Jason Induction of Logical Decision Trees (Jildt): A learning library and its application to Commitment

Alejandro Guerra-Hernández[1], Carlos Alberto González-Alarcón[1], and Amal El Fallah Seghrouchni[2]

[1] Departamento de Inteligencia Artificial
Universidad Veracruzana
Sebastián Camacho No. 5, Xalapa, Ver., México, 91000
`aguerra@uv.mx, dn_carlos@hotmail.com`

[2] Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie
4, Place Jussieu, Paris, France, 75005
`Amal.Elfallah@lip6.fr`

Abstract. This paper[3] presents JILDT, a library that defines two agent classes for Jason, the java-based implementation of *AgentSpeak*(L). The agents defined as instances of these classes can learn about their reasons to adopt and abandon intentions, performing induction of logical decision trees. The library enables collecting training examples composed by the beliefs of the agents when a plan is adopted as an intention, labeled as success or failed executions. The induced trees are used to refine the context of the plans in question and to form rules for dropping intentions. The library is tested studying commitment in relation to intentional learning: A simple problem in a world of blocks is used to compare the behavior of a default Jason agent that does not reconsider his intentions; a learning agent that reconsiders by experience when to adopt intentions; and a single-minded agent that also drops intentions when this is rational. Results are very promissory for justifying a formal theory of single-mind commitment based on learning.

Keywords: Intentional Learning, AgentSpeak(L), Inductive Logic Programming, Logical Decision Trees, Commitment.

1 Introduction

It is well known that the Belief-Desire-Intention (BDI) model of agency [10, 11] lacks of learning competences. Coping with this, we introduce JILDT (Jason Induction of Logical Decision Trees): A library that defines two learning agent classes for Jason [3], the well known java-based implementation of the *AgentSpeak*(L) model [12].

[3] Published partially in MICAI 2010, LNAI Vol. 6437:375–385.

Agents defined as instances of the JILDT *intentionalLearner* class can learn about their reasons to adopt intentions, performing first-order induction of logical decision trees [1]. A set of plans and actions are defined in the library for collecting training examples of executed intentions, labeling them as succeeded or failed executions, computing the target language for the induction, and using the induced trees to modify accordingly the plans of the learning agents. In this way, the intentional learning approach [6] can be applied to any Jason agent by declaring the membership to this class.

The second class of agents defined in JILDT deals with single-mind commitment [10], i.e., once an agent intends something, he maintains his intention until he believes it has been accomplished or he believes it is not possible to eventually accomplish it anymore. It is known that Jason agents are not single-minded by default [3, 7]. So, agents defined as instances of the JILDT $singleMinded$ class achieve single-mind commitment, performing a policy-based reconsideration, where policies are rules learned by the agents for dropping intentions . This is foundational and theoretical relevant, since the approach reconciles policy-based reconsideration, as defined in the theory of practical reasoning [4], with computational notions of commitment as the single-mind case [10]. Attending in this way the normative and descriptive aspects of reconsideration, opens the door for a formal theory of reconsideration in *AgentSpeak*(L) based on intentional learning.

Organization of the paper is as follows: Section 2 offers a brief introduction to the *AgentSpeak*(L) agent oriented programming language, as defined in Jason. An agent program, used in the rest of the paper, is introduced to exemplify the reasoning cycle of Jason agents. Section 3 introduces the Top-Down Induction of Logical Decision Trees (Tilde) method, emphasizing the way Jason agents can use it for learning. Section 4 describes the implementation of the JILDT library. Section 5 presents the experimental results for three agents in the blocks world: a default Jason agent, an intentional learner and a single-mind committed agent. Section 6 offers discussion, including related and future work.

2 Jason and AgentSpeak(L)

Jason [3] is a well known Java based implementation of the *AgentSpeak*(L) [12] abstract language for BDI agents. As usual an agent ag is formed by a set of plans ps and beliefs bs. Each belief $b_i \in bs$ is a ground first-order term. Each plan $p \in ps$ has the form $trigger\ event : context \leftarrow body$. A trigger event can be any update (addition or deletion) of beliefs (at) or goals (g).The context of a plan is an atom, a negation of an atom or a conjunction of them. A non empty plan body is a sequence of actions (a), goals, or belief updates. $\top$ denotes empty elements, e.g., plan bodies, contexts, intentions. Atoms (at) can be labelled with sources. Two kinds of goals are defined, achieve goals (!) and test goals (?).

