Jason Intentional Learning: an Operational Semantics

Carlos A. González-Alarcón[1], Francisco Grimaldo[1], and Alejandro Guerra-Hernández[2]

[1] Departament d'Informàtica, Universitat de València
Avinguda de la Universitat s/n, (Burjassot) València, España
{carlos.a.gonzalez, francisco.grimaldo}@uv.es
[2] Departamento de Inteligencia Artificial, Universidad Veracruzana
Sebastián Camacho No. 5, Xalapa, Ver., México, 91000
aguerra@uv.mx

Abstract. This paper introduces an operational semantics for defining Intentional Learning on *Jason*, the well known Java-based implementation of *AgentSpeak*(*L*). This semantics enables *Jason* to define agents capable of learning the reasons for adopting intentions based on their own experience. In this work, the use of the term *Intentional Learning* is strictly circumscribed to the practical rationality theory where plans are predefined and the target of the learning processes is to learn the reasons to adopt them as intentions. Top-Down Induction of Logical Decision Trees (TILDE) has proved to be a suitable mechanism for supporting learning on *Jason*: the first-order representation of TILDE is adequate to form training examples as sets of beliefs, while the obtained hypothesis is useful for updating the plans of the agents.

Keywords: Operational semantics, Intentional Learning, AgentSpeak(L), Jason .

1 Introduction

In spite of the philosophical and formal sound foundations of the Belief-Desire-Intention (BDI) model of rational agency [4,11,12], learning in this context has received little attention. Coping with this, frameworks based on Decision Trees [13,14] or First-Order Logical Decision Trees [7] have been developed to enable BDI agents to learn about the executions of their plans.

JILDT[3] [8] is a library that provides the possibility to define Intentional Learning agents in *Jason*, the well known Java-based implementation [3] of *AgentSpeak*(*L*) [10]. Agents of this type are able to learn about their reasons to adopt intentions, performing Top-Down Induction of Logical Decision Trees [1]. A plan library is defined for collecting training examples of executed intentions, labelling them as succeeded or failed, computing logical decision trees, and using

[3] Available on http://jildt.sourceforge.net/

the induced trees to modify accordingly the plans of learner agents. In this way, the Intentional Learning approach [9] can be applied to any *Jason* agent by declaring its membership to this type of agent.

The *AgentSpeak*(L) language interpreted by *Jason* does not enable learning by default. However it is possible to extend the language grammar and its semantics for supporting Intentional Learning. This paper focuses on describing this operational semantics, which enables *Jason* to define agents capable to learn the reasons for adopting intentions based on their own experience. Direct inclusion of the learning steps into the reasoning cycle makes this approach unique.

The organization of the paper is as follows: Section 2 briefly introduces the *AgentSpeak*(L) agent oriented programming language, as implemented by *Jason*. Section 3 introduces briefly the Top-Down Induction of Logical Decision Trees (TILDE) method. Section 4 describes the language grammar and operational semantics that define Intentional Learner agents on *Jason*. Finally, section 5 states the final remarks and discusses future work, particularly focusing on the issues related with social learning.

2 Jason and AgentSpeak(L)

Jason [3] is a well known Java-based implementation of the *AgentSpeak*(L) abstract language for rational agents. For space reasons, a simplified version of the language interpreted by *Jason* containing the fundamental concepts of the language that concerns this paper is shown in the Table 1 (the full version of the language is defined in [3]). An agent ag is defined by a set of beliefs bs and plans ps. Each belief $b \in bs$ can be either a ground first-order literal or its negation (a belief) or a Horn clause (a rule). Atoms at are predicates, where P is a predicate symbol and $t_1, \ldots, t_n$ are standard terms of first-order logic. Besides, atoms can be labelled with sources. Each plan $p \in ps$ has the form: $@lbl\ te : ct \leftarrow h$. $@lbl$ is an unique atom that identifies the plan. A trigger event (te) can be any update (addition or deletion) of beliefs or goals. The context (ct) of a plan is an atom, the negation of an atom or a conjunction of them. A non empty plan body (h) is a sequence of actions, goals, or belief updates. Two kinds of goals are defined, achieve goals (!) and test goals (?).

