

٩٧ / ١٤
٢٢ / ٢٥

A STUDY OF THE EFFICIENCY OF TMS-BASED PRODUCTION RULE SYSTEMS

by
Khalid Waleed Mahmoud

Supervisor

Assoc. Prof. Riad Jabri

Co-Supervisor

Asst. Prof. Khalil el Hindi


عميد كلية الدراسات العليا

Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science in
Computer Science

Faculty of Graduate Studies
University of Jordan

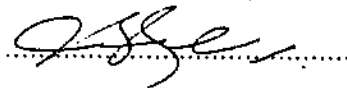
December 1997

This thesis was successfully defended and approved on 8 / 12 / 1997

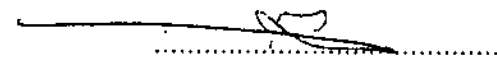
Examination Committee

Signature

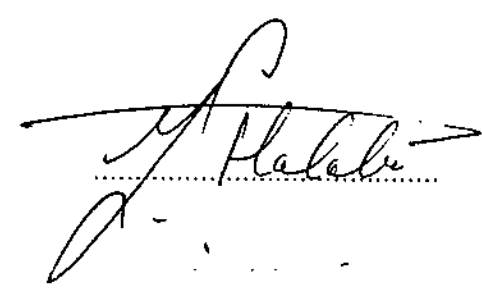
Dr. Riad Jabri, Chairman
Assoc. Prof. of Compiler



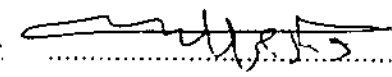
Dr. Khalil el Hindi, Member
Asst. Prof. of Artificial Intelligence



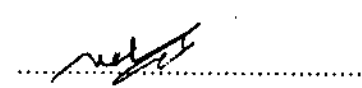
Dr. Yahia Halabi, Member
Prof. of Numerical Analysis



Dr. Ahmad AL-Jaber, Member
Assoc. Prof. of Algorithm



Dr. Arafat Awajan, Member
Asst. Prof. of Artificial Intelligence



Acknowledgement

I am gratefully acknowledge the guidance, and help of my supervisor Assoc. Prof. Dr. Riad Jabri, during the preparation of this work.

I would like to express my deepest thanks to my co.-supervisor Asst. Prof. Dr. Khalil el Hindi, for his guidance throughout the study.

Finally, I would like to thank the staff members and postgraduate students in the department of computer science, for the relaxed and friendly atmosphere they have provided.

Contents

| | |
|--|-----------|
| Acknowledgement | iii |
| List of Contents | iv |
| List of Appendices | vi |
| List of Tables | vii |
| List of Figures | ix |
| Abstract | xi |
| Chapter 1: Introduction | 1 |
| Chapter 2: Background | 7 |
| 2.1 The ATMS | 7 |
| 2.2 Production Rule Systems | 10 |
| 2.3 The Rete Algorithm | 12 |
| 2.4 Integrating Production Systems and Truth Maintenance Systems | 15 |
| 2.4.1 Loose coupling the ATMS and production system. | 16 |
| 2.4.2 Tight coupling of ATMS and production system : the Morgue system | 18 |
| 2.4.2.1 Drawbacks of the Morgue system | 22 |
| 2.4.3 Tight coupling of ATMS and production system : The Hindi system. | 24 |
| Chapter 3: Design and Implementation | 27 |
| 3.1 Assertions, and Rule design | 27 |
| 3.1.1 Assertions | 27 |
| 3.1.2 Rules | 28 |
| 3.1.2.1 Variable Restrictions | 29 |
| 3.2 The Rete Algorithm | 31 |

| | |
|--|-----------|
| 3.2.1 The match Process | 31 |
| 3.2.2 Structures | 32 |
| 3.2.3 The Rete Building Algorithm..... | 38 |
| 3.2.4 The Reasoning Process | 44 |
| 3.3 ATMS : Structures and Algorithms | 50 |
| 3.3.1 Structures..... | 50 |
| 3.3.2 Label update and Nogood handling | 52 |
| 3.4 Loose coupling of ATMS and production system | 54 |
| 3.5 Tight coupling the ATMS and production system: Morgue System. | 57 |
| 3.6 Tight coupling the ATMS and production system: the Hindi System. ... | 65 |
| Chapter 4: Performance Evaluation of the Algorithms | 74 |
| 4.1 The Queen problem..... | 74 |
| 4.1.1 Rules | 75 |
| 4.1.2 Results and Conclusions | 77 |
| 4.1.3 Different versions of the queen problem..... | 78 |
| 4.2 A Constraint Satisfaction Problem | 80 |
| 4.2.1 Rules | 80 |
| 4.2.2 Results and Conclusions..... | 82 |
| 4.3 Student Registration system-1 : a typical Database system | 85 |
| 4.3.1 Rules | 86 |
| 4.3.2 Results and Conclusions | 90 |
| 4.4 Student Registration system-2: a typical Database application | 91 |
| 4.4.1 Results and Conclusions | 92 |
| 4.4.2 Different data for the registration problem | 92 |

| | |
|------------------------------------|----|
| Chapter 5:Conclusions | 94 |
| 5.1 Conclusions..... | 94 |
| 5.2 Suggested Future Work | 95 |
| References | 97 |

Appendices

| | |
|---|-----|
| Appendix 1 : Rete Program..... | 99 |
| Appendix 2 : Loose Coupling Program | 119 |
| Appendix 3 : Morgue System Program | 149 |
| Appendix 4 : Hindi System Program | 177 |
| Appendix 5 : Rules..... | 208 |
| Abstract in Arabic | 212 |

List of Tables

| | |
|---|----|
| Table 1: ATMS and Production system Interaction. | 57 |
| Table 2: Four cases in The Morgue system | 61 |
| Table 3: Six cases in the Hindi system | 67 |
| Table 4: Time and join operation needed for 4-queen problem | 78 |
| Table 5: Number of assertions in alpha memory nodes (4-queen problem) | 78 |
| Table 6: Time and join operation needed for 3-queen problem | 78 |
| Table 7: Time and join operation needed for 5-queen problem | 78 |
| Table 8: Time and join operation needed for 6-queen problem | 79 |
| Table 9: An example of a constraint satisfaction problem | 80 |
| Table 10: Time, join operation, contradiction fire needed for Constraint satisfaction problem | 83 |
| Table 11: Number of tuples in alpha memory nodes (Constraint) | 83 |
| Table 12: Time, Join operation, Label Computation comparison for registration system-1 | 90 |
| Table 13: Execution Time, Join operation, and Label Computation comparison for registration system-2 | 92 |
| Table 14: Time, Join operation, Label Computation comparison for registration system-1 (example a) | 92 |
| Table 15 : Time, Join operation, Label Computation comparison for registration system-2 (example a) | 93 |
| Table 16 : Time, Join operation, Label Computation comparison for registration system-1 (example b) | 93 |

List of Figures

| | |
|---|----|
| Figure 1: Reasoning system = problem solver + TMS. | 3 |
| Figure 2: Rete Network | 14 |
| Figure 3: An Example of Rete network used in loose coupling. | 17 |
| Figure 4: An Example of Rete network used in tight coupling | 20 |
| Figure 5: An Example of Rete network used to show the drawback of Morgue system. | 23 |
| Figure 6: The next nodes of a type-checking node | 34 |
| Figure 7: The nodes that follow alpha-mem nodes | 35 |
| Figure 8: Rete Network | 39 |
| Figure 9: The Output of Rete building algorithm | 40 |
| Figure 10 : The Reasoning Process | 45 |
| Figure 11 : The Dependency Network | 52 |
| Figure 12 : The Reasoning Process in Loose Coupling | 55 |
| Figure 13 : 4-queens problem solutions | 74 |
| Figure 14 : Rete network corresponding to 4-queens problem | 76 |
| Figure 15 : Execution Time required to solve the queen problem | 79 |
| Figure 16 : Join operation in Queen problem | 79 |
| Figure 17: Rete network corresponding to constraint satisfaction problem | 82 |
| Figure 18 : Execution Time for Constraint satisfaction problem | 84 |
| Figure 19: Join operation and contradiction fire for Constraint satisfaction problem | 84 |
| Figure 20: Rete network corresponding to Registration problem(part of it) | 89 |
| Figure 21 : Execution Time for Registration problem | 91 |

Figure 22 : Number of Join operation and Label computation comparison for

Registration problem91

Abstract**A STUDY OF THE EFFICIENCY OF
TMS-BASED PRODUCTION RULE SYSTEMS**

by

Khalid Waleed Mahmoud

Supervisor

Assoc. Prof. Riad Jabri

Co-Supervisor

Asst. Prof. Khalil el Hindi

Truth maintenance systems (TMS) have become a common and very widely used in AI technology. TMS's give solutions for a lot of problems; such as retracting facts, explaining results, reasoning in dynamic environment and so on. Assumption-based truth maintenance system (ATMS) is one kind of TMS that support multiple contexts. It can be tightly or loosely coupled with production systems in order to find an optimal solution to a given knowledge based problem by investigating all possible contexts simultaneously.

Many algorithms to couple ATMS and production systems have been suggested. This thesis is aimed at empirical study of three coupling algorithms. The first algorithm is based on loose coupling; where the production system and ATMS are clearly separated. The other two algorithms are based on tight coupling; where the ATMS operation is integrated with the production system's operations, these two algorithms are: Morgue system and Hindi system.

Within the framework of this study, the performance of these algorithms has been studied and compared with each other, as well as their advantages and drawbacks have been described in details.

It has been found, using four different applications, that the tight coupling approaches are more efficient than the loose coupling approach, and the Hindi system is at least as efficient as the Morgue system, and in many cases much more efficient.

Chapter 1

Introduction

Production systems have played a very important role in the evolution of AI by introducing and performing search process needed in many AI application. Production systems consists of the following :

- A collection of **if-then** rules, each consisting of **if-part** that determines the conditions needed to fire the rule, and **then-part** which describe the action to be performed if the rule is fired. When the conditions of a rule (**if-part**) is true, then the rule's action (**then-part**) is executed and the rule is said to be fired.
- A set of data (assertions) stored in working memory, these data are the knowledge needed for the problem to be solved, sometimes called working memory elements (WME's).
- Control strategy that specify the order in which the rules will be used and a way of resolving conflicts when there is more than one applicable rule .
- Rule applier: a way to execute the **then-part** of the rule.

The flow control in reasoning system finds a list of all applicable rules, and selects one of them to be fired (conflict resolution). Firing a rule can change the content of the working memory and this can affect the list of applicable rules. The previous steps must be repeated until no rule can be fired.

Conventional reasoning systems have several shortcomings, these shortcoming are listed as follows:

- 1- Conventional reasoning systems do not provide explanation of how its conclusions were derived from the initial assertions. They just provide the answer which is not always enough. For example, in diagnosis system, it is not accepted that the system decides that a surgery is needed without giving the reason for this decision.
- 2- Conventional reasoning systems do not provide explanation of why something is impossible. Again if we have a way of generating an explanation about why it is impossible then we might be able to find the reasons.
- 3- When a reasoning system makes an inference based on the facts that we have at hand, the issue arises of what to do if one of the facts become false. We have to retract all facts (assertions) that was derived using this fact. This is can not be done in conventional reasoning system since we don't know which assertions have been derived using this retracted fact. This can be done if we know the data dependency. The data dependency means that how the truth or falsity of each assertion is related to the truth or falsity of other assertions. This issue of belief revision is important in many real world applications.
- 4- Since most AI reasoning system search, they often go over parts of the search space again and again. This is an overhead which can't be avoided in conventional reasoning system. If the system cached its inference then it would not need to re-derive conclusions that it had already derived earlier.
- 5- Conventional reasoning systems are designed to work with information that must be complete, i.e. all facts that are necessarily to solve the problem are present in the system or, at least, can be determined so that the user can be asked to provide. These systems can't deal with situation that depends on lack of some piece of knowledge (non-monotonic inference). For example, we would like to be able to say things like "if you have no reason to suspect person then assume he did not do it" , adding a new

assertion (some reason to suspect the person) may invalidate an inference that depends on the absence of that assertion. (The ABC Murder story (Quine, 1978), is a good example for non-monotonic reasoning).

The previous shortcoming led to the development of truth maintenance systems (TMS's) that have become a common and very widely used in AI technology. TMS's is a technique developed originally for the purpose of belief revision.

a TMS-based reasoning system usually consists of two parts: a problem solver and TMS as shown in figure 1.

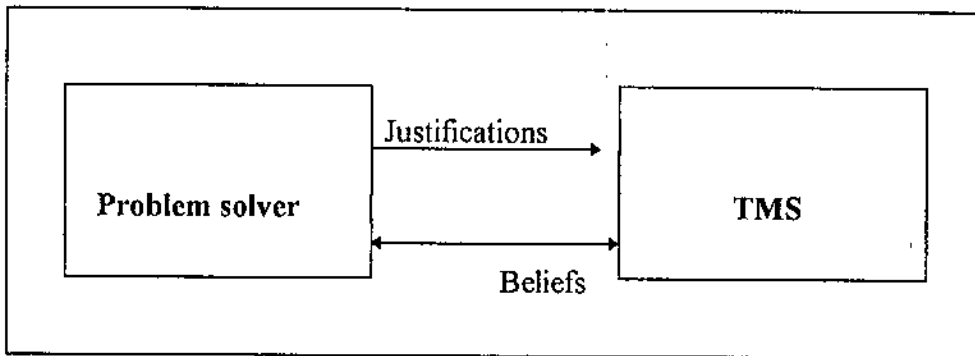


Figure 1: Reasoning system = Problem solver + TMS

Whenever the problem solver performs an inference step, it informs the TMS about the belief of the problem solver data and the reason for these beliefs by sending its justification (information about how the datum be derived) to the TMS. The TMS creates for each datum a node, and connect this node with nodes that were used to infer this node; these nodes appear in the justification. The result network (nodes, justifications) is called dependency network. The problem solver also informs the TMS of sets of mutually inconsistent beliefs. In return the TMS can inform the problem solver if there is a current set of mutually consistent beliefs.

TMS can help to overcome the previous shortcomings of the conventional reasoning system:

- 1- Justifications recorded by the TMS allow the problem solver to generate explanation by tracing the dependency network.
- 2- Justification again can be traced backward in case of contradictions to identify the piece of knowledge responsible.
- 3- When a fact (assertion) turns to be false, TMS use the dependency network in order to locate the inferred data that depend directly or indirectly on the retracted fact; and thus provide means to perform some sort of belief revision (Hindi, 1994).
- 4- Using the cached inference in the TMS, the problem solver need not to re-derive conclusions that had already been derived earlier in the search.
- 5- When a new assertion becomes available, the TMS located the inferred data that depend on the absence of this assertion and retract these inferred data; and thus provides means to perform non-monotonic reasoning.

Truth maintenance systems have the potential to significantly improve the efficiency of problem solving. There are many cases in which a TMS enhances the overall efficiency of the reasoning system. In many cases, caching inferences is more efficient than rerunning the inference rules that generated them in the first place. However, if the inference rules are inexpensive and the task does not require an exponential number of assertions, the TMS is probably inappropriate (Forbus, 1993).

There are several different families of TMS. Each type serve the problem solver in different ways, and hence support different types of AI problems. The main families of TMS's are as follows :

- Justification-based truth maintenance system (JTMS): These systems are the oldest type of TMS. They support non-monotonic reasoning, and allows the problem solver to make inference in single context.
- Logic-based truth maintenance system (LTMS): Unlike the JTMS, the LTMS allows negations to be expressed explicitly and therefore can represent any propositional calculus formula; otherwise, the JTMS, LTMS are very similar.
- Assumption-based truth maintenance system (ATMS): The ATMS is like a JTMS, but it allows the problem solver to make inferences in multiple contexts at once to generate all solutions, and this allows the ATMS to consider all possible combinations of assumptions. The basic ATMS doesn't support non-monotonicity. However, it has been extended to deal with non-monotonic reasoning (Dressler, 1988).

These various TMS systems have advantages and disadvantages with respect to each other. The JTMS and the LTMS are often better when only one single solution is required (reasoning in one single context) since they do not need to consider alternatives. However, the ATMS is more suitable for problems that require reasoning in multiple contexts at the same time (all solutions are required).

An optimal solution to a given knowledge based problem may be efficiently obtained by investigating all possible contexts simultaneously. Since the Assumption-based Truth Maintenance System is a multiple context TMS, we concern ourselves, in this thesis, with the ATMS.

AI researchers Ohta(1990), Morgue(1991), Hindi(1994) considered integrating problem solver and the ATMS in order to improve the overall efficiency of reasoning system. The main approaches to integrate the problem solver and ATMS was the loose coupling approach and the tight coupling approach. Morgue noticed some drawbacks of the loose coupling approach and so developed the tight coupling approach to overcome

these problems. However, Hindi noticed some drawbacks of Morgue approach and so suggested a new approach for tight coupling.

Within the framework of this study, all previous approaches have been implemented and their performance in four different problems has been evaluated

The rest of the thesis is structured as follows. In chapter two we review the main approaches of integrating production systems with truth maintenance system. In chapter three we present the structures and the algorithms needed to implement the main approaches. In chapter four we give the empirical performance comparison between the approaches. The main findings and possible future works are summarized in chapter five.

Chapter 2

Background

In this chapter we will review the ATMS, production rule systems, and the main approaches to integrate them. In section 2.1, the ATMS is presented. Production rule systems are discussed in section 2.2. In section 2.3 the Rete algorithm is described. In section 2.4, three approaches to integrate the Rete-based production rule system and ATMS are discussed.

2.1 The ATMS

The ATMS (de Kleer, 1986 a, b & c) (Forbus, 1993) is a search-control mechanism that can be combined with a problem solver to solve a wide range of problems. The problem solver and the ATMS interact with each other as follows:

- 1- The problem solver determines the inference (Datum) to be made and its justification.
- 2- Then, the problem solver transmits the justification to the ATMS.
- 3- The ATMS creates a node for the derived assertion.
- 4- Finally, The ATMS computes a label for the node by executing the label-update algorithm (discussed below and represented formally in chapter 3).

Let us present some definitions that will be useful (all the following definitions are taken from de Kleer, (1986)):

Problem solver Datum: Datum is used to refer to the problem solver implication, and it is the structure used to represent assertions and tuples.

ATMS node: Node is used to refer to ATMS node. An ATMS node has the form:

γ_{datum} : < datum, label, justifications >

Consider for example the following ATMS node: γ_{datum} : <P, {{A, B},{C}},{(a, b)}>. this means that the datum P holds under assumptions A and B or under C only (capital letter), and P was derived using a, b (small letter) as antecedent part of the justification.

Assumptions: An assumption is a special datum which the problem solver assumes but might later retract.

Justifications: An ATMS justification describes how a node is derivable from other nodes, for example, $a, b \Rightarrow c$; this mean that the node c is derived using the nodes a and b .

Environment: An environment is a set of assumptions. A node n is said to hold in an environment E if n can be derived from E and the current justifications.

Context: The context of a consistent environment is the set of facts that can be derived from the assumptions of that environment.

Label: The label is a set of environments associated with every node, n . Every environment E of n 's label must be consistent and n must holds in each environment E of n 's label. The label is constructed by the ATMS.

Dependency Network: It is the network created by the ATMS from nodes and justifications.

The label of a node consists of some environments in which node is hold in. Each environment consists of several basic assumptions that led (directly or indirectly) to the node. The assumption is a special datum which the problem solver assumes but might later retract.

An environment is inconsistent (nogood) if the assumptions of this environment entail a contradiction. Any environment E which is a superset of any environment "marked as nogood" is also nogood. This due to the monotonic nature of the ATMS i.e.

if a node (contradiction for example) can be inferred in an environment, E , it can also be inferred in any super environment of E .

The label of any node must be consistent, sound, complete and minimal. A label, L , for a node, n , is consistent if all of its environments are consistent. L is sound if n holds in all environments of L . L is complete if every consistent environment E , for which n holds, is a superset of some environment in L . L is minimal if no environment in L is a superset of any other environment in L .

Whenever a new justification is provided to a datum, the ATMS executes the label-update algorithm. This algorithm starts by computing the label for the consequent of the justification (derived assertion). For example, Consider the nodes a , b and c having the following labels:

$L(a) = \{\{A, U\}\}$; $L(a)$ means the label of a node a .

$L(b) = \{\{B\}, \{C, D\}, \{A, C\}\}$

$L(c) = \{\}$

478599

Now if a new justification for c , $a, b \Rightarrow c$, is supplied, the label of the node c must be updated. The first approximation of the new label consists of the union of all possible combinations of single environments; one environment from each antecedent node label. Thus for c we get $\{\{A, B, U\}, \{A, C, D, U\}, \{A, C, U\}\}$, the label constructed in this way is complete and sound. The second step is to make it minimal, this could be done by removing the environments that are superset of the others, so we have to remove $\{A, C, D, U\}$ because it is a superset of $\{A, C, U\}$. The third step is to make it consistent, this could be done by removing the environments that are superset of the nogood environment. Suppose that $\{A, C\}$ is nogood environment, so the environment $\{A, C, U\}$ should be removed because it is a superset of the nogood environment $\{A, C\}$. As a result we get a new label for the node c which is $\{\{A, B, U\}\}$.

After computing the label for the derived assertion, The label-update algorithm check if the newly constructed label is equal to the old one, if so, then, the processing of the derived node is finished. Otherwise, the label of all the consequent nodes (all nodes that are derived using this node) have to be updated in an analogous way.

Whenever a new nogood environment is discovered, this environment and all its superset environments must be removed from the label of all nodes.

Some typical queries that the problem solver can ask the ATMS are:

- Under what assumptions does the given datum hold.
- Are a given set of assumptions consistent.

In summary, ATMS performs the following basic operations: creating a node to represent a datum, adding justifications to the dependency network, maintain the label of each node, handling nogoods. However, label updating and nogoods handling are costly operations and their efficient implementation is a key point in the successful use of ATMS in real word problem.

2.2 Production Rule Systems

Production (rule-based) systems are popular architecture for problem solvers. A typical system consists of:

- Domain knowledge in the form of rules or productions.
- Working memory which holds all the assertions (tuples) that are already known for the current problem.
- A rule interpreter which determines which rules are applicable for the current problem, and select one of the rules in order to execute it.

By convention, any rule contains two main parts: the “if part” of a rule which is called LHS (left-hand side), sometimes called an antecedent part, and its “then part”

which is called RHS (right-hand side), sometimes called consequent part. The LHS of any rule consists of a set of patterns.

A forward chaining rule system starts with a collection of assertions and tries all available rules over and over, adding a new assertions as it goes (incremental), until no rule can produce a new assertion. In Contrast, a backward chaining rule system starts with a goal and tries to verify that goal using assertions that are already known.

Because TMS are particularly useful for incremental systems (to be able to consider any new relevant data that become available during reasoning), and because forward chaining is incremental by nature, we concentrate, in this thesis, on the forward chaining systems.

The basic inference cycle for a production systems goes into three phases: match, select, and act. These phases are described below:

Match: The patterns in the left hand side (LHS) of the rules in the knowledge base are matched against the content of the working memory, in order to determine which rules have their LHS's satisfied with consistent bindings to the working memory assertions. Such Rules are said to be applicable and are put in a queue called conflict set.

Select (Conflict Resolution): From the conflict set, one of the rules is selected to be executed. The selection strategies may depend on any criteria, for example, first in first out (FIFO).

Act: The rule that have been selected from the previous step is executed by carrying out the consequence of the rule (RHS), they may involve any operation (add, delete, or change the content of the working memory).

The above cycle is repeated until no rules can be applicable.

2.3 The Rete Algorithm

Among the previous steps, the match step is known to require a large part of the execution time. Naive algorithm for matching would try to match the rules patterns one by one against the working memory assertions. This algorithm is not efficient for large systems (typical knowledge base may contains hundreds or even thousands of rules, each with several patterns, and hundreds of assertions as well). The problems of the naive algorithm are described below:

1. In each cycle, the naive algorithm starts fresh matching our rules against all the working memory assertions although the contents of the working memory changes very little from one cycle to another; this makes a lot of the matching unnecessary.
2. The same patterns may appear in several rules. The naive algorithm does not care, and the testing is repeated for each one of them.

The Rete (Forgy, 1982) algorithm used in the OPS-5 production system, was the first to address these problem. The algorithm suggests two methods to minimize the time needed for the match step, these two methods are described below:

1. Forgy suggests that by saving match information, the rule interpreter does not have to start fresh in each cycle. Instead, the rule interpreter processes only the new working memory assertion by checking if the new assertion in conjunction with the assertions entered earlier can complete the instantiation of any rule; if so the ground instance (the rule's patterns are replaced by a corresponding assertions) of this rule is saved in the conflict set.
2. Repeated testing of the same pattern shared between some rules could be avoided. A single set of tests for the same pattern could be performed by grouping rules which share the same patterns and linking them according to their common patterns.

The Rete algorithm first, loads all the rules and examines their LHS's before the reasoning starts, compiles the rules into a Rete network, which connects all the rules having a common patterns in their LHS.

A Rete network consists of several nodes, these nodes and their functions is listed below:

- A Root node: It is the first node in the network, where all assertions go through. It is followed by the type checking node.
- Type checking nodes: One node is created for each different class of patterns. It is followed by t-const nodes or an alpha memory node.
- t-const nodes: one t-const node is created for each pattern consists of some tests. It is followed by an alpha memory node.
- An alpha memory node (α node): is a place to hold assertions (tuples).
- *AND* nodes: this node used to join two alpha memory nodes. It is followed by alpha memory nodes, sometimes called beta memory node.
- P memory nodes: this is the last node created for each rule. It is a place to hold the joined tuples used to instantiate the rule (i.e. ground instance of the applicable rules).

For example, take the following rules :

if A($x > 50$, y), B(y, $z > 25$) then

if A($x > 10$, y), C(y > 20 , z) then

The complete Rete network built for these rules is shown in figure 2. A, B and C are type checking nodes. ' $x > 50$ ', ' $x > 10$ ', ' $z > 25$ ', and ' $y > 20$ ' are t-const nodes. α_1 , α_2 , α_3 , α_4 , α_5 , and α_6 are alpha memory nodes. AND1, AND2 are *AND* nodes. P1 and P2 are p-mem nodes.

The Rete matching algorithm works as follows:

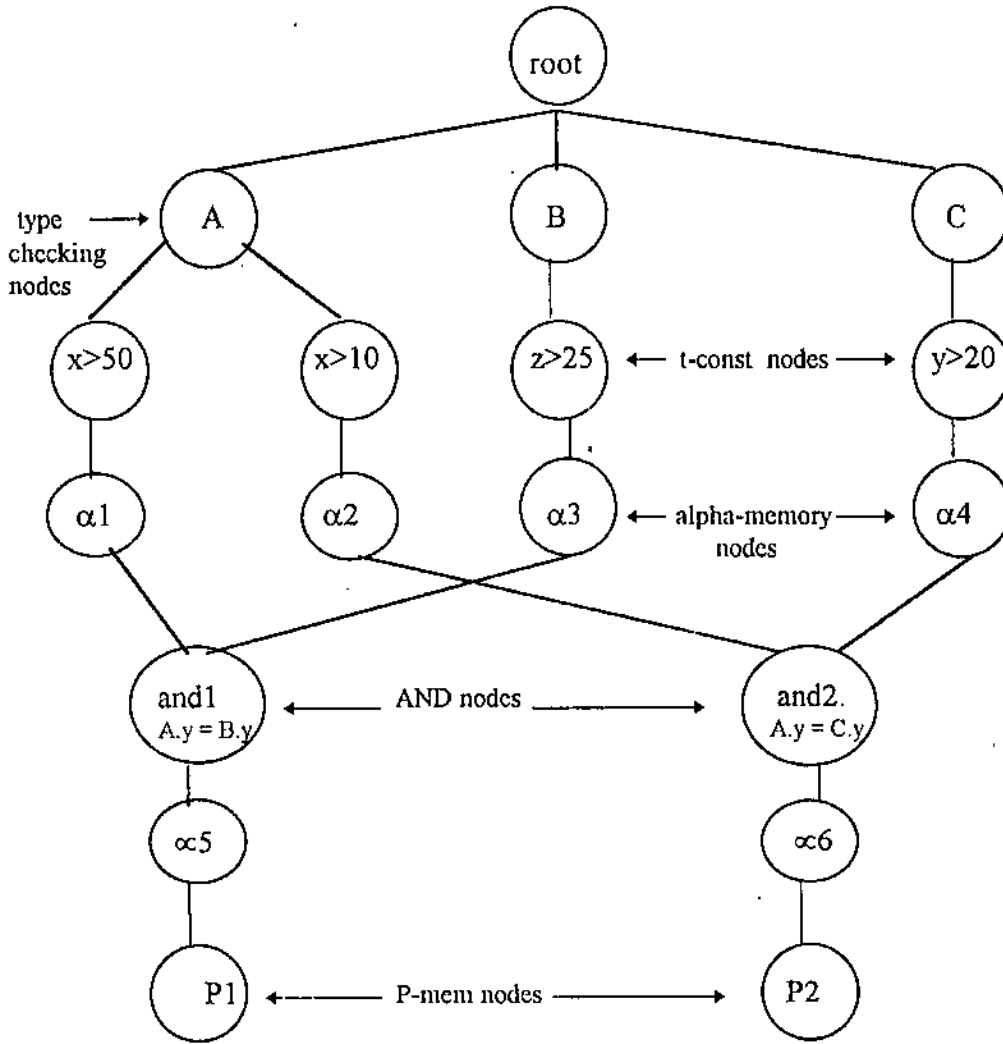


Figure 2 : Rete Network

- After building the Rete network that corresponds to the rules, all the initial working memory assertions are entered to the network from the root node. The root node receives assertions as inputs, and propagates a copy of each assertion that it received to each of its successors.
- The type checking node checks the name of the assertion, if the assertion has the right type, it forwards it to its successors (t-const or α nodes), otherwise it discards it.