The operational semantics [3] of the language, is given by a set of rules that define a transition system (see figure 1) between configurations $\langle ag, C, M, T, s\rangle$, where:

– *ag* is an agent program formed 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 stores the current applicable plans, relevant plans, intention, etc.
– s labels the current step in the reasoning cycle of the agent.

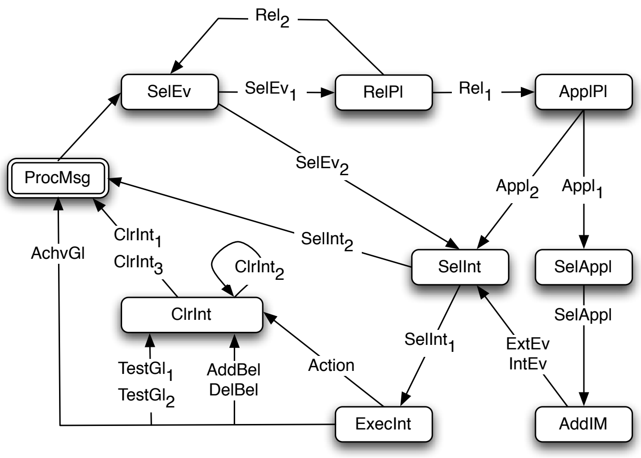


Fig. 1. The transition system for AgentSpeak(L) operational semantics.

An artificially simplified agent program for the blocks world environment, included in the distribution of Jason, is listed in the table 1. Examples in the rest of this paper are based on this agent program. Initially he believes that the *table* is clear (line 3) and that something with nothing on is clear too (line 2). He has a plan labeled *put* (line 10) expressing that to achieve putting a block X on Y, in any context (*true*), he must move X to Y. Our agent is bold about putting things somewhere else.

Now suppose the agent starts running in his environment, where someone else asks him to put b on c. A reasoning cycle of the agent in the transition system shown in Figure 1 is as follows: at the configuration labeled as *procMsg* the beliefs about *on*/2 are perceived (lines 5–8) reflecting the state of the environment; and an event $+!put(b, c)$ is pushed on C_E. Then this event is selected at configuration *SelEv* and the plan *put* is selected as relevant at configuration *RelPl*. Since the context of *put* is true, it is always applicable and it will be selected to form a new intention in C_I at *AddIM*. Once selected for execution at *SelInt*, the action $move(b, c)$ will be actually executed at *ExecInt* and since there is nothing else to be done, the intention is dropped from C_I at *ClrInt*. Coming back to *ProcMsg* results in the agent believing $on(b, c)$ instead of $on(b, a)$.

Table 1. A simplified agent in the blocks world.

```
// Beliefs
clear(X) :- not(on(_,X)).
clear(table).
// Beliefs perceived
on(b,a).
on(a,table).
on(c,table).
on(z,table).
// Plans
@put
+!put(X,Y) : true <- move(X,Y).
```

Now, what if something goes wrong? For instance, if another agent puts the block z on c before our agent achieves his goal? Well, his intention will fail. And it will fail every time this happens. The following section introduces the induction of logical decision trees, and the way they can be used to learn that *put* is applicable only when X and Y are clear.

3 Tilde

The Top-down Induction of Logical Decision Trees (Tilde) [1] is an inductive logic programming technique, that we have adopted for learning in the context of BDI agents [6]. The first-order representation of Tilde is adequate to form training examples as sets of beliefs, i.e., the beliefs of the agent supporting the adoption of a plan as an intention; and the obtained hypothesis are useful for updating the plans and beliefs of the agents, i.e., a logical tree expresses hypotheses about the successful or failed executions of the intentions, as illustrated in figure 2.

In what follows, Tilde is briefly introduced, emphasizing the compatibility with the agents in Jason. First, a Logical Decision Tree is a binary first-order decision tree where:

- Each node is a conjunction of first-order literals; and
- 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).

Three inputs are required to compute a logical decision tree: First, a set of training examples, where each trainning example, known as model, is composed by the set of beliefs the agent had when the intention was adopted; a literal coding what is intended; and a label indicating a successful or failed execution of the intention. Models are computed every time the agent believes an intention has been achieved (success) or dropped (failure). Table 2 shows two models corresponding to the examples in figure 2. The class of the examples is introduced

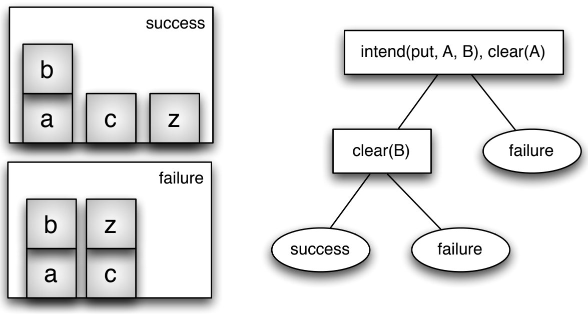


Fig. 2. A Tilde simplified setting: two training examples and the induced tree, when intending to put *b* on *c*.

