief revision function should work.
Jason also provides inter-agent communication based on the speech act theory and cooperation[1]. The communication elements that Jason provides are currently based on KQML performatives with some additional MAS specific performatives (e.g. for the exchange of plans). Messages in a Jason MAS are sent and received asynchronously which means with time delay in between. Selection functions for which message the agent should process first on the beginning of the reasoning cycle described above can be specified by the programmer. Messages can as well be defined in a social way, giving the information on what the social position of a sender agent is and it can be chosen then if the message received by this agent should be regarded as 'compulsive' or powerful in its illocutionary act, or if it is altruistically acceptable regardless from the agent's social power. Similarly, trust functions can be implemented. The annotation option can be used for beliefs to mark which belief is acquired e.g. due to a received message of another agent.

3.1 Applying Jason

The simple way of creating a MAS in Jason makes it easy to work with a stable system that is practicable. Once the Java classes are created and the relevant functions customized the MAS runs. Since Jason is implemented in Java, there is no problem on using it on different operation systems which makes it possible to have a flexible distribution of a MAS over the net. The system architectures of Jason of which one uses SACI are only specified for that certain distribution. But since the architectures can be created customly it would also be possible to use other mechanisms.
Jason itself has not been used for industrial purposes since it is a young platform with needs for extensions and formalization. Nonetheless, the fact that it is an AgentSpeak interpreter makes it applicable in plenty developments concerning MAS (e.g. PRS and dMARS). Possible applications of Jason are seen in Social Simulation and the Semantic Web [4].

4 IMPACT vs. Jason

In this section we are going to introduce another platform for developing multiagent systems called IMPACT. In the next subsection a summery of the main

[1] The Coo-AgentSpeak mechanism for introducing cooperation in BDI multiagent systems is not yet available in Jason

features of IMPACT is given and then we are going to compare IMPACT to Jason and show the important differences as well as the pro and contra sides concerning both platforms.

4.1 Features of IMPACT

IMPACT (Interactive Maryland Platform) is a platform that provides a set of servers to improve the interoperability between agents which is independent form the application used [8]. The servers are yellow pages, thesaurus, registration, type and interface, offering access to distributed and heterogeneous data sources and an infrastructure on which the agents can interact. Agents can be designed on top of existing legacy code. IMPACT also provide means for coordination in MAS. The basis of IMPACT are manifested in formal methods from computational logic using mathematical notions. An agent in IMPACT is build up of two parts, a set of data structures or types that can be manipulated by the agent and a set of functions that manipulate the data types. The agent's program is 'wrapped' which means that several components are used that are responsible for the program's structure. The components are the followings:

1. Message Box
2. State
3. Action Base
4. Query Language
5. Concurrent action mechanism
6. Action Constraints
7. Integrity Constraints
8. Agent Program

The services used in IMPACT are restricted to an HTML like form with certain defined features (Service name, inputs, output and attributes).
Agents are programmed in a specific language and once an agent is created it is possible to register it in the IMPACT server and to use the services offered by it. A so called Agent Roost can take up many agents and forward messages to its agents. The Roost offers a certain control mechanism for the agents. Another component of IMPACT are the Connections that facilitates agent framework connections and connections to other systems. IMPACT has a central management for Log messages.

4.2 Differences to Jason

IMPACT provides a very effective way for agent interoperability and information exchange. While IMPACT was specifically designed to meet this criteria, Jason lacks predefined interoperability. Another feature not integrated in IMPACT but not in Jason is a secure agent environment which is e.g. useful and necessary in e-commerce. On the other hand, Jason is much more flexible in adding and modifying features and functions while IMPACT has a more rigid architecture

for agents. The language in IMPACT is specific for IMPACT implementation and the programmer has to learn that language first. Jason is based on AgentSpeak which is very simple in its syntax and semantics because of its first-order logic like structure. Jason itself is written in Java and can be worked with instantly if some programming knowledge is present. Another advantage of Jason is its availability: it is an open source project. The IMPACT community only offers documentation material while versions of the platforms itself are hardly obtainable.