ag	$::=$	$bs \quad ps$		h_1	$::=$	$a \mid g \mid u \mid h_1; h_1$	
bs	$::=$	$b_1 \ldots b_n$	$(n \geq 0)$	at	$::=$	$P(t_1, \ldots, t_n)$	$(n \geq 0,$
ps	$::=$	$p_1 \ldots p_n$	$(n \geq 1)$			$\mid P(t_1, \ldots, t_n)[s_1, \ldots, s_m]$	$m > 0)$
p	$::=$	$@lbl\ te : ct \leftarrow h$		s	$::=$	$\texttt{percept} \mid \texttt{self} \mid id$	
te	$::=$	$+at \mid -at \mid +g \mid -g$		a	$::=$	$A(t_1, \ldots, t_n)$	$(n \geq 0)$
ct	$::=$	$ct_1 \mid \top$		g	$::=$	$!at \mid ?at$	
ct_1	$::=$	$at \mid \neg at \mid ct_1 \wedge ct_1$		u	$::=$	$+b \mid -b$	
h	$::=$	$h_1; \top \mid \top$					

Table 1. *Jason* language grammar. Adapted from [3].

The operational semantics of the language is given by a set of rules that define a transition system between configurations, as depicted in Figure 2(a). A configuration is a tuple $\langle ag, C, M, T, s \rangle$, where:

- ag is an agent program defined by a set of beliefs bs and plans ps.
- An agent circumstance C is a tuple $\langle I, E, A \rangle$, where: I is a set of intentions; E is a set of events; and A is a set of actions to be performed in the environment.
- M is a set of input/output mailboxes for communication.
- T is a tuple $\langle R, Ap, \iota, \varepsilon, \rho \rangle$ that keeps track of temporary information. R and Ap are the sets of relevant and applicable plans, respectively. ι, ε, ρ record the current intention, event and selected plan, respectively.
- s labels the current step in the reasoning cycle of the agent.

Transitions are defined in terms of semantic rules with form:

$$\frac{cond}{C \rightarrow C'}(\textbf{rule id})$$

where $C = \langle ag, C, M, T, s \rangle$ is a configuration that can become a new configuration C' if a *cond*ition is satisfied. Appendix A shows the operational semantic rules extracted from from [3,2] that are relevant for the purposes of this paper.

3 Top-down Induction of Logical Decision Trees

Top-down Induction of Logical DEcision Trees (TILDE) [1] is an Inductive Logic Programming technique adopted for learning in the context of rational agents [9]. The first-order representation of TILDE is adequate to form training examples as sets of beliefs, e.g., the beliefs of the agent supporting the adoption of a plan as an intention; and the obtained hypothesis is useful for updating the plans and beliefs of the agents.

A Logical Decision Tree is a binary first-order decision tree where: (a) Each node is a conjunction of first-order literals; and (b) The nodes can share variables, but a variable introduced in a node can only occur in the left branch below that node (where it is true). Unshared variables may occur in both branches.

Three inputs are required to compute a Logical Decision Tree: A set of training examples, the background knowledge of the agent and the language bias. Training examples are atomic formulae composed of an atom referring to the plan that was intended; the set of beliefs the agent had when the intention was adopted or when the intention failed; and the label indicating a successful or failed execution of the intention. Examples are collected every time the agent believes an intention has been achieved (success) or dropped (failure). The rules believed by the agent, constitute the background knowledge of the agent, i.e., general knowledge about the domain of experience of the agent. The language bias is formed by *rmode* directives that indicate which literals should be considered as candidates to form part of a Logical Decision Tree.

The TILDE algorithm is basically a first-order version of the well known C4.5 algorithm. The algorithm is not described in this paper, due to space limitations, but it is advisable to consult the original report of TILDE [1] or the version of the algorithm reported in [8] for further details.

4 Extending the Language: a TILDE-Learning approach

The extension to the grammar that is required to incorporate the induction of Logical Decision Trees [8] into *Jason* is shown in Table 2. As any agent, a learner agent ag_{lrnr} is formed by a set of beliefs and a set of plans. Beliefs can be either normal beliefs *nbs* (as defined by *bs* in Table 1) or learning beliefs *lbs*. Learning beliefs are related to the learning process input and configuration. These beliefs can be of three types: *rmode* directives, *settings* and training *examples*. *rmode* literals are directives used to represent the language bias (as introduced in Section 3). *settings* literals customize the learning process configurations, e.g., the metrics used for building a new hypothesis. In turn, *examples* are literals used for defining training examples. A training example defines the relation between the label of the plan chosen to satisfy an intention, the perception of the environment, and the result of the execution of the plan (*succ*essful or *fail*ed) captured as the class of the examples. A plan can be either normal (non learnable) or *learnable*, i.e., a plan in which new contexts can be learned. To become a learnable plan (*lp*), a plan just need to annotate its label as such.