Table 2. The training examples from figure 2 as models for Tilde. Labels at line 2.

```
1  begin(model(1))              begin(model(2))
2     succ.                        fail.
3     intend(put,b,c).             intend(put,b,c).
4     on(b,a).                     on(b,a).
5     on(a,table).                 on(a,table).
6     on(c,table).                 on(z,c).
7     on(z,table).                 on(c,table).
8  end(model(1))                end(model(2))
```

at line 2, and the associated intention at line 3. The rest of the model corresponds to the beliefs of the agent when he adopted the intention.

Second, the rules believed by the agent, like *clear*/1 in table 1 (lines 2–3), do not form part of the training examples, since they constitute the background knowledge of the agent, i.e., general knowledge about the domain of experience of the agent.

And third, the language bias, i.e., the definition of which literals are to be considered as candidates to be included in the logical decision tree, is defined combinatorially after the literals used in the agent program, as shown in table 3. The *rmode* directives indicate that their argument should be considered as a candidate to form part of the tree. The *lookahead* directives indicate that the conjunction in their argument should be considered as a candidate too. The last construction is very important since it links logically the variables in the intended plan with the variables in the candidate literals, enabling generalization.

For the considered example, the induced decision tree for two successful examples and one failed is showed in table 4. Roughly, it is interpreted as: when intending to put a block A on B, the intention succeeds if A is clear and B is clear (line 3); and fails otherwise (lines 4 and 5). This tree is equivalent to the one shown in figure 2.

Table 3. The language bias defining the vocabulary to build the decision tree.

```
rmode(clear(V1)). rmode(on(V1,V2)). rmode(on(V2,V1)).
rmode(intend(put,V1,V2)).
lookahead(intend(put,V1,V2),clear(V1)).
lookahead(intend(put,V1,V2),clear(V2)).
lookahead(intend(put,V1,V2),on(V1,V2)).
lookahead(intend(put,V1,V2),on(V2,V1)).
```

Table 4. The induced Logical Decision Tree.

```
intend(put,A,B),clear(A) ?
+ yes:  clear(B) ?
|       + yes:  [succ]   [1 examples(s)]
|       + no:  [fail]   [1 example(s)]
+ no:  [fail]   1
```

3.1 Building a Logical Desicion Tree

Logical Decision Trees are computed recursively, as in ID3-like algorithms (table 1). Given an inicial query $Q = true$ and a set of labeled examples E, a set of refinement candidates $\rho(Q)$ is computed after the language bias (rmodes and lookaheads) and the query Q. Candidates are especializations of Q. The candidate that maximizes gain ratio (Eq. 1) is selected (line 2). The procedure finishes when a stop criteria (line 3) is reached, e.g., the gain ratio is not improved by the candidate. If this is the case, a leaf with the majority class for the example is returned. Otherwise, an internal node has to be computed. In order to do that, the content of the internal node $Conj$ is computed extracting the query Q from the candidate Q_b (line 6). The set of examples E is partitioned (lines 7–8) in those examples where the query succeeds (E_1) and those where it does not (E_2). Then the procedure is called recursively to build the left and right branches of the internal node with content $Conj$ (lines 9-11). The procedures returns the built tree T.

The best candidate Q_b is selected (Algorithm 2) as follows: Each refinement candidate $r_i \in \rho(\leftarrow Q)$ induces a partition in the examples E. A quality criterion of such split is computed using gain ratio (Eq. 1) as measure. For this, a matrix *counter* stores the number of examples that succeds and fails of each class c (lines 7–12). The candidate selected is the one that maximizes gain ration (line 17). Gain ration is an entropy based measure of information. The entropy of a set of examples E is:

$$s(E) = -\sum_{i=1}^{k} p(c_i, E) \log p(c_i, E)$$

Algorithm 1 Induction of Logical Decision Trees.

```
1: procedure BUILDTREE(E,Q)                          ▷ $E$ is a set of examples, $Q$ a query
2:     $\leftarrow Q_b := best(\rho(\leftarrow Q))$               ▷ $best$ max information gain
3:     if $stopCriteria(\leftarrow Q_b)$ then              ▷ E.g., No information gain obtained
4:         $T := leaf(majority\_class(E))$
5:     else
6:         $Conj \leftarrow Q_b \backslash Q$
7:         $E_1 \leftarrow \{e \in E \mid e \wedge B \models Q_b\}$
8:         $E_2 \leftarrow \{e \in E \mid e \wedge B \not\models Q_b\}$
9:         $buildTree(Left, E_1, Q_b)$ ;
10:        $buildTree(Right, E_2, Q)$
11:        $T \leftarrow nodei(Conj, Left, Right)$
12:    end if
13:    return $T$                                        ▷ The built tree
14: end procedure
```

where k is the number of classes, c_i are the classes and $p(c_i, E)$ is the proportion of examples in E that belongs to class c_i.

We are interested in refinement candidates r_i that reduces the entropy of the examples, i.e., those that increases information. The Information Gain of a candidate, as usual, is computed as follows:

$$infoGain(E, counter) = s(E) - \sum_{V \in \{true, false\}} \frac{counter(V)}{|E|} s(E_{r_i})$$

where *counter* is the matrix computed in algorithm 2 and $E_{r_i} \subseteq E$ is the partition induced by r_i in the examples E. The maximal gain that a refinement candidate r_i can get is:

$$maxGain(E) = - \sum_{E_{r_i} \subseteq E} \frac{|E_{r_i}|}{|E|} \log \frac{|E_{r_i}|}{|E|}$$

Finally, Gain Ratio is computed as follows:

$$gainRatio(E, counter) = \frac{infoGain(E, counter)}{maxGain(E)} \tag{1}$$

4 Implementation

JILDT implements two classes of agents: The first one is the *intentionalLearner* class, that defines agents capable of refining the context of their plans accordingly to the induced decision trees. In this way, the reasons to adopt a plan that has failed, as an intention in future deliberations, are reconsidered. The second one is the *singleMindedLearner* class, that implements agents that are also capable

Algorithm 2 Best candidate selection.

```
1: procedure BEST(E,R)    ▷ $E$ is a set of examples, $R = \rho(\leftarrow Q)$ refinements of $Q$
2:     i := 1;
3:     for all $r_i \in R$ do
4:         for all $c \in C$ do                ▷ $C = \{success, failure\}$
5:             counter[true][c] := 0; counter [false][c] := 0;
6:         end for
7:         for all $e \in E$ do
8:             if $e \wedge B \models r_i$ then
9:                 counter[true][class(e)]++;
10:            else
11:                counter[false][class(e)]++;
12:            end if
13:            $s_i$ := gainRatio(E, counter);
14:        end for
15:        i++;
16:    end for
17:    return $Q_b := r_i$ s.t. $max(s_i)$
18: end procedure
```

of learning rules that express when it is rational to drop an intention. The body of these rules is obtained from the branches in the induced decision trees that lead to failure. For this, the library defines a set of plans to allow the agents to autonomously perform inductive experiments, as described in section 3, and to exploit their discoveries. Table 5 lists the main internal actions implemented in java to be used in the plans of the library. The rest of the section describes the use of these plans by a learning agent.

Both classes of agents define a plan `@initialLearningGoal` to set the correct learning mode (intentional or singleMinded) by extending the user defined plans to deal with the learning process. For example, such extensions applied to the plan *put*, as defined for the agent listed in table 1, are shown in the table 6. The original body of the plan is at line 6. If this plan is adopted as an intention and correctly executed, then the agent believes (line 8) a new *succ*essful training *example* about *put*, including his beliefs at the time the plan was adopted.

Fun starts when facing problems: First, if the execution of an intention fails, for instance, because *move* could not be executed correctly, then a failure event is produced, e.g., $-!put(X, Y)$. JILDT adds the plan shown in table 7 to process these failure events, resulting in a $fail$ure training example added to the beliefs of the agent (line 4) and an inductive process intended to be achieved (line 5). The plan in table 8 deals with failures due to the lack of applicable plans. situation. It is rational to avoid commitment if there is no applicable plans for a given event.

The plan to achieve learning is shown in table 9. If the agent succeeds in computing a logical decision tree with the examples already collected, then he uses the tree to construct a new context for the associated plan (branches leading to

Table 5. Internal actions defined in the JILDT library.