- A t-const node checks the assertion that it received against a simple test (if needed), if the assertion passes the test, the node will forward the assertion to the next node (α node), otherwise the assertion gets discarded.
- An *AND* node has two input memories and one output memory. An *and* node checks whether the incoming assertion matches any saved assertions in the other input memory node; if so, the *and* node will join each matching pair and then forwarded to the output memory node, otherwise, the *and* node will discards the arriving assertion.
- The assertions or joined assertions that reaches the P memory node, will be used to instantiate the corresponding rule.

After all assertions have been entered in a cycle (propagate down through the Rete network), one rule instance in the conflict set is fired, and the remaining instantiations in the conflict set are saved. The result of the fired production rule, which may be a new assertion, is entered to the system, and propagate through the network to find if any rule instantiation can be completed by that assertion. These new instantiations are added to the conflict set saved from the previous cycle, and one rule instance in the conflict set is fired and removed. The reasoning process terminates when no rules can fire (i.e. when conflict set is empty).

2.4 Integrating Production Systems and Truth Maintenance Systems

Integrating production systems and TMS, will increase the overall efficiency of the production system if a suitable interface is used between the Rete-based production system and the TMS; Otherwise this integration could needlessly waste computer resources (Morgue, 1991).

In this section we will discuss three approaches to integrate the ATMS and Rete-based production systems. These are the loose coupling approach, the Morgue tight coupling approach, and the Hindi system tight coupling approach.

2.4.1 Loose coupling the ATMS and production system.

The loose coupling approach maintains the dependency network of the ATMS and the Rete network as a separate entities. It only, modified the “select” step in the inference cycle in order to integrate the ATMS and the Rete-based production system. This modification and the main ideas in this approach are described below:

- When the inference engine selects a rule instance to be fired, the ATMS is called to compute the label of the consequence of the selected rule. If the label is not empty, the rule is fired; Otherwise, the inference engine select another rule instant, because there is no point in inferring a datum that holds in no environment.
- Contradiction rules have the responsibility to discover the nogood environments. Since the nogoods and all its superset, must be removed from the labels of all nodes, the process of label-update will become easier. So discovering the nogoods assertions as early as possible will simplify the label-update process, and that can prevent other rules from firing because their consequences will have empty labels. This could be done by giving contradiction rules more priority in execution than other rules, i.e., by executing the contradiction rules before executing any other normal rule.

Morgue,(1991), noticed two main drawbacks of this approach. The first is that some work might be done repeatedly by the ATMS. The second is that some work might be done unnecessarily by Rete.

Consider the following example (taken from Hindi, 1994): (The corresponding Rete network is shown in Figure 3).

Rules :

R1 : if $p_1(X, Y), p_2(Y, Z), p_3(Z)$ then $q(X, Z)$

R2 : if $p_1(X, Y), p_2(Y, Z), p_4(X, Z)$ then \perp (this is a contradiction rule)

ATMS nodes:

$\gamma_{p_1(a, b)}$: $\langle p_1(a, b), \{A\}, \{(\dots)} \rangle$

$\gamma_{p_2(b, c)}$: $\langle p_2(b, c), \{B\}, \{(\dots)} \rangle$

$\gamma_{p_3(c)}$: $\langle p_3(c), \{E\}, \{(\dots)} \rangle$

$\gamma_{p_4(a, c)}$: $\langle p_4(a, c), \{A, B\}, \{(\dots)} \rangle$

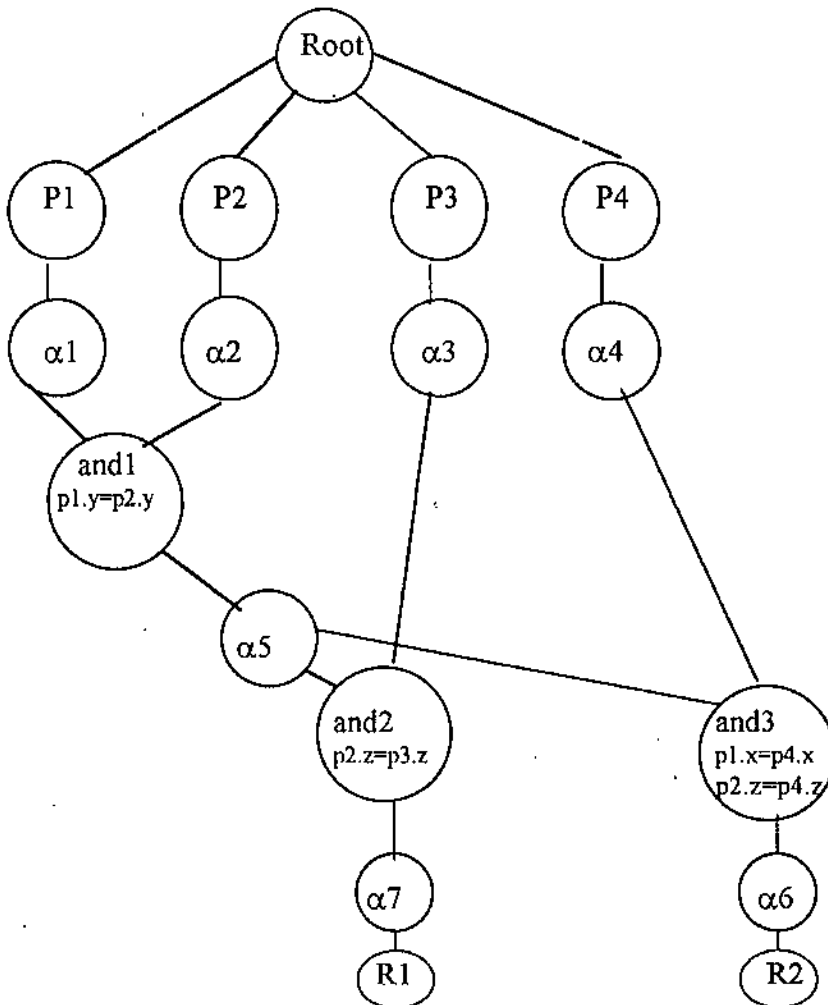


Figure 3: An Example of Rete network used in loose coupling

After the initial working memory elements entered the system, the two rules are found to be applicable. During the select step, the two rules will compute the label of the shared conjunct $p_1(a, b) \wedge p_2(b, c)$, which is a repeated step that could be avoided.

To illustrate the second drawback, the environment $\{A, B\}$ will be nogood after executing the second rule. Thus, the label of the conjunct $p_1(a, b) \wedge p_2(b, c)$ will be empty, but since this information is not available during the match step, The Rete would still match the conjunct with the tuple $p_3(c)$. This is unnecessarily step because the label of the resulting instance of the first rule will be empty also, and will not be used in the act step.

We clearly need to integrate the ATMS label computation with the match step of the inference engine, which was the basic principle of the Morgue, (1991) approach.

2.4.2 Tight coupling of ATMS and production system : the Morgue system

The tight coupling approach proposed by Morgue,(1991) (a very similar approach was described in (Ohta, 1990)), avoided the loose coupling drawbacks. The approach integrates the dependency network of the ATMS with the Rete network. The approach involves some modifications to the Rete algorithm in the match step. During the match step, the approach perform the following:

- 1-Calculate the label of every tuple (assertion) or joined tuple in α memory nodes and store the labels there with the tuple.
- 2-If the label of any tuple or joined tuple becomes empty (i.e. cannot instantiate a new rule), then it is removed from all the memory nodes.

Doing so, eliminates the drawbacks of the loose coupling approach because:

- 1-Discarding these tuples or joined tuples with empty label, will prevent the propagation of the tuple down in the Rete network. Thus in the previous example the tuple $p_3(c)$ will

not be matched with an empty label joined tuple, i.e. the approach avoided unnecessary match work done by Rete.

2-Storing the label of a tuple or joined tuple in the memory nodes, makes it shared among different rules. This will reduce the label re-computation for the same tuple, i.e. much of the work done by the ATMS is saved and shared. In our previous example the label of conjunct $p_1(a, b) \wedge p_2(b, c)$ will be computed only once.

The main drawback of this approach is described as follows:

label update is a costly operation in the ATMS, and is even made worse with the tight coupling approach. The Morgue tight coupling approach labels more tuples (the joined tuples in addition to the usual tuples) than the loose coupling approach, and thus any operation that updates the labels (contradiction handling, add a new justification, retract a tuple) must also update the labels in the Rete memory nodes.

Morgue's tight coupling approach minimize the effect of this drawback by reducing the number of tuples in the Rete network and by performing more data recording.

Discovering contradictions as early as possible will shortcut a great amount of computations and purge a large part of the search space. Since the contradiction rules produce new nogood environments, and the new nogoods must be removed from the label of all nodes, the labels of some tuples may become empty label, and so, get removed from the Rete memory nodes. This reduces the amount of data to be matched by Rete. Thus firing contradiction rules as soon as they get instantiated would probably save some matching work. To do so, when a tuple or a joined tuple is used to instantiate both a contradiction rule and other rules, we need to fire the contradiction rule before transmitting the data to the other rules. In other words, we delay the propagation of the

data (in Rete) to match the normal rules until the contradiction rule is fired. Consider the previous example (The corresponding Rete network is repeated in Figure 4).

Rules :

R1 : if $p_1(X, Y)$, $p_2(Y, Z)$, $p_3(Z)$ then $q(X, Z)$

R2 : if $p_1(X, Y)$, $p_2(Y, Z)$, $p_4(X, Z)$ then \perp (*this is a contradiction rule*)

ATMS nodes:

$\gamma_{p_1(a, b)}$: $\langle p_1(a, b), \{A\}, \{(\dots)} \rangle$

$\gamma_{p_2(b, c)}$: $\langle p_2(b, c), \{B\}, \{(\dots)} \rangle$

$\gamma_{p_3(c)}$: $\langle p_3(c), \{E\}, \{(\dots)} \rangle$

$\gamma_{p_4(a, c)}$: $\langle p_4(a, c), \{A, B\}, \{(\dots)} \rangle$

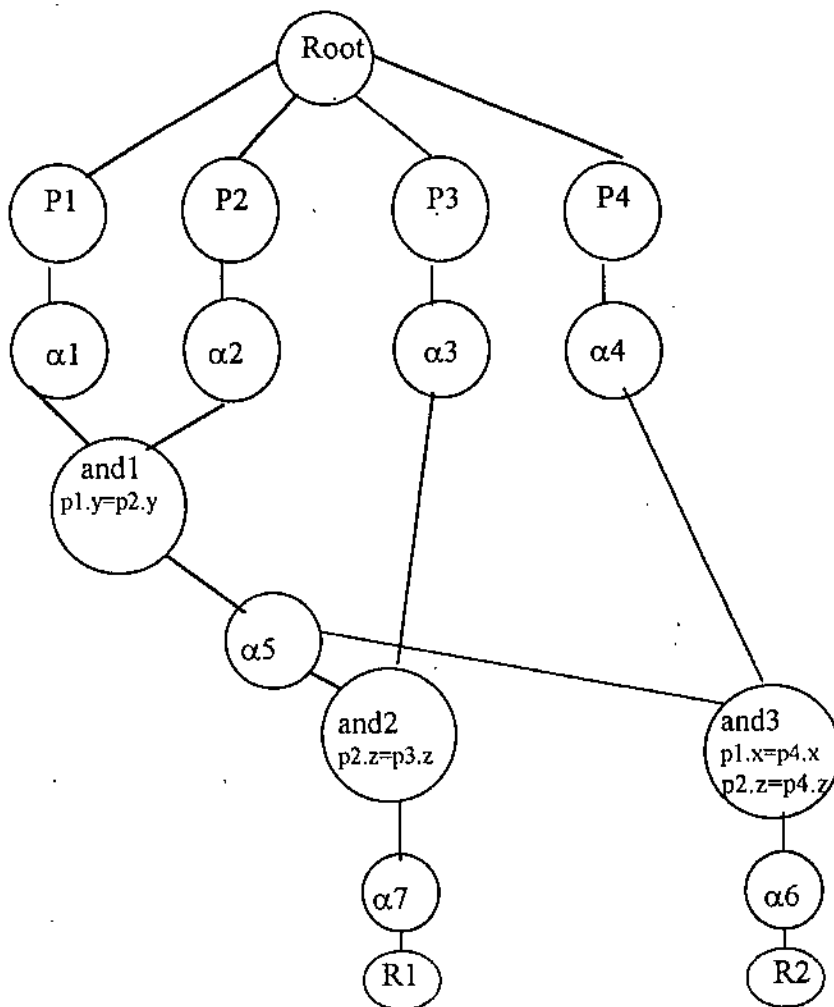


Figure 4: An Example of Rete network used in tight coupling

Since node α_5 is shared between R1 (normal rule) and R2 (contradiction rule), any data that reach the node α_5 must be propagated first to *and3*, and when the instantiated rule is fired, data can be propagated to *and2*. When the system executes the contradiction rule (R2), the system will discover that the environment $\{A, B\}$ is nogood, and this environment must be removed from the label of all nodes. Since the label of the compound tuple in α_5 has only this environment, then after deleting this environment the label becomes empty and so the tuple gets discarded. Thus *and2* will not make any matching effort because the left input memory has no tuples.

The tight coupling approach performs the following extra data recording:

- 1-every tuple or joined tuple records the α memory in which it is stored.
- 2-each tuple is linked through intermediate joined tuple to other tuples.
- 3-each environment records the tuples and joined tuples which have this environment in their label.

When a new nogood environment is discovered, all the tuples and joined tuples recorded in the new nogood environment have to update their labels by removing the nogood environment from them. If the label of any tuple or joined tuple becomes empty then :

1. record the Rete memory node in which it is stored.
2. traverse the links between the tuple and other tuples to determine the affected tuples and record its memory nodes.
3. After finishing the previous step, traverse all the recorded memory nodes and remove the marked tuples or joined tuples.

To Summarize: The main points of the tight coupling approach are :

- Rete and the dependency network of the ATMS are integrated together.
- The label for any tuple (or joined tuple) that reaches any memory node is computed.
- All tuples with empty label get deleted.
- The algorithm establishes the needed links between tuples in the network.
- If an α memory node is shared between a contradiction rule and another rule, the contradiction rule is given the priority in execution over all other rules.
- When a new nogood is discovered during the reasoning process, the algorithm updates the label of the affected nodes using the links established previously.

2.4.2.1 Drawbacks of the Morgue system

Hindi (1994) has noticed some serious drawbacks of Morgue's approach. These are described below.

First, Label update (adding a new environment) is now more complex because it may require rejoining tuples. Deleting tuples with empty label may be complicated and make the approach (the tight coupling approach) inefficient. This is because, the label of such a tuple may gain a new environment (due to a new justification), and thus becomes relevant again. This tuple needs to be regenerated. If the tuple was part of a joined tuple then it needs to be re-joined.

What makes the problem even worse is the fact that we do not know whether adding the new environment will cause some previously discarded tuple become relevant again, before we actually generate such a tuple, and compute its label.

Consider the previous example (The corresponding Rete network is repeated in Figure 5).

Rules :

R1 : if $p_1(X, Y), p_2(Y, Z), p_3(Z)$ then $q(X, Z)$

R2 : if $p_1(X, Y), p_2(Y, Z), p_4(X, Z)$ then \perp (*this is a contradiction rule*)

ATMS nodes:

$\gamma_{p_1(a, b)}: \langle p_1(a, b), \{\{A\}\}, \{\{\dots\}\} \rangle$

$\gamma_{p_2(b, c)}: \langle p_2(b, c), \{\{B\}\}, \{\{\dots\}\} \rangle$

$\gamma_{p_3(c)}: \langle p_3(c), \{\{E\}\}, \{\{\dots\}\} \rangle$

$\gamma_{p_4(a, c)}: \langle p_4(a, c), \{\{A, B\}\}, \{\{\dots\}\} \rangle$

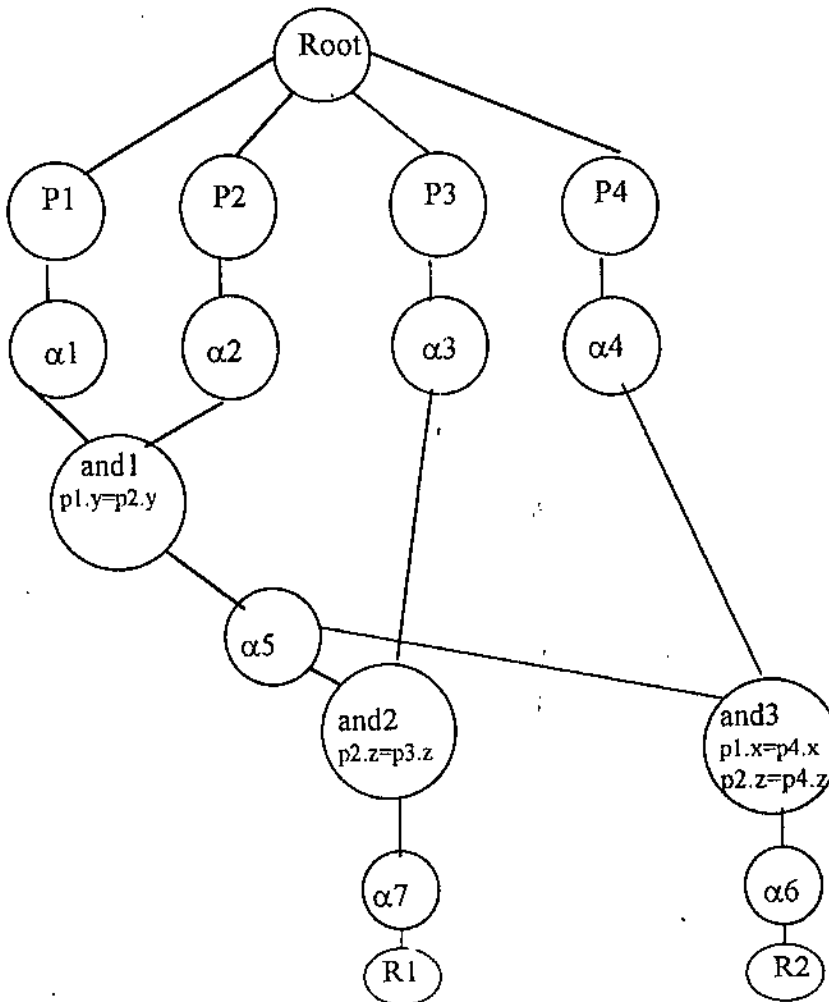


Figure 5: An Example of Rete network used to show the drawback of Morgue system.

The joined tuple $p_1(a, b), p_2(b, c)$ produced by *and1* holds in the environment $\{A, B\}$, this environment is a nogood environment, so it must be removed from the label of the conjunct $p_1(a, b), p_2(b, c)$, leaving it with empty label, and hence the joined tuple

must be discarded. Now, suppose that the system adds a new environment to the label of tuple $p_1(a, b)$ say $\{C\}$, This means that the tuple $p_1(a, b)$ with a new environment must be re-joined with the matched tuple in the other input node and so regenerate the joined tuple $p_1(a, b), p_2(b, c)$, but now with a new label $\{B, C\}$.

Thus adding a new environment to the label of any tuple requires rejoining the tuple with all matched tuple in the other input memory, in addition to computing the labels such tuples. Since the ATMS is usually used in domains that require generating solution in multiple contexts; adding an environment to the label of a tuple is an operation that takes place frequently, and also adding environment to the label of in one tuple may affect another tuple. In short, the addition of a new environment could add extra overhead to the Morgue approach, which increases the number of operations done by the Rete instead of decreasing it.

Second, discarding any tuple from the Rete (because it hold in no environment), will also discard all the work which Rete has performed to generate this tuple, and also all other joined tuples of which this tuple is a part of it. So, if this tuple is regenerated because it gained a new environment, all of the work done by Rete to match it is a repeated work that might include some expensive join operations.

Thus from the previous discussion, it is clear that discarding any tuple from the network, will not reduce the amount of work done by the Rete, instead, it could make label update more expensive operation that may require a re-matching and re-joining tuples.

2.4.3 Tight coupling of ATMS and production system : The Hindi system

To improve Morgue's system, we should consider the following problems :

1. When the label of a tuple becomes empty, the Rete will discard all the works it performed to match the tuple.

2. When a new environment is added to the label of any tuple, The Rete will perform extra expensive operations such as join operation in addition to label update.

The main idea in the solution (Hindi, 1994) was to put tuples with an empty label in a special inactive part of a Rete memory node (called *OUT*) and tuples with non-empty labels in an active part of the memory node (called *IN*).

Only tuples in the *IN* part are involved in the match process. Thus, Rete does not have to do any work on matching a tuple with an empty label, and at the same time need not discard such tuples. Therefore, it will not need to regenerate such tuples when their label becomes not empty.

If the label of a node becomes empty, then the label of all joined tuples of which the tuple is a part of becomes empty. Instead of removing all these tuples from the network (as required by the Morgue system) and so, discard all the work performed to match these tuples, all these tuples are moved to the *OUT* part of their memory nodes, and can move back if their label become non-empty. Doing so, solves the first problem.

When the label of any tuple in the *OUT* part becomes non-empty again (because of a new environment is added to the system), this tuple needs only to be matched with those tuple in the *IN* part of the other input memory that have been inserted after they were moved to the *OUT* parts of their memory nodes. To do so, Hindi used time stamp to determine the order in which tuples were inserted, and it is described as follows :

- When a tuple is inserted or moved to the *IN* part, it will record the current time.
- When a tuple becomes *OUT*, it will record the current time and advance the time.

When the tuple is moved to the *IN* part and if it was a part of some joined tuples, this tuple must not be rejoined with the other tuples. Instead, the joined tuples it is linked to are moved to the *IN* part of the their memory. Doing so, solves the second problem.

Hindi, distinguished between two types of tuples with empty labels. The first is a tuple that have not become non-empty yet, this means that this tuple was never in the *IN* part; such a tuple needs to be matched with every tuple in the *IN* part. To guarantee this, this tuple is stamped with a time less than all possible time stamps (e.g. -1). The other type is, a tuple that was in the *IN* part in a previous step, and due to discovering a new nogood the label becomes empty; such tuples must be stamped with the current time.

In The next chapter, we will investigate all the previous algorithms in more detail, and describe all the implementation issues.

Chapter 3

Design and Implementation

In this chapter, loose coupling approach, Morgue approach, and Hindi approach are discussed in more details. Structures and algorithms for their implementation are described.

The design of rules, assertions and patterns are presented in section 3.1. The implementation of the Rete algorithm is explained in section 3.2. The implementation of ATMS is described in section 3.3. The algorithm and implementation of the loose coupling is described in section 3.4. The algorithm and implementation of Morgue system is described in section 3.5. Finally, The algorithm and implementation of Hindi system is described in section 3.6. The performance of the algorithms will be studied in the next chapter (Note: Datum is the corresponding structure for assertion (i.e. for tuple)).

3.1 Assertions, and Rule design

We will start by studying assertions, and their format, then we will describe the format of the rules as well as the restrictions over the variables.

3.1.1 Assertions

Assertions represent simple facts. An assertion takes the form of a list containing the assertion name which is a symbol that identify the assertion, field names which are a symbols preceded by the exclamation mark, and a value for each field. Assertions have the following syntax:

(assertion-name !field-name1 value1 !field-name2 value2 . . .)

Examples:

- (Add !Arg1 5 !Arg2 7 !Result 12) : This assertion says that “5+7 = 12”.
- (Father !F Ali !S Ahmad) : This assertion says that Ali is father of Ahmad.

Assertions are stored in the Rete memory nodes, some are stored before the beginning of the reasoning process, these assertions are called initial working memory elements. Others are inferred during the reasoning process.

3.1.2 Rules

Procedural knowledge are expressed as rules; these rules are used to build the Rete network. Each rule has two main parts; a trigger (or condition part) and a body (or action part). The trigger is a list of patterns that specify the kind of assertions which the rule intends to respond to. The body of the rule consists of a pattern that specifies the assertion to be inferred. Patterns are similar to assertions but they may contain variables which are symbols preceded by question mark (e.g. : ?x is a variable).

Whenever a set of assertions match the trigger part of the rule, the rule is fired (i.e. it's body is executed in the environment formed by matching the patterns with the assertions).

The general form of rules is:

(Rule rule-name (trigger) ==> (body)).

A rule body has the form:

(Rassert! pattern).

Example: “if ?x is father of ?y and ?y is brother to ?z then ?x is father of ?z” is represented using the following rule :

(Rule R1

((Father !f ?x !s ?y)

(brother !b1 ?y !b2 ?z))

```
==>(Rassert! (father !f ?x !s ?z))
```

Another kind of rules called contradiction rules, which are used to detect contradictions. Since contradiction handling is very important operation in all the algorithms we are studying, we encode them as explicit rules of the form:

```
(Contradiction-Rule rule-name
```

```
  (trigger-list)
```

```
  ==>(Rassert-nogood!))
```

Example: “if ?x is greater than ?y, and ?y is greater than ?x then assert a contradiction” is represented using the rule:

```
(Contradiction-rule R2
```

```
  ((Greater !arg1 ?x !arg2 ?y)
```

```
  (Greater !arg1 ?y !arg2 ?x))
```

```
  ==>
```

```
  (Rassert-nogood!))
```

3.1.2.1 Variable Restrictions

In the trigger part we sometimes need to put some restrictions over the values that the variables (in patterns) may take. In order to have this ability, we encode such restrictions using lisp functions that take all field values in the assertion, in the same order as they appear in the pattern, as parameters. The functions return “true” if the corresponding condition is satisfied. The functions are associated with the corresponding patterns according to the following syntax:

```
(pattern) :test !'function-name.
```

For example, if we need to force the following pattern (success !student-name ?x !mark ?y) to match only the assertions that have the value of the “mark” field greater than or equal to 50, then the previous pattern can be rewritten in the following manner:


```
(success !student-name ?x !mark ?y): test #'check-mark
```

where “check-mark” is a function defined as :

```
(defun check-mark (x y) (>= y 50)).
```

If more than one restriction exists you can encode all these restrictions in the function. Similarly, if you need to put restrictions on all patterns in the trigger part, you can write a function that check these restrictions, and put the function name at the end of the trigger part, using the following format:

```
(rule rule-name
  (trigger-list): test #'function-name
  ==>
  (body))
```

This function takes all the field values in all the patterns in the trigger part in the same order as they appear in the trigger part. For example, in the n-queen problem (see section 4.1 for a description for the problem), we need every queen in a different row. The rule used to enforce that is:

```
(Rule Q
  ((q !r ?x1 !c ?y1)
   (q !r ?x2 !c ?y2)
   ....): test #'check-rows
  ==>
  (body))
```

“check-rows” is a function that ensures that each queen is in different row and is defined as:

```
(defun check-rows (x1 y1 x2 y2 .....
```

(and (not (= x1 x2)) (not (= x1 x3)))

The general rule syntax is:

In case of a normal rule (not a contradiction rule):

```
(Rule rule-name
  ((pattern1) [:test #'function-name]
  (pattern n) [:test #'function-name] ) [:test #'function-name]
  ==>(Rassert! (pattern)))
```

Contradiction rule have, on the other hand, the following form:

```
(Contradiction-Rule rule-name
  ((pattern1) [:test #'function-name]
  (pattern n) [:test #'function-name] ) [:test #'function-name]
  ==>(Rassert-nogood! ))
```

3.2 The Rete Algorithm

In this section the Rete algorithms were studied in terms of their match process and structures used in the algorithms. Based on these terms, two algorithms are introduced, one to build the Rete network and the other to do reasoning.

3.2.1 The match Process

The Rete algorithm aims at improving the efficiency of the matching process, necessary in all production systems.

The Rete algorithm compiles the trigger part of the rules into some tests that must be satisfied in order for rules to become applicable (firable). There are two kind of tests: intra-tests and inter-tests. An intra-test involves one pattern, for example, for the following pattern:

(P1 !f1 ?x !f2 ?y !f3 ?x !f4 10)

the following tests must be generated:

1. (f1 = f3); because the variable ?x is associated with !f1 and !f3, which implies variable equality.
2. (f4 = 10); because !f4 have a constant value rather than variable (constant test).

An inter-test involves two or more patterns, for example the patterns:

((p1 !f1 ?x !f2 ?y)

(p2 !f3 ?z !f4 ?x))

imply that !f1 = !f4, which is an inter test that involves two patterns, so the following test must be generated: (f1 = f4); because the variable ?x associated with !f1 and !f4.