ag_{lrnr}	::=	$bs\ \ ps$		$class$	::=	`succ` | `fail`	
bs	::=	$nbs\ \ lbs$		ps	::=	$nps\ \ lps$	
nbs	::=	$b_1 \dots b_n$	$(n \geq 0)$	nps	::=	$p_1 \dots p_n$	$(n \geq 1)$
lbs	::=	$lb_1 \dots lb_n$	$(n \geq 0)$	lps	::=	$lp_1 \dots lp_n$	$(n \geq 1)$
lb	::=	`rmode`(b)		lp	::=	@lbl[`learnable`]	
	|	`settings`$(t_1, \dots, t_n)$	$(n \geq 2)$			$te : ct \leftarrow h$	
	|	`example`$(lbl, bs, class)$					

Table 2. Extension to the *Jason* grammar language enabling *Intentional Learning*.

Semantics is extended by adding a Learning component (L) into configurations $\langle ag, C, M, T, L, s\rangle$. L is a tuple $\langle \rho, Exs, Cnds, Bst, Build, Tree\rangle$ where:

- The plan that triggered the learning process is denoted by ρ,
- Exs is the set of training examples related to the executed intention,
- $Cnds$ is the set of literals (or conjunctions of literals) that are candidates to form part of the induced logical decision tree,
- Bst is a pair $\langle at, gain\rangle$ (where $gain \in \mathbb{R}$) keeping information about the candidate that maximizes the gain ratio measure,
- $Build$ is a stack of *building tree parameters (btp)* tuples. Every time a set of training examples is split, a new *btp* is added into $Build$ for computing a new inner tree afterwards. Each *btp* is a tuple $\langle Q, Exs, Branch\rangle$, where : Q is the conjunction of literals in top nodes; Exs is the partition of examples that satisfies Q, which will be used for building the new tree; and $Branch \in \{left, right\}$ indicates where the tree being computed must be placed.
- $Tree$ is a stack of lists. A tree can be represented as a list, where the first element denotes the node label and the remaining elements represent left and right branches, respectively. Figure 1 shows how this works.

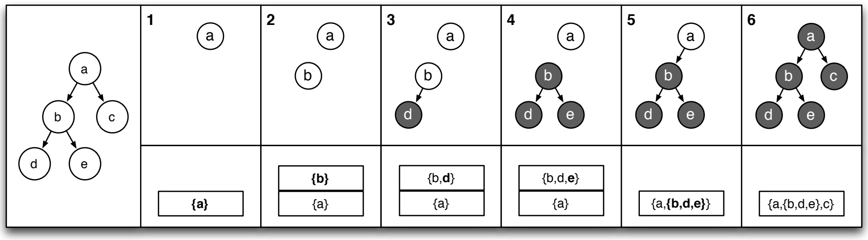

Fig. 1. Every time an inner tree is being built, a new list is added into the stack (1,2). When a leaf node is reached, this is added into the list on the top of the stack (3,4); in case of a right branch, the whole list is removed and added into the next list (5). If there is not a list under the top of the stack the main tree has been computed (6).

For the sake of readability, we adopt the following notational conventions in semantic rules:

- We write L_ρ to make reference to the component ρ of L. Similarly for all the other components of a configuration.
- A stack is denoted by $[\alpha_1 \ddagger \ldots \ddagger \alpha_z]$, where α_1 is the bottom of the stack and α_z is the top of the stack. $\ddagger$ delimits the elements of the stack. $L_{Build}[\alpha]$ denotes the α-element on the top of stack L_{Build}. Similarly for L_{Tree}.
- If p is a plan on the form $@lbl\ te : ct \leftarrow h$, then $Label(p) = lbl$, $TrEv(p) = te$, $Ctxt(p) = ct$ and $Body(p) = h$.
- $Head(lst)$ and $Tail(lst)$ denote the head and the tail of a list, respectively.

The reasoning cycle is then extended for enabling Intentional Learning as can be seen in Figure 2(b). Two rules enable agents to collect training examples labelled as *succ* when the execution of a plan is successful ($ColEx_{succ}$) or *fail* otherwise ($ColEx_{fail}$).

