Journal of Computer Science 3 (11): 898-906, 2007
ISSN 1549-3636

Structure and Performance of the Meta Inference Engine

Jean-Marc NIGRO and Yann BARLOY
Institut Charles Delaunay (FRE CNRS 2848), Universite de Technologie de Troyes,
Troyes, France

Abstract: This study presents the structure and the performance of a new inference engine: the MIE (Meta Inference Engine). It is able to manipulate not only the rules but also the metarules. The article first describes the architecture of the MIE and gives an example to illustrate its use. A comparison of performance between an RETE network and the MIE is then made. This shows that the MIE is more efficient at manipulating metaknowledge (metarules) but that an RETE network is quicker when the system inserts or deletes a fact.

Key words: Metaknowledge, artificial intelligence, inference engine, OPS and RETE network, meta-rule

INTRODUCTION

The domain of metaknowledge was conceived in the 1970s and 80s[10] at the same time as the emergence of rule-based systems. A metaknowledge can be defined as being knowledge about knowledge. Different classes of metaknowledge were established by Jacques Pitrat[20] metaknowledge for acquiring, for explaining, for using or for stocking knowledge.

The study and manipulation of metaknowledge are transverse in several domains in Artificial Intelligence (AI). They are often used to model many levels of decisions or structures such as meta-systems developed in LCF[7] and FOL[24]. Metaknowledge was evoked for decision support[18], learning[1,16], monitoring[11], generation of comments[17], manipulation of the temporal graph[9], strategy of a inference engine[24], problem solving in geometry[2], checking on the coherence of a rules-based systems[21] and the discovery of new knowledge[14].

There is no specific architecture (or programming language) for manipulating metaknowledge: MALICE[19] metaknowledge is written in language C; those of GénéCom[17] and SNARK[13] with rules and those of SADE[11] with Lisp. However, rules-based systems have the advantage of catering for the building of different levels of knowledge[9] a chronology to execute packages of rules for example. The other advantage of the rules is they allow the developer to be focused on the transcription of methods in the form of rules without being concerned about their triggering. An inference engine takes responsibility for the matching of conditions and the execution of rules.

It is in this context that the idea to conceive a new inference engine, called MIE (Meta-Inference Engine) appeared. This idea had already been introduced in a theoretical way by Clancey[5] or Torsun[22] who had developed a logic allowing the programming of a "meta" level by using a language such as PROLOG. The development of a tool allowing the manipulation of metaknowledge[20,12] would facilitate the implementation of AI systems.

So, the MIE allows the developer to build systems based on rules and meta-rules (a meta-rule is executed as a rule). Contrary to most systems which use meta-rules[3,21] in a static way, the triggering of meta-rules can dynamically modify the rules structure during the session. Also, reflective systems can be made by the MIE.

Two types of inference engines are known: those based on a filter algorithm and those using an RETE network[8]. Various models were developed according to the principle of an RETE network. One of the differences between all these models comes from the memorization of facts in the nodes of the network. SNARK[13] and TREAT[15,23] use fewer memorization techniques whereas OPS[3] and TANGO[6] memorise all the facts and sets of facts (representing instances).

The use of an RETE network is much more faster than a filtering technique. On the other hand, it has a physical structure which is more complex: various types of nodes and joints. If this implementation had been chosen, every execution of a meta-action (for example to modify or to delete a rule) would have entailed, every time, an expensive reorganization of the RETE network. The idea is to imagine a new structure

Corresponding Author: Jean-Marc NIGRO, Institut Charles Delaunay (FRE CNRS 2848), Université de Technologie de Troyes, Troyes, France

which would be a compromise between RETE and filtering. This new structure would be faster than the filtering solution and more flexible than the RETE network. This flexibility would make the use of metaknowledge easier when a rule needs to be created, modified or deleted during an execution.

The goal of this article is to present the structure used by the MIE and to compare its performances with an RETE network.

In the following section, the structure of the meta-inference engine is presented. Several algorithms are described to explain how the performances of the MIE and an RETE structure have been calculated to insert and to delete a fact, a condition or a rule. The last part shows the result of the comparison with graphs and commentaries.