3.2.2 Structures

After analyzing the patterns in all rules, the rules used in the problem solver are compiled into nodes in the Rete network. These nodes are described in detail below: (see Def.lsp in appendix 1.2 for lisp implementation for the nodes)

1- Root Node: It's the first node in the network where all data must pass through. This node is attached to a distinct set of type-checking nodes. Its task is to forward a copy of each tuple it receives to each attached type-checking node. The structure (record) used to define the root node consists of:

Title: An identifier used to identify the Rete.

Type-checking: A list of the next type-checking nodes.

2- Type-checking node: All patterns found in the trigger part of all rules are classified into groups according to their names (the first symbol that appears in the pattern), a type-checking node is created for each class of patterns and is attached to the root node. Each assertion that passes the root node must be tested by these type-checking nodes. A

type-checking node discards the tuple if it not of the corresponding type, and passes it to the following node if it is. The following node could be a t_const nodes and/or alpha memory node. A type-checking node has one input and several output nodes. The structure used to define a type-checking node consists of :

Name: A pattern name.

Next-nodes: A list of the next nodes (alpha-mem or/and t-const nodes)

for example the pattern (loc !r ?x !c ?y), a corresponding type-checking node is created for it with name 'loc'.

3- T-const node (Test for Constant): After the Rete classified the patterns into classes according to their name, the patterns within the same class must be classified also according to some restrictions related to each pattern (intra-tests). (see section 3.1.2.1,3.2.1) for example: Suppose we have the following patterns that appear in different rules and all have the same pattern name:

1- (loc !r ?x !c ?y)

2- (loc !r 1 !c ?y)

3- (loc !r ?x !c ?x)

4- (loc !r ?x !c ?1): test #'function name

All the previous patterns have the same type-checking node because all of them have the same pattern name 'loc', but each of them has different conditions that must be satisfied in order that they can be instantiated:

1-The first pattern has no condition.

2-The second one has a test (!r = 1) which is a constant test (t-const1).

3-The third one has a test (!r = !c) which is a variables equality (t-const2).

4-The fourth one has two tests: (!c = 1), and an external test function(t-const3).

According to the previous cases, the type-checking node ('loc') must be attached directly to alpha-mem node to store all the assertions that match with the first pattern (loc !r ?x !c ?y), also it must be attached to three t-const nodes, one for each pattern of the patterns 2, 3, 4. All assertions that pass the tests in any t-const node are stored in a corresponding alpha-mem node (see figure 6).

Some tuples can pass the test of more than one t-const node. For example: the assertion (loc !r 1 !c 2) satisfying the corresponding t-const nodes for pattern 1 and pattern 2, therefor, a copy of the assertion should be stored in α_1 and another in α_2 .

The structure used to define a t-const node consist of:

Id: Unique identifier.

Check1: Constant check.

Check2: Variable equality.

Check3: External test function.

Next-node: An alpha memory node that store all the passed assertions.

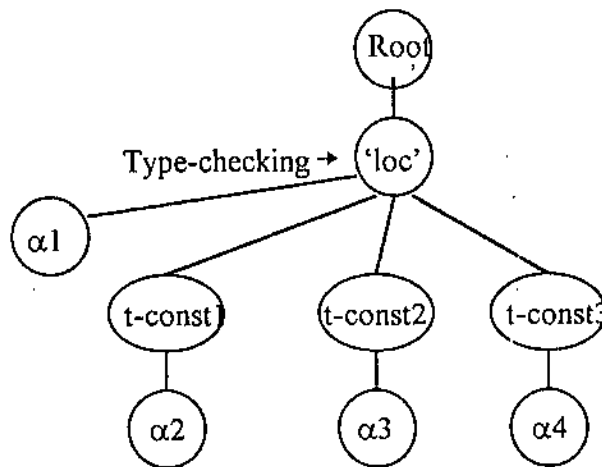


Figure 6: The next nodes of a type-checking node

4- Alpha-mem node: It's a place to hold assertions (compound or single) that can pass the previous nodes. Some rules have only one pattern, others have several patterns. In

case of rules with one pattern, the node that follows the alpha-mem node is p-mem node (a node corresponding to each rule). In the second case, the node that follows the alpha-mem node is an AND-node which test inter-tests (conditions that involve more than one pattern, see figure 7)

In figure 7, two AND node follows α_2 , AND1 is it's left AND-node, AND2 is it's right AND-node, α_2 also has a p-mem node as a next node. The structure used to define the alpha-memory consist of:

Id: A unique identifier for the node.

Data: A list of tuples that are stored by the node.

R-ands: A list of right AND-nodes linked to the alpha-mem node.

L-ands: A list of left AND-nodes linked to the alpha-mem node .

P-mems: A list of possible p-mem nodes linked to the alpha-mem node.

Prev-node: A pointer to the previous node.

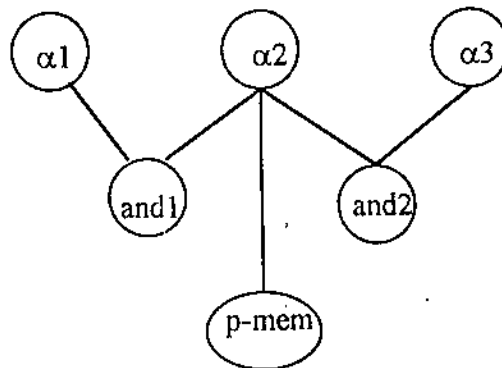


Figure 7: The nodes that follow alpha-mem nodes

5- AND-node: An AND-node has two input memory nodes and one output memory node, An AND-node does the following:

- 1- Matches the two incoming assertions to see if they satisfy the corresponding inter-test.

- 2- If the input assertion pass the test, the AND-node joins the two assertions in one compound tuple (assertion) and stores it in the output memory node.

The structure used to define an AND-node consists of:

Id: Unique identifier.

Check1: Inter-test between two assertions.

L-mem: Left input alpha-mem node.

R-mem: Right input alpha-mem node.

O-mem : Output alpha-mem node.

6- P-mem node: There is a corresponding p_mem node for each rule. All assertions that pass all the tests related to the rule are found in the alpha memory node which is located immediately before the p-mem node. Those compound tuples are ready to instantiate the body of the rule if they pass the external function test associated with the trigger part of the rule (see section 3.1.2.1). These tuples are called justification tuples "A tuple in a p-mem corresponds to a ground instant of a rule" (Hindi 1994). These justification tuples are used to instantiate the body of the rule using the variable locations that are generated during the compilation of the rules. Consider for example the rule:

```
(rule RI
  ((father !f ?x !s ?y)
   (brother !b1 ?x !b2 ?z))
==>
  (Reassert! (uncle !u ?z !m?y))
```

In order to instantiate the pattern found in the body of the rule (uncle !u ?z !m ?y), the variables in the pattern (?z and ?y) must be assigned values. The values must be taken from the trigger part. In order to know where to find the corresponding values for each variable we must generate the variable locations for this rule and store it in the p-

mem node, so it can be used in the reasoning algorithm, the variable locations for this rule are:

$(?x \Rightarrow !f)$ or $(?x \Rightarrow !b1)$; ?x is found in the field !f or !b1

$(?y \Rightarrow !s)$; ?y is found in the field !s

$(?z \Rightarrow !b2)$; ?z is found in the field !b2

The structure used to define the p-mem consists of:

Rule-name: The rule name corresponding to this p-mem node.

Inter-test: External function used to test some restriction over all the patterns in the trigger part.

Consequence: The body of the rule. (uninstantiated)

Var-loc: A list containing all the variables in the trigger part and in which field they can be found.

7- **Instance:** Using the justification tuples and variable locations from the p-mem node, we can now generate the binding environment that will be used to assign the right values for each variable in the body of the rule. For example, In the previous example the only variables found in the body are ?z and ?y, so we need to generate a binding environment that consists of the ?z and ?y variables, this binding environment is:

$((?z = \text{value}(!b2) ; \text{the value of } ?z \text{ equal the value of the field } !b2$

$(?y = \text{value}(!s) ; \text{the value of } ?y \text{ equal the value of the field } !s)$

A rule instance is an instantiation of the rule. i.e. after substituting a value for each variable in left and right hand sides.

The structure used to define instance consists of :

Rule-name: The name of the rule to which the instance correspond to.

Consequence: The body of the rule (uninstantiated).

Vars: A list of all variables in the body.

Binding: An environment generated from the trigger part.

An instance is inserted in the conflict set in order to be processed by the conflict resolution strategy. All assertions generated by firing the instantiated rules are stored in a queue. Elements in the queue will be inserted in the root node in order to be processed later by the reasoning algorithm.

8- Datum: is the corresponding structure for the assertion(tuple). A Datum's structure consists, only at this stage, of the assertion that it represents, and a list of alpha memory nodes where it can be found. The structure will become more complex as we consider other algorithms.

9- Rete Production System (REPS): A REPS is represented as a structure that encapsulates all nodes and subsystems used in the production system (e.g. ATMS, Rete). The component of the structure used to define the production system are:

Title: The name of the problem solver.

ATMS: A pointer to the ATMS associated with the problem solver.

Rete: A pointer to the first node in the Rete network.

Derived: A special memory to hold all the tuples that can not be inserted in any alpha memory nodes (because they are not used in the Rete).

Alpha: A list of alpha memory nodes that contain a single datum.

Conflict-set: A place where all instantiated rules are stored.

Queue: A place where all unprocessed tuples are stored.

3.2.3 The Rete Building Algorithm.

In this section we will discuss the algorithm used to build the Rete network (i.e. the algorithm that compile the rules into a corresponding Rete network). This algorithm takes a list of rules that constitute the problem solver, and translates them into nodes in

the Rete network (see build.lisp in appendix 1.3 for complete lisp implementation for building algorithm), for example the following rule:

```
(defun test1 (x1 x2) (> x1 10))

(defun test2 (x1 x2 x3 x4 x5) (not (= x1 x4)))

(Rule R1

(( Cond1 !x 1 !y ?c1 !m ?c1)

 ( Cond2 !z ?c1 !w ?c3) :test #'test1) :test #'test2

==> (Rassert! (loc !!1 ?c1 !!2 ?c3)))
```

In figure 8, the corresponding Rete for the previous rule is given. In figure 9, the output of Rete building algorithm is given.

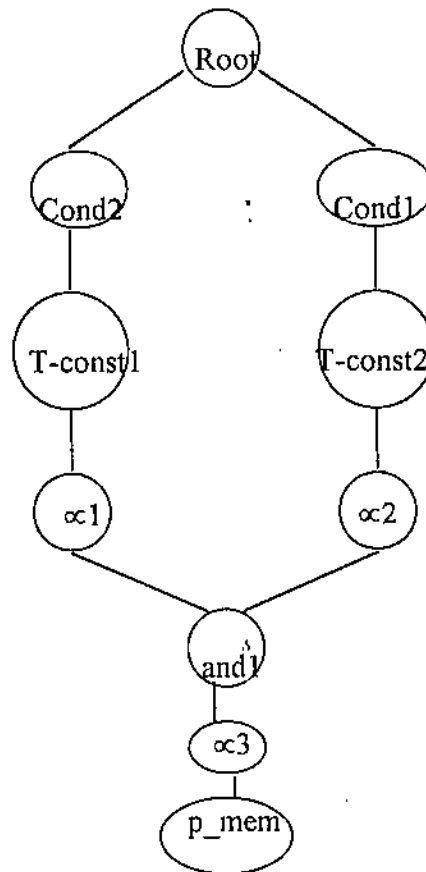


Figure 8: Rete Network

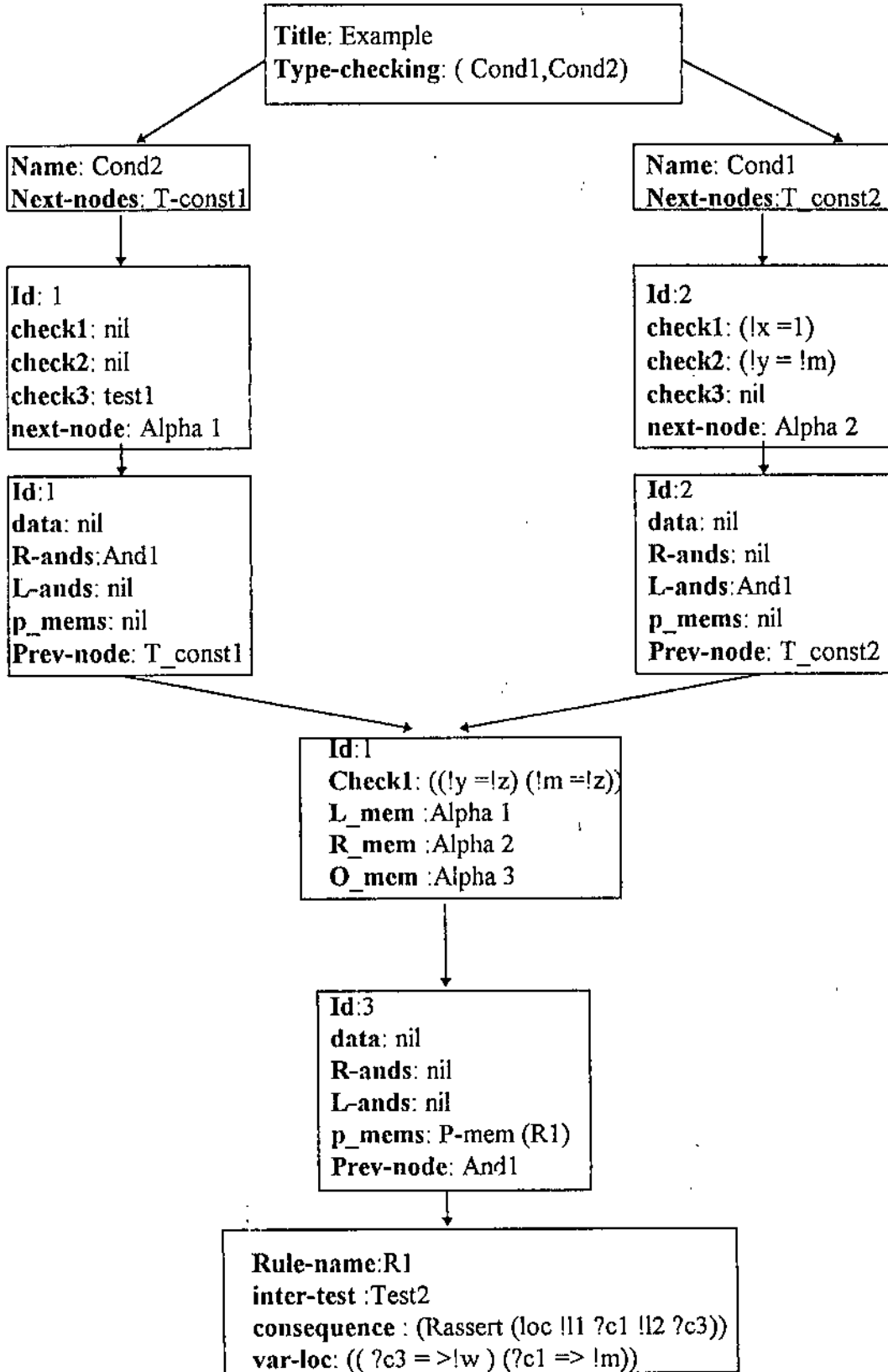


Figure 9: The Output of Rete building algorithm

In this algorithm we will use *RULES* to refer to the rules of the problem solver, *ALPHA* to refer to an alpha memory node, *T-CONST* to refer to a t-const node, *AND* to refer to an AND-node, and *TCHECK* to refer to a type-checking node. This algorithm is described below:

build-Rete (RULES)

create the root node, ROOT

for each rule, R, in RULES do

let ANTECEDENTS = a list of all patterns in the trigger part of rule, R.

call process-antecedents (ANTECEDENTS, ALPHA); result will be in ALPHA

create p-mem node and attach it to ALPHA.

endfor

end build-Rete

The procedure *build-Rete* simply calls the procedure *process-antecedents* for each rule of the problem solver. *Process-antecedents* creates the Rete network corresponding to given rule, augments it with the main network, and returns the last alpha memory node created for it. Before processing the next rule, *build-Rete* creates the p-mem node and attaches it to the last alpha memory node generated by *process-antecedents*.

Next, we will consider *process-antecedents* in details.

process-antecedents (ANTECEDENTS, ALPHA)

for each pattern, A, in ANTECEDENTS do

call create-tch-&-t-const (A, ALPHA1); the result will be in ALPHA1

push ALPHA1 onto a temporary stack, L

endfor

let ALPHA = first alpha in L

pop alpha from the stack, L

while L is not empty do

call anded (first alpha in L , ALPHA) ; the result will be in ALPHA

pop alpha from the stack, L

endwhile

end process-antecedents

This procedure *process-antecedents* calls two procedures. First, it calls *create-tch&t-const*, which takes one antecedent from the list of all antecedents of the rule under examination, creates the necessarily type-checking and t-const nodes, and returns the last alpha memory node created for it (ALPHA1). Then, it calls the procedure *anded* which takes the first two alpha memory nodes from the list, L, joins the two alpha nodes using AND node, and returns the output memory (ALPHA) of the AND node. This output memory is then used by *process-antecedents* to join with the next alpha memory node in the list, L. This process continues until all alpha memory nodes in the list, L, are anded.

Next , we will consider the *create-tch&t-const* in details.

create-tch&t-const(A ,ALPHA)

let NAME = the name of the antecedent, A. (first symbol)

search for a previously created type-checking node with the same NAME.

if it exists

let TCHECK = the old type-checking node

else

create a new type-checking node

```

    let TCHECK = the new type-checking node
    attach TCHECK to the ROOT node

endif

let CHECK1 = generate constant test(A)
let CHECK2 = generate variables equality test(A)
let CHECK3 = the external test function associated with this antecedent
if CHECK1, CHECK2, and CHECK3 are all empty then
    if there exists an alpha memory node associated with TCHECK
        let ALPHA = the old alpha memory node
    else
        create a new alpha memory node, ALPHA
        attach ALPHA to TCHECK.
    endif
else { CHECK1, CHECK2, or CHECK3 are not empty }
    look for a previously created t-const node which perform the same tests and
    attached to TCHECK
    if such a node exists
        let ALPHA = the alpha memory node attached to the old t-const node
    else
        create a new t-const node
        let T-CONST = the new t-const node
        create a new alpha memory node for T-CONST
        set ALPHA = the new alpha memory node.
    endif
endif

```

endif

end create-tch&-t-const

The procedure *create-tch&-t-const*, analyzes the antecedent under consideration carefully, in order to find if there already exists a suitable type-checking node and t-const node, if so; it will return the old ones. (to maximize the sharing between the rules). If no suitable nodes were found then a new nodes are created.

anded (ALPHA1 , ALPHA2)

let INTER-TEST = generate inter-test between ALPHA1 , ALPHA2

look for an AND-node associated with ALPHA1 and ALPHA2, and have the same INTER-TEST

if such a node exists

let ALPHA2 = the output memory of the old AND-node

else

create AND-node

let ALPHA2 = the output memory of the new AND-node

endif

end anded

This procedure simply takes two alpha memory nodes as input, looks for an existent AND-node which perform the same inter-test needed between the two input nodes (ALPHA1, ALPHA2). if such a node doesn't exist, the procedure creates a new one and returns the output memory for the and node, otherwise it returns the output memory of the old one.

3.2.4 The Reasoning Process

The reasoning process will start immediately after the Rete network is completely build, it begins by forming a queue, QUEUE, that contains all the initial working memory

element (WME) (the initial assertions that represent simple facts of the problem). Then each WME is dequeued and inserted into Rete to determine the conflict set. This process continues until all WME's have been dequeued and matched by Rete i.e. until QUEUE is empty.

The conflict set is then examined to select a rule to fire (resolve conflicts). We use a simple conflict resolution strategy; we choose the last rule entered the conflict set i.e. last in first out (LIFO). Firing the chosen rule may result in a new assertion to be inserted into (matched by) Rete, which in turn may affect the conflict set. Then a new rule is chosen from the conflict set and fired. This cycle continues until the conflict set becomes empty, at that point the reasoning process is complete (figure 10 illustrates the process).

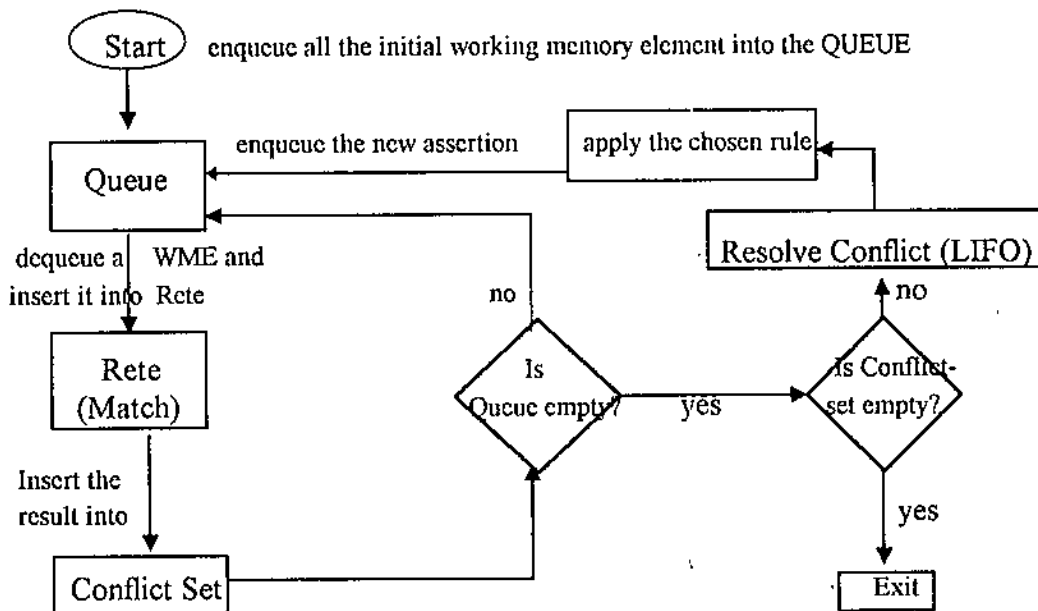


Figure 10 : The Reasoning Process

The reasoning algorithm is given below: (see reason.lsp in appendix 1.4 for lisp implementation for the algorithm).

Let QUEUE refers to the queue used to store all the unprocessed assertions, CONFLICT-SET refers to the conflict set.


```

start-reasoning (QUEUE)

  let CONTINUE = true

  while CONTINUE do

    while QUEUE is not Empty do

      let DATUM = dequeue (QUEUE)

      call process-datum (DATUM)

    endwhile {QUEUE not empty}

    let CONTINUE = false

    while CONFLICT-SET is not Empty do

      INST = pop(CONFLICT-SET)

      fire the instance INST

      if a new datum is inserted in the QUEUE

        let CONTINUE = true

        exit from the while loop

      endif

    endwhile {conflict set not empty}

  endwhile

end start-reasoning.

```

Next, the procedure *process-datum* which process each datum enter the Rete network is described below.

```

process-datum (DATUM)

  let TCHECK = the corresponding type checking node for DATUM

  if there is no TCHECK for DATUM

    push DATUM in the derived memory if the DATUM does not exist previously

```

```

    return
endif

for each node, NODE, that follows TCHECK do

    if type(NODE) = alpha memory then

        if DATUM not in NODE then

            insert DATUM in NODE

            call process-alpha(NODE, DATUM)

        endif

    else { type(NODE) = t-const node }

        if DATUM not in the alpha memory associated with NODE then

            if DATUM satisfy all tests in NODE then

                let ALPHA = alpha memory associated with NODE

                insert DATUM in ALPHA

                call process-alpha(ALPHA, DATUM)

            endif

        endif

    endif ; {type(NODE)}

endfor

end process-datum

```

The procedure *process-datum*, searches for a type-checking node for the input datum, if there is no such node, this means that this datum is a derived datum not used in the reasoning process, therefore, it must be stored in a special place for derived assertions (the derived field in REPS).

The type of the node that follows the type-checking node can be an alpha memory node or a t-const node (look for the definition of the type-checking structure). When the

next node is an alpha memory node, the datum must be inserted in the memory if it was not there. In the other hand, if the next node was a t-const node, the datum can be inserted in the alpha memory of the t-const node only if the datum passes all the tests and was not already stored in the alpha memory node.

process-alpha (ALPHA , NEW-TUPLES)

if there exist left AND nodes associated with ALPHA then

call process-l-ands (ALPHA, NEW-TUPLES)

endif

if there exist right AND nodes associated with ALPHA then

call process-r-ands (ALPHA, NEW-TUPLES)

endif

if there exist p-mem nodes associated with ALPHA then

call process-p-mem (ALPHA, NEW-TUPLES)

endif

end process-alpha

The procedure *process-alpha*, calls the suitable procedure depending on the node that follows it, which could be left AND-node, right AND-node or p_mem node.

process-l-ands (ALPHA, NEW-TUPLES)

for each left ands, LAND associated with ALPHA do

call join(NEW-TUPLES, tuples in the other input memory for LAND, NEW-JOINED-TUPLES)

if NEW-JOINED-TUPLES is not empty

insert NEW-JOINED-TUPLES in the output memory of the LAND

call process-alpha (output memory of LAND , NEW-JOINED-TUPLES)

endif

endfor

end process-l-and

Procedure *process-l-and* joins the new tuples with all tuples that exist in the other input of the AND-node if they pass the inter-test in the AND-node under consideration. All the new joined tuples are inserted in the output memory of the AND-node. Finally, the procedure call *process-alpha* to process the new joined tuples. The procedure *process-r-and* is the same as *process-l-and* procedure with some minor modifications (use right AND instead of left AND).

process-p-mem (ALPHA, NEW-TUPLES)

for each p-mem, P-MEM, associated with ALPHA do

for each datum, DATUM, in the NEW-TUPLES do

if the DATUM satisfy the external function test then

create instance and push it to the conflict set

endif

endfor { for each datum }

endfor { for each p_mem }

end process-p-mem

Procedure *process-p-mem* checks each new datum to see if it satisfies the external test function (if it exists), if the datum satisfies the test, it will used to create a new instance that will be pushed in the conflict set.

join (LTUPLES, RTUPLES, RESULT)

for each LTUPLE in the LTUPLE do

for each tuple, RTUPLE in RTUPLES do

```

    if LTUPLE and RTUPLE satisfy the join test
        join the two tuples and insert the joined tuple in RESULT
    endif
endfor {for each RTUPLE}
endifor {for each LTUPLE}
end join

```

Procedure *join* joins each tuple in one set with all the tuples in the other set. If the joined tuple passes the inter-test between the two tuples it will be stored in the output memory node, *RESULT*.

3.3 ATMS : Structures and Algorithms

ATMS performs the following basic operations:

1. Creating a node corresponding to each datum in the problem solver.
2. Adding a new justification to the dependency network.
3. Maintaining the labels of nodes and handling nogood environments.

3.3.1 Structures

We represent ATMS, nodes, justifications, and environments as structures. These are described below (de Kleer, 1986 a).

1- ATMS structure

The ATMS structure is used as a place to hold all the component of the ATMS, the structure used to define the ATMS is (see *ATMS.lsp* in appendix 2.5 for complete lisp implementation for ATMS algorithm):

Nodes: a list that contains all the nodes in the ATMS.

Justs: a list that contains all the justifications installed in the dependency network.

Good-env: a list that contains all the good environments.

Nogood-env: a list that contains all the nogood environments.

Contra-node: a contradiction node used by contradiction rules.

2- Node structure

ATMS creates a node in the ATMS for each datum in the problem solver, and each node has the following structure:

Datum: a pointer to the corresponding datum.

Label: a list of environments, this node holds in.

Justs: a list of justifications that support the node.

Consequences: A list of justifications, this node belongs to it's antecedents.

3-Justification structure

Justification describes the dependency between the derived nodes and nodes used to infer them (figure 11 shows the relation between nodes and justifications) . Each justification consists of :

Antecedents: a set of nodes used in deduction.

Consequence: a node that follows from the antecedents.

Informant: A string describes the deduction.

4- Environment structure

Each label consists of a set of environments, each environment consists of a set of assumption nodes, and each environment records all nodes that it is label contains this environment. The structure used to define the environment is:

Assumptions: a list of all the assumption nodes belong to this environment.

Nodes: a list of all nodes containing this environment in their label.

From the previous structure's definitions we can see how all components of the ATMS are related to each other, we summarize these relations below:

- Each node stores the label.
- Each label consists of a set of environments.
- Each environment is a set of assumption nodes.
- Each environment records all the nodes that have this environment in its label.
- Each node records all the justification that use this node.
- Each justification stores the antecedents and the consequence nodes.
- Nodes, justifications, and environments are stored in the ATMS structure.

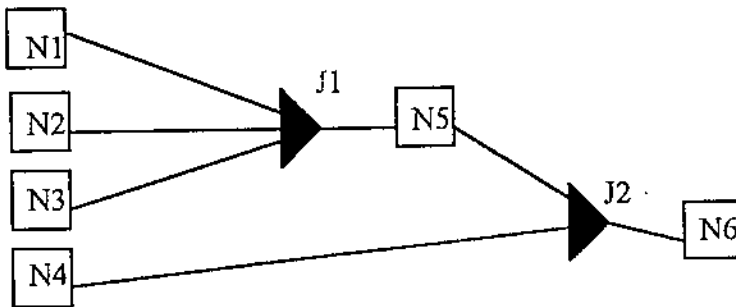


Figure 11 : The Dependency Network
 N1, N2, N3 and N4 are assumption nodes.
 N5 and N6 are derived Nodes.
 J1 is a justification installed on N5.
 J2 is a consequence justification for N5.