Rule $\mathbf{ColEx_{succ}}$ adds a training example labelled as *succ* when the selected event T_ε is an achievement goal addition event, the selected plan T_ρ is a learnable plan, and the execution of an intention is done, (i.e., when the reasoning cycle is in the step $ClrInt$ and there is nothing else to be executed). This rule removes the whole intention T_ι like rule $ClrInt_1$ in the default operational semantics (Appendix A) but for learnable plans.

$$\frac{T_\varepsilon = \langle +!at, i\rangle \qquad T_\rho \in ag_{lps} \qquad T_\iota = [head \leftarrow \top]}{\langle ag, C, M, T, L, ClrInt\rangle \rightarrow \langle ag', C', M, T, L, ProcMsg\rangle} \ (\mathbf{ColEx_{succ}})$$

s.t. $ag'_{lbs} = ag_{lbs} + example(Label(T_\rho), intend(at) \cup ag_{bs}, succ)$
$C'_I = C_I \backslash \{T_\iota\}$

In a similar way, rule $\mathbf{ColEx_{fail}}$ adds a training example labelled as *fail* when the reasoning cycle is on the step $ExecInt$, the selected event T_ε is an achievement goal deletion event and the selected plan T_ρ is a learnable plan. Besides

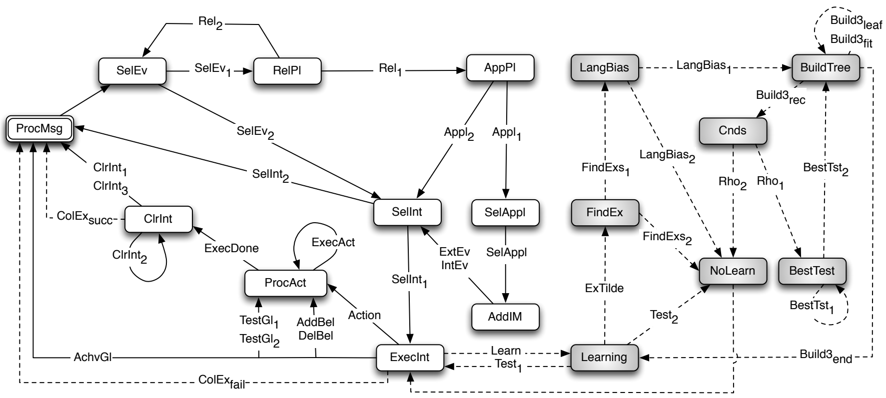

Fig. 2. Extended reasoning cycle. a) Unshaded states and solid lines define the basic reasoning cycle. b) Shaded states and dashed lines represent the extension of the reasoning cycle.

adding a new training example, this rule adds an achievement goal learning event. The current intention T_ι is suspended and associated to the new event. Since a new event is added, the reasoning cycle is moved towards *ProcMsg*, as it does the rule *AchvGl* in the default operational semantics (Appendix A).

$$\frac{T_\varepsilon = \langle -!at, i\rangle \qquad T_\rho \in ag_{lps} \qquad T_\iota = i[head \leftarrow h]}{\langle ag, C, M, T, L, ExecInt\rangle \rightarrow \langle ag', C', M, T, L', ProcMsg\rangle} \; (\mathbf{ColEx_{fail}})$$

s.t. $ag'_{lbs} = ag_{lbs} + example(Label(T_\rho), intend(at) \cup ag_{bs}, fail)$
$C'_E = C_E \cup \{\langle +!learning, T_\iota\rangle\}$
$L'_\rho = T_\rho$
$C'_I = C_I \backslash \{T_\iota\}$

The learning process starts in *ExecInt* by rule **Learn** when the selected event is $\langle +!Learning, i\rangle$. Rule **ExTilde** fires when L_{Tree} is an empty stack, and starts the construction of the Logical Decision Tree.

$$\frac{T_\varepsilon = \langle +!learning, i\rangle}{\langle ag, C, M, T, L, ExecInt\rangle \rightarrow \langle ag, C, M, T, L, Learning\rangle} \; (\mathbf{Learn})$$

$$\frac{L_{Tree} = \top}{\langle ag, C, M, T, L, Learning\rangle \rightarrow \langle ag, C, M, T, L, FindEx\rangle} \; (\mathbf{ExTilde})$$