STRUCTURE OF THE META INFERENCE ENGINE

The structure shown on Fig. 1 is used to encode a set of rules. Each encoded rule contains three lists: the list of conditions, the list of actions and the list of variables. The latter is composed of a set of variables used in the conditions of the rules. This is useful when the MIE has to create a new instance. The list of actions contains the sequence of actions which will be executed if the information encoded in an instance satisfies all the conditions of a rule.

Each condition is characterized by a name, an object, an attribute, a value, a list of facts which satisfy the condition, a list of instances (see below the explanation of its use), and a complexity value which evaluates the matching calculation between the condition and a fact. The list of conditions is sorted from the simplest to the most complex with the use of Table 1 and 2. For example, the condition (cube1 = color !color) will have the compexity 2 while the condition (!cube > height (+ !var 10)) (the condition will be satisfied if the cube !cube has a height greater than the value of the variable !val incremented with 10) will have the complexity 12.

This sorted list of conditions allows for the progressive building of instances as the conditions are satisfied. The idea is to insert in the first places the conditions which give rise to few calculations and which determine the values of the variables. The complex conditions (the costliest) will be computerized at the very end.

A condition contains a list of facts and a list of instances. Each fact (from the list of facts) is matched with the condition and is used to create or modify an instance (from the list of instances). The list of instances possesses partial instances which have been constructed from the first condition (the less complex) to the condition being tested. The last condition of the rule (the most complex) contains the complete instances usable to execute the rule.

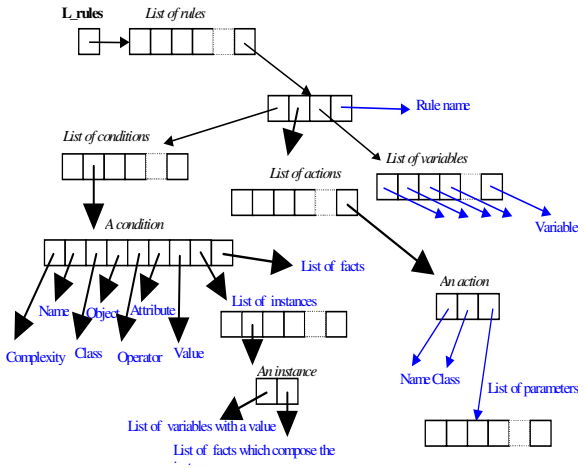

Fig. 1: The structure used by the MIE

Table 1: The complexity of a condition with the operator =

Operator =		Object	
		Variable	Not Variable
Value	Variable	3	2
	Not Variable	1	0
	Variable in an expression	5	4
	Variable not in an expression	1	0

Table 2: The complexity of a condition with an operator different from =

Operator different from		Object	
		Variable	Not Variable
value	Variable	9	6
	Not Variable	3	0
	Variable in an expression	15	12
	Variable not in an expression	3	0

For the condition at position i, an instance is composed of a list of facts and a list of variables which allows for matching all the conditions from position 1 to position i and which allocates a value to all the variables of all the conditions from position 1 to position i. To be more explicit, an example is described below.

An example to illustrate the use of the MIE structure: Supposing that the rule written by the developer (respecting the syntax described in annexe A1) is the example1 rule, the first task of the MIE is to calculate the complexity of each condition and to sort them Table 3.

Table 3: Calculation of the complexity and ascending sort of the condition condition

Condition	Complexity
(!cube1 = color red)	1
(!cube2 = below table)	1
(!cube1 = below !cube2)	3
(!cube1 = height !val)	3
(!cube2 > height !val)	9

```
(Defrule example1
      (!cube1 = color red)
      (!cube1 = below !cube2)
      (!cube1 = height !val)
      (!cube2 = below table)
      (!cube2 > height !val)
=>
      (assert agregate shape pyramid)
)
```

Once the conditions have been sorted into ascending order, the MIE can build the structure which allows the facts to match the conditions. Now, if the three facts (CubeA below CubeB), (CubeB below CubeC), (CubeC below table) are sent, the two first facts match the third condition (they are stored in the list of facts) and the last fact matches the second condition.