3.3.2 Label update and Nogood handling

The most expensive operations done by the ATMS are label update and nogood handling.

1- Label update:

The label update starts whenever a new justification is installed for a node, this operation is described as follows:

1- Calculate the label of the consequence of the justification by:

- Find a list of all environments that can be formed by union one environment from each antecedent node's label (to make it complete).

- Remove from the new list (formed in previous step) any environment that is a superset of any environment that exists in the new set (to make it minimal).
 - Remove all nogood environments from the new list (to make it sound).
- 2- If the consequence node of the justification is new (have not been already derived), then
- stop {label update is complete}
- else,
- Compute the label of the this node, take into account the old label for the node.
 - If the new label is the same as the old one, then
 - stop { no need to update the label for any consequence nodes }
 - else,
 - Look for all the consequence justifications for the consequent node and repeat the algorithm for each of them.

2- Nogood handling :

Whenever a new justification is installed for the contradiction node (i.e. a new nogood environment have been found), the following operations must be done:

- 1-Compute the new label for the consequent of the justification.
- 2-All the environments generated from step 1 become nogood and must be moved to the nogood-env list in ATMS.
- 3-Remove any of these nogood environments from the labels of nodes that mention them (recall that these nodes are recorded in the environment structure).
- 4-Find all the environments in the ATMS's good-envs list that are a superset of one of the new nogood environments, these are also nogood environments, so steps 2 and 3 must be repeated for each of them.

3.4 Loose coupling of ATMS and production system.

The main points in this algorithm are: (see section 2.4.1 for more details)

1. Fire all contradiction rules first, so that nogood environments are discovered as soon as possible.
2. The conflict resolution will prevent any instant with empty label for it's consequence from firing.

According to the previous points, we must have the ability to distinguish between a rule instance that corresponds to a contradiction rule from a rule instance that corresponds to a normal rule. To be able to do that, we have add to the definition of the p-mem and instance's structure a new boolean field called *contradiction?*, this field is true only if the instance or the p-mem corresponds to a contradiction rule. To have the ability to compute the label of the consequence of the instance, a new field *antecedents* is added to the instance's structure, which contains the tuples that are used to create this instance.

The Rete building algorithm which was discussed in 3.2.3, needed to be slightly modified so it can set the correct value for *contradiction?* field during the build process. If the created node corresponds to a contradiction rule then building algorithm will set this field to TRUE, else set it to FALSE.

The reasoning strategy and the conflict resolution method of the reasoning algorithm which was described in section 3.2.4 need to be modified. These modifications are described below.

Two things in the reasoning algorithm must be changed: first, before we apply any of the normal rules, we have to fire all the instances of contradiction rules. Second, the conflict resolution must select the last (in) rule instance in the conflict set which has a

non-empty label (rule instance with empty labels need not be fired). Figure 12, illustrates the reasoning process using the loose coupling approach.

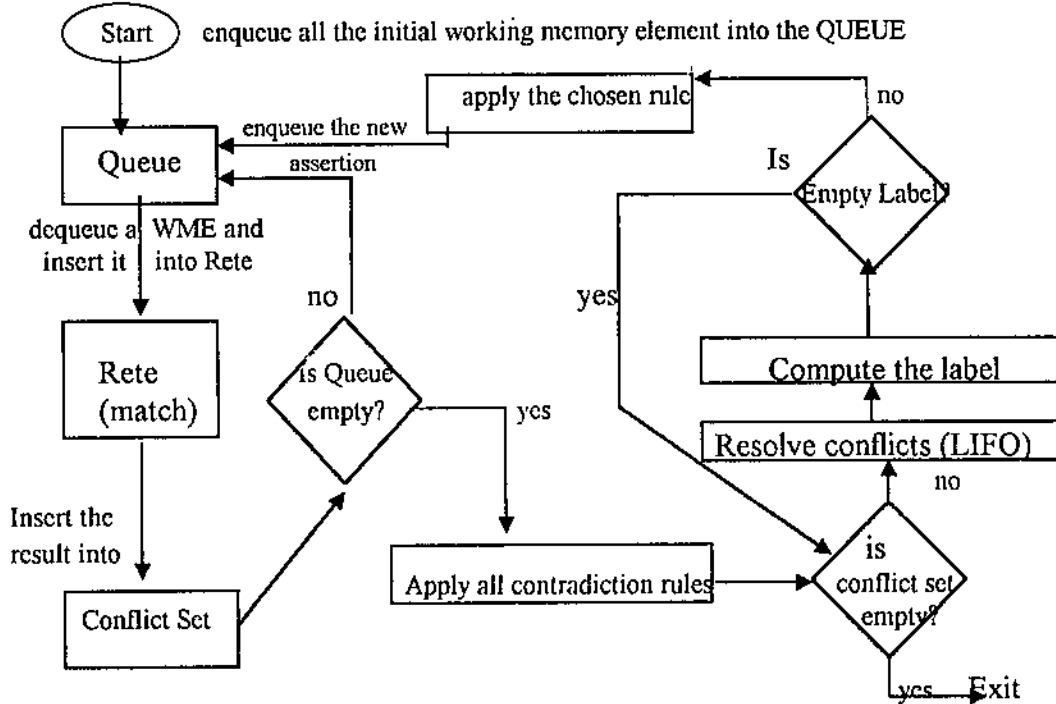


Figure 12 : The Reasoning Process in Loose Coupling

The following algorithm described the reasoning process used by the loose coupling approach in more details (see appendix 2 for complete lisp implementation for loose coupling algorithm):.

start-reasoning (QUEUE)

let CONTINUE = true

while CONTINUE do

while QUEUE is not Empty do

let DATUM = dequeue (QUEUE)

call process-datum (DATUM)

endwhile

```

act all instance 's of the contradiction rules that are in the conflict set

let CONTINUE = false

while CONFLICT-SET is not Empty do

    call conflict-resolution(CONFLICT-SET, INST)

    fire the instance INST

    if a new datum is inserted in the QUEUE

        let CONTINUE = true

        exit {from the while loop}

    endif

endwhile {conflict set is not empty}

endwhile {continue}

end start-reasoning

conflict-resolution (CONFLICT-SET, INST)

while CONFLICT-SET is not empty do

    let INST = pop (CONFLICT-SET)

    let LABEL = the label of the consequence node in INST

    if LABEL is not empty then return

    endwhile

end conflict-resolution

```

Finally, we can imagine the loose coupling system as two separated systems; a Rete-based production system and an ATMS. They are communicating with each other in three events; these are summarized in table 1.

| # | Event | Production system Action | ATMS Action |
|---|---|---|---|
| 1 | <ul style="list-style-type: none"> • apply a contradiction rule. | <ul style="list-style-type: none"> • send the corresponding justification to ATMS. | <ul style="list-style-type: none"> • find the new nogood environments. • remove those environments from the labels. |

| # | Event | Production system Action | ATMS Action |
|---|--|--|--|
| 2 | <ul style="list-style-type: none"> • conflict resolution. | <ul style="list-style-type: none"> • ask the ATMS to compute the label of the consequence of the rule instance. | <ul style="list-style-type: none"> • compute the label and return the result. |
| 3 | <ul style="list-style-type: none"> • apply a normal rule. | <ul style="list-style-type: none"> • send the corresponding justification to ATMS. | <ul style="list-style-type: none"> • install the justification in the dependency network. • update the label of all the related nodes. |

Table 1 : ATMS and Production system Interaction

3.5 Tight coupling the ATMS and production system: Morgue System.

The main points in Morgue system can be summarized as follows (see section 2.4.2 for a detail discussion of the Morgue system):

1. Compute the label of each datum in a Rete memory node.
2. Remove any datum with empty label from memory nodes.
3. Execute contradiction rules as soon as they are instantiated (i.e. become applicable).

From the previous points, we can make the following observations:

1. All responsibilities of the ATMS are integrated within the Rete-based production system.
2. The labels are associated with tuples rather than nodes.
3. A justification is installed in the Rete-based production system instead of the ATMS.

So we have to make some modifications on some structures used in loose coupling approach, These modifications are listed below:

- 1- Add a new field to all Rete nodes, called *contradiction?*, this field will be set to "true" if this node is shared by a contradiction rule, so that we can give these nodes high priority during matching process in order to fire contradiction rules as soon as they are instantiated.

2- Use *datum* instead of *node* in all structures used in the ATMS.

3- The datum structure is redefined as :

Fact: The assertion that this datum represents.

Assumption?: A Boolean field which indicate whether or not the corresponding datum is an assumption.

Alpha-list?: A list of all memory nodes, that a copy of this datum can be found in.

Justs: A list of all justifications that support this datum.

Consequences: A list of all justifications that this datum is one of their antecedents

Label: The label of the datum.

Del: Flag to indicate, whether or not this datum must be deleted (because it's label is empty).

For the ATMS, all it's operations (label update, contradiction handling), is done directly on the tuples in the production systems.

Some changes must also be made on the Rete building algorithm, these changes are listed below:

1. Set the *contradiction?* field to "true" if this node belongs to (or shared with) a contradiction rule.
2. Rearrange the list stored in the *next-nodes* field, so that the contradictory node appears at the beginning of the list. This is very important because when the reasoning algorithm decides to forward the datum to the next node(s) in the network, this datum will be forward first to the contradictory node (if it exists). In this way, the contradiction rule is given priority to be instantiated and fired before other normal rules.

All these modifications are described in more details by the following algorithm: (see appendix 3 for complete lisp implementation for Morgue algorithm), let QUEUE be a queue for all initial working memory elements.

start-reasoning (QUEUE)

let CONTINUE = true

while CONTINUE do

while QUEUE is not Empty do

dequeue (QUEUE , DATUM)

call process-datum (DATUM)

endwhile {QUEUE is not empty}

let CONTINUE = false

while CONFLICT-SET is not Empty do

call conflict-resolution(CONFLICT-SET, INST) {to select a rule instance}

fire the rule instance INST

if as a result a new datum is derived enqueue it in QUEUE

let CONTINUE = true

exit from the while loop

endif

endwhile {conflict set is empty}

endwhile {CONTINUE}

end start-reasoning

In *start-reasoning* procedure, nothing is mentioned about contradiction rules, and that is because they are fired as soon as they are instantiated, and will not be stored in the conflict set. The conflict resolution strategy select the last rule instance enter the

conflict set, note that there is no need to check the label of the consequence because it can not be empty.

The procedure *process-datum* deals with two cases: First, if there is no type checking node for the given datum, in this case the datum must be inserted in a special memory node called derived memory if it is not already stored there. If it is already stored in derived memory we just update its label. Second, if there is a corresponding type checking node, then we have to check the datum against the four cases described in table 2. This is done by the procedure *check-single-tuple*. Note that the tests in a t-const node will not be reexamined if the datum already exists in the memory node.

process-datum (DATUM)

```

let TCHECK = the corresponding type checking node for DATUM
if there is no TCHECK for DATUM
    call check-single-tuple (DATUM, DERIVED-MEM)
    return
endif
for each node, NODE, that follows TCHECK do
    if type(NODE) = alpha memory then
        call check-single-tuple (DATUM, ALPHA)
    else { type(NODE) = t-const node }
        let ALPHA = the alpha memory node associated with NODE
        if DATUM is in ALPHA then call check-single-tuple(DATUM,ALPHA)
        if DATUM is not in ALPHA and it satisfies all tests of NODE then
            call check-single-tuple (DATUM, ALPHA)
        endif
    endif
endfor

```

endif

endfor

end process-datum

When a datum is to be matched (i.e. inserted into Rete), the Morgue system must take into account 4 different cases; these cases are summarized in table 2 along with the necessary action to be taken. These cases are dealt with by *check-single-tuple* and *check-joined-tuple*.

| # | Cases | Action |
|---|---|---|
| 1 | <ul style="list-style-type: none"> • The given datum exists in a memory node and • the new label of the datum is the same as the old label. | <ul style="list-style-type: none"> • justify the datum i.e. link it to the corresponding justification |
| 2 | <ul style="list-style-type: none"> • The given datum exists in the memory node and • the new label of the datum differs from the old label. | <ul style="list-style-type: none"> • justify the datum. • update the label of the old datum. • propagate through the Rete network for label update. |
| 3 | <ul style="list-style-type: none"> • The given datum does not exist in any memory node and • its label is not empty. | <ul style="list-style-type: none"> • push the datum in the corresponding memory node. • justify the datum. • propagate through the Rete network for join and label update. |
| 4 | <ul style="list-style-type: none"> • The given datum does not exist in any memory node and • its label is empty. | <ul style="list-style-type: none"> • do nothing |

Table 2: Four cases in The Morgue system

check-single-tuple (DATUM, ALPHA)

if DATUM already exists in ALPHA then { i.e. old datum }

let NEW = new environments generated after computing the new label for DATUM, and do not exist in the old label of DATUM.

justify DATUM


```

if NEW is not empty then
    update the label of the old datum
    call process-alpha(ALPHA, DATUM)
endif
else {new datum}
    if the DATUM label is not empty then
        insert DATUM in ALPHA
        justify DATUM
        call process-alpha(ALPHA, DATUM)
    endif {DATUM label is not empty}
endif {DATUM already exists}
end check- single-tuple

```

Procedure *process-alpha* propagate the datum that reaches an alpha memory node through the remaining part of Rete network. Following an alpha memory node there could be p-mem nodes, contradiction left AND nodes, contradiction right AND nodes, normal left AND nodes, and normal right AND nodes. In order to give contradiction rules more priority over normal rules; *process-alpha* forwards the datum first to the nodes that correspond to contradiction rules before nodes that correspond normal rules.

process-alpha (ALPHA , NEW- TUPLES)

call *process-p-mem* (ALPHA, NEW- TUPLES)

call *process-ands* (ALPHA, Contradiction-left AND nodes , NEW- TUPLES)

call *process-ands* (ALPHA, Contradiction-right AND nodes, NEW- TUPLES)

call *process-ands* (ALPHA, Left AND nodes, NEW- TUPLES)

call *process-ands* (ALPHA, Right AND nodes, NEW- TUPLES)

end process-alpha

Procedure *process-ands* is the same as the procedure *process-l-ands* described in section 3.2.4, it joins each tuple in NEW-TUPLES with each tuple stored in the other input memory for each AND node (listed as the second argument), all joined tuples that pass the inter-test will be stored in the output memory node (if they not already there) for the AND node, and call *process-alpha*.

Procedure *process-p-mem* checks weather *p-mem* node corresponds to a contradiction rule or not, if so, the rule instance must be fired, otherwise, the procedure stores it in the conflict set.

process-p-mem (ALPHA, TUPLES)

for each p-mem, P-MEM associated with ALPHA do

for each datum, DATUM in the NEW- TUPLES do

if DATUM satisfy the external test function then

if P-MEM corresponds to a contradiction rule then

create a rule instance, INST

call act-contradiction-rule (INST)

else

create instance and push it into the conflict set

endif { p-mem correspond }

endif {DATUM satisfy }

endfor {each DATUM}

endfor {each p-mem}

end process-p-mem

Procedure *JOIN*, takes two list of tuples, and join each tuple in the first list with all tuples in the second list, for each joined tuple the procedure calls *check-joined-tuple* to

check the joined tuple against the join test and against the four cases described in table 2 and takes the corresponding action for each of them.

join (LTUPLES ,RTUPLES ,ALPHA, RESULT)

for each datum, LDATUM in the LTUPLES do

for each datum, RDATUM in RTUPLES do

call check-joined-tuple (LDATUM, RDATUM, ALPHA, RESULT)

endfor

endfor

end join

check-joined-tuple (LDATUM, RDATUM, ALPHA, RESULT)

let LABEL = compute the label of the joined-tuple-of(LDATUM, RDATUM)

if joined-tuple-of(LDATUM, RDATUM) exists previously in ALPHA then

let NEW = new environments exists in LABEL and does not exist in the old label for the joined tuple.

justify DATUM

if NEW is not empty then

update the label of the joined tuple

insert the joined tuple in RESULT

endif

else {new datum}

if (LDATUM, RDATUM satisfy the join test)and (LABEL is not empty) then

join the two tuples and insert the joined tuple in RESULT

justify the new compound-tuple

endif

endif

end check-joined-tuple

When a contradiction rule is fired, new nogood environments may be discovered, therefore the following operations must be done:

1. The nogood handling process in the ATMS must be started (see section 3.3.2 for more details about nogood handling).
2. For each tuples, TUPLE, that its label become empty as a result of contradiction handler, the following operation must be done:

for each alpha memory node, ALPHA, that have a copy of TUPLE, (these memory nodes can be found in *alpha-list* field in the datum's structure) follow the consequent justification for TUPLE, and delete linked tuples that have an empty label, and then delete the TUPLE itself.

3.6 Tight coupling the ATMS and production system: the Hindi System.

As discussed in section 2.4.3, the main ideas in the Hindi system is: store tuples that its label is empty in a special inactive part (called *OUT* part) of the same Rete memory node, and tuples with non-empty label in an active part of the memory node (called *IN* part). The Hindi system uses time stamp to decide what tuples to use in the join operation.

When a datum is to be matched (i.e. inserted into Rete), the HINDI system must take into account six different cases; these cases are summarized in table 3 along with the necessary action to be taken. These cases are dealt with by into *check-single-tuple* and *check-joined-tuple*.

| # | Cases | Actions |
|---|--|--|
| 1 | <ul style="list-style-type: none"> • The given datum is new (does not already exist in a memory node) and • its label is empty. | <ul style="list-style-type: none"> • store the new datum in the OUT part of the memory node • stamp the datum with time stamp -1. |
| 2 | <ul style="list-style-type: none"> • The given datum is new and • its label is not empty. | <ul style="list-style-type: none"> • store the new datum in the IN part of the memory node. • stamp the datum with the current time. • this datum must be joined with all tuples stored in the IN part of the other input memory of the AND-node. • justify the datum. • propagate through the Rete for join and label update. |
| 3 | <ul style="list-style-type: none"> • The datum is old (already exists) and • the old datum stored in the IN part of the memory node and • the new label differs from the old label. | <ul style="list-style-type: none"> • update the label of the old datum • justify the datum. • this datum will not be used in the next join operations. • propagate through the Rete for label update. |
| 4 | <ul style="list-style-type: none"> • The given datum is old and • the old datum stored in the IN part of the memory node and • the new label is the same as the old label. | <ul style="list-style-type: none"> • justify the datum. |
| 5 | <ul style="list-style-type: none"> • The datum is old and • the old datum stored in the OUT part of the memory node and • the new label is not empty. | <ul style="list-style-type: none"> • move the datum to the IN-part of the memory node. • update the label of the old datum • justify the datum. • join this datum with all tuples that its time stamp \geq the time stamp of this datum¹ and stored in the IN part of the other input memory of the AND-node. • propagate through the Rete for join and label update. |

¹ The Hindi system assert that we need to join the datum with all tuples stored in the IN part of the other input memory of the AND-node whose its time stamp is greater than the time stamp of this datum (Hindi-1994). This will prevent the datum from being joined with other tuples stored in the IN part of the other memory of the AND node that have time stamp equal to the time stamp of this datum, this will lead to uncompleted result. In our implementation we join with all tuples that have time stamp greater than or equal the time stamp of the tuple.

| | | |
|---|--|--|
| 6 | <ul style="list-style-type: none"> • The datum is old and • the old datum stored in the OUT part of the memory node and • the new label is empty. | <ul style="list-style-type: none"> • do nothing |
|---|--|--|

Table 3: Six cases in the Hindi system

So we have to make some modifications on the structures and algorithms used in MORGUE system, these modifications are listed below:

- 1- We use the field *del* in the datum's structure to distinguish between tuples stored in the OUT part and tuples stored in the IN part. If a tuple is stored in the OUT part of a memory node then its *del* field value must be set to TRUE, otherwise set it to FALSE (this field was also used in Morgue system to indicate that this datum must be deleted).
- 2- We add to the datum's structure a new field to record the time stamp called *stime*.
- 3- The join operation deal with three cases: join the tuple with all other tuples in the IN part of the other input memory of the AND node, join the tuple with all other tuples in the IN part of the other input memory that have time stamp greater than or equal to the datum's time stamp, the third case is when no join operation is needed. To reflect these cases in the datum's structure we add a new field called *jtime*, which has three cases:
 - if $jtime = 0$, then no join operation is needed for this datum.
 - if $jtime = -1$, then join the datum with all the tuples stored in the IN part of the other input memory of the AND-node.
 - if $jtime \diamond (0 \text{ or } -1)$, then join the datum with all tuples stored in the IN part of the other input memory of the AND-node, that has time stamp greater than or equal to the datum's time stamp.

In order to apply the previous modifications, we can rewrite some procedures (from the Morgue's procedures discussed above), the modified procedures are discussed below (see appendix 4 for complete lisp implementation for Hindi algorithm):

Procedure *conflict-resolution* look for the first instance in the conflict set where it's antecedents (justification tuple) stored in the IN part of the memory node.

conflict-resolution (CONFLICT-SET, INST)

while CONFLICT-SET is not empty do

let INST = pop (CONFLICT-SET)

if IN(INST) then return {instance antecedents stored in the IN part}

endwhile

end conflict-resolution

Procedure *process-datum* is not modified here, but obviously, it will not have to reexamine the tests in t-const node even if the datum is in the OUT part of a memory node. *Process-datum* call procedure *check-single-tuple* to check the tuple against the six cases described in table 3 and take the corresponding action.

check-single-tuple (DATUM, ALPHA)

if DATUM already exists in ALPHA then {old datum }

if in(DATUM) then {DATUM stored in the IN part}

let NEW = new environments generated after computing the new label for DATUM, and does not exist in the old label.

justify DATUM

if NEW is not empty then

update the label of the old datum

set jtime in the old datum to 0 { no join operation is needed }

```

    call process-alpha(ALPHA, DATUM)
endif
else {out(DATUM)}
    if the label of DATUM is not empty then
        insert DATUM in the IN part of ALPHA
        update the label of the old datum
        set jtime for old datum to stime(old datum) {join with all IN tuples in the
        other memory which have stime >= stime(old datum)}
        justify DATUM
        call process-alpha(ALPHA, DATUM)
    endif { if the label of ..}
endif { if in(datum) }
else { new datum }
    if the label of DATUM is empty then
        insert DATUM in the OUT part of ALPHA
        let stime(DATUM) = -1
    else { the label is not empty}
        insert DATUM in the IN part of ALPHA
        let stime(DATUM) = current-time
        let jtime(DATUM) = -1 {join with all IN tuples in the other memory}
        justify DATUM
        call process-alpha(ALPHA, DATUM)
    endif { if the label is empty}
endif { if datum already exist }

```


end check-single-tuple

Procedure *process-p-mem* is the same as that used by the Morgue system, except that tuples stored in the IN part only are used to instantiate the rules. Note here, that the external test function is not reexamined for the tuples stored in the OUT part, but in the Morgue you have to reexamine the test because all the tuples with empty labels was deleted.

Join procedure, call *check-joined-tuples* with a suitable flag according to the value of *jtime* field in datum's structure, all joined tuples will be returned in RESULT.

join (LTUPLES ,RTUPLES ,ALPHA, RESULT)

for each tuple stored in the IN part, LDATUM, from LTUPLES do

for each tuple stored in the IN part, RDATUM from RTUPLES do

case jtime(LDATUM)

0:call check-joined-tuple(LDATUM, RDATUM, ALPHA, RESULT,
"nojoin")

-1:call check-joined-tuple(LDATUM, RDATUM, ALPHA, RESULT,
"all")

otherwise

call check-joined-tuple(LDATUM, RDATUM, ALPHA, RESULT,
"part")

endcase

endfor

endfor

end join

Procedure *check-joined-tuple* checks the joined tuple against the join test and the six cases (summarized by table 3), and apply the corresponding action for each case.

check-joined-tuple (LDATUM, RDATUM, ALPHA, RESULT, FLAG)

let LABEL = compute the label of the joined-tuple-of(LDATUM, RDATUM)

if joined-tuple-of(LDATUM, RDATUM), D, already exists in ALPHA then

if in(D) then {D stored in the IN part}

let NEW = new environments exists in LABEL and does not exist in the old label of the joined tuple.

justify D

if NEW is not empty then

update the label of the old datum

set jtime of the old datum to 0 { no join operation is needed }

insert D in RESULT

endif

else {out(D)}

if LABEL is not empty then

insert D in the IN part of ALPHA

update the label of the old datum

set jtime of D to stime(D) {join with all IN tuples in the other memory which have stime \geq stime(D)}

justify D

insert D in RESULT

endif {label is not empty}

endif { in(d)}

else { new datum }

if FLAG = "nojoin" then return

```

if (FLAG = "part" ) and (stime(LDATUM) > stime(RDATUM)) then return
let LABEL = compute the label of the consequence of LDATUM, RDATUM
if (LDATUM, RDATUM satisfy the join test ) then
    let D = the join of the two tuples
    if LABEL is empty then
        insert D in the OUT part of ALPHA
        let stime(D) = -1
    else {label is not empty }
        insert D into the IN part of ALPHA
        let stime(D) = current-time
        let jtime(D) = -1 {join with all IN tuples in the other memory}
        justify D
        push D to the RESULT
    endif {if label is empty }
endif {LDATUM, RDATUM satisfy the join test }
endif {if joined tuple already exist}
end check-joined-tuple

```

Finally, when a contradiction rule is fired, new nogood environments may be discovered, therefore the following operation must be done:

1. The nogood handling process in the ATMS must be started (see section 3.3.2 for more details about nogood handling).
2. For each tuples, TUPLE, that its label become empty as a result of the previous step, the following operation must be done:

move the datum to the OUT-part of the memory node.

stamp TUPLE with *current-time*.

for each alpha memory node, ALPHA, that have a copy of TUPLE, (these memory nodes can be found in *alpha-list* field in the datum's structure)

follow the consequent justification for TUPLE, and move all linked tuples that have an empty label to the OUT part and stamp each of them with current time.
advance the current time.

In the next chapter, we will present an empirical performance study of the three methods discussed in this chapter.

Chapter 4

Performance Evaluation of

The Algorithms

In this chapter, four problems are solved using the three algorithms explained in the previous chapters. These are the queen problem, constraint satisfaction problem, student registration, and a modified student registration problem. We present a performance comparison between the three algorithms. In section 4.1, the queen problem is discussed. In section 4.2, the constraint satisfaction problem is discussed. In section 4.3, a student registration system is discussed. In section 4.4, the problem in 4.3 is modified and discussed again. All experiments were performed using a PC with a Pentium-133 processor and a 32 M-Byte RAM.

4.1 The Queen problem.

The N-queen puzzle is a classic example of combinatorial problems. The problem is: given an $N \times N$ chessboard and N queens, in how many different ways can we place the queens on the board so that none of them can capture any other? (each must be in different column, row, and diagonal). For $N = 4$ there are two solutions illustrated in figure 13, and the number of solutions rapidly goes up with N .

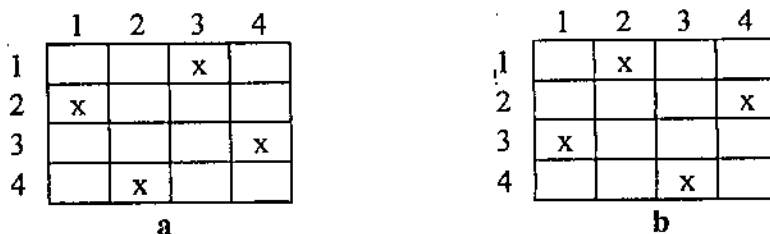


Figure 13 : 4-queens problem solutions

Using ATMS to solve the queen problem can help in minimizing the time needed to find the solution by storing the inconsistent combinations of queen location (nogood environment) in ATMS so that they never tried again.

4.1.1 Rules

we can solve the problem using two rules, one to detect contradictions, and the other to generate all the permutations on N queens in N x N chessboard (see appendix 5.1 for complete lisp implementation for queen problem).

(Contradiction-rule C1

```
((q !x ?r1 !y ?c1) (q !x ?r2 !y ?c2)) :test #'queens-not-ok
==> (rassert-nogood!))
```

(Rule R1

```
((q !x 1 !y ?c1) (q !x 2 !y ?c2) (q !x 3 !y ?c3) (q !x 4 !y ?c4))
==> (rassert! (loc !c1 ?c1 !c2 ?c2 !c3 ?c3 !c4 ?c4)))
```

Note: the field !x mean row and !y mean column.

Initial working memory element: all the locations in 4x4 chessboard are needed as an initial working memory elements (WME), i.e. (q !x 1 !y 1) to (q !x 4 !y 4).

The function “queens-not-ok”, used by the first rule, will take the location of the first queen (?r1, ?c1), and the location of the second queen (?r2, ?c2) as input, and test them to see if they are in the same row, column or diagonal, if so the rule will fire and assert a new nogood environment (see appendix 5.1 for a lisp listing of the function). The second rule generates all the possible situations that 4 queens can be placed on the chessboard.