Once the tree has been built, rules $\mathbf{Test_1}$ and $\mathbf{Test_2}$ check whether something new has been learned. Rule $\mathbf{Test_1}$ is fired when the new hypothesis does not subsume the prior context (i.e., it is not a generalization of it). Then, it parses the Tree into a Logical Formula (through function `parseLF`), that is used to update the context of the failed plan, and restarts the learning component for future calls. Instead, when the learned hypothesis subsumes the prior context, rule $\mathbf{Test_2}$ moves the learning cycle towards the $NoLearn$ state. Note that a hypothesis subsuming a prior context means that nothing new has been learned, since learning was triggered because of a plan failure. The agent cycle is automatically lead from the $NoLearn$ to the $ExecInt$ state. Recovering from a situation in which individual learning has been unsuccessful is out of the scope of this paper and remains part of the future work, as discussed in Section 5.

$$\frac{L_\rho = @lbl\ te : ct \leftarrow h \quad \texttt{parseLF}(L_{Tree}) = lct \quad lct \not\preceq ct}{\langle ag, C, M, T, L, Learning\rangle \rightarrow \langle ag', C', M, T, L', ExecInt\rangle}(\mathbf{Test_1})$$

s.t. $ag'_{ps} = \{ag_{ps} \backslash L_\rho\} \cup \{@lbl[learnable]\ te : lct :\leftarrow h\}$
$L' = \langle \top, \{\}, \{\}, \top, [], []\rangle$
$C'_E = C_E \backslash T_\varepsilon$

$$\frac{L_\rho = te : ct \leftarrow h \quad \texttt{parseLF}(\mathtt{L_{Tree}}) = lct \quad lct \preceq ct}{\langle ag, C, M, T, L, Learning\rangle \rightarrow \langle ag', C, M, T, L, NoLearn\rangle}(\mathbf{Test_2})$$

4.1 Semantics for Building Logical Decision Trees

This section presents the transition system for building a Logical Decision Tree. The first thing a learner agent needs for executing the TILDE algorithm is to get the set of training examples regarding the failed plan. Each example related to the failed plan is represented as $example(lbl, bs, class)$, where the first argument is the label of the failed plan; the rest of the arguments are the state of the world when the example was added and the label class of the example. Rule $\mathbf{FindExs_1}$ applies when at least one example regarding L_ρ is a logical consequence of the learning beliefs of the agent. Here, the set L_{Exs} is updated and the learning cycle goes forward the $LangBias$ step. If there is no training example, rule $\mathbf{FindExs_2}$ moves the cycle forward the $NoLearn$ step.

$$\frac{Exs = \{lbs | lbs = example(Label(L_\rho), bs, class) \wedge ag_{lbs} \models lbs\}}{\langle ag, C, M, T, L, FindEx\rangle \rightarrow \langle ag, C, M, T, L', LangBias\rangle}\ (\mathbf{FindExs_1})$$

s.t. $L'_{Exs} = Exs$

$$\frac{ag_{lbs} \not\models example(Label(L_\rho), bs, class)}{\langle ag, C, M, T, L, FindEx\rangle \rightarrow \langle ag, C, M, T, L, NoLearn\rangle}\ (\mathbf{FindExs_2})$$

The language bias is generated by rule $\mathbf{LangBias_1}$, through the function `getLangBias()`, whose only parameter is the set of training examples in L_{Exs}. If the set of *rmode* directives is not empty, the cycle goes forward the step

$BuildTree$. In this transition, the whole directives in LB are added as beliefs in the learning beliefs of the agent. Besides, a *building tree parameters* tuple is added in L_{Build}: the initial query is a literal indicating the intention the agent was trying to reach; the initial examples are those in L_{Exs}; and the symbol $\top$ denotes that this is the configuration for building the main node. The rule **LangBias$_\mathbf{2}$** moves the cycle forward the $NoLearn$ step when is not possible to generate the language bias (training examples had no information about the beliefs of the agent when these were added).