Action	**Description**
`getCurrentBels(Bs)`	*Bs* unifies with the list of current beliefs of the agent.
`getCurrentCtxt(C)`	*C* unifies with the context of the current plan.
`getCurrentInt(I)`	*I* unifies with the current intention.
`getLearnedCtxt(P,LC,F)`	*LC* unifies with the learned context for plan *P*. *F* is true if a new different context has been learned.
`changeCtxt(P,LC)`	Changes the context of plan *P* for *LC*.
`setTilde(P)`	Builds the input files for learning about plan *P*.
`execTilde(T,G)`	Executes Tilde saving inputs and results. If *T* is *true*, a trace of Tilde algorithm is shown. If *G* is true, a visual tree is throwed.
`addDropRule(LC,P)`	Adds the rule to drop plan *P* accordingly to *LC*.
`setLearningMode`	Modifies plans to enable learning (intentionalLearner).
`setSMLearningMode`	Modifies plans to enable learning and dropping rules (singleMindedLearner class).

Table 6. JILDT extensions for plan *put* (original body at line 6).

```
1  @put
2  +!put(X,Y) : true <-
3    jildt.getCurrentInt(I);
4    jildt.getCurrentBels(Bs);
5    +intending(I,Bs);
6    move(X,Y);
7    -intending(I,Bs);
8    +example(I,Bs,succ);
```

success) and a set of rules for dropping the plan when it is appropriate (branches leading to failure). Two plans in the library are used to verify if something new has been learned.

There is a small ontology associated to the inductive processes. Table 10 lists the atomic formulae used with this purpose. These formulae should be treated as a set of reserved words.

The Tilde algorithm has been implemented as an action of JILDT. Logical consequence is computed using the Jason logical consequence method for logic formulae. Convergence of our implementation has been tested satisfactorily against ACE/Tilde [2] (same output for Bongard problems). The induced tree can be traced during induction and inspected in a graphic viewer (Figure 3).

5 Experiments

We have designed a very simple experiment to compare the behavior of a default Jason agent, an intentional learner, and single-minded agent that learns his

Table 7. A plan added by JILDT to deal with *put* failures requiring induction.

```
@put_failCase
-!put(X,Y) : intending(put(X,Y), Bs) <-
  -intending(I,Bs);
  +example(I,Bs,fail);
  !learning(put);
  +example_processed;
```

Table 8. A plan added by JILDT to deal with *put* being non applicable.

```
@put_failCase_NoRelevant
-!put(X,Y) : not .intend(put(X,Y)) <-
   .print("Plan ",put," non applicable.");
   +non_applicable(put).
```

policies for dropping intentions. For the sake of simplicity, these three agents are defined as shown in table 1, i.e., they are all bold about putting blocks somewhere else; and that is their unique competence.

The experiment runs as illustrated in figure 4: The *experimenter* asks the other agents to achieve putting the block b on c, but with certain probability $p(N)$, he introduces noise in the experiment by putting the block z on c, or on b. There is also a latency probability $p(L)$ for the last event: The *experimenter* could put block z before or after it asks the others agents to put b on c. This means that the other agents can perceive noise before or while intending to put b on c.

Numerical results are shown in table 11 (average of 10 runs, each one of 100 experiments) for a probability of latency of 50%. The probability of noise varies (90%, 70%, 50%, 30%, and 10%). Lower values configure less dynamic environments free of surprises and effectively observable. The performance of the agent is interpreted as more or less rational as follows: dropping an intention because of the occurrence of an error, is considered irrational. Refusing to form an intention because the plan is not applicable; dropping the intention because of a reason to believe it will fail; and achieving the goal of putting b on c are considered rational behaviors.

Figure 5 summarizes the result of all the executed experiments, where the probabilities of noise and latency range on $\{90\%, 70\%, 50\%, 30\%, 10\%\}$. As expected the performance of the *default* agent is proportionally inverse to the probability of noise, independently of the probability of latency.

The *learner* agent reduces the irrationality due to noise before the adoption of the plan as intention, because eventually he learns that in order to intend to put a block X on a block Y, X and Y must be clear:

Table 9. The learning plan

```
@learning
+!learning(P): true <-
   .print(\"Trying to learn a better context...\");
   jildt.setTilde(P);
   jildt.execTilde(false,false);
   jildt.getLearnedCtxt(P,LC,F);
   !learningTest(P,LC,F).
```

Table 10. A small ontology used by JILDT.