		List of instances	
Condition	**List of facts**	**List of variables**	**List of facts**
(!cube1 = color red)			
(!cube2 = below table)	(CubeC below table)		
(!cube1 = below !cube2)	(CubeA below CubeB) (CubeB below CubeC)		
(!cube1 = height !val)			
(!cube2 > height !val)			

Let us now suppose that the two following facts (CubeA color red) and (CubeB color red) are sent to the structure. These two facts match the first condition and they are stored in the list of facts. Two partial instances are built in the first condition and they are propagated in the second condition. Two partial instances are also built and they are propagated in the third condition. But only one partial instance is built to respect the link between the variables !cube1 member of condition 1 and 3 and !cube2 member of condition 2 and 3.

		List of instances	
Condition	List of facts	List of variables	List of facts
(!cube1 = color red)	(CubeA color red) (CubeB color red)	1:(!cube1 = CubeA) 2:(!cube1 = CubeB)	1:(CubeA color red) 2:(CubeB color red)
(!cube2 = below table)	(CubeC below table)	1: (!cube1 = CubeA ; !cube2 = CubeC) 2: (!cube1 = CubeB ; !cube2 = CubeC)	1: (CubeA color red) ; (CubeC below table) 2: (CubeB color red) ; (CubeC below table)
(!cube1 = below !cube2)	(CubeA below CubeB) (CubeB below CubeC)	2: (!cube1 = CubeB ; !cube2 = CubeC)	2: (CubeB color red) ; (CubeC below table) ; (CubeB below CubeC)
(!cube1 = height !val)			
(!cube2 > height !val)			

If we go on to send the fact (CubeC height 40). It matches the two last conditions but no new partial condition are created because the variable !cube1 has to have the value CubeB and not CubeC. So, the fourth condition does not build an instance.

		List of instances	
Condition	List of facts	List of variables	List of facts
(!cube1 = color red)	(CubeA color red) (CubeB color red)	1:(!cube1 = CubeA) 2:(!cube1 = CubeB)	1:(CubeA color red) 2:(CubeB color red)
(!cube2 = below table)	(CubeC below table)	1: (!cube1= CubeA ;	1: (CubeA color red) ; (CubeC below table) 2:

		!cube2= CubeC) 2: (!cube = CubeB ; !cube2= CubeC)	(CubeB color red) ; (CubeC below table)
(!cube1 = below !cube2)	(CubeA below CubeB) (CubeB below CubeC)	2: (!cube1 = CubeB ; !cube2 = CubeC)	2: (CubeB color red) ; (CubeC below table) ; (CubeB below CubeC)
(!cube1 = height !val)	(CubeC height 40)		
(!cube2 > height !val)	(CubeC height 40)		

Finally we can send the fact (CubeB height 20). It matches the two last conditions. The fourth condition can create a partial instance because !cube1 (object of the condition) matches with the value CubeB. This instance is propagated in the last condition which can create a complete instance.

		List of instances	
Conditio n	List of facts	List of variables	List of facts
(!cube1 = color red)	(CubeA color red) (CubeB color red)	1:(!cube1 = CubeA) 2:(!cube1 = CubeB)	1:(CubeA color red) 2:(CubeB color red)
(!cube2 = below table)	(CubeC below table)	1: (!cube1= CubeA ; !cube2= CubeC) 2: (!cube1= CubeB ; !cube2 = CubeC)	1: (CubeA color red) ; (CubeC below table) 2: (CubeB color red) ; (CubeC below table)
(!cube1 = below !cube2)	(CubeA below CubeB) (CubeB below CubeC)	2: (!cube1= CubeB ; !cube2= CubeC)	2: (CubeB color red) ; (CubeC below table) ; (CubeB below CubeC)
(!cube1 = height !val)	(CubeC height 40) (CubeB height 20)	2: (!cube1= CubeB ; !cube2=	2: (CubeB color red) ; (CubeC below table) ; (CubeB below

		CubeC ; !val= 20)	CubeC) ; (CubeB height 20)
(!cube2 > height !val)	(CubeC height 40) (CubeB height 20)	2: (!cube1= CubeB ; !cube2 = CubeC ; !val= 20)	2: (CubeB color red) ; (CubeC below table) ; (CubeB below CubeC) ; (CubeB height 20) ; (CubeC height 40)