$$\frac{\texttt{getLangBias}(L_{Exs}) = LB}{\langle ag, C, M, T, L, LangBias\rangle \rightarrow \langle ag', C, M, T, L', BuildTree\rangle}\ (\mathbf{LangBias_1})$$

s.t. $\forall(rmode(X) \in LB)\ .\ ag'_{lbs} = ag_{lbs} + rmode(X)$
$L_\rho = @lbl\ {+}!at : ct \leftarrow h$
$L'_{Build} = [\langle intend(at), L_{Exs}, \top\rangle]$

$$\frac{\texttt{getLangBias}(L_{Exs}) = \{\}}{\langle ag, C, M, T, L, LangBias\rangle \rightarrow \langle ag, C, M, T, L, NoLearn\rangle}\ (\mathbf{LangBias_2})$$

At this point, the necessary data for executing the TILDE algorithm [1] has been processed. Next rules define the transitions for building a Logical Decision Tree. Rule **Build3$_\mathbf{leaf}$** is applied when a stop condition is reached (e.g., the whole examples belong to the same class). A boolean function like `stopCriteria()` returns *true* if the examples in its argument satisfy a stop criteria, and *false* otherwise. The *leaf* node is obtained through the function `majority_class` and it is added into the list on the top of the L_{Tree} stack. If the leaf node is in a *Right* branch, the whole list on the top is removed from the top and added into the list under the top of the stack (see Figure 1). The stack L_{Build} is updated removing the tuple on the top of it. If no stop condition is found, the learning cycle moves towards the next step (**Build3$_\mathbf{rec}$**).

$$\frac{L_{Build}[\langle Q, Exs, Branch\rangle] \quad \texttt{stopCriteria}(Exs) = true}{\langle ag, C, M, T, L, BuildTree\rangle \rightarrow \langle ag, C, M, T, L', BuildTree\rangle}(\mathbf{Build3_{leaf}})$$

s.t. $leaf = \texttt{majority\_class}(Exs)$,
$L_{Tree} = [T_z \ddagger \ldots \ddagger T_2 \ddagger T_1]$,
$Tree = T_1 \cup \{leaf\}$,

$$L'_{Tree} = \begin{cases} [T_z \ddagger \ \ldots \ddagger\ T_2 \ddagger\ Tree] & \text{if } Branch = Left, \text{ or} \\ & (Branch = Right \text{ and } T_2 = \top) \\ [T_z \ddagger\ \ldots\ \ddagger \{T_2 \cup Tree\}] & \text{if } Branch = Right \text{ and } T_2 \neq \top \end{cases}$$

$L'_{Build} = L_{Build} \backslash \langle Q, Exs, Branch\rangle$

$$\frac{L_{Build}[\langle Q, Exs, Branch\rangle] \quad \texttt{stopCriteria}(Exs) = false}{\langle ag, C, M, T, L, BuildTree\rangle \rightarrow \langle ag, C, M, T, L, Cnds\rangle}(\mathbf{Build3_{rec}})$$

Sometimes, the L_{Tree} stack has more than one element but there are no more elements for building inner nodes (e.g. the right side of a tree is deeper than the left one). In this cases, rule $\mathbf{Build3_{fit}}$ flats the L_{Tree} stack adding the list on the top of the stack inside the one below until there is only one list in the stack.

$$\frac{L_{Tree}[T_2 \ddagger T_1] \quad T_2 \neq \top \quad L_{Build}[\top]}{\langle ag, C, M, T, L, BuildTree\rangle \rightarrow \langle ag, C, M, T, L', BuildTree\rangle} (\mathbf{Build3_{fit}})$$

s.t. $L'_{Tree} = [T_z \ddagger \ \ldots \ \ddagger \{T_2 \cup T_1\}]$

Rule $\mathbf{Rho_1}$ generates the candidates to form part of the tree using the function `rho()` whose parameters are a query Q and the language bias. This rule updates the element L_{Cnds} when a non-empty set of candidates has been generated; otherwise, rule $\mathbf{Rho_2}$ moves the cycle forwards the $NoLearn$ step.

$$\frac{\begin{array}{c} LB = \{lb | lb = rmode(RM) \wedge ag_{lbs} \models lb\} \\ L_{Build}[\langle Q, Exs, Branch\rangle] \quad \texttt{rho}(Q, LB) = Candidates \end{array}}{\langle ag, C, M, T, L, Cnds\rangle \rightarrow \langle ag, C, M, T, L', BestTest\rangle} (\mathbf{Rho_1})$$

s.t. $L'_{Cnds} = Candidates$
$\forall (cnd \in Candidates) \ . \ cnd = at_1 \wedge ... \wedge at_n \qquad (n \geq 1)$

$$\frac{\begin{array}{c} LB = \{lb | lb = rmode(RM) \wedge ag_{lbs} \models lb\} \\ L_{Build}[\langle Q, Exs, Branch\rangle] \quad \texttt{rho}(Q, LB) = \{\} \end{array}}{\langle ag, C, M, T, L, Cnds\rangle \rightarrow \langle ag, C, M, T, L, NoLearn\rangle} (\mathbf{Rho_2})$$