References

1. Shoham, Y.: Agent-oriented programming. *Artificial Intelligence*. Volume 60, Issue 1, 1993, Pages 51–92.
2. Weiss, G. (ed.): *Multiagent Systems – A Modern Approach to Artificial Intelligence*. MIT Press, 1999, Chapter 1, Pages 54–61.
3. Weerasooriya, D., Rao, A.S., Ramamohanarao, K.: Design of a Concurrent Agent-Oriented Language. *ECAI Workshop on Agent Theories, Architectures, and Languages*, 1994, Pages 386–401.
4. Bordini, R.H., Hübner, J.F., Vieira, R.: Jason and the golden fleece of agent-oriented programming. *Bordini, R.H., Destani, M., Dix, J., Seghrouchni, A. (eds.): Multi-Agent Programming : Languages, Platforms and Applications (Multiagent Systems, Artificial Societies, and Simulated Organizations)*, Springer-Verlag, 2005, Chapter 1, Pages 3–37.
5. Bordini, R.H., Hübner, J.F.: Jason – A java-based AgentSpeak interpreter used with SACI for mulit-agent distribution over the net. Online tutorial: *http://jason.sourceforge.net*
6. Rao, A.S.: AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. MAAMAW, 1996, Pages 42–55.
7. D'Inverno, M., Luck, M.: Engeneering AgentSpeak(L) – A Formal Computational Model. *Journal of Logic Computation*, Oxford University Press, 1998, Volume 8 No. 3, Pages 1–27.
8. Arisha, K., Ozcan, K., Ross, R., Subrahmanian, V., Eiter,T., Kraus, S.: IMPACT: A Platform for Collaborating Agents. *IEEE Intelligent Systems*. 1999, Volume 14, March/April, Pages 64–72.

Appendix

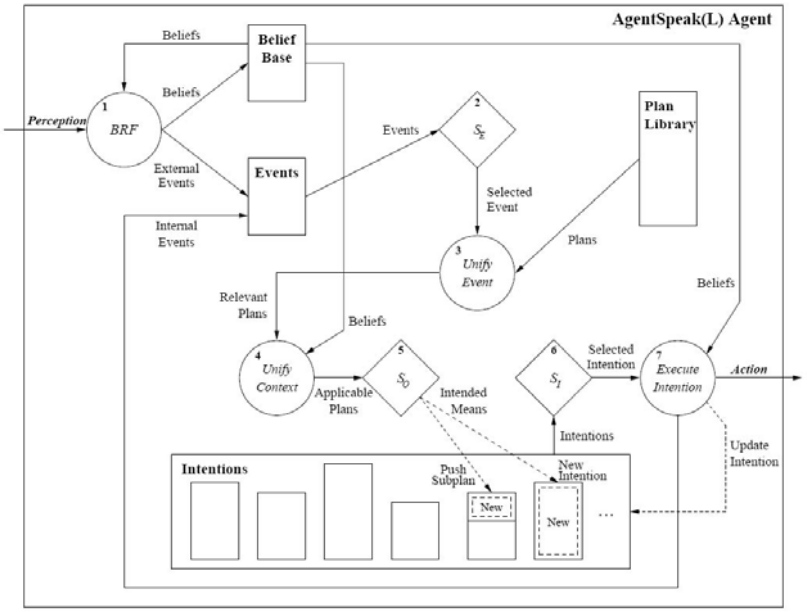

Figure 1

$$\frac{S_E(C_E) = \langle te, i\rangle}{\langle ag, C, M, T, \mathsf{SelEv}\rangle \longrightarrow \langle ag, C', M, T', \mathsf{RelPl}\rangle} \quad (\mathbf{SelEv_1})$$

$$\textit{where:} \quad C'_E = C_E \setminus \{\langle te, i\rangle\} \qquad T'_\varepsilon = \langle te, i\rangle$$

$$\frac{C_E = \{\}}{\langle ag, C, M, T, \mathsf{SelEv}\rangle \longrightarrow \langle ag, C, M, T, \mathsf{SelInt}\rangle} \quad (\mathbf{SelEv_2})$$

$$\frac{T_\varepsilon = \langle te, i\rangle \qquad \mathsf{RelPlans}(ag_{ps}, te) \neq \{\}}{\langle ag, C, M, T, \mathsf{RelPl}\rangle \longrightarrow \langle ag, C, M, T', \mathsf{ApplPl}\rangle} \quad (\mathbf{Rel_1})$$