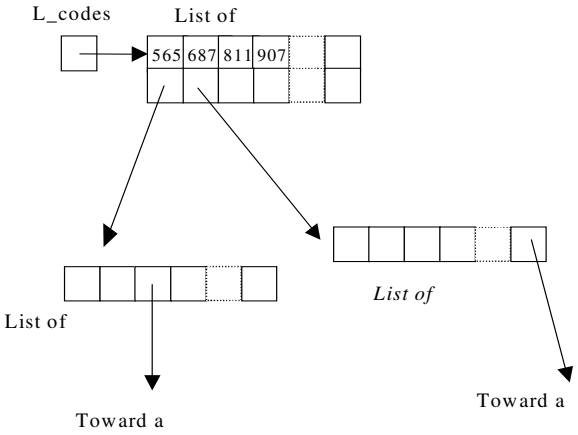

Fig. 2: Structure to route the facts to the condition

The insertion of a new fact uses a special structure which allows us to ignore uninteresting conditions, rather than testing all the conditions of all the rules. This structure offers a quick propagation of facts for all the conditions Fig. 2. The MIE works with a first order logic: all conditions have an attribute which is not a variable. To have a match between a fact and a condition, they (the fact and the condition) must have the same attribute. The idea is to encode the attributes with the conditions and to regroup them in an ordered list.

Each code reaches a set of conditions having the same attribute zone. When the MIE has to look for a new fact in the structure, it calculates the code of the attribute (of the fact) and seeks it out in the list of codes. If the code is found then the MIE can test all the conditions linked to this code with the fact.

DEFINITION AND ALGORITHMS

The aim of this section is to show how the comparison of performance between the MIE structure and the RETE network has been done. First, some

variables are defined which are used in several algorithms: insert a fact, delete a fact, insert a condition, delete a condition, insert a rule and delete a rule.

DEFINITION OF VARIABLES

Number of facts : nb_facts
Number of rules : nb_rules
Number of conditions in a rule : nb_conditions
Number of original conditions in a rule : nb_original
Number of common conditions in a rule : nb_common = nb_conditions - nb_original

If we suppose that a common condition appears on average in 1/3 of the rules :
The total number of conditions :

total_nb_conditions = nb_rules * nb_original + (nb_rules/ 3)*(nb_conditions - nbr_original)
total_nb_conditions = (nbr_règles/ 3)*(nbr_conditions + 2*nbr_originals)

The number of copies of a common condition *(see annexe A2)*
nb_copies_common_condition=(3*nb_original+nb_common*nb_rules)/(3*nb_conditions)

We will assume that the facts are equitably distributed in test nodes :
nb_facts_in_a_test_node = nb_facts / total_nb_conditions.

The number of outputs for a joint node :
nb_outputs_joint = log (total_nb_conditions - nb_common)

The number of outputs for a test node (see annexe A2)
nb_outputs_test = (3* nb_original + nb_common * nb_rules) / (3*nb_conditions)

The coefficient of matching in a joint node between 2 facts or between 1 fact and 1 instance: Coef_matching = 20%

The number of instances in a joint node:
nb_instances_joint_node = $(Uo(1-r^{n+1})/(1-r))/n$
with Uo=nb_facts/ total_nb_conditions;
r= nbr_facts/ total_nb_conditions * coef_matching;
n=nb_conditions-1

The number of instances in a condition in the MIE strucure (an average) :

nb_instances_condition_mie = (nb_facts/ total_nb_conditions) * Coef_matching

To simplify counting, we assume that a fact matches only one test node.

ALGORITHMS

Four algorithms are presented below: Insert_a_fact_OPS, Insert_a_fact_MIE, Insert_a_condition_OPS and Insert_a_condition_MIE. The other algorithms are described in annexe A3.