Rule $\mathbf{BestTst_1}$ evaluates iteratively each candidate in L_{Cnds} for selecting the candidate that maximizes gain ratio.

$$\frac{\texttt{gainRatio}(Head(L_{Cnds})) = G}{\langle ag, C, M, T, L, BestTest\rangle \rightarrow \langle ag, C, M, T, L', BestTest\rangle} (\mathbf{BestTst_1})$$

s.t. $L'_{Bst} = \begin{cases} G & \text{if } G > L_{Bst} \\ L_{Bst} & \text{otherwise} \end{cases}$
$L'_{Cnds} = Tail(L_{Cnds})$

When all candidates have been evaluated, rule $\mathbf{BestTst_2}$ splits the training examples in those satisfying $Q \wedge L_{Bst}$ and those that do not. Two new *btp* tuples are added into the L_{Build} stack for building inner trees afterwards, and a new list is added into the L_{Tree}.

$$\frac{L_{Cnds} = \{\} \quad L_{Build}[\langle Q, Exs, Branch\rangle}{\langle ag, C, M, T, L, BestTest\rangle \rightarrow \langle ag, C, M, T, L', BuildTree\rangle} (\mathbf{BestTst_2})$$

s.t. $bg = \{bs \in ag_{bs} | bs = at\text{:-}body \ \wedge \ body \neq \top\}$
$Exs_L = \{Ex \in Exs | Ex = example(lbl, bs, class) \ \wedge \ (bs \cup bg) \models (Q \wedge L_{Bst})\}$
$Exs_R = \{Ex \in Exs | Ex = example(lbl, bs, class) \ \wedge \ (bs \cup bg) \not\models (Q \wedge L_{Bst})\}$
$L_{Build} = [btp_z \ddagger \ \ldots \ \ddagger \langle Q, Exs, Branch\rangle]$
$L'_{Build} = [btp_z \ddagger \ \ldots \ \ddagger \langle Q, Exs_R, Right\rangle \ddagger \langle Q \wedge L_{Bst}, Exs_L, Left\rangle]$

Finally rule $\mathbf{Build3_{end}}$ indicates the end of the building process when there is no building tree parameters tuple in the stack L_{Build}. The flow of the cycle goes forward the step *Learning* for processing the learned hypothesis.

$$\frac{L_{Build}[\top]}{\langle ag, C, M, T, L, BuildTree\rangle \rightarrow \langle ag, C, M, T, L, Learning\rangle} (\mathbf{Build3_{end}})$$

As mentioned before, once the Logical Decision Tree has been built, the learned hypothesis is used for updating the plans of the agent when more specific hypothesis is learned (see rule $\mathbf{Test_1}$). If the learned hypothesis is either more general or similar to prior knowledge means that there was no learning, and therefore the reasoning cycle continues with its default operation.

5 Discussion and future work

The operational semantics presented in this paper defines Intentional Learning on *Jason*, which has served to create agents capable of learning new reasons for adopting intentions, when the executions of their plans failed. Learning is achieved through Top-Down Induction of Logical Decision Trees (TILDE), that has proved to be a suitable mechanism for supporting learning on Jason since the first-order representation of these trees is adequate to form training examples as sets of beliefs, while the obtained hypothesis is useful for updating the plans of the agents. Current work provides a formal and precise approach to incorporate Intentional Learning into BDI multi-agent systems, and a better understanding of the reasoning cycle of agents performing this type of learning. For reasons of space, a demonstration scenario showing the benefits of this approach is not presented here but can be found in [8], where we evaluate how agents improve their performance by executing Intentional Learning whenever the execution of a plan is failed.