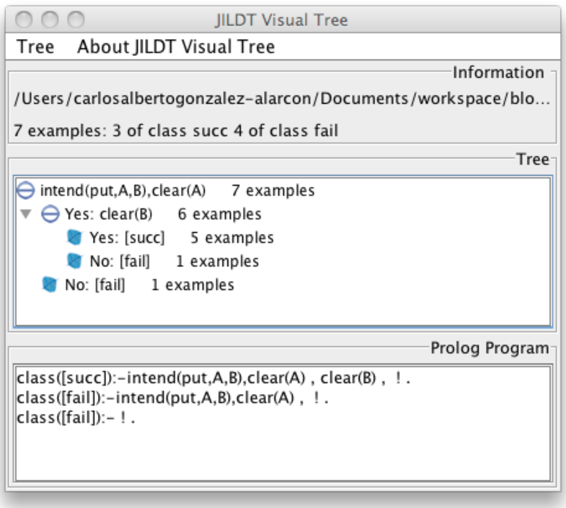
Atom	Description
`drop(I)`	*I* is an intention to be dropped. Head of dropping rules.
`root_path(R)`	*R* is the current root to Tilde experiments.
`current_path(P)`	*P* is the current path to Tilde experiments.
`dropped_int(I)`	The intention *I* has been dropped.
`example(P,Bs,Class)`	A training example for plan *P*, beliefs *Bs* and *Class*.
`intending(I, Bs)`	*I* is being intended yet. *Class* is still unknown.
`non_applicable(TE)`	There were no applicable plans for the trigger event *TE*.

Table 11. Experimental results (average from 10 runs of 100 iterations each one) for a probablity of latency of p(L)=0.5 and different probabilities of noise p(N).

Agent	p(N)	Irrational			Rational			
		after	before	total	refuse	drop	achieve	total
default	90	43.8	48.2	92.0	00.0	00.0	08.0	08.0
learner	90	48.7	37.3	86.0	04.5	00.0	09.5	14.0
singleMinded	90	44.5	38.8	83.3	03.2	03.8	09.7	16.7
default	70	34.5	36.0	70.5	00.0	00.0	29.5	29.5
learner	70	33.2	13.3	46.5	20.6	00.0	32.9	53.5
singleMinded	70	18.4	16.4	34.8	16.3	17.5	31.4	65.2
default	50	22.5	26.3	48.8	00.0	00.0	51.2	51.2
learner	50	26.1	05.4	31.5	20.7	00.0	47.8	68.5
singleMinded	50	11.6	09.9	21.5	16.1	14.9	47.5	78.5
default	30	14.2	15.0	29.2	00.0	00.0	70.8	70.8
learner	30	15.1	02.4	17.5	11.8	00.0	70.7	82.5
singleMinded	30	03.3	03.7	07.0	10.9	12.0	70.1	93.0
default	10	04.2	05.5	09.7	00.0	00.0	90.3	90.3
learner	10	05.3	01.0	06.3	04.9	00.0	88.8	93.7
singleMinded	10	00.9	00.9	01.8	03.8	03.4	91.0	98.2

```
put(X,Y) : (clear(X)  & clear(Y)) <- move(X,Y).
```

Then, the *learner* can refuse to intend putting *b* on *c* if he perceives *c* is not clear. So, for low latency probabilities, he performs better than the default agent, but of course his performance decays as the probability of latency increases; and, more importantly: there is nothing to do if he perceives noise after the intention



```
/Users/carlosalbertogonzalez-alarcon/Documents/workspace/blo...
7 examples: 3 of class succ 4 of class fail
```

```
intend(put,A,B),clear(A)    7 examples
  Yes: clear(B)    6 examples
    Yes: [succ]    5 examples
    No: [fail]    1 examples
  No: [fail]    1 examples
```

```
class([succ]):-intend(put,A,B),clear(A) , clear(B) ,  ! .
class([fail]):-intend(put,A,B),clear(A) ,  ! .
class([fail]):- ! .
```

Fig. 3. The viewer displaying a Logical Decision Tree for the blocks world.

has been adopted. In addition, the *singleMinded* agent learns the following rules for dropping the intention when blocks X and Y are not clear:

```
drop(put(X,Y)) :- .intend(put(X,Y)) & not(clear(X)).
drop(put(X,Y)) :- .intend(put(X,Y)) & not(clear(Y)).
```

Every time a $singleMindedLearner$ agent is going to execute an intention, he verifies that no reasons to drop the intention exist; otherwise the intention is dropped. So, when the *singleMinded* agent already intends to put b on c and the experimenter puts the block z on c, he rationally drops his intention. In fact, the *singleMinded* agent only fails when it is ready to execute the primitive action *move* and noise appears.

For high probabilities of both noise and latency, the chances of collecting contradictory training examples increases and the performance of the *learner* and *the singleMinded* agents decay. By contradictory examples we mean that for the same blocks configuration, examples can be labeled as success, but also as failure. This happens because the examples are based on the beliefs of the agent when the plan was adopted as an intention, so that the later occurrence of noise is not included.