The corresponding Rete is shown in figure 14. In the figure, we have one type checking node, TCHECK, because we have only one kind of patterns. We also have four t-const nodes, t-const1, t-const2, t-const3, and t-const4; they check the values of the

field lx for 4, 3, 2, and 1 respectively. The figure contains two p-mem nodes: p1 corresponding to the contradiction rule, C1, and p2 corresponding to the normal rule (not a contradiction rule), R1.

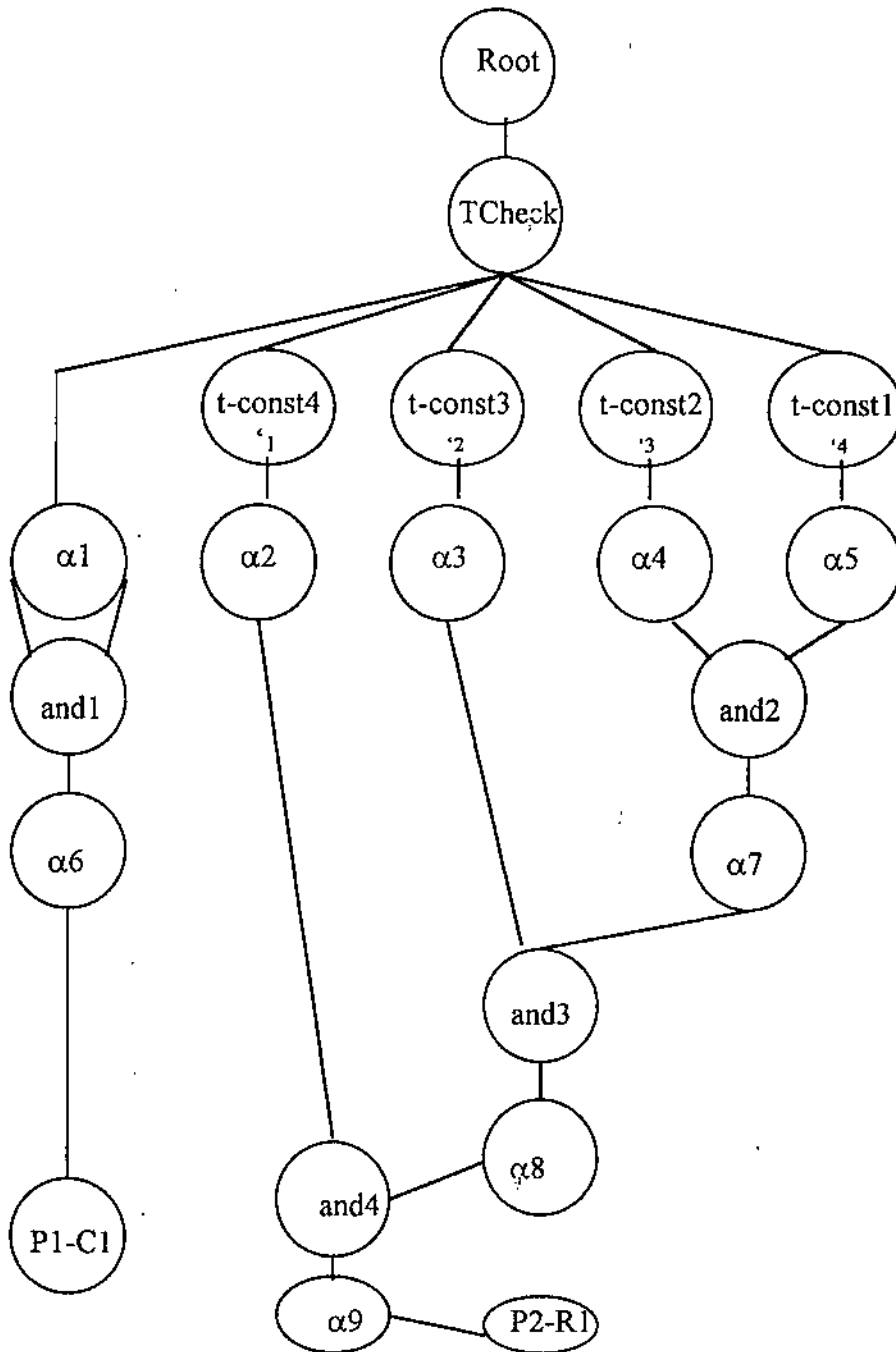


Figure 14 : Rete network corresponding to 4-queens problem

4.1.2 Results and Conclusions

This section presents a performance comparison of the algorithms on different versions of the queen problem (3, 4, 5, 6-queen problem) in term of time and basic operation.

Table 4, 6, 7 and 8 show the execution time needed by each algorithm and the number of join operations performed by each algorithm to solve 4-queen, 3-queen, 5-queen and 6-queen problems respectively. Graphical representation of table 4 is shown in figures 15 and 16. Table 5 shows the number of assertions (tuples) in each memory node in all algorithms to solve the 4-queen problem.

Tables 4, 5, 6, 7, and 8 illustrate how the tight coupling approaches are more efficient than the loose coupling. That is apparent in the time needed, join operations, and the number of tuples stored in the memory nodes. Table 5, shows that the number of tuples stored in the memory nodes by tight coupling approaches are less than the loose coupling approach. Tables 4, 6 and 7 show that the time needed by the tight coupling approaches is less than the time needed by the loose coupling approach, and the same thing can be said for the number of join operations. This is due to the fact that the tight coupling approaches do not use tuples with empty label during the match step, which will reduce the number of tuples involved in the join operations and hence reduce the execution time.

Tables 4, 6, 7, and 8 also show that the Morgue system and the Hindi system performed the same number of join operations, and therefore required the same execution time (3, 4, and 5-queen problem). This is expected because the Morgue system didn't need, in this case, to rejoin any tuples.

Table 8, shows that the execution time needed to solve the 6-queen problem using the Hindi system is less than the time needed by the Morgue system. This is because the Hindi system needs only to change a flag in the datum structure when it needs to move

the tuple to the OUT part, and this operation takes less time than the delete operation performed by the Morgue system.

Table 6, also shows that the loose coupling approach failed to solve the 6-queen problem on our system, because it needs more memory space to store the tuples with empty label.

Table 5, also shows that the Hindi system need more space than the Morgue system, this is expected because the Hindi system stores tuples with empty label in the OUT part of the alpha memory.

| | | Time(sec) | Join |
|----------------|------------|-----------|------|
| Loose coupling | | 58 | 472 |
| Tight coupling | Morgue sys | 41 | 192 |
| | Hindi sys | 41 | 192 |

Table 4: Time and join operation needed for 4-queen problem

| | | | $\alpha 1$ | $\alpha 2$ | $\alpha 3$ | $\alpha 4$ | $\alpha 5$ | $\alpha 6$ | $\alpha 7$ | $\alpha 8$ | $\alpha 9$ |
|----------------|-------------|-----|------------|------------|------------|------------|------------|------------|------------|------------|------------|
| Loose coupling | | | 16 | 4 | 4 | 4 | 4 | 136 | 16 | 64 | 256 |
| Tight coupling | Morgue sys. | | 16 | 4 | 4 | 4 | 4 | 60 | 6 | 4 | 2 |
| | Hindi sys. | IN | 16 | 4 | 4 | 4 | 4 | 60 | 6 | 4 | 2 |
| | | OUT | 0 | 0 | 0 | 0 | 0 | 76 | 10 | 20 | 14 |

Table 5: Number of assertions in alpha memory nodes (4-queen problem)

| | | Time(sec) | Join |
|----------------|---------------|-----------|------|
| Loose coupling | | 3.63 | 81 |
| Tight coupling | Morgue system | 3.57 | 60 |
| | Hindi system | 3.57 | 60 |

Table 6: Time and join operation needed for 3-queen problem

| | | Time(sec) | Join |
|----------------|---------------|-----------|------|
| Loose coupling | | 950 | 4225 |
| Tight coupling | Morgue system | 345 | 540 |
| | Hindi system | 339 | 540 |

Table 7: Time and join operation needed for 5-queen problem

| | | Time(sec) | Join |
|----------------|---------------|--------------|------------------|
| Loose coupling | | no more room | for lisp objects |
| Tight coupling | Morgue system | 1905 | 1554 |
| | Hindi system | 1886 | 1554 |

Table 8: Time and join operation needed for 6-queen problem

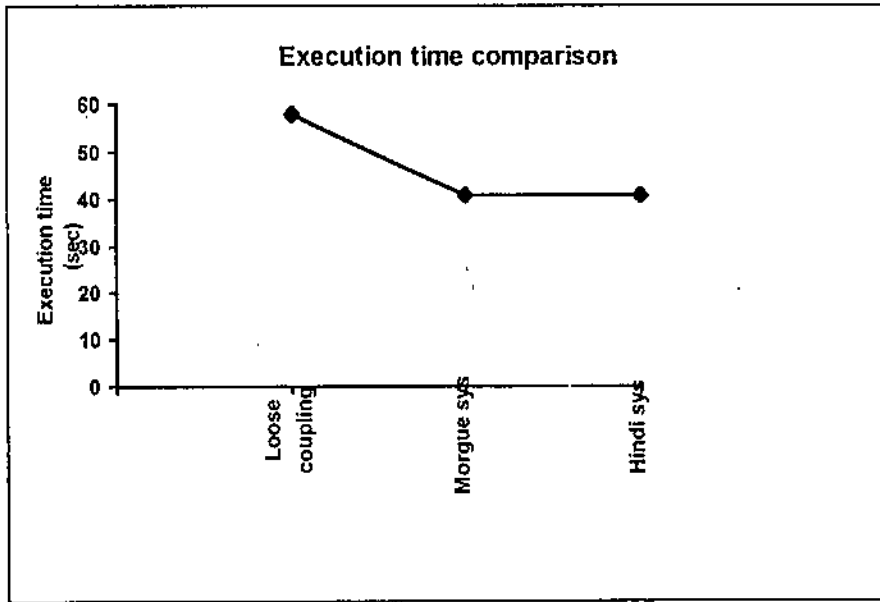


Figure 15 : Execution Time required to solve the queen problem

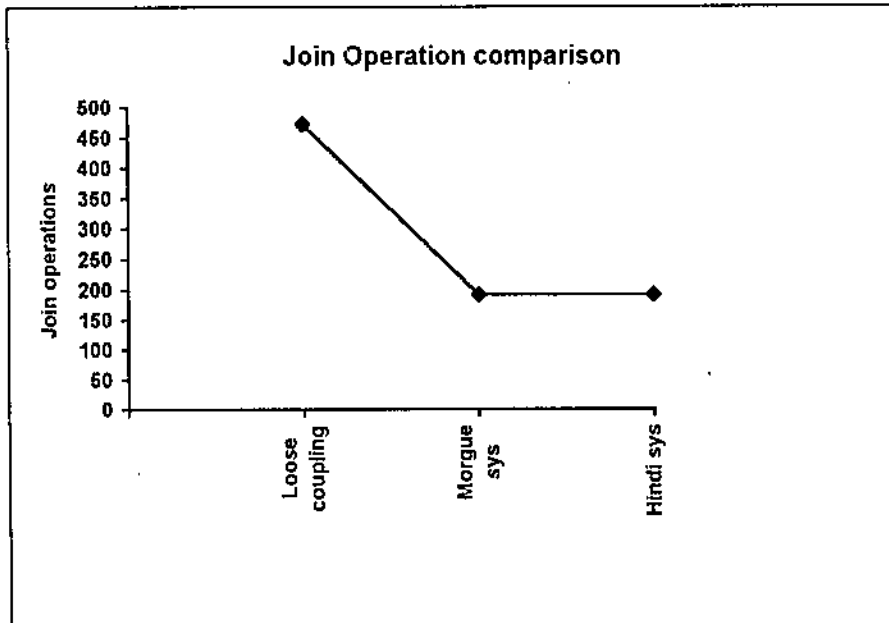


Figure 16 : Join operation in Queen problem

4.2 A Constraint Satisfaction Problem

A constraint satisfaction problem is defined as a triple consisting of: a finite set of variables, a finite set of candidate values for each variable, and a finite set of constraints on the values that variables can be assigned simultaneously (Bodington, 1988). The aim is to find a set of consistent value assignments to the variables that do not violate any of the constraints.

An example taken from Bodington, 1988, is shown in table 9. This example has only one solution, the variables a , b , and c should be assigned the values 5, 2 and 1 respectively.

| Variable | Domain | Constraints |
|----------|---------|-----------------|
| a | {3,5} | $a + c > 4$ |
| b | {2,3} | $b + c < 5$ |
| c | {1,3,5} | $a + b + c < 9$ |

Table 9: An example of a constraint satisfaction problem

Using ATMS to solve the constraint satisfaction problem, can help in minimizing the time needed to find the solution by storing the inconsistent assignments (nogood environment) in ATMS so that they never tried again.

4.2.1 Rules

We use four rules, (see appendix 5.2 for complete lisp implementation for constraint satisfaction problem) the first three to detect contradictions, and the fourth to generate all the possible assignments of the variables. The corresponding Rete is represented in figure 17.

(Contradiction-rule C1

```
((ass !var c !val ?v2)(ass !var a !val ?v3)) :test #'c-a-not-ok
```

```
==> (rassert-nogood!))
```

This rule says: if variable C is assigned a value $?v2$, and a variable A is assigned a value $?v3$, and these assignments didn't satisfy the first constraint represented by the function "c-a-not-ok", then the rule will fired to assert a new nogood environment.

(Contradiction-rule C2

```
((ass !var b !val ?v1 ) (ass !var c !val ?v2 )) :test #'b-c-not-ok
==> (rassert-nogood!))
```

This rule says: if variable B is assigned a value $?v1$, and a variable C is assigned a value $?v2$, and these assignments didn't satisfy the second constraint represented by the function "b-c-not-ok", then the rule will fired to assert a new nogood environment.

(Contradiction-rule C3

```
((ass !var b !val ?v1 ) (ass !var c !val ?v2 ) (ass !var a !val ?v3 )) :test #'b-c-a-not-ok
==> (rassert-nogood!))
```

This rule says: if variable B is assigned a value $?v1$, a variable C is assigned a value $?v2$, and a variable A is assigned a value $?v3$, and these assignments didn't satisfy the third constraint represented by the function "b-c-a-not-ok", then the rule will fired to assert a new nogood environment.

(Rule R1

```
((ass !var b !val ?v1 ) (ass !var c !val ?v2 )(ass !var a !val ?v3 ))
==> (rassert! (sol !a ?v3 !b ?v1 !c ?v2 )))
```

This rule will generate all the value assignments permutations for the three variables.

```
initial-working-memory : (ass !var a !val 3 ) (ass !var a !val 5 ) (ass !var b !val 2 )
(ass !var b !val 3 ) (ass !var c !val 1 ) (ass !var c !val 3 )(ass !var c !val 5 ))
```

The functions *c-a-not-ok*, *b-c-not-ok*, and *b-c-a-not-ok* check the constraints in the problem and return true if some constraint is not applicable(see appendix 5.2 for a complete listing of these functions).

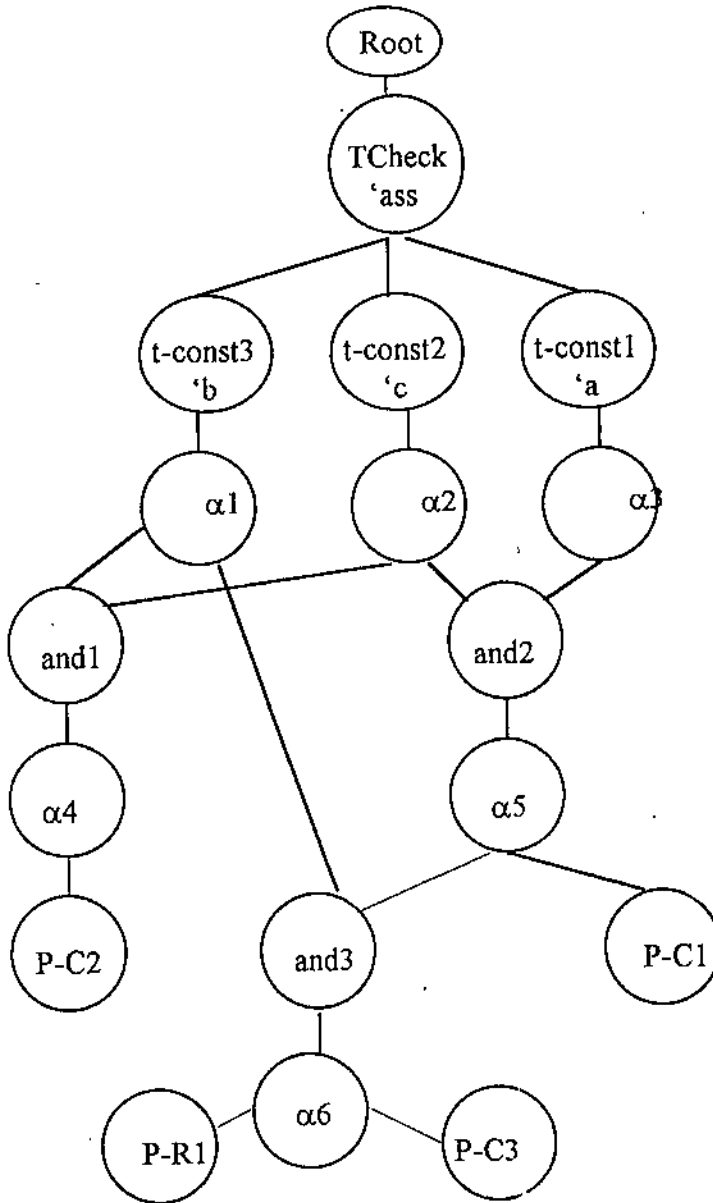


Figure 17: Rete network corresponding to constraint satisfaction problem

4.2.2 Results and Conclusions

This section presents a performance comparison of the algorithms on the constraint satisfaction problem described in 4.2.

Table 10, shows the execution time needed by each algorithm, the number of join operations needed to solve the problem, and the number of times the contradiction rules

were fired. A graphical representation is shown in figures 18 and 19. Table 11 shows the number of assertions maintained in each memory node by the algorithms.

Table 10, shows that the time needed to solve the problem by the tight coupling approaches is less than the time needed by the loose coupling approach, and the same thing can be said for join operation. Table 10 also shows that Morgue system and Hindi system required the same execution time, because in this problem it doesn't need to perform rejoin operation.

Table 10, shows that in the tight coupling approaches (Hindi and Morgue system), the number of contradiction rules that was fired are less than the loose coupling approach, that is because the first and the second contradiction rule is a part of the third one (shared), and any joined tuples used to fire the first or the second contradiction will not be used to fire the third contradiction rule.

Table 11, shows that the Hindi system need more space than the Morgue system, this is expected because the Hindi system stores tuples with empty label in the out part of the alpha memory.

| | | Time(sec) | Join | Contradiction fire |
|----------------|---------------|-----------|------|--------------------|
| Loose coupling | | 1.3 | 24 | 13 |
| Tight coupling | Morgue system | 1 | 22 | 6 |
| | Hindi system | 1 | 22 | 6 |

Table 10: Time, join operation, contradiction fire needed for Constraint satisfaction problem

| | | | $\alpha 1$ | $\alpha 2$ | $\alpha 3$ | $\alpha 4$ | $\alpha 5$ | $\alpha 6$ |
|----------------|------------|-----|------------|------------|------------|------------|------------|------------|
| Loose coupling | | | 2 | 3 | 2 | 6 | 6 | 12 |
| Tight coupling | Morgue sys | | 2 | 3 | 2 | 2 | 5 | 1 |
| | Hindi sys | IN | 2 | 3 | 2 | 2 | 5 | 1 |
| | | OUT | 0 | 0 | 0 | 4 | 1 | 9 |

Table 11: Number of tuples in alpha memory nodes (Constraint)

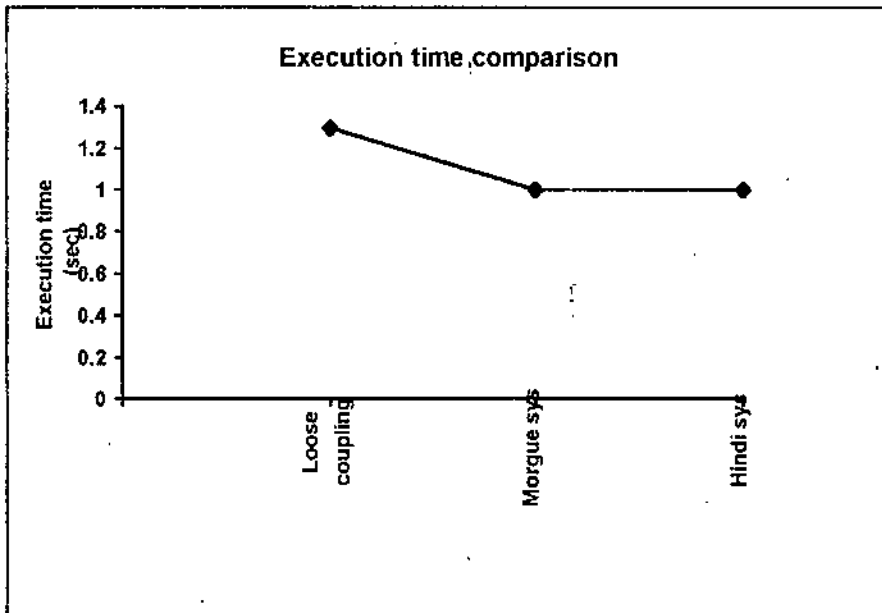


Figure 18 : Execution Time for Constraint satisfaction problem

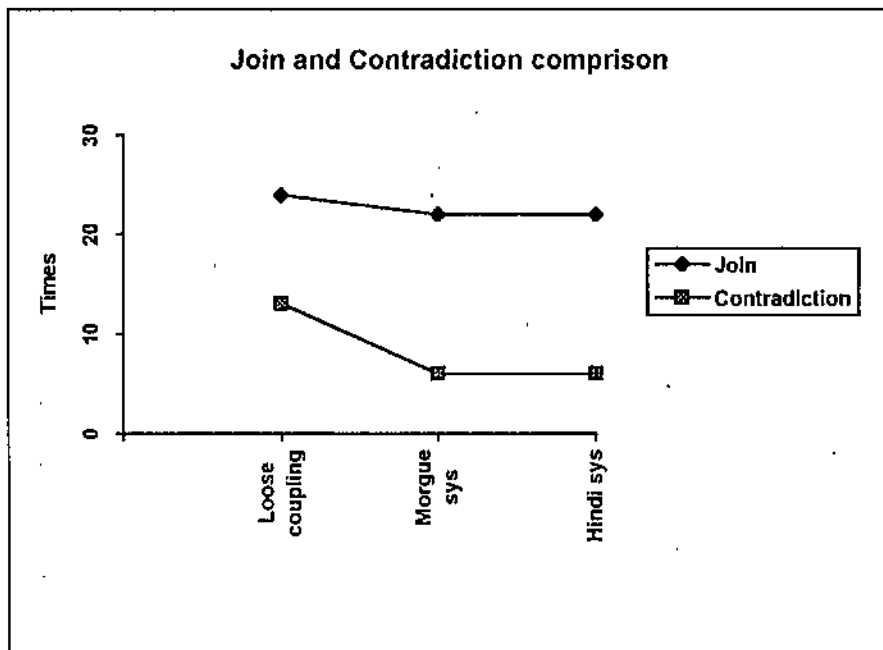


Figure 19: Join operation and contradiction fire for Constraint satisfaction problem

4.3 Student Registration system-1 : a typical Database system

In the previous two examples, we have shown that the tight coupling approaches are more efficient than the loose coupling approach. In order to present the advantages of the Hindi system, we need an application that needs to rejoin tuples. A database application is a good example, because of the need to add and delete data (tuples). To delete any tuple, we form a nogood environment that consists of the corresponding assumption, and to add tuple we insert the tuple in the Rete network to start reasoning.

Integrating ATMS-based production system with database system has been presented by Hindi, 1994, in a new method to couple an active databases and an ATMS expert systems. The approach called "monitor coupling". In monitor coupling the ATMS revises the beliefs of the reasoning system every time some relevant data in the database is modified. Active rules in the database monitor the data relevant to the reasoning system and alert it when any relevant data is modified or new relevant data becomes available.

In our example, we used a Student registration system. In this system the student is trying to register/drop some elective courses. These elective courses are divided into groups and subgroups, for example, one group of courses consist of four subgroups; the student can register only one course from each subgroup. When he registers any course from some subgroup.all the remaining courses in the subgroup can't be registered, but if the student fails the course all the courses in the same subgroup that he was not allowed to register become available for him. The system aims at helping the user to register the allowed elective courses. The user only use (register *course-number*) to register a course, and (retract *course-number*) to drop a course, it's the responsibility of the system to check the allowance to register this course, and so if it's not retract it.

The registration system is a good example for a dynamic data, that can be used to compare between the two tight coupling approaches. Here, we will not compare the tight coupling approaches with the loose coupling approach, because it is obvious, that it will not outperform the tight coupling approaches. Instead we will focus on comparing the Hindi system and the Morgue system.

4.3.1 Rules

In this subsection we present the rules¹ we used in our system (see appendix 5.3 for complete lisp implementation for registration problem). To register any course, all the following rules must be satisfied:

Note : the syntax used in TryReg is: !N: the course number, !G-1: the main group number, !G-2 :the subgroup number.

**** you can't register two courses from the same subgroup within the same group (group 2).**

(Contradiction-Rule G2-1

```
((TryReg !N ?x1 !G-1 2 !G-2 ?z)
 (TryReg !N ?x2 !G-1 2 !G-2 ?z)) :test #'diff-courses
==> (rassert-nogood!))
```

**** you can register only four courses from group 2.**

(Contradiction-Rule G2-2

```
((TryReg !N ?x1 !G-1 2 !G-2 ?z1) (TryReg !N ?x2 !G-1 2 !G-2 ?z2)
 (TryReg !N ?x3 !G-1 2 !G-2 ?z3) (TryReg !N ?x4 !G-1 2 !G-2 ?z4)
 (TryReg !N ?x5 !G-1 2 !G-2 ?z5)) :test #'diff-courses
==> (rassert-nogood!))
```

¹ these rules was taken from the Art faculty of Jordan university.

**** you can't register more than two courses from the same subgroup within the same group (group 5).**

(Contradiction-Rule G5-1

```
((TryReg !N ?x1 !G-1 5 !G-2 ?z) (TryReg !N ?x2 !G-1 5 !G-2 ?z)
 (TryReg !N ?x3 !G-1 5 !G-2 ?z)) :test #'diff-courses (x1 <> x2 <> x3)
==> (rassert-nogood!))
```

**** you can register only five courses from group 5.**

(Contradiction-Rule G5-2

```
((TryReg !N ?x1 !G-1 5 !G-2 ?z1) (TryReg !N ?x2 !G-1 5 !G-2 ?z2)
 (TryReg !N ?x3 !G-1 5 !G-2 ?z3) (TryReg !N ?x4 !G-1 5 !G-2 ?z4)
 (TryReg !N ?x5 !G-1 5 !G-2 ?z5)
 (TryReg !N ?x6 !G-1 5 !G-2 ?z6)) :test #'diff-courses
==> (rassert-nogood!))
```

**** you can't register more than four courses from the same subgroup within the same group (group 3).**

(Contradiction-Rule G3-1

```
((TryReg !N ?x1 !G-1 3 !G-2 ?z) (TryReg !N ?x2 !G-1 3 !G-2 ?z)
 (TryReg !N ?x3 !G-1 3 !G-2 ?z) (TryReg !N ?x4 !G-1 3 !G-2 ?z)
 (TryReg !N ?x5 !G-1 3 !G-2 ?z)) :test #'diff-courses
==> (rassert-nogood!))
```

**** you have to register seven courses from group 3, and they must be distributed among all the subgroups.**

(Contradiction-Rule G3-2

```
((TryReg !N ?x1 !G-1 3 !G-2 ?z1) (TryReg !N ?x2 !G-1 3 !G-2 ?z2)
```

(TryReg !N ?x3 !G-1 3 !G-2 ?z3) (TryReg !N ?x4 !G-1 3 !G-2 ?z4)
 (TryReg !N ?x5 !G-1 3 !G-2 ?z5) (TryReg !N ?x6 !G-1 3 !G-2 ?z6)
 (TryReg !N ?x7 !G-1 3 !G-2 ?z7)) :test #'G3-2 (Z's must be different)
 ==> (rassert-nogood!)

**** you can register only seven courses from group 3.**

(Contradiction-Rule G3-3

((TryReg !N ?x1 !G-1 3 !G-2 ?z1) (TryReg !N ?x2 !G-1 3 !G-2 ?z2)
 (TryReg !N ?x3 !G-1 3 !G-2 ?z3) (TryReg !N ?x4 !G-1 3 !G-2 ?z4)
 (TryReg !N ?x5 !G-1 3 !G-2 ?z5) (TryReg !N ?x6 !G-1 3 !G-2 ?z6)
 (TryReg !N ?x7 !G-1 3 !G-2 ?z7)
 (TryReg !N ?x8 !G-1 3 !G-2 ?z8)) :test #'diff-courses
 ==> (rassert-nogood!)