Figure 3 proposes the propagation of a fact in an RETE network. This network has 5 test nodes: the first one contains 1 fact, the second and the fourth contain 3 facts and the fifth 4 facts. The first joint node has 2 partial instances. When a new fact is sent in the RETE network, it is tested with all test nodes. In figure 3, the new fact satisfies the matching fonction of the third test node. It is memorised and it is propagated in the joint node B. Then, this node verifies if a new partial instance can be made from the partial instance of the node A and the new fact. In the example, two new partial instances are created and stored in node B. They are propagated towards node C which verifies if new partial instances can be built from partial instances of node B and the facts of the test node 4. So, three new partial instances are created and stored in node C. The same comparison is done with node D which builds 4 instances from node C and node 5. Then, these 4 instances are proposed to the node rule which can execute them.
Algorithm Insert_a_fact_OPS

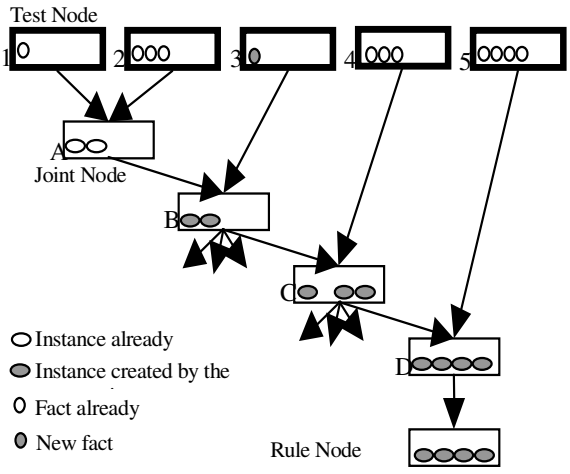

Fig. 3: The insertion of a fact in OPS structure

```
Begin
  Levels_Counter ← 1

  // Test for the first joint node
  Instructions_Counter  ←+ nb_instances_joint_node *
1 + 1

  // Test for the storage of the first joint node
  Instructions_Counter ←+ nb_instances_joint_node * 1
* Coef_matching + 1

  Un ← nb_instances_joint_node *1 *Coef_matching

  Loop (nb_conditions div 2) - 1 times
    // Test for a joint node
    Instructions_Counter  ←+ Un * Levels_Counter  *
(Nb_facts/ total_nb_conditions) + 1

    // Test for the storage of a joint node
    Instructions_Counter  ← + Un * Levels_Counter  *
(Nb_facts/ total_nb_conditions) * Coef_matching + 1

    Levels_Counter  ←+ 1

    Un   ← Un * (nb_facts/ total_nb_conditions) *
Coef_matching
  End Loop

  Return (Instructions_Counter)
End
```

The first step of the insertion of a fact in the RETE network consists in testing the fact with all the test nodes. This is not taken into account because its number of instructions is insignificant in comparison with the other steps. The Insert_a_fact_OPS algorithm begins with the test and the storage of the first joint node. Secondly, a loop used to count the instructions of the propagation of partial instances in all joint nodes of the RETE network.

Figure 4 shows the insertion of a fact in the MIE structure. The number of instructions for the calculation of the code and the route of the new fact to the node 3 are considered as insignificant in comparison to the total number of instructions. The first step is to test if the new fact can be memorized in the node. Then, the MIE has to try to combine it with the instances of the previous node to build the new instances. In the example, 3 new instances are created in node 3. The MIE tries to propagate these instances in the following node by matching up the fact of node 4.

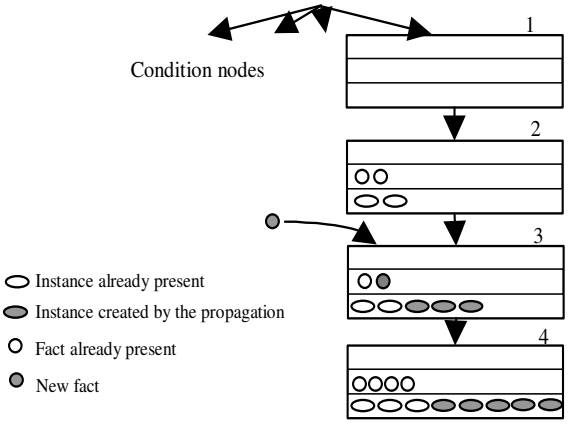

Fig 4: The insertion of a fact in the MIE structure

```
Algorithm Insert_a_fact_MIE
Begin

  Loop nb_copies_common_condition times

    // Test for the first condition
    Instructions_Counter                          ←+
nb_instances_condition_mie * 1 + 1

    // Test for the storage of the first condition
    Instructions_Counter                          ←+
nb_instances_condition_mie * 1 * Coef_matching + 1

    Un      ← nb_instances_condition_mie  *  1  *
Coef_matching

    Loop (nb_conditions div 2) - 1 times
      // Test for a condition
      Instructions_Counter  ←+  Un  *  (nb_facts/
total_nb_conditions) + 1

      // Test for the storage of the first condition
      Instructions_Counter  ←+  Un  *  nb_instances
_condition_mie + 1

      Un ← Un * nb_instances_condition_mie
    End Loop

  End Loop

  Return (Instructions_Counter)
End
```