In normal situations, an agent is expected to have different relevant plans for a given event. Refusing should then result in the adoption of a different relevant plan as a new intention. That is the true case of policy-based reconsideration, abandon is just an special case. Abandon is interpreted as rational behavior: the agent uses his learned policy-based reconsideration to prevent a real failure.

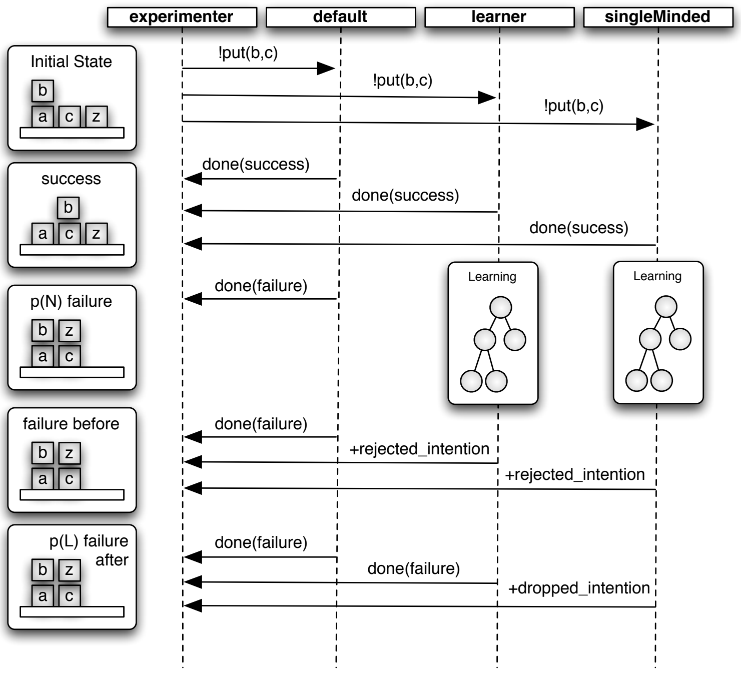


Fig. 4. The experiment process.

6 Discussion and future work

Experimental results are very promising. When compared with other experiments about commitment [8], it is observed that the *intentionalLearner* and *singleMinded* agents are adaptive: they were bold about *put*, and then they adopt a cautious strategy after having problems with their plan. Using intentional learning provides convergence to the right level of boldness-cautiousness based on their experience. And also, it seems that a bold attitude is adopted toward successful plans, and a cautious one toward failed plans; but more experiments are required to confirm this hypothesis.

The JILDT library provides the extensions to *AgentSpeak*(L) required for defining intentional learning agents. Using the library, it was easy to implement a single-mind committed class of agents. We obtained a better understanding of the inductive method, that will enable us to improve it. For instance, it is possible to use the initial query of TILDE to avoid the use of lookaheads, reducing the number of candidates while computing the logical decision tree. Experimental results suggest that induction could be enhanced if the training examples represent not only the beliefs of the agent when the intention was adopted, but also when it was accomplished or dropped, in order to minimize the effects of the latency in noise. Also, incremental versions of the inductive method are now envisioned, as well as social learning protocols [5] exploiting distributed knowledge.