*** if the student try to register some courses and this course is available then assert TryReg predicate for this course in order to check all the constraint.**

(Rule R1

((Register !Cou ?x)
 (Course !No ?x !G1 ?y !G2 ?z))
 ==> (rassert! (TryReg !N ?x !G-1 ?y !G-2 ?z)))

*** if the course passes all the above constraints then assert YouCanReg for this course.**

(Rule R2

((TryReg !N ?x1 !G-1 ?y1 !G-2 ?z1) (TryReg !N ?x2 !G-1 ?y2 !G-2 ?z2)
 (TryReg !N ?x3 !G-1 ?y3 !G-2 ?z3) (TryReg !N ?x4 !G-1 ?y4 !G-2 ?z4)
 (TryReg !N ?x5 !G-1 ?y5 !G-2 ?z5) (TryReg !N ?x6 !G-1 ?y6 !G-2 ?z6)

(TryReg !N ?x7 !G-1 ?y7 !G-2 ?z7) (TryReg !N ?x8 !G-1 ?y8 !G-2 ?z8))

==> (rassert! (YouCanReg !Na ?x1)))

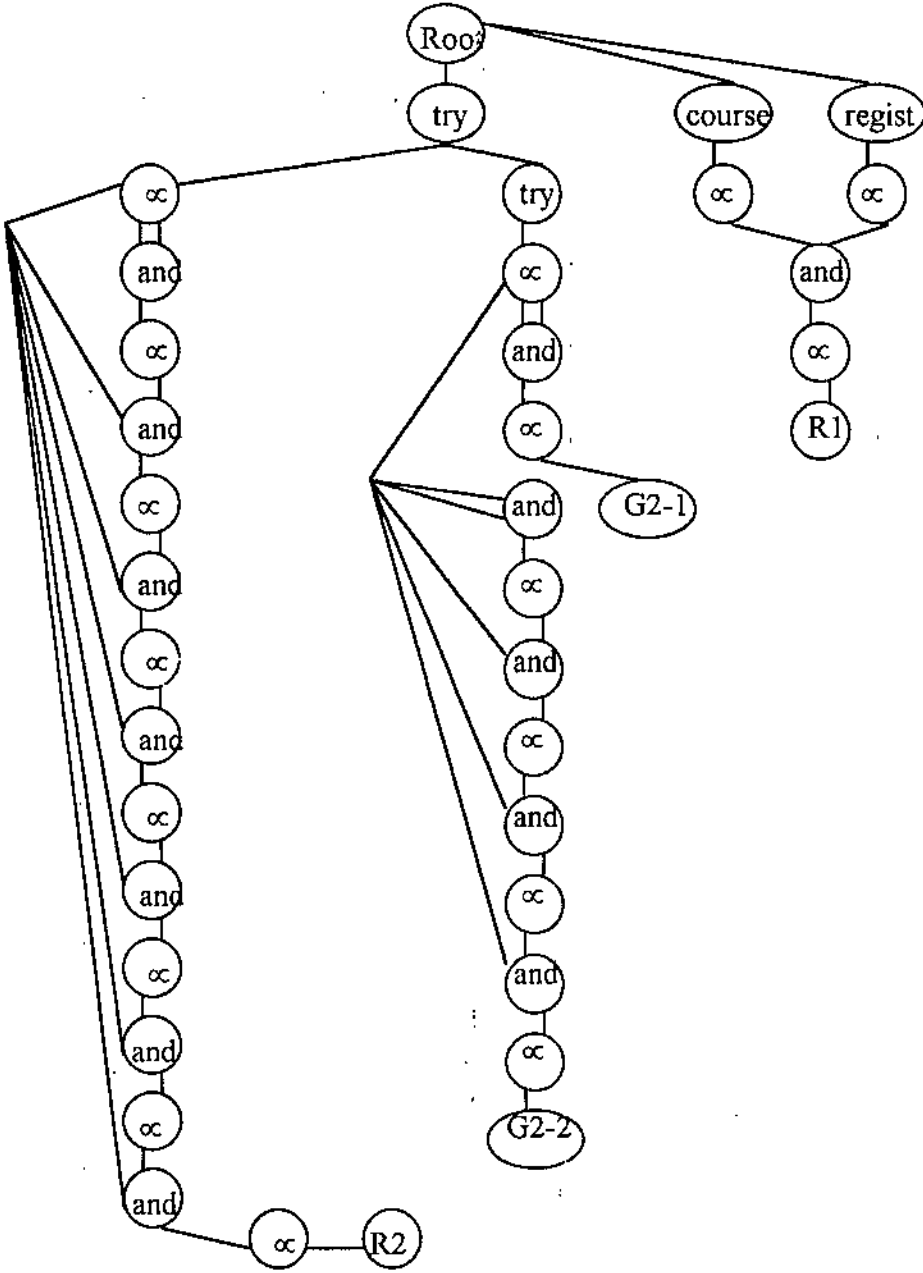


Figure 20: Rete network corresponding to Registration problem(part of it)

Initial working memory element: one Course assertion for each course. For example, a course with ID 1100, that belongs to the second subgroup of the first group is represented by: (Course !No 1100 !G1 1 !G2 2)). Our system use 73 courses. (see

appendix 5.3 for more detail about the rules and WME). Part of the corresponding Rete is shown in figure 20.

4.3.2 Results and Conclusions

We have used the following assertions to compare the performance of the Morgue and the Hindi systems:

(course !No 1100 !G1 1 !G2 1)(course !No 4100 !G1 2 !G2 1) (course !No 5100 !G1 2 !G2 1) (register 1100) (register 4100) (register 5100) (retract 5100) (retract 4100) (register 5100)

| | Time(sec) | Join | Label computation |
|----------------------|-----------|------|-------------------|
| Morgue system | 179 | 1841 | 2012 |
| Hindi system | 109 | 1389 | 1724 |

Table 12 : Time, Join operation, Label Computation comparison for registration system-1

Table 12 shows the time needed to solve the same problem using the two approaches, as well as the number of join operations, and label computation needed to solve the problems. A graphical representation of the results is shown in figure 21 (without a test function), and figure 22. Another examples are given in section 4.3.2, and the results are given in Tables 14 and 16.

It's clear from Tables 12, 14 and 16 that Hindi system is more efficient than the Morgue system. It saves a lot of time because it performs less join operations, and label computations. In Hindi system all the join operations and label computations done to register a course are saved even if the course is retracted. The saved work can be used whenever the course is registered again without the need to repeat it again. While in Morgue system all the works done in the first registration are discarded and all the works is repeated again, and that was the main improvement of the Hindi system.

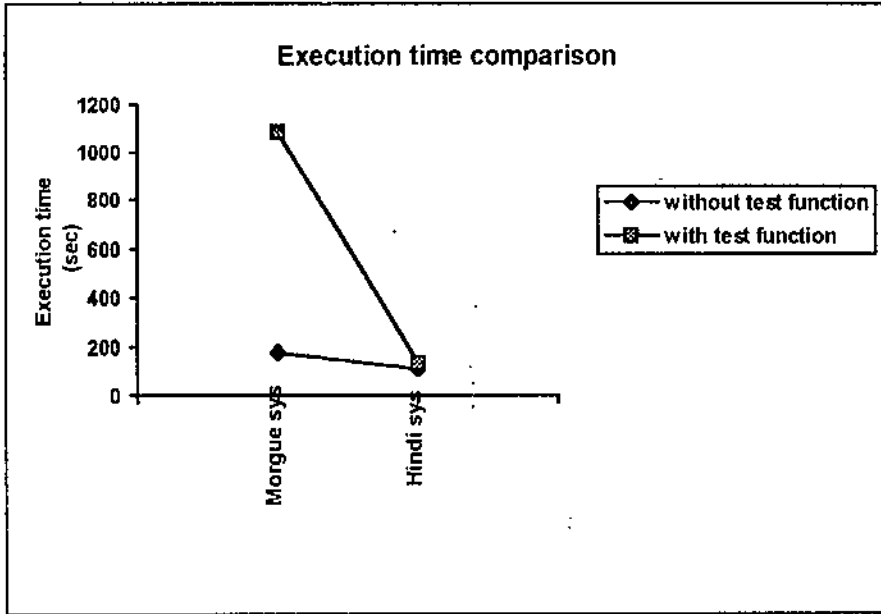


Figure 21 : Execution Time for Registration problem

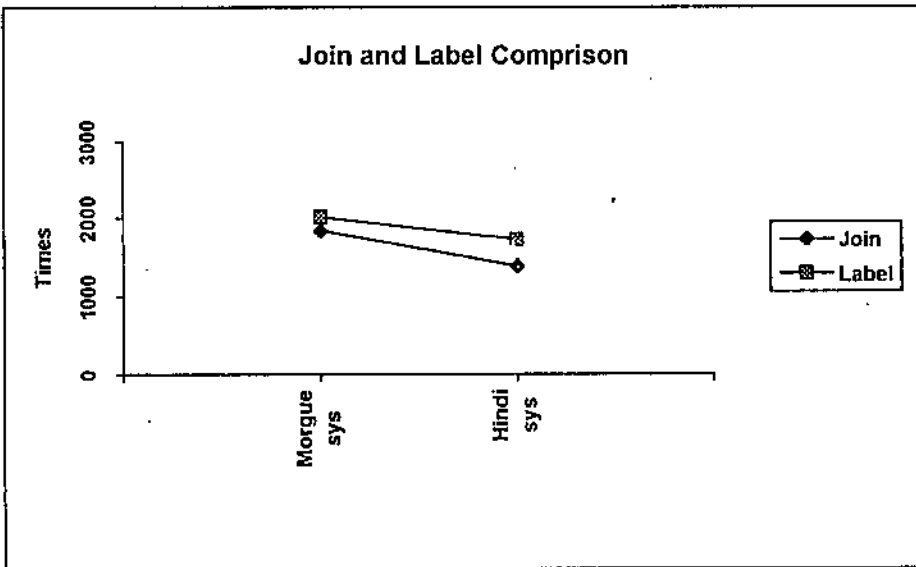


Figure 22 : Number of Join operation and Label computation comparison for Registration problem

4.4 Student Registration system-2: a typical Database application

Recall from chapter 3, that the Hindi system does save the work done in the match process, while Morgue system may need to rematch some tuples. In order to study the effect of this on the registration system, we augmented a test function that must be tested whenever a pair of tuples are joined. Here we used the same problem in the previous section and a delay function as a test function for evaluation process.

The existence of such functions is familiar in AI problem, since we need sometimes to restrict the rules to match only assertions that satisfy the test function.

4.4.1 Results and Conclusions

The results are shown in Table 13, and the corresponding graphs are given in the figure 21,22. Another examples are given in section 4.3.2, and the results are given in Tables 15 and 17.

| | Time(sec) | Join | Label computation |
|----------------------|-----------|------|-------------------|
| Morgue system | 1086 | 1841 | 2012 |
| Hindi system | 130 | 1389 | 1724 |

Table 13 : Execution Time, Join operation, and Label Computation comparison for registration system-2

Tables 13, 15, and 17 show that the Hindi system takes a ~12% of the time needed by the Morgue system to solve the problem. The only thing that have been added to this system compared to the previous example is the test function, so the large difference of time is expected because the Hindi system does test the function only once, where Morgue system does it every time it needs to rejoin tuples.

4.4.2 Different data for the registration problem

The following examples supports the previous conclusions:

Example A:

(course !No 314 !G1 5 !G2 1) (course !No 315 !G1 5 !G2 1) (course !No 411 !G1 5 !G2 1) (register 314) (register 315)(register 411)(retract 411) (retract 314)(register 411)

| | Time(sec) | Join | Label computation |
|----------------------|-----------|------|-------------------|
| Morgue system | 318 | 2972 | 3200 |
| Hindi system | 278 | 2430 | 2912 |

Table 14 : Time, Join operation, Label Computation comparison for registration system-1 (example a)

| | Time(sec) | Join | Label computation |
|----------------------|-----------|------|-------------------|
| Morgue system | 1840 | 2972 | 3200 |
| Hindi system | 297 | 2430 | 2912 |

Table 15 : Time, Join operation, Label Computation comparison for registration system-2 (example a)

Example B:

(course !No 211 !G1 4 !G2 1) (course !No 212 !G1 4 !G2 1) (register 211)(register 212) (retract 211)(retract 212)(register 211)(register 212)

| | Time(sec) | Join | Label computation |
|----------------------|-----------|------|-------------------|
| Morgue system | 68 | 1054 | 1168 |
| Hindi system | 41 | 599 | 880 |

Table 16 : Time, Join operation, Label Computation comparison for registration system-1 (example b)

| | Time(sec) | Join | Label computation |
|----------------------|-----------|------|-------------------|
| Morgue system | 547 | 1054 | 1168 |
| Hindi system | 51 | 599 | 880 |

Table 17 : Time, Join operation, Label Computation comparison for registration system-2 (example b)

In the next chapter, generalized conclusions, final remarks, and suggestion for future work, will be presented.

Chapter 5

Conclusions

In this chapter, we present our conclusions derived from all the three systems, as well as some suggested future work.

5.1 Conclusions

1. We had shown that the Tight coupling approaches are more efficient than the loose coupling approach, using four different applications. This is because tight coupling approaches use the label of the tuple during reasoning to determine whether we use the tuple in the next join operation or not. This leads to a reduction in the number of tuples in the memories and so reduces the time needed for the join operations. Moreover, the computation of labels in tight coupling approaches is much easier, since we compute the label using two antecedent only each time. Moreover this computation is shared between rules because of the nature of the Rete network. The loose coupling approach, on the other hand, has to compute the label using all the antecedents of the rule, and no work sharing can be gained, since the label computations are done outside the Rete.
2. For application that may need to rejoin tuples, Hindi system saves a lot of time, because it performs less join operations, and less label computations. Moreover it becomes much more efficient than the Morgue system when there is hard match inter-test that involves two tuples or more. (e.g. complex equation)

3. When there is no rejoin operation, Hindi system can at worst perform as good as the Morgue system, but in many cases it requires more memory, since it does not eliminate nodes with empty label, but, instead it keeps them in the OUT part of memory nodes.

4. In case of a large space problem or high order combinatorial problem, even if there is no rejoin operation (e.g. queen problem), Hindi system saves some time, and this is because the "delete" operation used by the Morgue system takes more time to execute than changing the flag operation used by Hindi system.

5. The Morgue system, have discarded a main advantage of the ATMS which is the ability to explain how it reached its conclusions. The ATMS will then traverse its dependency network and return all the assumptions that supports its conclusion. Because the Morgue system delete all tuples with empty label we can't ask it to trace its dependency network to explain how something become nogood. On the other hand, Hindi system doesn't delete any tuple, and so it can answer the Why? queries.

5.2 Suggested Future Work

1. We can improve the efficiency of the loose coupling approach itself by making some modifications on the way it deals with contradictions. In the loose coupling approach all instantiated contradiction rules are fired first; that can help in preventing some instantiated rule from firing which saves some time. However, all tuples with empty labels will still be stored in Rete memory nodes and used in the join operations which is unnecessary. The modification we suggest here is to move all the tuples with empty label to the out part of the corresponding memory node whenever a contradiction rule is fired. This will decrease the number of join operations performed and saves some work. Although, this modification can improve the efficiency of the loose coupling approach, but we expect it will not become more efficient than the

tight coupling approaches. This is because the tight coupling approaches uses the label during the match process to decide whether or not to match the node any further or not; This will reduce execution time of the matching step.

2. We suggest redesigning the reasoning algorithm so that it stores the tuple with empty labels in the OUT part only when there is enough memory space available, otherwise it must delete the empty label tuples i.e. return to Morgue system when there is no space available.

3. Redesign the rules, so we can determine which rules are more dynamic and can be treated according to the Hindi system, while for the other less dynamic rules use the basic Morgue system.

4. The Morgue and Hindi system, merges the Rete with the ATMS, which complicate the implementation. To simplify the implementation, I suggest implementing the ATMS and the Rete network as separate modules that can communicate via message passing.

References

- Bodington, R. and Ellaby E.1988. Justification And Assumption Based Truth Maintenance Systems: When And How To Use Them For Constraint Satisfaction .*In Reason Maintenance Systems and Their Applications*. Ellis Horwood limited.
- de Kleer, J. 1986. An Assumption-based TMS. *Artificial Intelligence*, 22(2):178-196.
- de Kleer, J. 1986. Extending the ATMS. *Artificial Intelligence*, 22(2): 193-197.
- de Kleer, J. 1986. Problem Solving With The ATMS. *Artificial Intelligence*, 22 (2) :197-224.
- Dresslar, O. 1988. An extended basic ATMS. *Proceeding of the second international workshop on Non-monotonic Reasoning*: 143-163.
- Forbus, K.D. and de Keer, J. 1993. *Building Problem Solvers, 1st.edition*.A Bradford book, England.
- Forgy c. 1982. A Fast Algorithm For the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*,19:17-37.
- Hindi, K. 1994. *Integrating Truth Maintenance Systems with Active Database Systems for Next Generation Cooperative Systems*. Ph.D. thesis, Department of computer Science, University of Exeter.

Hindi, K. and Lings, B. 1994. Using Truth Maintenance Systems To Solve The Data Consistency Problem . In *Proceeding of the second International Conference on Cooperative Information Systems: CoopIS-94*, University of Toronto Press.

McAllester, D. 1980. An Outlook On Truth Maintenance. Technical Report AIM-551, AI Laboratory, Massachusetts Institute of Technology.

Morgue, G. and Chehire, T. 1991. Efficiency of production systems when coupled with an assumption based truth maintenance system. In *Proceedings of the Ninth national Conference on artificial Intelligence, AAAI*, 268-274.

Ohta, Y. and Inoue, K. 1990. A Forward-Chaining Multiple-Context Reasoner And Its Application To Logic Design. In *IEEE second International Conference on Tools for Artificial Intelligence*. pages 386-392.

Quine, W.V. and J.S. Ullian. 1978. *The web of Belief*. New York: Random House.

Appendix 1

Rete Program

1.1 Rete.lsp

```
;;; RETE Production System = REPS
(IN-PACKAGE :RETE)
(LOAD "DEF.LSP") ;Structure definition
(LOAD "BUILD.lsp") ;Build RETE network
(LOAD "REASON.lsp") ;Reasoning system
(Defvar RULE-FILE ) ;Rule-file
(Setf rule-file "q.lsp")
(Defvar Problem "Queen") ;Problem name
;;; Measurement variable
(defvar joperation) ;;; Join operation
(defvar erule) ;;; Execute normal rule
(defvar ecrule) ;;; Execute Contradiction Rule
(defvar lcomp) ;;; Label Computation
(defvar instrule) ;;; The no. of rule instantiation
;;; Initialization .
(setf joperation 0) (setf erule 0) (setf ecrule 0) (setf lcomp 0) (setf instrule 0)
;;; Start function
(defun start ()
  (gc)
  (in-reps (create-reps problem))
  (load RULE-FILE)
  (setf rule-file nil)
  (start-reasoning))

(defun in-reps (ps) (setq *reps* ps))
(defun in-rete (rt) (setq *rete* rt))

;;;Create Function :REPS , Datum
(defun create-reps (title)
  (setq ps (make-reps
    :TITLE title
    :RETE (in-rete (make-rete :title (list :RETE-OF title))))))
  ps)

;;; Input :(1 (fact)) -- this is a tuple? (not a compound tuple)
;;; Output :Datum
;;; not exist previously .
(defun create-datum (wme)
  (make-datum :fact (list wme)))

;;; Create *rule* and set new data
```

```

(defun set-rule (name lhs test rhs vars flds)
  (setq *rule*
    (make-Rule
      :name name :lhs lhs
      :rhs rhs :test test
      :vars vars :flds flds)))

;;; Macros
(defun initial-working-memory (&rest wmes)
  `(set-queue ',wmes))

(defun rule (name ant &rest body )
  `(decompose-rule ',name ',ant ',body nil))

(defun Contradiction-rule (name ant &rest body )
  `(decompose-rule ',name ',ant ',body t))

(defun rassert! (fact)
  `(assert! ,(quotize fact) ))

;;; Decompose rule into its component and store that in *Rule*
(defun decompose-rule (name Lhs body contradiction &aux rhs test vars)
  (setf RHS (extract-rhs body))
  (when (eq (first body) :test) (setf test (second body)))
  (setf vars (extract-vars Lhs))
  (setf flds (reverse(extract-flds Lhs)))
  (set-rule name lhs test rhs vars flds) (create-rete))

;;; Extract the right hand side of the rule
(defun extract-rhs (body) (setf l (rest(member '==> body))))

;;; Extract a list of variables
(defun extract-vars (Lhs &aux (var-list nil) )
  (cond ((variable? lhs) (list lhs))
        ((atom lhs) nil)
        (t (append (extract-vars (first lhs)) (extract-vars (rest lhs) )))))

;;; Extract a list of fields
;;; Input : list of predicates
;;; without remove duplicate
(defun extract-flds (Lhs)
  (let ((fld-list nil))
    (dolist (pattern lhs)
      (when (listp pattern)
        (dolist (item pattern)
          (when (field? item) (setf fld-list (cons item fld-list) )))))
    fld-list))

;;; Take a list of initial working memory element

```

```

;;; Create a datum for each,Store it in queue
(defun set-queue (wmes)
  (dolist (wme wmes)
    (push (create-datum (cons '1 (list wme))) (reps-queue *reps*))))

;;; Is x a variable ?c ?
(defun variable? (x)
  (and (symbolp x) (char= #\? (elt (symbol-name x) 0))))

;;; Is x a field !c ?
(defun field? (x)
  (and (symbolp x) (char= #\! (elt (symbol-name x) 0))))

;;;Data Retrieive
;;; Fetch a pattern
;;; Show all assertions
;;; Note : this may print duplicate data becuase the tuple may stored
;;; : in more than one alpha-mem
(defun show-data ()
  ;; Look in alpha assosiated with type-checking,t-const
  ;; Which contain only a single tuple
  (format t "~%Working Memory :-")
  (dolist (alpha (reps-alpha *reps*))
    (dolist (datum (alpha-beta-datums alpha))
      (format t "~% ==> ~a" (second (first(datum-fact datum))))))
  ;; Look in derived attribute which contain all tuple that have not alpha
  (dolist (datum (reps-derived *reps*))
    (format t "~% ==> ~a" (second (first(datum-fact datum))))))

;;; Input :(1 (pat-name !fld1 ?d !fld2 3....))
;;; Output :list of instantiated pattern
(defun fetch (pattern &aux (result nil))
  ;; look if there is an alpha correspond to predicate name
  (dolist (alpha (reps-alpha *reps*))
    (when (eq (first (second pattern))
              (get-pred-name (first(alpha-beta-datums alpha))))
      (dolist (datum (alpha-beta-datums alpha))
        (setf bindings (match (second pattern)
                              (second (first(datum-fact datum))))))
        (unless (eq bindings 'fail)
          (push (sublis bindings (second pattern)) result))) )
    (dolist (datum (reps-derived *reps*))
      (setf bindings (match (second pattern)
                            (second (first(datum-fact datum))))))
      (unless (eq bindings 'fail)
        (push (sublis bindings (second pattern)) result)))
  (remove-duplicates result :test 'equal))

;;; Input :(1 (pat-name !fld1 ?d !fld2 3....))

```



```

;;; Output :datum if it exist or nil else
(defun is-exist? (pattern &aux (result nil))
  ;; look if there is an alpha correspond to predicate name
  (dolist (alpha (reps-alpha *reps*))
    (when (eq (first (second pattern))
              (get-pred-name (first(alpha-beta-datums alpha))))
      (dolist (datum (alpha-beta-datums alpha))
        (setf bindings (match (second pattern)
                              (second (first(datum-fact datum)))))
          (unless (eq bindings 'fail)
            (return-from is-exist? datum))))))
;;; look in derived datums
(dolist (datum (reps-derived *reps*))
  (setf bindings (match (second pattern)
                        (second (first(datum-fact datum)))))
    (unless (eq bindings 'fail)
      (return-from is-exist? datum))) nil)

;;; Input : (pat-name !fld1 ?d !fld2 3....)
;;; Output : list of instantiated pattern
(defmacro Rfetch (pattern)
  `(fetch (cons 'I (list ',pattern))))
;;; Input : datum
;;; output : a predicate name
(defun get-pred-name (wme)
  (when wme
    (first(second(first(datum-fact wme))))))

;;;MATCH PART
;;; MATCH A PATTERN WITH AN ASSERTION
;;; RETURN THE BINDING IF IT EXIST ELSE FAIL
(defun match (a b &optional (bindings nil))
  (cond ((equal a b) bindings)
        ((variable? a) (match-variable a b bindings))
        ((or (not (listp a)) (not (listp b))) 'FAIL)
        ((not (eq 'FAIL (setq bindings (match (first a) (first b) bindings))))
         (match (cdr a) (cdr b) bindings))
        (t 'FAIL)))

(defun match-variable (var exp bindings &aux binding)
  (setq binding (assoc var bindings))
  (cond (binding
         (match (cdr binding) exp bindings))
        (t (cons (cons var exp) bindings) )))

(time(start))
(dotimes (r 10)
  (format t "~a" (code-char 7)))

```

1.2 Def.lsp

```

;;; Queen Problem Using RETE Production System REPS
;;; Declaration Part
;;; 1- SPECIAL VARIABLES .
;;; 2- STRUCTURE USED .
;;; 3- PRINT FUNCTION .
(IN-PACKAGE :RETE)
;;; RETE Production System
(defvar *REPS* nil "Queen Problem-RETE")
(defvar *rule*) (defvar *RETE* nil "Queen Problem-RETE")
(defstruct (reps (:PREDICATE reps?) (:PRINT-FUNCTION print-reps))
  (title nil)
  (atms nil) ;Assumption beased truth maintenance system
  (rete nil) ;RETE network
  (alpha nil) ;Memory-node where all datum are stored there
  (derived nil) ;Datums that have no alpha are stored here
  (conflict-set nil) ;Conflict-set (instantiated rules)
  (queue nil)) ;Queue used conflict Resolution.

(defun print-reps (reps st ignore)
  (format st "REPS:~A" (reps-title reps)))

(defun show-reps ()
  (format t "~%Title:~a" (reps-title *reps*))
  (format t "~%Atms:~a" (reps-atms *reps*))
  (format t "~%RETE:~a" (reps-rete *reps*))
  (format t "~%QUEUE:~a" (reps-queue *reps*))
  (format t "~%To show All Data Echo (show-Data)")
  (format t "~%To show Conflict-set Echo (show-conflict-set)"))

;;; INSTANCE STRUCTURE
;;; Print Functions
;;; SHOW CONFLICT SET.
(defstruct (instance (:PREDICATE instance?)
  (:PRINT-FUNCTION print-instance))
  (rule-name nil)
  (consequence nil) ;;un-instantiated consequence
  (vars nil) ;;variable in consequence
  (binding nil)) ;;assosiation list with variable binding

(defun print-instance (inst st ignore)
  (format st "~%Instance:-")
  (format st "~%Rule:~a" (instance-rule-name inst))
  (format st "~%Consequence:~a" (instance-consequence inst))
  (format st "~%Vars:~a" (instance-vars inst))
  (format st "~%Binding:~a" (instance-binding inst)))

(defun show-conflict-set (&optional (conflict-set (reps-conflict-set *reps*)))
  (dolist (instance conflict-set) (print-instance instance t nil)))

```

```

;;; DATUM STRUCTURE <Fact>
;;; PRINT FUNCTIONS
(defstruct (datum (:PREDICATE datum?)
                (:PRINT-FUNCTION print-datum))
  (fact nil)
  (node nil)
  (assumption? nil))

(defun print-datum (datum st ignore)
  (format st "Datum:~a" (datum-fact datum)))

(defun show-datums (datums)
  (dolist (datum datums)
    (format t "~%Datum ~A: ~%Node:~A ~%Assumption: ~A "
            (datum-fact datum)
            (datum-node datum)
            (datum-assumption? datum))))

;;; Temporary structure used in rete builder
(defstruct (Rule) ;temp rule
  (name nil) (lhs nil) (test nil)
  (rhs nil) (flds nil) (vars nil))

;;;RETE NETWORK
;;; Print Functions
(defstruct (RETE (:PREDICATE rete?)
                (:PRINT-FUNCTION print-rete)) ;;; ROOT NODE
  (title 'Rete-nets)
  (type-checking nil) ;; Nodes to Check the predicate name

(defun print-rete (rete st ignore)
  (format st "RETE:~A" (rete-title rete)))

(defun show-rete ()
  (format t "~%Title:~a" (rete-title *rete*))
  (format t "~%Type checking Nodes:~a" (rete-type-checking *rete*)))

;;;Type Checking Nodes
(defstruct (Tcheck-node (:PREDICATE tcheck-node?)
                       (:PRINT-FUNCTION print-Tcheck-node))
  (name nil) ;; Name of the predicate
  (next-nodes nil) ;; alpha-beta or t-const - list of nodes

(defun print-Tcheck-node (Tcheck-node st ignore)
  (format st "TCHECK:~A" (Tcheck-node-name Tcheck-node)))

(defun show-Tcheck-node (&optional (Tchecks (rete-type-checking *rete*)))
  (dolist (Tcheck Tchecks)