The Insert_a_fact_MIE algorithm has 2 overlapping loops. The first one consists in testing the

fact with all conditions which have the same attribute. For each condition, the algorithm counts the test, the memorization of the fact and the creation of new instances. Then, it propagates these instances at the end of the list of conditions.

Algorithm Insert_a_condition_OPS
Begin
// Assumption that the condition is a new condition:

- Search to see if the condition already exists = total_nb_conditions
- Search for the links (object or attribute) with other conditions of the rule = nb_conditions *2
- Creation of test and joint nodes = 2
- Creation of links = 3
- Delete instances in rule node = nb_instances_joint_node
- Propagation of facts = insert_a_fact_ops * nb_facts
- Delete instances in the joint node downstream = nb_conditions / 2 * nb_instances_joint_node
- Search for the place where they are going to insert the condition = the insertion of an element in a sorted list = Ln (nb_conditions+1);
- Update functions of validation in all nodes downstream = (Uo(1-rn+1)/(1-r))/n with Uo=10; r= nb_outputs_joint, n=nb_conditions / 2.

End;

The insertion of a condition in an RETE network implies the creation of new nodes and links, the modification of the several functions of validation (which belong to several nodes), and the propagation of facts (and instances) in the new nodes (and the nodes downstream)

Algorithm Insert_a_condition_MIE
Begin
- Creation of the condition = 1
- Counting of the condition complexity = 1
- Insertion of the test node in the rule = nb_conditions/2
- Delete instances in nodes which are downstream of the condition = (nb_conditions / 2) * nb_instances_condition_mie
- Propagation of facts: insert_a_fact_MIE * (nb_facts / total_nb_conditions)
- Propagation of metafacts:

a. Insertion of metafacts representing the condition = 6 * insert_a_fact_MIE
b. Modification of metafacts representing the rule = 2 * insert_a_fact_MIE + 2 * delete_a_fact_MIE
End,

The insertion of a condition in an MIE implies the creation of a node (a condition) and a link, the propagation of the facts from the new node (the new condition) to the last node of the rule, and the managing of metafacts to taking into account this insertion.

RESULTS AND ANALYSE

All algorithms are tested in 3 different situations:

- the number of facts increases and the number of rules is constant
- the number of rules increases and the number of facts is constant
- the number of facts and the number of rules increase simultaneously

Figures 5, 6 and 7 concern the execution of the algorithms in order to insert and to delete a fact in an RETE network and the MIE structure. The figures show that for the three cases, the performance of the RETE network is clearly better than the MIE.

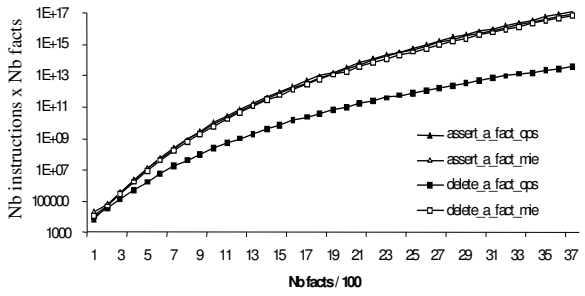

Fig. 5: Performance of the insertion and the deletion of a fact with a number of fact variations

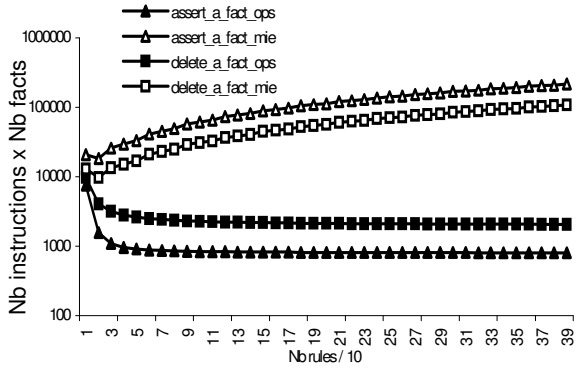

Fig. 6: Performance of the insertion and the deletion of a fact with a number of rule variations