$$\textit{where:} \quad T'_R = \mathsf{RelPlans}(ag_{ps}, te)$$

$$\frac{T_\varepsilon = \langle te, i\rangle \qquad \mathsf{RelPlans}(ag_{ps}, te) = \{\}}{\langle ag, C, M, T, \mathsf{RelPl}\rangle \longrightarrow \langle ag, C, M, T, \mathsf{SelEv}\rangle} \quad (\mathbf{Rel_2})$$

$$\frac{\mathsf{AppPlans}(ag_{bs}, T_R) \neq \{\}}{\langle ag, C, M, T, \mathsf{ApplPl}\rangle \longrightarrow \langle ag, C, M, T', \mathsf{SelAppl}\rangle} \quad (\mathbf{Appl_1})$$

$$\textit{where:} \quad T'_{Ap} = \mathsf{AppPlans}(ag_{bs}, T_R)$$

$$\frac{\mathsf{AppPlans}(ag_{bs}, T_R) = \{\}}{\langle ag, C, M, T, \mathsf{ApplPl}\rangle \longrightarrow \langle ag, C, M, T, \mathsf{SelInt}\rangle} \quad (\mathbf{Appl_2})$$

$$\frac{S_{Ap}(T_{Ap}) = (p, \theta)}{\langle ag, C, M, T, \mathsf{SelAppl}\rangle \longrightarrow \langle ag, C, M, T', \mathsf{AddIM}\rangle} \quad (\mathbf{SelAppl})$$

$$\textit{where:} \quad T'_p = (p, \theta)$$

$$\frac{T_\varepsilon = \langle te, \mathrm{T}\rangle \qquad T_p = (p,\theta)}{\langle ag, C, M, T, \mathsf{AddIM}\rangle \longrightarrow \langle ag, C', M, T, \mathsf{SelInt}\rangle} \quad \textbf{(ExtEv)}$$

$$\textit{where:} \quad C_I' = C_I \cup \{\ [p\theta]\ \}$$

$$\frac{T_\varepsilon = \langle te, i\rangle \quad T_p = (p,\theta)}{\langle ag, C, M, T, \mathsf{AddIM}\rangle \longrightarrow \langle ag, C', M, T, \mathsf{SelInt}\rangle} \quad \textbf{(IntEv)}$$

$$\textit{where:} \quad C_I' = C_I \cup \{\ (i[p])\theta\ \}$$

$$\frac{C_I \neq \{\} \qquad S_I(C_I) = i}{\langle ag, C, M, T, \mathsf{SelInt}\rangle \longrightarrow \langle ag, C, M, T', \mathsf{ExecInt}\rangle} \quad \textbf{(IntSel}_\mathbf{1}\textbf{)}$$

$$\textit{where:} \quad T_i' = i$$

$$\frac{C_I = \{\}}{\langle ag, C, M, T, \mathsf{SelInt}\rangle \longrightarrow \langle ag, C, M, T, \mathsf{ProcMsg}\rangle} \quad \textbf{(IntSel}_\mathbf{2}\textbf{)}$$

$$\frac{T_i = i[head \leftarrow a\,; h]}{\langle ag, C, M, T, \mathsf{ExecInt}\rangle \longrightarrow \langle ag, C', M, T, s\rangle} \quad \textbf{(Action)}$$

$$\begin{array}{rcl} \textit{where:} \quad C_A' & = & C_A \cup \{a\} \\ C_I' & = & (C_I \setminus \{T_i\}) \cup \{i[head \leftarrow h]\} \\ s & = & \begin{cases} \mathsf{ClrInt} & \text{if } h = \top \\ \mathsf{ProcMsg} & \text{otherwise} \end{cases} \end{array}$$

$$\frac{T_i = i[head \leftarrow\, !at\,; h]}{\langle ag, C, M, T, \mathsf{ExecInt}\rangle \longrightarrow \langle ag, C', M, T, \mathsf{ProcMsg}\rangle} \quad \textbf{(Achieve)}$$