```

```

(format t "~%Name:~a" (Tcheck-node-name Tcheck))
(format t "~%Next Nodes:~a" (Tcheck-node-next-nodes Tcheck)))
;;;
;;; T-const nodes
(defun T-const-node (:PREDICATE t-const-node?)
  (:PRINT-FUNCTION print-T-const-node))
  (id 0) ;; indexer
  (check1 nil) ;; check the constant value ((field value)...)
  (check2 nil) ;; check the field in one predicate ((field field)...)
  (check3 nil) ;; intra-test-function name
  (next-node nil) ;; alpha-beta -one node

(defun print-T-const-node (t-const-node st ignore)
  (format st "TC:~a" (t-const-node-id t-const-node))
  (if (t-const-node-check1 t-const-node) (format st " Const-test,"))
  (if (t-const-node-check2 t-const-node) (format st " Equal-var,"))
  (if (t-const-node-check3 t-const-node) (format st " Intra-Ext-fun,")))

(defun show-t-const-node (t-const-list)
  (dolist (t-const t-const-list)
    (format t "~%Id:~a" (t-const-node-id t-const))
    (format t "~%Const-test:~a" (t-const-node-check1 t-const))
    (format t "~%Equal-var:~a" (t-const-node-check2 t-const))
    (format t "~%Intra-Ext-fun:~a" (t-const-node-check3 t-const))
    (format t "~%Next-node:~a" (t-const-node-next-node t-const))))
;;;
;;; Alph and beta memory nodes
(defun alpha-beta (:PREDICATE alpha-beta?)
  (:PRINT-FUNCTION print-alpha-beta))
  (id 0) ;; indexer
  (Datums nil) ; List of facts
  (R-Ands nil) ; Right and node -list
  (L-Ands nil) ; Left and node -list
  (p-mem nil) ; P-mem node if it the next -list
  (prev-node nil)); pointer to the previous node -one element

(defun print-alpha-beta (alpha-beta st ignore)
  (format st "Alpha-beta:~A" (alpha-beta-id alpha-beta)))

(defun show-alpha-beta (mem-list)
  (dolist (mem mem-list)
    (format t "~%Id:~a" (alpha-beta-id mem))
    (format t "~%Datums:~a" (alpha-beta-datums mem))
    (format t "~%No of datums :~a" (length (alpha-beta-datums mem)))
    (format t "~%R-Ands:~a" (alpha-beta-R-ands mem))
    (format t "~%L-Ands:~a" (alpha-beta-L-ands mem))
    (format t "~%P-mems:~a" (alpha-beta-p-mem mem))
    (format t "~%Prev-node :~a" (alpha-beta-prev-node mem))))
;;;

```

```

;;; AND nodes
(defstruct (AND-node (:PREDICATE and-node?)
                   (:PRINT-FUNCTION print-and-node))
  (id 0)      ;; indexer
  (check1 nil) ;; check the var between predicates ((fld2 ind fld2 ind)..)
  (L-mem nil) ;; Left Memory Nodes -one element
  (R-mem nil) ;; Right Memory Nodes -one element
  (O-mem nil)) ;; Output Memory Nodes -one element

(defun print-and-node (and-node st ignore)
  (format st "And:~A" (and-node-id and-node)))

(defun show-and-node (and-list)
  (dolist (and-node and-list)
    (format t "~%Id:~a" (and-node-id and-node))
    (format t "~%Eq-var:~a" (and-node-check1 and-node))
    (format t "~%Left mem :~a" (and-node-L-mem and-node))
    (format t "~%Right mem :~a" (and-node-R-mem and-node))
    (format t "~%Output mem :~a" (and-node-O-mem and-node))))

;;; P-memory nodes
(defstruct (P-mem (:PREDICATE p-mem?) (:PRINT-FUNCTION print-p-mem))
  (rule-name nil) ; the name of the rule corresponding to it
  (inter-test nil) ; the name of the inter test function
  (consequent nil) ; the consequent of the tule (un-instantiated)
  (var-loc nil)) ; ((?x fld index)...)

(defun print-p-mem (p-mem st ignore)
  (format st "P-mem :~A" (p-mem-rule-name p-mem)))

(defun show-p-mem (p-mem-list)
  (dolist (p-mem p-mem-list)
    (format t "~%Rule Name :~a" (p-mem-rule-name p-mem))
    (format t "~%Test Name :~a" (p-mem-inter-test p-mem))
    (format t "~%Consequent:~a" (p-mem-consequent p-mem))
    (format t "~%Var-loc :~a" (p-mem-var-loc p-mem))))

```

1.3 Build.lsp

```

;;; Create the rete network for a set of rules
;;; It will be more iffecient if you write the rules according to the
;;; Following condition:
;;; 1- put the commom predicate in the right hand side of the rule
;;;    and in the same order over the rules
;;; 2- put the more changable predicate at the left hand side of the rule
;;; 3- any condition you need over any predicate use :test function-name
;;; 4- you can use :test function for the whole set of predicate
;;; 5- if you use the predicate in different place
;;;    use the same arrangment of field name
;;; General syntax of a rule is :

```

```

;;; (rule r-name ((ant1)[:test #'function-name]
;;;             (ant2)[:test #'function-name]
;;;             .....))
;;;             [:test #'function-name]    ==>
;;;             (rassert! (consequent)))
;;; General syntax of fact
;;; (predicate-name !field-name field-value ..... )
;;; if the field value is variable use ?varname.
;;; To Enter initial working memory element :
;;; (initial-working-memory (father !f1 ahmad !f2 sami) (...) ...)
;;; Intra-test: use all fields-value in a predicate under examine
;;; as an input to the function -
;;; with duplicate - (do not remove duplicate)
;;; and in order as they are appear in the predicate
;;; Inter-test: use all field values in all rule under examine
;;; as an input to the function with the same order as
;;; they appear in the rules predicate - with duplicate
;;; Rassert! ,Rassert-nogood! :insert data
(in-package :RETE)
;;; Special variable declaration
(defvar *dir* 'r) ; special variable to determine the direction of build rete net
(defvar *r-env* nil) ; place to hold the binding from right
(defvar *l-env* nil) ; place to hold the binding from left
(defvar *mem-id* 0) ; Counter for alpha-node
(defvar *con-id* 0) ; Counter for t-const-node
(defvar *and-id* 0) ; Counter for and-node
(setf *con-id* 0)(setf *and-id* 0)(setf *mem-id* 0)
;;; Variable Needed for debug and tree traverse
(defvar *tc* nil)(setf *tc* nil)(defvar *tch* nil)(setf *tch* nil)
(defvar *al* nil)(setf *al* nil)(defvar *ad* nil)(setf *ad* nil)
(defvar *p* nil)(setf *p* nil)

;;; Main function
;;; *rule*-test is a test over all predicate
(defun create-rete ()
  (setf *r-env* nil)(setf *l-env* nil)(setf *dir* 'r)
  (create-p-mem (process-antecedents (decompose-ants))))

;;; Decompose each antecedent into a antecedent and a test
;;; Input: ((p1 ..):test #'f (p2 ..))
;;; Output: (((p1 ..) #'f) ((p2 ..) nil))
(defun decompose-ants (&optional (lhs (rule-lhs *rule*)) &aux nlhs)
  (do ((ants lhs (rest ants))
      (nlhs nil)
      ((endp ants) (reverse nlhs))
      (cond ((and (listp (first ants)) (eq (second ants) :test))
             (push (list (first ants) (third ants)) nlhs)
             (setf ants (rest(rest ants))))
          (t (push (list (first ants) nil) nlhs)))))

```

```

;;; Main function in build a rete for each rule
;;; Input : (((p1 ..) #f) ((p2 ..) nil)) .
;;; Output: alpha-beta Memory Node .
(defun process-antecedents (lhs)
  (cond ((endp lhs) nil)
        (t (anded (process-antecedents (rest lhs))
                   (process (first lhs)) ))))
;;; Process one antecedent and create type-checking ,t-const node,alpha-beta
;;; Input :((p1 ..) #f)
;;; Output: alpha-beta Memory Node .
(defun process (ant)
  (update1-l-r-env ant) ;; update r-env,l-env
  (setf check1 (compute-check1 ant)) ;; constant check
  (setf check2 (compute-check2 ant)) ;; var equality
  (setf check3 (second ant)) ;; function name
  ;;If there is a test create t-const else create type-check-node only
  (if (or check1 check2 check3)
      (alpha (t-const check1 check2 check3 (type-check ant)))
      (alpha (type-check ant))))

;;; Update the r-env ,l-env according to the new pattern
;;; Input :((p1 ..) #f) , *dir*
;;; Output :((fld1 var index) ....)
(defun update1-l-r-env (ant)
  (cond ((eq *dir* 'r) (setf *dir* 'l) (setf *r-env* (generate-env ant)))
        ((eq *dir* 'l) (setf *l-env* (generate-env ant)))))

;;; Input :((p1 ..) #f)
;;; Output :((fld1 l var1 ) ... )
(defun generate-env (ant &aux (env nil))
  (do ((ant1 (first ant) (rest ant1))
      (env nil))
      ((endp ant1) (reverse env))
      (when (and (field? (first ant1)) (variable? (second ant1)))
          (push `,(first ant1) 1 ,(second ant1) ) env))))

;;; Check the constant field in a predicate
;;; Input :((p1 ..) #f)
;;; Output :((fld val)...)
(defun compute-check1 (ant &aux (env nil) )
  (do ((ant1 (first ant) (rest ant1))
      (env nil))
      ((endp ant1) (reverse env))
      (when (and (field? (first ant1)) (not (variable? (second ant1))))
          (push `,(first ant1) ,(second ant1) ) env))))

;;; Check if two or more variable are equal
;;; Input :((p1 ..) #f)

```

```

;;; Output :(fld1 fld2)
(defun compute-check2 (ant &aux (env nil))
  (do ((ant1 (first ant) (rest ant1))
      (env nil)
      ((endp ant1) (reverse env))
      (when (field? (first ant1))
        (setf fld (first ant1))
        (setf var (second ant1))
        (do ((rem (rest(rest ant1)) (rest rem) ))
            ((endp rem))
            (when (and (field? (first rem)) (eq (second rem) var))
              (push `(.fld ,(first rem)) env)
              (return t) )))))

;;; Create type checking node , If it exist return it
;;; Input :((p1 ..) #f)
;;; Output :Type Checking node
(defun type-check (ant)
  (setf tch (lookfor-tch ant))
  (if tch tch (create-type-checking ant)))

;;; Create alpha-mem node and associate it with the node
;;; Input : type-checking node or t-const or and node
;;; Output : Alpha-beta memory-node
(defun alpha (node)
  (cond
    ((tcheck-node? node)
     ;; if it exist return it else create a new one
     (lookfor-alpha-beta node))
    ((and (t-const-node? node)
         (alpha-beta? (t-const-node-next-node node)))
     (t-const-node-next-node node))
    ((t-const-node? node)
     (setf *mem-id* (1+ *mem-id*))
     (setf (t-const-node-next-node node)
           (make-alpha-beta :id *mem-id* :prev-node node))
     (push (t-const-node-next-node node) (reps-alpha *reps*)))
    ((and (and-node? node) (and-node-o-mem node))
     (and-node-o-mem node))
    ((and-node? node)
     (setf *mem-id* (1+ *mem-id*))
     (setf (and-node-o-mem node)
           (make-alpha-beta :id *mem-id* :prev-node node))
     (and-node-o-mem node) )))

;;; look if one of the node is a mem node ;; associated with type-checking
(defun lookfor-alpha-beta (tcheck-node)
  (dolist (node (tcheck-node-next-nodes tcheck-node))

```



```

    (when (alpha-beta? node)
      (return-from lookfor-alpha-beta node)))
  (create-alpha-beta tcheck-node))

```

```

;;; create memory node
;;; join it to the previous node
(defun create-alpha-beta (tcheck-node)
  (setf *mem-id* (1+ *mem-id*))
  (setf ab (make-alpha-beta :id *mem-id* :prev-node tcheck-node))
  (push ab (tcheck-node-next-nodes tcheck-node))
  (push ab (reps-alpha *reps*)) ab)

```

```

;;; Look if a type-checking node for the ant exist previously
(defun lookfor-tch (ant)
  (setf p-name (first(first ant)))
  (dolist (tch (rete-type-checking *rete*))
    (when (eq p-name (tcheck-node-name tch))
      (return-from lookfor-tch tch) )))

```

```

;;; Create a structure and return it
;;; and join it to the root
(defun create-type-checking (ant)
  (setf tc (make-tcheck-node :name (first(first ant))))
  (push tc (rete-type-checking *rete*)) tc)

```

```

;;; if there exist t-const return it else create a t-const node
(defun t-const (check1 check2 check3 tcheck-node)
  (setf t-const-node (lookfor-t-const check1 check2 check3 tcheck-node))
  (if t-const-node
      t-const-node ;; to make closure around the function
      (eval `(create-t-const ',check1 ',check2 ',check3 ',tcheck-node))))

```

```

;;; Look if a t-const node for the type-checking node. is exist previously
;;; must have the same condition
(defun lookfor-t-const (ck1 ck2 ck3 tcheck-node)
  (dolist (node (tcheck-node-next-nodes tcheck-node))
    (when (and (t-const-node? node)
                (t-const-equal? node ck1 ck2 ck3))
      (return-from lookfor-t-const node))))

```

```

;;; create t-const node structure and return it
;;; join it to the previous type-checking node
(defun create-t-const (ck1 ck2 ck3 tcheck-node)
  (setf *con-id* (1+ *con-id*))
  (setf tc (make-t-const-node
            :id *con-id*
            :check1 ck1
            :check2 ck2
            :check3 ck3))

```

```

(push tc (tcheck-node-next-nodes tcheck-node)) tc)

;;; check if the tests are in this t-const-node or not
(defun t-const-equal? (node ck1 ck2 ck3 )
  (and
    (equal (t-const-node-check3 node) ck3)
    (equal-ck (t-const-node-check2 node) ck2)
    (equal-ck (t-const-node-check1 node) ck1)))

(defun equal-ck (ck1 ck2)
  (unless (= (length ck1) (length ck2)) (return-from equal-ck nil))
  (dolist (item ck1)
    (unless (member item ck2 :test #'equal) (return-from equal-ck nil))) t)

;;; Main function to join between antecedents
;;; Create and-node , p memory node
;;; Input :two alpha-beta nodes
;;; Output :one alpha-beta
(defun anded (r-mem l-mem)
  (if (null r-mem) (return-from anded l-mem))
  (setf inter-test (update2-l-r-env)) ;; update r-env,l-env,dir,compute inter-test
  (setf Out-mem-and-node (lookfor-and-node inter-test l-mem r-mem))
  (if Out-mem-and-node
    Out-mem-and-node
    (alpha(create-and-node inter-test l-mem r-mem))))

;;; Input :*r-env* *l-env* *dir*
;;; Output :*r-env* after put l-env at the beginning of the r-env
;;; and increment the index of the old *r-env*
;;; and return the inter-test
(defun update2-l-r-env (&aux (nrenv nil))
  (dolist (renv *r-env*)
    (push `,(first renv) ,(1+ (second renv)) ,(third renv) ) nrenv))
  (setf nrenv (reverse nrenv))
  (setf inter-test (compute-inter-test *l-env* nrenv))
  (dolist (lenv *l-env*) (push lenv nrenv))
  (setf *r-env* nrenv) (setf *l-env* nil)
  inter-test)

;;;Compute the inter test between the right and the left env
;;;Input : R-env L-env
;;;Output : Test
(defun compute-inter-test (l-env r-env &aux (test nil))
  (dolist (L l-env)
    (setf lvar (third L))
    (dolist (R r-env)
      (setf rvar (third R))
      (when (eq lvar rvar)
        (push `,(first l) ,(second l) ,(first R) ,(second R)) test)))) test)

```

```

;;; create and-node and join it to the mem-nodes
(defun create-and-node (test l r)
  (setf *and-id* (1+ *and-id*))
  (setf and-node (make-and-node
                  :id *and-id*
                  :check1 test
                  :l-mem l
                  :r-mem r))
  (push and-node (alpha-beta-r-and) r)
  (push and-node (alpha-beta-l-and) l) and-node)

;;; look if there exist a previously created and node
;;; if yes return the output mem
(defun lookfor-and-node (test l r)
  (dolist (left-and (alpha-beta-l-and) l)
    (when (and (member left-and (alpha-beta-r-and) r) :test #'eq)
      (equal-and? left-and test))
    (return-from lookfor-and-node (and-node-o-mem left-and))))))

(defun equal-and? (anode test)
  (unless (= (length test) (length (and-node-check1 anode)))
    (return-from equal-and? nil))
  (dolist (item (and-node-check1 anode))
    (unless (member item test :test #'equal) (return-from equal-and? nil))) t)

;;; After finish build a rete nets for a rule you need to create a p_mem
;;; for this rule and associate it to the last node
;;; if there is a test in whole rule you must process this test
;;; Input :alpha-beta
;;; Output :p-mem
(defun create-p-mem (node &optional (inter-test-fun (rule-test *rule*)))
  (eval `(setf p (make-p-mem
                  :rule-name ,(rule-name *rule*)
                  :inter-test ,inter-test-fun
                  :consequent ,(rule-rhs *rule*)
                  :var-loc ,(compute-var-loc))))
  (push p (alpha-beta-p-mem node)) p)

;;; find the var loc using the *r-env* and vars in *rule*
;;; Output : ((?x fld index)...)
(defun compute-var-loc (&aux (loc nil))
  (dolist (var (rule-vars *rule*))
    (dolist (binding *r-env*)
      (when (eq (third binding) var)
        (push `(,var ,(first binding) ,(second binding)) loc)
        (return t))))
  loc)

```

1.4 Reason.lsp

```

(in-package :rete)
;;; Rete Reasoning System .
(defun start-reasoning (&aux (repeat t))
(loop
  (unless repeat
    (format t "~%*****")
    (format t "~%Reasoning terminated")
    (format t "~%To look for the data use - show-data ")
    (format t "~%To look for a specific data use - rfetch (...) ")
    (format t "~%Join operation      =~a" Joperation)
    (format t "~%Rule instantiated    =~a" instrule)
    (format t "~%Execute Normal Rule    =~a" ERule)
    (format t "~%Execute Contradiction Rule =~a" ECRule)
    (format t "~%Label Computation      =~a" Lcomp)
    (format t "~%*****")
    (return))
  ;;Find all applicable rule and store the instants in the conflict set
  (loop
    (setf wme (pop (reps-queue *reps*)))
    (when (null wme) (return))
    (process-wme wme))
  (setf repeat nil)
  ;; loop until a new datum is inserted or no instant in conflict set
  (loop
    (setf inst (resolve-conflicts))
    (when (null inst) (return)) ; Empty conflict-set: finish resolution.
    (when (act inst) ;; When the act insert a new datum in queue-process it.
      (setf repeat t)
      (return))) ))

;;; You can put here any criteria you want ;
;;; I use a simple strategy (the last in first out) ;
(defun resolve-conflicts ()
  (pop (reps-conflict-set *reps*)))

;;; Input :Datum
;;; Insert any completely instantiated rule in conflict set.
(defun process-wme (wme)
  ;; Find all next-node of the correspond type-check node
  (setf tch (find-tcheck (get-pred-name wme)))
  (unless tch ;; No type checking for this predicate
    (push wme (reps-derived *reps*))
    (return-from process-wme t))
  (setf next-nodes (tcheck-node-next-nodes tch))
  ;;Next node :alpha or t-const
  (dolist (node next-nodes)
    (when (alpha-beta? node)
      (push wme (alpha-beta-datums node))
      (process-alpha node (list wme)))

```

```

(when (t-const-node? node)
  (when (and
    (process-check1 node wme)
    (process-check2 node wme)
    (process-check3 node wme))
    (push wme (alpha-beta-datums.(t-const-node-next-node node)))
    (process-alpha (t-const-node-next-node node)
      (list wme))))))
;;; Input : Predicate name
;;; Output : type checking node
(defun find-tcheck (p-name)
  (dolist (tch (rete-type-checking *rete*))
    (when (eq p-name (tcheck-node-name tch))
      (return-from find-tcheck tch))) nil)

;;; Input : t-const-node , datum
;;; Output : t , nil
;;; Check the constant values ((fld value)(fld value)...)
;;; Datum here is a single tuple.
(defun process-check1 (t-const datum)
  (dolist (check (t-const-node-check1 t-const))
    (unless (eq (fld-value (first check) 1 datum) (second check))
      (return-from process-check1 nil))) t)

;;; Input : t-const-node , datum
;;; Output : t , nil
;;; Check the fields equality ((fld1 fld2)(fld4 fld5)...)
;;; Datum here is a single tuple.
(defun process-check2 (t-const datum)
  (dolist (check (t-const-node-check2 t-const))
    (unless (eq (fld-value (first check) 1 datum)
      (fld-value (second check) 1 datum))
      (return-from process-check2 nil))) t)

;;; Input : t-const-node , datum
;;; Output : t , nil
;;; Check the test function ((fld1 fld2)(fld4 fld5)...)
;;; Datum here is a single tuple.
(defun process-check3 (t-const datum)
  (unless (t-const-node-check3 t-const)
    (return-from process-check3 t))
  (setf tuple (second(first(datum-fact datum))))
  (setf flds (reverse(extract-flds (list tuple))))
  (apply (t-const-node-check3 t-const)
    (mapcar #'(lambda (fld)
      (fld-value fld 1 datum))
      flds)))

;;; Input : The Field name, Index ,datum

```

;;; Output : The value of the field; if it exist

```
(defun fld-value (fld index datum)
  (setf tuples (datum-fact datum))
  (dolist (tuple tuples)
    (when (= (first tuple) index)
      (do ((pattern (second tuple) (rest pattern)))
          ((null pattern) nil)
        (when (eq (first pattern) fld)
          (return-from fld-value (second pattern))))))))
```

;;;Input : List of New Datums

;;;Process first Left and-node ,then right and-node ,

;;; finally look for p-mem if it exist.

```
(defun process-alpha (alpha new-datums)
  (process-l-and alpha new-datums)
  (process-r-and alpha new-datums)
  (process-p-mem alpha new-datums))
```

;;; Join the new datums with fact stored in the right mem of the left and

;;; Process the out memory of the and-node

```
(defun process-l-and (alpha new-datums)
  (setf lands (alpha-beta-l-ands alpha))
  (dolist (land lands)
    ;;; the following condition is used to prevent repeat work
    ;;; when the l and r mem for node AND node is the same and is
    ;;; traversed only in the right direction
    (unless (equal (and-node-r-mem land) (and-node-l-mem land))
      (setf new-joind-tuples
        (join new-datums (alpha-beta-datums (and-node-r-mem land)
          (and-node-o-mem land) (and-node-check1 land))))
      (insert-joined-tuple new-joind-tuples (and-node-o-mem land))
      (when new-joind-tuples
        (process-alpha (and-node-o-mem land) new-joind-tuples))))))
```

;;; Join the new datums with fact stored in the right mem of the right and

;;; Process the out memory of the and-node

```
(defun process-r-and (alpha new-datums)
  (setf rands (alpha-beta-r-ands alpha))
  (dolist (rand rands)
    (setf new-joind-tuples
      (join (alpha-beta-datums (and-node-l-mem rand)) new-datums
        (and-node-o-mem rand) (and-node-check1 rand)))
    (insert-joined-tuple new-joind-tuples (and-node-o-mem rand))
    (when new-joind-tuples
      (process-alpha (and-node-o-mem Rand) new-joind-tuples))))
```

;;; Check each new datums with inter-test function (if it exist)

;;; Store the good one in the conflict set

```
(defun process-p-mem (alpha new-datums)
```

```

(unless (alpha-beta-p-mem alpha)
  (return-from process-p-mem t))
(setf p-mems (alpha-beta-p-mem alpha))
(dolist (p-mem p-mems)
  (dolist (datum new-datums)
    (when (process-inter-test p-mem datum)
      (update-conflict-set
        (p-mem-rule-name p-mem)
        (p-mem-consequent p-mem)
        (generate-binding p-mem datum))))))

```

```

(defun update-conflict-set (rname consequence binding)
  (incf instrule)
  (setf lambda-vars (extract-vars consequence))
  (eval `(push (make-instance
    :rule-name ',rname
    :consequence #'(lambda ,lambda-vars ,@consequence)
    :binding ',binding
    :vars ',lambda-vars)
    (reps-conflict-set *reps*))))

```

```

;;; Input : p-mem , datum
;;; Output : t , nil
;;; Check the inter test function if it exist
(defun process-inter-test (p-mem datum)
  (unless (p-mem-inter-test p-mem) (return-from process-inter-test t))
  (when (p-mem-inter-test p-mem)
    (setf vals (extract-vals datum))
    (apply (p-mem-inter-test p-mem) vals)))

```

```

;;; Input : Datum
;;; Output : list of variable values
(defun extract-vals (datum &aux (vals nil) (temp nil))
  (dolist (tp (reverse(datum-fact datum)))
    (setf temp nil)
    (do ((tuple (second tp) (rest tuple)))
      ((null tuple) t)
      (when (field? (first tuple))
        (push (second tuple) temp))))
    (setf vals (append (reverse temp) vals)) vals)

```

```

;;; Input : p-mem node ,datum
;;; Output : ((?x . value)(..))
(defun generate-binding (p-mem datum &aux (binding nil))
  (dolist (loc (p-mem-var-loc p-mem))
    (push (cons (first loc)
      (fld-value (second loc) (third loc) datum))
      binding)) binding)

```

;;; Input :right and left datums , out-mem of the and-node ,test

;;; Output : a list of datums (joined tuple test)

```
(defun join (ldatums rdatums alpha test &aux (result nil) )
  (dolist (ld ldatums)
    (dolist (rd rdatums)
      (incf joperation)
      (setf updated-r (inc-index rd))
      (setf joined-tuple (append (datum-fact ld) updated-r))
      (setf d (make-datum :fact joined-tuple))
      (when (process-andcheck d test)
        (push d result)))))) result)
```

;;; Input : new-joined tuple

;;; Operation : push them at the datum for the alpha mem

```
(defun insert-joined-tuple (new-joind-tuples alpha)
  (setf (alpha-beta-datums alpha)
    (append new-joind-tuples (alpha-beta-datums alpha))))
```

;;; Input :Datums

;;; Output :fact in the datum after increment index

;;; increment the index in each tuple by one

```
(defun inc-index (datum &aux (result nil))
  (setf facts (datum-fact datum))
  (dolist (fact facts)
    (push (list (1+ (first fact)) (second fact)) result)) (reverse result))
```

;;; Input : joined tuple , test ((fld1 index fld2 ind)(...))