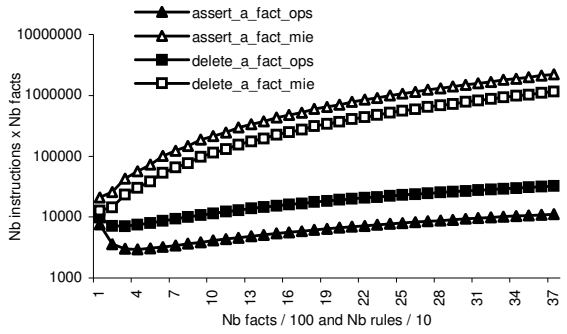

Fig. 7: Performance of the insertion and the deletion of a fact with a number of fact and rule variations

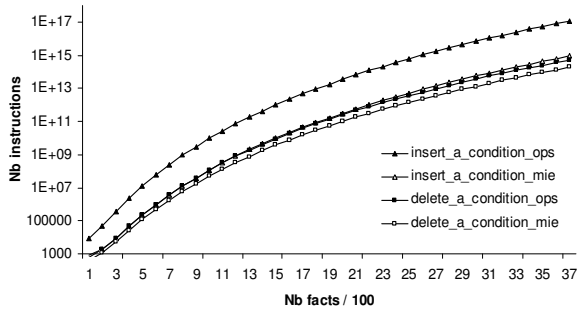

Fig. 8: Performance of the insertion and the deletion of a condition with a number of fact

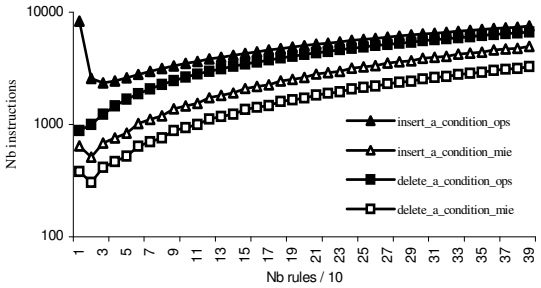

Fig. 9: Performance of the insertion and the deletion of a condition with a number of rule variations

Figures 8, 9 and 10 concern the insertion and the deletion of a condition in an RETE network and MIE structure. The figures show that for the three cases, the performance of the MIE is clearly better than the RETE network. The annexe A4 concerning the insertion and deletion of a rule gives the same result.

These comparisons show that the MIE has a better performance when the system is needed to work at a metalevel (with metarules). But a classic inference engine is preferable for executing simple rules.

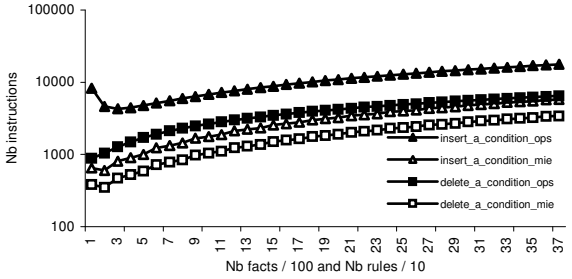

Fig. 10: Performance of the insertion and the deletion of a condition with a number of fact and rule variations

CONCLUSION AND PERSPECTIVES

The version of meta inference engine presented in this paper is the first one which has been found to be usable. The first results and the comparison of performances show that the MIE gives a better performance than a classic inference engine in the case of the use of metaknowledge even if it is slower to insert and delete a fact. Nevertheless, it is still experimental and incomplete: it does not take into account negative conditions and it needs a dedicated environment which will allow for the easy application of learning techniques based on meta rules. For example, some modules, which manipulate execution traces, can be developed. Many classes of traces can be imagined: a short term trace, a middle term trace and a long term trace. The short term trace will contain all the executed rules and possible instances. The middle term trace will synthesize the short term trace by counting the number of executions for each rule, or the number of times when a rule would be executed. The long term trace will regroup statistics and evaluate the performance of a rule or a package of rules. All these traces will allow the MIE to learn techniques during the running time (with short term and middle term traces) and after the execution (with the long term trace).