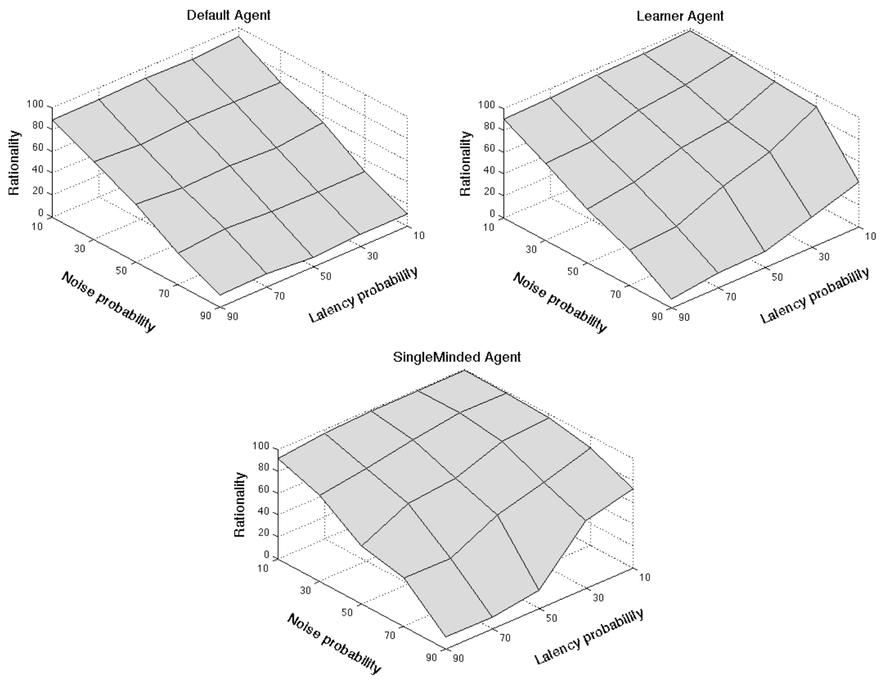


Fig. 5. The experiment results. Left: *Default* performance. Right: *Learner* performance. Center: *SingleMinded* performance

The transition system (Figure 1) for the *singleMinded* agents has been modified to enable dropping intentions. Basically, every time the system is at *execInt* and a drop learned rule fires, the intention is dropped instead of being executed. It is possible now to think of a formal operational semantics for *AgentSpeak*(*L*) commitment based on policy-based reconsideration and intentional learning.

In [13] an architecture for intentional learning is proposed. Their use of the term intentional learning is slightly different, meaning that learning was the goal of the BDI agents rather than an incidental outcome. Our use of the term is strictly circumscribed to the practical rationality theory [4] where plans are predefined and the target of the learning processes is the BDI reasons to adopt them as intentions. A similar goal is present in [9], where agents can be seen as learning the selection function for applicable plans. The main difference with our work is that they propose an *ad hoc* solution for a given non BDI agent. Our approach to single-mind commitment evidences the benefits of generalizing intentional learning as an extension for Jason.

Acknowledgments. Authors are supported by Conacyt CB-2007 fundings for project 78910. The second author is also supported by scholarship 273098.

References

1. Blockeel, H., De Raedt, L.: Top-down induction of first-order logical decision trees. Artificial Intelligence, 101(1–2):285–297 (1998)
2. Blockeel, H., Raedt, L., Jacobs, N., Demoen, B.: Scaling up inductive logic programming by learning from interpretations. Data Mining and Knowledge Discovery, 3(1):59–93 (1999)
3. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. Wiley, England (2007)
4. Bratman, M.: Intention, Plans, and Practical Reason. Harvard University Press, Cambridge (1987)
5. A. Guerra-Hernández, A. El-Fallah-Seghrouchni, and H. Soldano. Distributed learning in BDI Multiagent Systems. In R. Baeza-Yates, M. J.L., and E. Chávez, editors, *Fifth Mexican International Conference on Computer Science*, pages 225–232, USA, 2004. Sociedad Mexicana de Ciencias de la Computación (SMCC), IEEE Computer Society.
6. Guerra-Hernández, A., Ortíz-Hernández, G.: Toward BDI sapient agents: Learning intentionally. In: Mayorga, R.V., Perlovsky, L.I. (eds.) Toward Artificial Sapience: Principles and Methods for Wise Systems, pp. 77–91. Springer, London (2008)
7. Guerra-Hernández, A., Castro-Manzano, J. M., El Fallah Seghrouchni, A.: CTL AgentSpeak(L): a Specification Language for Agent Programs. Journal of Algorithms, (64):31–40 (2009)
8. Kinny, D., Georgeff, M. P.: Commitment and effectiveness of situated agents. In Proceeding of the Twelfth International Conference on Artificial Intelligence IJCAI-91, pp. 82–88, Sidney, Australia (1991)
9. Nowaczyk, S., Malec, J.: Inductive Logic Programming Algorithm for Estimating Quality of Partial Plans. In MICAI 2007, LNAI, vol. 4827, pp. 359–369 Springer Verlag, Heidelberg (2007)
10. Rao, A.S., Georgeff, M.P.: Modelling Rational Agents within a BDI-Architecture. In: Huhns, M.N., Singh, M.P., (eds.) Readings in Agents, pp. 317–328. Morgan Kaufmann (1991)
11. Rao, A.S., Georgeff, M.P.: Decision procedures for BDI logics. Journal of Logic and Computation 8(3), pp. 293–342 (1998)
12. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: de Velde, W.V., Perram, J.W. (eds.) MAAMAW. LNCS, vol. 1038, pp. 42–55. Springer Verlag, Heidelberg (1996)
13. Subagdja, B., Sonennberg, L., Rahwan, I.: Intentional learning agent architecture. Autonomous Agents and Multi-Agent Systems, 18:417–470 (2008)