$$\begin{array}{rcl} \textit{where:} \quad C_E' & = & C_E \cup \{\langle +!at, i[head \leftarrow h]\rangle\} \\ C_I' & = & C_I \setminus \{T_i\} \end{array}$$

$$\frac{T_i = i[head \leftarrow ?at\,; h] \qquad \mathsf{Test}(ag_{bs}, at) \neq \{\}}{\langle ag, C, M, T, \mathsf{ExecInt}\rangle \longrightarrow \langle ag, C', M, T, s\rangle} \quad \textbf{(Test}_\mathbf{1}\textbf{)}$$

$$\begin{array}{rcl} \textit{where:} \quad C_I' & = & (C_I \setminus \{T_i\}) \cup \{(i[head \leftarrow h])\theta\} \\ & & \theta \in \mathsf{Test}(ag_{bs}, at) \\ s & = & \begin{cases} \mathsf{ClrInt} & \text{if } h = \top \\ \mathsf{ProcMsg} & \text{otherwise} \end{cases} \end{array}$$

$$\frac{T_i = i[head \leftarrow ?at\,; h] \qquad \mathsf{Test}(ag_{bs}, at) = \{\}}{\langle ag, C, M, T, \mathsf{ExecInt}\rangle \longrightarrow \langle ag, C', M, T, s\rangle} \quad \textbf{(Test}_\mathbf{2}\textbf{)}$$

$$\begin{array}{rcl} \textit{where:} \quad C_E' & = & C_E \cup \{\langle +?at, i[head \leftarrow h]\rangle\} \\ C_I' & = & C_I \setminus \{T_i\} \\ s & = & \begin{cases} \mathsf{ClrInt} & \text{if } h = \top \\ \mathsf{ProcMsg} & \text{otherwise} \end{cases} \end{array}$$

$$\frac{T_i = i[head \leftarrow +b\,; h]}{\langle ag, C, M, T, \mathsf{ExecInt}\rangle \longrightarrow \langle ag', C', M, T, s\rangle} \quad \textbf{(AddBel)}$$

$$\begin{array}{rcl} \textit{where:} \quad ag_{bs}' & = & ag_{bs} + b[\mathtt{self}] \\ C_E' & = & C_E \cup \{\langle +b[\mathtt{self}], \mathrm{T}\rangle\} \\ C_I' & = & (C_I \setminus \{T_i\}) \cup \{i[head \leftarrow h]\} \\ s & = & \begin{cases} \mathsf{ClrInt} & \text{if } h = \top \\ \mathsf{ProcMsg} & \text{otherwise} \end{cases} \end{array}$$

$$\frac{T_i = i[head \leftarrow -b; h]}{\langle ag, C, M, T, \mathsf{ExecInt}\rangle \longrightarrow \langle ag', C', M, T, s\rangle} \quad (\textbf{DelBel})$$

$$\begin{array}{lrcl} \textit{where:} & ag'_{bs} & = & ag_{bs} - b[\mathtt{self}] \\ & C'_E & = & C_E \cup \{\langle -b[\mathtt{self}], \mathsf{T}\rangle\} \\ & C'_I & = & (C_I \setminus \{T_i\}) \cup \{i[head \leftarrow h]\} \\ & s & = & \begin{cases} \mathsf{ClrInt} & \text{if } h = \top \\ \mathsf{ProcMsg} & \text{otherwise} \end{cases} \end{array}$$

$$\frac{j = [head \leftarrow \mathsf{T}], \text{ for some } j \in C_I}{\langle ag, C, M, T, \mathsf{ClrInt}\rangle \longrightarrow \langle ag, C', M, T, \mathsf{ProcMsg}\rangle} \quad (\textbf{ClrInt}_1)$$

$$\textit{where:} \quad C'_I = C_I \setminus \{j\}$$

$$\frac{j = i[head \leftarrow \mathsf{T}], \text{ for some } j \in C_I}{\langle ag, C, M, T, \mathsf{ClrInt}\rangle \longrightarrow \langle ag, C', M, T, \mathsf{ProcMsg}\rangle} \quad (\textbf{ClrInt}_2)$$

$$\textit{where:} \quad C'_I = (C_I \setminus \{j\}) \cup \{i\}$$