ACKNOWLEDGEMENTS

Research supported in part by Champagne-Ardenne Regional Council (district grant) and the European Social Fund.

REFERENCES

1. Anderson, M., T. Oates and D. Perlis, 2006. ReGiKAT: (Meta-)Reason-Guided Knowledge Acquisition and Transfert-or-Why Deep Blue Can't Play Checkers and Why Today's Smart Systems Aren't Smart. IPMU, 4-7 july.

2. Bazin, J.M., P. Castells, R. Moriyon and F. Saiz, Kiev 1993. Aknowledge based problem solver conceived for intelligent tutoring application, ICCTE 93.
3. Brownston, L., R. Farrel, E. Kant and N. Martin, 1985. Programming Expert Systems in OPS5: An Introduction to Rule Base Programming. Addison-Wesley.
4. Clancey, W.J., 1992. Model construction operators. Artificial Intelligence 53: 1-115.
5. Cazenave, T., 2003. Metarules to Improve Tactical Go Knowlegde. Information Sciences, 154 (3-4) : 173-188.
6. Cordier, M.O. and M.C. Rousset, 1984. Interactive operators in expert systems. European Conference on Artificial Intelligence, ECAI 84, Pise.
7. Cohn, A., 1979. High level proof in LCF. In WAD, pp: 73-80.
8. Forgy, C.L.,1982. A Fast Algorithme for the Many Pattern/Many Object Pattern Match Problem. Artificial Intel., 19.
9. Genesereth, M.R. and N.J. Nilsson, 1987. Logical Fundations of Artificial Intel., Los Altos, CA Morgan Kaufmann.
10. Hayes, P.J., 1973. Computation and deduction. Proceeding 2nd. Symposium on Mathematical Foundations of Computer Science, Czechoslovakian Academy of Sciences, 105-118.
11. Kornman, S., 1995. A Meta-level Architecture for self-monitoring. workshop IJCAI'95, On Reflection and Meta-Level Architecture and their Applications in AI, 125-131.
12. Laurière, J.L., 1986. Snark : A langage to represent declarative knowledge and an inference engine which uses heuristics, Information Processing 86, Ed Elsevier Science Publisher, pp: 811-816.
13. Laurière, J.L. and M. Vialatte, 1986. Snark a language to represent declarative knowledge. IFIP, North-Holland.
14. Lenat, D., 1984. Why AM and EURISKO appear to work. Artificial Intelligence. 23 (3): 269-294.
15. Miranker, D.P., 1987. TREAT: A better match algorithm for ai production systems. National Conference on Artificial Intelligence. Seattle.
16. Nigro, J.M. and Y. Barloy, 2006. The Meta Inferences Engine: A tool to use metaknowledge, IPMU 06, Paris, July.
17. Nigro, J.M., 1995. GeneCom: A program which uses meta concepts, Workshop IJCAI 95, pp: 132-139.
18. Pinson, S., 1989. Credit Risk Assessment and meta-judgement, Theory and decision, 27 : 117-133.
19. Pitrat, J.,2006. Meta-explanation in a Constraint Satisfaction solver. IPMU 2006, 4-7 july.
20. Pitrat, J., 1990. An intelligent system must and can observe its own behaviour. Cognitiva 90.
21. Spreeuwenberg, S., R. Gerrits and M. Boekenoogen, 2000. VALENS: A Knowledge Based Tool to Validate and Verify an Aion Knowledge Base. ECAI 2000, pp: 731- 735.
22. Torsun, I.S., 1995. Foundations of Intelligent Knowledge-Based Systems. Academic Press.
23. Wang, Y.W.,1992. EN. Hanson. A performance Comparison of the RETE and TREAT Algorithme for Testing Database Rule Conditions. Eighth International Conference on Data Engineering. Tempe (USA).
24. Weyhrauch, R., 1980. Prolegomena to a theory of mechanized formal reasoning. Artificial Intelligence, 13 (1,2): 133-170.