The semantics presented in this paper paves the way for future research on Social Learning, as an alternative for recovering from individual learning failures. Social Learning has been defined as the phenomenon by means of which a given agent can update its own knowledge base by perceiving the positive or negative effects of any given event undergone or actively produced by another agent on a state of the world within which the learning agent has as a goal [6]. It would be interesting to identify the mechanisms that must be implemented at the agent level to enable them to learn from one another. A first answer is that the intentional level of the semantics presented in this paper is required to define distributed learning protocols as a case of collaborative goal adoption [5], where a group of agents sharing a plan has as social goal learning a new context for the plan in order to avoid possible future failures. As an example of learning protocol, agents in a group could share experiences (training examples) with the learner agent (the one that discovered the plan execution failure) to achieve this social goal.

Acknowledgements

This work has been jointly supported by the Spanish MICINN and the European Commission FEDER funds, under grant TIN2009-14475-C04. First author is supported by Conacyt doctoral scholarship number 214787. Third author is supported by Conacyt project number 78910.

References

1. H. Blockeel, L. D. Raedt, N. Jacobs, and B. Demoen. Scaling up inductive logic programming by learning from interpretations. *Data Mining and Knowledge Discovery*, 3(1):59–93, 1999.
2. R. H. Bordini and J. F. Hübner. Semantics for the jason variant of agentspeak (plan failure and some internal actions). In *ECAI*, pages 635–640, 2010.
3. R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in Agent-Speak using Jason*. John Wiley & Sons Ltd, 2007.
4. M. E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA., USA, and London, England, 1987.
5. C. Castelfranchi. Modelling social action for ai agents. *Artif. Intell.*, 103(1-2):157–182, Aug. 1998.
6. R. Conte and M. Paolucci. Intelligent social learning. *Journal of Artificial Societies and Social Simulation*, 4(1), 2001.
7. A. Guerra-Hernández, A. El-Fallah-Seghrouchni, and H. Soldano. Learning in BDI Multi-agent Systems. In J. Dix and J. Leite, editors, *Computational Logic in Multi-Agent Systems: 4th International Workshop, CLIMA IV, Fort Lauderdale, FL, USA, January 6–7, 2004, Revised and Selected Papers*, volume 3259 of *Lecture Notes in Computer Science*, pages 218–233, Berlin Heidelberg, 2004. Springer-Verlag.
8. A. Guerra-Hernández, C. González-Alarcón, and A. El FallahSeghrouchni. Jason Induction of Logical Decision Trees (Jildt): A learning library and its application to commitment. EUMAS 2010, 2010.
9. A. Guerra-Hernández and G. Ortíz-Hernández. Toward BDI sapient agents: Learning intentionally. In R. V. Mayorga and L. I. Perlovsky, editors, *Toward Artificial Sapience*, pages 77–91. Springer London, 2008.
10. A. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In R. van Hoe, editor, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands, 1996.
11. A. Rao and M. Georgeff. Modelling rational agents within a BDI-Architecture. Technical Report 14, Carlton, Victoria, February 1991.
12. A. S. Rao and M. P. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):293–342, 1998.

13. D. Singh, S. Sardina, and L. Padgham. Extending BDI plan selection to incorporate learning from experience. *Robotics and Autonomous Systems*, 58(9):1067 – 1075, 2010. Hybrid Control for Autonomous Systems.
14. D. Singh, S. Sardiña, L. Padgham, and G. James. Integrating learning into a BDI agent for environments with changing dynamics. In *IJCAI*, pages 2525–2530, 2011.

A Jason Semantic Rules

The following $AgentSpeak(L)$ operational semantic rules are relevant for the purposes of this paper, in particular for defining the relation with the rules defined for collecting training examples and the way that an achievement goal deletion event is triggered. For a detailed reviewing of this rules, is highly recommended to consult the text in [3,2].

$$\frac{T_\iota = i[head \leftarrow !at; h]}{\langle ag, C, M, T, ExecInt\rangle \rightarrow \langle ag, C', M, T, ProcMsg\rangle}(\textbf{AchvGl})$$

s.t. $C'_E = C_E \cup \{\langle +!at, T_\iota\rangle\}, C'_I = C_I \setminus \{T_\iota\}$

$$\frac{T_\iota = [head \leftarrow \top]}{\langle ag, C, M, T, ClrInt\rangle \rightarrow \langle ag, C', M, T, ProcMsg\rangle}(\mathbf{ClrInt_1})$$

s.t. $C'_I = C_I \setminus \{T_\iota\}$

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

s.t.

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

$C'_I = C_I \cup \{i'[te : ct \leftarrow h]\}$, 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 $\% \in \{!, ?\}$

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