;;; Output : t or nil

```
(defun process-andcheck (d tests)
  (dolist (test tests)
    (unless
      (eq
        (fld-value (first test) (second test) d)
        (fld-value (third test) (fourth test) d))
      (return-from process-andcheck nil))) t)
```

;;; Input :instance

;;; Output : t . if it insert a new data

```
(defun act (inst)
  (incf erule)
  (apply (instance-consequence inst)
    (instantiate-variables (instance-vars inst)
      (instance-binding inst))))
```

```
(defun instantiate-variables (consequent binding)
  (sublis binding consequent))
```

;;;Input : fact

;;;Insert in the queue if not exist

```
(defun assert! (fact)
  (if (is-exist? (cons 'I (list fact)))
```



```
(return-from assert! nil)
(push (create-datum (cons '1 (list fact))) (reps-queue *reps*))))
```

```
;Copyright (c) 1986-1993 Kenneth D. Forbus, Johan de Kleer and Xerox
;Corporation. All Rights Reserved.
```

```
(defun quotize (pattern)
  (cond ((null pattern) nil)
        ((variable? pattern) pattern)
        ((not (listp pattern)) (list 'QUOTE pattern))
        (t `(cons ,(quotize (car pattern))
                   ,(quotize (cdr pattern))))))
```

Appendix 2

Loose Coupling Program

2.1 Rete.lsp

```
;;; Loose Coupling between RETE Production System with ATMS
```

```
(IN-PACKAGE :IRETE)
(LOAD "DEF.LSP")      ;Structure definition
(LOAD "BUILD.lsp")   ;Build RETE network
(Load "ATMS.lsp")    ;Atms system
(LOAD "REASON.lsp")  ;Reasoning system
```

```
(Defvar RULE-FILE ) ;Rule-file
(Defvar Problem "Regist") ;Problem name
(Setf rule-file "crule.lsp")
```

```
;;; Measurement variable
(defvar joperation)   ;;; Join operation
(defvar erule)       ;;; Execute normal operation
(defvar ecrule)      ;;; Execute Contradiction Rule
(defvar lcomp)       ;;; Label Computation
(defvar instrule)    ;;; The no. of rule instantiation
```

```
;;; Initialization
(setf joperation 0)
(setf erule 0)
(setf ecrule 0)
(setf lcomp 0)
(setf instrule 0)
```

```
;;; Start function
(defun start ()
  (gc)
  (in-reps (create-reps problem))
  (load RULE-FILE)
  (setf rule-file nil)
  (start-reasoning))
```

```
(defun in-reps (ps) (setq *reps* ps))
(defun in-rete (rt) (setq *rete* rt))
```

```
;;;Create Function :REPS , Datum
(defun create-reps (title)
  (setq ps (make-reps
            :TITLE title
```

```

      :ATMS (create-atms (list :ATMS-OF title))
      :RETE (in-rete (make-rete :title (list :RETE-OF title))))))
(setq false (make-datum :fact '(1 '(FALSE)) ))
(setf (datum-node false) (atms-contranode *atms*))
(setf (node-datum (datum-node false)) false)
ps)

```

```

;;; Input   : (1 (fact)) -- this is a tuple? (not a compound tuple)
;;;      :asn ;weather it's assumption or not
;;;      :must not exist previously .
;;; Operation :create datum
;;;      :create a correspond node
;;; Output   :Datum

```

```

(defun create-datum (wme asn)
  (setq datum (make-datum
                :fact (list wme)
                :assumption? asn))
    ;;; this tuple consist of it self only
  (setf (datum-datum-list datum) (list datum))
  (if asn
      (setf (datum-node datum) (assume! datum))
      (setf (datum-node datum) (create-node datum))))
  datum)

```

```

;;; Input :Datum ,Not exist previously
;;; Output :Node
(defun assume! (datum)
  (assume-node datum))

```

```

;;; Create *rule* and set new data
(defun set-rule (name lhs test rhs vars flds cont)
  (setq *rule*
        (make-Rule
          :name name :lhs lhs
          :rhs rhs :test test
          :vars vars :flds flds
          :contradiction cont)))

```

```

;;; Macros

```

```

(defunmacro initial-working-memory (&rest wmes)
  `(set-queue ',wmes))

```

```

(defunmacro rule (name ant &rest body )
  `(decompose-rule ',name ',ant ',body nil))

```

```

(defunmacro Contradiction-rule (name ant &rest body )
  `(decompose-rule ',name ',ant ',body t))

```

```
(defmacro rassert! (fact)
  `(assert! ,(quotize fact)))
```

```
(defmacro rassert-nogood! ()
  `(assert-nogood!))
```

;;; Decompose rule into its component and store that in *Rule*

```
(defun decompose-rule (name Lhs body contradiction &aux rhs test vars)
  (setf RHS (extract-rhs body))
  (when (eq (first body) :test) (setf test (second body)))
  (setf vars (extract-vars Lhs))
  (setf flds (reverse(extract-flds Lhs)))
  (set-rule name lhs test rhs vars flds contradiction)
  (create-rete))
```

;;; Extract the right hand side of the rule

```
(defun extract-rhs (body)
  (setf l (rest(member '==> body))))
```

;;; Extract a list of variables

```
(defun extract-vars (Lhs &aux (var-list nil) )
  (cond ((variable? lhs) (list lhs))
        ((atom lhs) nil)
        (t (append (extract-vars (first lhs))
                    (extract-vars (rest lhs ))))))
```

;;; Extract a list of fields
 ;;; Input : list of predicates
 ;;; without remove duplicate

```
(defun extract-flds (Lhs)
  (let ((fld-list nil))
    (dolist (pattern lhs)
      (when (listp pattern)
        (dolist (item pattern)
          (when (field? item) (setf fld-list (cons item fld-list) )))))
    fld-list))
```

;;; Take a list of initial working memory element
 ;;; Create a datum for each,Store it in queue

```
(defun set-queue (wmes)
  (dolist (wme wmes)
    (push (create-datum (cons 'l (list wme)) t) (reps-queue *reps*))))
```

```

;;; Is x a variable ?c ?
(defun variable? (x)
  (and (symbolp x)
        (char= #\? (elt (symbol-name x) 0))))

;;; Is x a field !c ?

(defun field? (x)
  (and (symbolp x)
        (char= #\! (elt (symbol-name x) 0))))

;;;Data Retrieve
;;; Fetch a pattern
;;; Show all assertions
;;; Note : this may print duplicate data because the tuple may stored
;;;       : in more than one alpha-mem

(defun show-data (&optional (show-label nil))
  ;; Look in alpha associated with type-checking,t-const
  ;; Which contain only a single tuple
  (format t "~%Working Memory :-")
  (dolist (alpha (reps-alpha *reps*))
    (dolist (datum (alpha-beta-datums alpha))
      (when (node-label (datum-node datum))
        (format t "~% ==> ~a" (second (first(datum-fact datum))))
        (when show-label
          (format t "~% Label ==> ~a" (node-label (datum-node datum)))))))
  ;; Look in derived attribute which contain all tuple that have not alpha
  (dolist (datum (reps-derived *reps*))
    (when (node-label (datum-node datum))
      (format t "~% ==> ~a" (second (first(datum-fact datum))))
      (when show-label
        (format t "~% Label ==> ~a" (node-label (datum-node datum)))))))

;;; Input :(1 (pat-name !fld1 ?d !fld2 3....))
;;; Output :list of instantiated pattern

(defun fetch (pattern show-label &aux (result nil))
  ;; look if there is an alpha correspond to predicate name
  (dolist (alpha (reps-alpha *reps*))
    (when (eq (first (second pattern))
              (get-pred-name (first(alpha-beta-datums alpha))))
      (dolist (datum (alpha-beta-datums alpha))
        (when (node-label (datum-node datum))
          (setf bindings (match (second pattern)
                                (second (first(datum-fact datum))))))

```

```

(unless (eq bindings 'fail)
  (when show-label
    (push (node-label (datum-node datum)) result)
    (push (sublis bindings (second pattern)) result))))))
(dolist (datum (reps-derived *reps*))
  (when (node-label (datum-node datum))
    (setf bindings (match (second pattern)
      (second (first(datum-fact datum)))))
    (unless (eq bindings 'fail)
      (when show-label
        (push (node-label (datum-node datum)) result)
        (push (sublis bindings (second pattern)) result))))))
(remove-duplicates result :test 'equal))

```

```

;;; Input : (1 (pat-name !fld1 ?d !fld2 3....))
;;; Output : datum if it exist or nil else

```

```

(defun is-exist? (pattern &aux (result nil))
  ;; look if there is an alpha correspond to predicate name
  (dolist (alpha (reps-alpha *reps*))
    (when (eq (first (second pattern))
      (get-pred-name (first(alpha-beta-datums alpha))))
      (dolist (datum (alpha-beta-datums alpha))
        (setf bindings (match (second pattern)
          (second (first(datum-fact datum)))))
        (unless (eq bindings 'fail)
          (return-from is-exist? datum))))))
  ;; look in derived datums
  (dolist (datum (reps-derived *reps*))
    (setf bindings (match (second pattern)
      (second (first(datum-fact datum)))))
    (unless (eq bindings 'fail)
      (return-from is-exist? datum)))
  nil)

```

```

;;; Input : (pat-name !fld1 ?d !fld2 3....)
;;; Output : list of instantiated pattern

```

```

(defmacro Rfetch (pattern)
  `(fetch (cons '1 (list ',pattern)) nil))

```

```

;;; Input : datum
;;; output : a predicate name

```

```

(defun get-pred-name (wme)
  (when wme
    (first(second(first(datum-fact wme)))))

```

```
;;;MATCH PART
;;; MATCH A PATTERN WITH AN ASSERTION
;;; RETURN THE BINDING IF IT EXIST ELSE FAIL
```

```
(defun match (a b &optional (bindings nil))
  (cond ((equal a b) bindings)
        ((variable? a) (match-variable a b bindings))
        ((or (not (listp a)) (not (listp b))) 'FAIL)
        ((not (eq 'FAIL (setq bindings (match (first a) (first b) bindings))))
         (match (cdr a) (cdr b) bindings))
        (t 'FAIL)))
```

```
(defun match-variable (var exp bindings &aux binding)
  (setq binding (assoc var bindings))
  (cond (binding
         (match (cdr binding) exp bindings))
        (t (cons (cons var exp) bindings) )))
```

```
(time(start))
(dotimes (r 1)
  (format t "~a" (code-char 7)))
```

2.2 Def.lsp

```
;;; Queen Problem Using RETE Production System REPS
;;; Declaration Part
;;; 1- SPECIAL VARIABLES .
;;; 2- STRUCTURE USED .
;;; 3- PRINT FUNCTION .
```

```
(IN-PACKAGE :IRETE)
```

```
;;; RETE Production System
(defvar *REPS* nil "Queen Problem-RETE")
(defvar *rule*)
(defvar *RETE* nil "Queen Problem-RETE")
```

```
(defstruct (reps (:PREDICATE reps?)
                (:PRINT-FUNCTION print-reps))
  (title nil)
  (atms nil) ;Assumption beased truth inaintenance system
  (rete nil) ;RETE network
  (alpha nil) ;Memory-node where all datum are stored there
  (derived nil) ;Datums that have no alpha are stored here
  (conflict-set nil) ;Conflict-set (instantiated rules)
  (queue nil) ;Queue used conflict Resolution.
```

```
(defun print-reps (reps st ignore)
  (format st "REPS:~A" (reps-title reps)))
```

```
(defun show-reps ()
  (format t "~%Title:~a"      (reps-title *reps*))
  (format t "~%Atms:~a"      (reps-atms *reps*))
  (format t "~%RETE:~a"      (reps-rete *reps*))
  (format t "~%QUEUE:~a"     (reps-queue *reps*))
  (format t "~%To show full atms Echo (print-full-atms)")
  (format t "~%To show All Data Echo (show-Data)")
  (format t "~%To show Conflict-set Echo (show-conflict-set)"))
```

```
;;; INSTANCE STRUCTURE
;;; Print Functions
;;; SHOW CONFLICT SET.
```

```
(defstruct (instance (:PREDICATE instance?)
  (:PRINT-FUNCTION print-instance))
  (rule-name nil)
  (contradiction nil) ;weather its contradiction or not.
  (consequence nil) ;;un-instantiated consequence
  (vars nil) ;;variable in consequence
  (antecedents nil) ;;list of antecedents datum
  (label nil) ;;label of the antecedents
  (binding nil)) ;;assosiation list with variable binding
```

```
(defun print-instance (inst st ignore)
  (format st "~%Instance:-")
  (format st "~%Rule:~a" (instance-rule-name inst))
  (format t "~%Contradiction:~a" (instance-contradiction inst))
  (format st "~%Consequence:~a" (instance-consequence inst))
  (format st "~%Vars:~a" (instance-vars inst))
  (format st "~%Antecedents:~a" (instance-antecedents inst))
  (format st "~%Labels:~a" (instance-label inst))
  (format st "~%Binding:~a" (instance-binding inst)))
```

```
(defun show-conflict-set (&optional (conflict-set (reps-conflict-set *reps*)))
  (dolist (instance conflict-set)
    (print-instance instance t nil)))
```

```
;;; DATUM STRUCTURE <Fact>
;;; PRINT FUNCTIONS
```

```
(defstruct (datum (:PREDICATE datum?)
  (:PRINT-FUNCTION print-datum))
  (fact nil)
```



```
(datum-list nil) ;; If it's a joint tuple ;this list consist the
                ;; corresponddatum for each component fact
(node nil)      ;; Corresponding Node in the atms
(assumption? nil));; Weather it is an assumption or not
```

```
(defun print-datum (datum st ignore)
  (format st "Datum:~a" (datum-fact datum)))
```

```
(defun show-datums (datums)
  (dolist (datum datums)
    (format t "~%Datum ~A: ~%Node:~A ~%Assumption: ~A "
            (datum-fact datum)
            (datum-node datum)
            (datum-assumption? datum))))
```

```
;;; Temporry structure used in rete builder
```

```
(defstruct (Rule) ;temp rule
  (name nil)
  (lhs nil)
  (test nil)
  (rhs nil)
  (flds nil)
  (vars nil)
  (contradiction nil))
```

```
;;;RETE NETWORK
;;; Print Functions
```

```
(defstruct (RETE (:PREDICATE rete?)
  (:PRINT-FUNCTION print-rete)) ;;; ROOT NODE
  (title 'Rete-nets)
  (type-checking nil) ;; Nodes to Check the predicate name
```

```
(defun print-rete (rete st ignore)
  (format st "RETE:~A" (rete-title rete)))
```

```
(defun show-rete ()
  (format t "~%Title:~a" (rete-title *rete*))
  (format t "~%Type checking Nodes:~a" (rete-type-checking *rete*)))
```

```
;;;
;;;
;;;Type Checking Nodes
;;;
```

```
(defstruct (Tcheck-node (:PREDICATE tcheck-node?)
  (:PRINT-FUNCTION print-Tcheck-node))
  (name nil) ;; Name of the predicate
  (next-nodes nil) ;; alpha-beta or t-const - list of nodes
```

```
(defun print-Tcheck-node (Tcheck-node st ignore)
```

```

(format st "TCHECK:~A" (Tcheck-node-name Tcheck-node)))

(defun show-Tcheck-node (&optional (Tchecks (rete-type-checking *rete*)))
  (dolist (Tcheck Tchecks)
    (format t "~%Name:~a" (Tcheck-node-name Tcheck))
    (format t "~%Next Nodes:~a" (Tcheck-node-next-nodes Tcheck))))
;;;
;;; T-const nodes
;;;
(defstruct (T-const-node (:PREDICATE t-const-node?)
                        (:PRINT-FUNCTION print-T-const-node))
  (id 0) ;; indexer
  (check1 nil) ;; check the constant value ((field value)...)
  (check2 nil) ;; check the field in one predicate ((field field)...)
  (check3 nil) ;; intra-test-function name
  (next-node nil) ;; alpha-beta -one node

(defun print-T-const-node (t-const-node st ignore)
  (format st "TC:~a" (t-const-node-id t-const-node))
  (if (t-const-node-check1 t-const-node) (format st " Const-test,"))
  (if (t-const-node-check2 t-const-node) (format st " Equal-var,"))
  (if (t-const-node-check3 t-const-node) (format st " Intra-Ext-fun,")))

(defun show-t-const-node (t-const-list)
  (dolist (t-const t-const-list)
    (format t "~%Id:~a" (t-const-node-id t-const))
    (format t "~%Const-test:~a" (t-const-node-check1 t-const))
    (format t "~%Equal-var:~a" (t-const-node-check2 t-const))
    (format t "~%Intra-Ext-fun:~a" (t-const-node-check3 t-const))
    (format t "~%Next-node:~a" (t-const-node-next-node t-const))))
;;;
;;; Alph and beta memory nodes
;;;
(defstruct (alpha-beta (:PREDICATE alpha-beta?)
                      (:PRINT-FUNCTION print-alpha-beta))
  (id 0) ;; indexer
  (Datums nil) ; List of facts
  (R-Ands nil) ; Right and node -list
  (L-Ands nil) ; Left and node -list
  (p-mem nil) ; P-mem node if it the next -list
  (prev-node nil)); pointer to the previous node -one element

(defun print-alpha-beta (alpha-beta st ignore)
  (format st "Alpha-beta:~A" (alpha-beta-id alpha-beta)))

(defun show-alpha-beta (mem-list)
  (dolist (mem mem-list)
    (format t "~%Id:~a" (alpha-beta-id mem)).

```

```

(format t "~%Datums:~a" (alpha-beta-datums mem))
(format t "~%No of datums :~a" (length (alpha-beta-datums mem)))
(format t "~%R-Ands:~a" (alpha-beta-R-ands mem))
(format t "~%L-Ands:~a" (alpha-beta-L-ands mem))
(format t "~%P-mems:~a" (alpha-beta-p-mem mem))
(format t "~%Prev-node :~a" (alpha-beta-prev-node mem))
(format t "~%=====")
(read)))
;;;
;;;
;;; AND nodes
;;;
(defstruct (AND-node (:PREDICATE and-node?)
                  (:PRINT-FUNCTION print-and-node))
  (id 0)      ;; indexer
  (check1 nil) ;; check the var between predicates ((fld2 ind fld2 ind)..)
  (L-mem nil) ;; Left Memory Nodes -one element
  (R-mem nil) ;; Right Memory Nodes -one element
  (O-mem nil) ;; Output Memory Nodes -one element

(defun print-and-node (and-node st ignore)
  (format st "And:~A" (and-node-id and-node)))

(defun show-and-node (and-list)
  (dolist (and-node and-list)
    (format t "~%Id:~a" (and-node-id and-node))
    (format t "~%Eq-var:~a" (and-node-check1 and-node))
    (format t "~%Left mem :~a" (and-node-L-mem and-node))
    (format t "~%Right mem :~a" (and-node-R-mem and-node))
    (format t "~%Output mem :~a" (and-node-O-mem and-node))))

;;;
;;; P-memory nodes
;;;
(defstruct (P-mem (:PREDICATE p-mem?)
                 (:PRINT-FUNCTION print-p-mem))
  (rule-name nil) ; the name of the rule corresponding to it
  (inter-test nil) ; the name of the inter test function
  (consequent nil) ; the consequent of the tule (un-instantiated)
  (contradiction nil); weather its contradiction or not.
  (var-loc nil) ; ((?x fld index)...)

(defun print-p-mem (p-mem st ignore)
  (format st "P-mem :~A" (p-mem-rule-name p-mem)))

(defun show-p-mem (p-mem-list)
  (dolist (p-mem p-mem-list)
    (format t "~%Rule Name :~a" (p-mem-rule-name p-mem))
    (format t "~%Contradiction:~a" (p-mem-contradiction p-mem)))

```

```
(format t "~%Test Name :~a" (p-mem-inter-test p-mem))
(format t "~%Consequent:~a" (p-mem-consequent p-mem))
(format t "~%Var-loc :~a" (p-mem-var-loc p-mem))))
```

2.3 Build.lsp

```
(in-package :IRETE)
;;; Special variable declaration

(defvar *dir* 'r) ; special variable to determine the direction of build rete net
(defvar *r-env* nil) ; place to hold the binding from right
(defvar *l-env* nil) ; place to hold the binding from left
(defvar *mem-id* 0) ; Counter for alpha-node
(defvar *con-id* 0) ; Counter for t-const-node
(defvar *and-id* 0) ; Counter for and-node
(setf *con-id* 0)
(setf *and-id* 0)
(setf *mem-id* 0)

;;; Variable Needed for debug and tree traverse
(defvar *tc* nil)(setf *tc* nil)
(defvar *tch* nil)(setf *tch* nil)
(defvar *al* nil)(setf *al* nil)
(defvar *ad* nil)(setf *ad* nil)
(defvar *p* nil)(setf *p* nil)

... *****
;;;
;;; Main function
;;; *rule*-test is a test over all predicate
... *****
;;;

(defun create-rete ()
  (setf *r-env* nil)(setf *l-env* nil)(setf *dir* 'r)
  (create-p-mem (process-antecedents (decompose-ants))))

;;; Decompose each antecedent into a antecedent and a test
;;; Input: ((p1 ..):test #'f (p2 ..))
;;; Output: (((p1 ..) #'f) ((p2 ..) nil))

(defun decompose-ants (&optional (lhs (rule-lhs *rule*)) &aux nlhs)
  (do ((ants lhs (rest ants))
      (nlhs nil)
      ((endp ants) (reverse nlhs))
      (cond ((and (listp (first ants)) (eq (second ants) :test))
             (push (list (first ants) (third ants)) nlhs)
             (setf ants (rest(rest ants))))
          (t (push (list (first ants) nil) nlhs)))))

;;; Main function in build a rete for each rule
;;; Input : (((p1 ..) #'f) ((p2 ..) nil)) .
```

;;; Output: alpha-beta Memory Node .

```
(defun process-antecedents (lhs)
  (cond ((endp lhs) nil)
        (t (anded (process-antecedents (rest lhs))
                   (process (first lhs)) ))))
```

;;; Process one antecedent and create type-checking ,t-const node,alpha-beta

;;; Input :((p1 ..) #f)

;;; Output: alpha-beta Memory Node .

```
(defun process (ant)
  (update1-l-r-env ant) ;; update r-env,l-env
  (setf check1 (compute-check1 ant)) ;; constant check
  (setf check2 (compute-check2 ant)) ;; var equality
  (setf check3 (second ant)) ;; function name
  ;;If there is a test create t-const else create type-check-node only
  (if (or check1 check2 check3)
      (alpha (t-const check1 check2 check3 (type-check ant)))
      (alpha (type-check ant))))
```

;;; Update the r-env ,l-env according to the new pattern

;;; Input :((p1 ..) #f) , *dir*

;;; Output :((fld1 var index))

```
(defun update1-l-r-env (ant)
  (cond ((eq *dir* 'r)
        (setf *dir* 'l)
        (setf *r-env* (generate-env ant)))
        ((eq *dir* 'l)
        (setf *l-env* (generate-env ant))))))
```

;;; Input :((p1 ..) #f)

;;; Output :((fld1 1 var1) ...)

```
(defun generate-env (ant &aux (env nil))
  (do ((ant1 (first ant)) (rest ant1))
      (env nil)
      ((endp ant1) (reverse env))
      (when (and (field? (first ant1)) (variable? (second ant1)))
          (push `((first ant1) 1 ,(second ant1)) env))))
```

;;; Check the constant field in a predicate

;;; Input :((p1 ..) #f)

;;; Output :((fld val)...) ;

```
(defun compute-check1 (ant &aux (env nil))
  (do ((ant1 (first ant)) (rest ant1))
      (env nil))
```

```
((endp ant1) (reverse env))
(when (and (field? (first ant1)) (not (variable? (second ant1))))
  (push `,(first ant1) ,(second ant1) ) env))))
```

```
;;; Check if two or more variable are equal
;;; Input :((p1 ..) #f)
;;; Output :(fld1 fld2)
```

```
(defun compute-check2 (ant &aux (env nil))
  (do ((ant1 (first ant) (rest ant1))
      (env nil)
      ((endp ant1) (reverse env))
      (when (field? (first ant1))
        (setf fld (first ant1))
        (setf var (second ant1))
        (do ((rem (rest(rest ant1)) (rest rem) ))
            ((endp rem))
            (when (and (field? (first rem)) (eq (second rem) var))
              (push `,(fld ,(first rem)) env)
              (return t) ))))))
```

```
;;; Create type checking node , If it exist return it
;;; Input :((p1 ..) #f)
;;; Output :Type Checking node
```

```
(defun type-check (ant)
  (setf tch (lookfor-tch ant))
  (if tch tch (create-type-checking ant)))
```

```
;;; Create alpha-mem node and associate it with the node
;;; Input : type-checking node or t-const or and node
;;; Output : Alpha-beta memory-node
```

```
(defun alpha (node)
  (cond
    ((tcheck-node? node)
     ;; if it exist return it else create a new one
     (lookfor-alpha-beta node))
    ((and (t-const-node? node)
         (alpha-beta? (t-const-node-next-node node)))
     (t-const-node-next-node node))
    ((t-const-node? node)
     (setf *mem-id* (1+ *mem-id*))
     (setf (t-const-node-next-node node)
           (make-alpha-beta :id *mem-id* :prev-node node))
     (push (t-const-node-next-node node) *al*) ;;=>test
     (push (t-const-node-next-node node) (reps-alpha *reps*))
     (t-const-node-next-node node))
```

```

((and (and-node? node) (and-node-o-mem node))
  (and-node-o-mem node))
((and-node? node)
  (setf *mem-id* (1+ *mem-id*))
  (setf (and-node-o-mem node)
    (make-alpha-beta :id *mem-id* :prev-node node))
  (push (and-node-o-mem node) *al*) ;;=>test
  (and-node-o-mem node)
  )))

```

;;; look if one of the node is a mem node ;; associated with type-checking

```

(defun lookfor-alpha-beta (tcheck-node)
  (dolist (node (tcheck-node-next-nodes tcheck-node))
    (when (alpha-beta? node)
      (return-from lookfor-alpha-beta node)))
  (create-alpha-beta tcheck-node))

```

;;; create memory node
 ;;; join it to the previous node

```

(defun create-alpha-beta (tcheck-node)
  (setf *mem-id* (1+ *mem-id*))
  (setf ab (make-alpha-beta :id *mem-id* :prev-node tcheck-node))
  (push ab *al*) ;;;test ==>
  (push ab (tcheck-node-next-nodes tcheck-node))
  (push ab (reps-alpha *reps*))
  ab)

```

;;; Look if a type-checking node for the ant exist previosly

```

(defun lookfor-tch (ant)
  (setf p-name (first(first ant)))
  (dolist (tch (rete-type-checking *rete*))
    (when (eq p-name (tcheck-node-name tch))
      (return-from lookfor-tch tch) )))

```

;;; Create a structure and return it
 ;;; and join it to the root

```

(defun create-type-checking (ant)
  (setf tc (make-tcheck-node :name (first(first ant))))
  (push tc *tch*) ;;;=>
  (push tc (rete-type-checking *rete*))
  tc)

```

;;; if there exist t-const return it else create a t-const node

```

(defun t-const (check1 check2 check3 tcheck-node)
  (setf t-const-node (lookfor-t-const check1 check2 check3 tcheck-node))

```

```
(if t-const-node
    t-const-node
    ;; to make closure around the function
    (eval `(create-t-const ',check1 ',check2 ',check3 ',tcheck-node))))
```

```
;;; Look if a t-const node for the type-checking node. is exist previously
;;; must have the same condition
```

```
(defun lookfor-t-const (ck1 ck2 ck3 tcheck-node)
  (dolist (node (tcheck-node-next-nodes tcheck-node))
    (when (and (t-const-node? node)
               (t-const-equal? node ck1 ck2 ck3))
      (return-from lookfor-t-const node))))
```

```
;;; create t-const node structure and return it
;;; join it to the previous type-checking node
```

```
(defun create-t-const (ck1 ck2 ck3 tcheck-node)
  (setf *con-id* (1+ *con-id*))
  (setf tc (make-t-const-node
            :id *con-id*
            :check1 ck1
            :check2 ck2
            :check3 ck3))
  (push tc (tcheck-node-next-nodes tcheck-node))
  (push tc *tc*) ;;;=>
  tc)
```

```
;;; check if the tests are in this t-const-node or not
(defun t-const-equal? (node ck1 ck2 ck3 )
  (and
   (equal (t-const-node-check3 node) ck3)
   (equal-ck (t-const-node-check2 node) ck2)
   (equal-ck (t-const-node-check1 node) ck1))))
```

```
(defun equal-ck (ck1 ck2)
  (unless (= (length ck1) (length ck2)) (return-from equal-ck nil))
  (dolist (item ck1)
    (unless (member item ck2 :test #'equal) (return-from equal-ck nil)))
  t)
```

```
;;; Main function to join between antecedents
;;; Create and-node , p memory node
;;; Input :two alpha-beta nodes
;;; Output :one alpha-beta
```

```
(defun anded (r-mem l-mem)
  (if (null r-mem) (return-from anded l-mem))
  (setf inter-test (update2-l-r-env)) ;; update r-env,l-env,dir,compute inter-test
  (setf Out-mem-and-node (lookfor-and-node inter-test l-mem r-mem))
```



```
(if Out-mem-and-node
  Out-mem-and-node
  (alpha(create-and-node inter-test l-mem r-mem))))
```

```
;;; Input : *r-env* *l-env* *dir*
;;; Output : *r-env* after put l-env at the begining of the r-env
;;;         and increment the index of the old *r-env*
;;;         and return the inter-test
```

```
(defun update2-l-r-env (&aux (nrenv nil))
  (dolist (renv *r-env*)
    (push `((first renv) ,(1+ (second renv)) ,(third renv) ) nrenv))
  (setf nrenv (reverse nrenv))
  (setf inter-test (compute-inter-test *l-env* nrenv))
  (dolist (lenv *l-env*)
    (push lenv nrenv))
  (setf *r-env* nrenv)
  (setf *l-env* nil)
  inter-test)
```

```
;;; Compute the inter test between the right and the left env
;;; Input : R-env L-env
;;; Output : Test
```

```
(defun compute-inter-test (l-env r-env &aux (test nil))
  (dolist (L l-env)
    (setf lvar (third L))
    (dolist (R r-env)
      (setf rvar (third R))
      (when (eq lvar rvar)
        (push `((first l) ,(second l) ,(first R) ,(second R)) test))))
  test)
```

```
;;; create and-node and join it to the mem-nodes
```

```
(defun create-and-node (test l r)
  (setf *and-id* (1+ *and-id*))
  (setf and-node (make-and-node
    :id *and-id*
    :check1 test
    :l-mem l
    :r-mem r))
  (push and-node *ad*) ;;;=>
  (push and-node (alpha-beta-r-and r))
  (push and-node (alpha-beta-l-and l))
  and-node)
```

```
;;; look if there exist a previously created and node
;;; if yes return the output mem
```

```

(defun lookfor-and-node (test l r)
  (dolist (left-and (alpha-beta-l-and l))
    (when (and (member left-and (alpha-beta-r-and r) :test #'eq)
              (equal-and? left-and test))
      (return-from lookfor-and-node (and-node-o-mem left-and))))))

(defun equal-and? (anode test)
  (unless (= (length test) (length (and-node-check1 anode)))
    (return-from equal-and? nil))
  (dolist (item (and-node-check1 anode))
    (unless (member item test :test #'equal) (return-from equal-and? nil)))
  t)

;;; After finish build a rete nets for a rule you need to create a p_mem
;;; for this rule and associate it to the last node
;;; if there is a test in whole rule you must process this test
;;; Input :alpha-beta
;;; Output :p-mem
(defun create-p-mem (node &optional (inter-test-fun (rule-test *rule*)))
  (eval `(setf p (make-p-mem
                  :rule-name ,(rule-name *rule*)
                  :contradiction ,(rule-contradiction *rule*)
                  :inter-test ,inter-test-fun
                  :consequent ,(rule-rhs *rule*)
                  :var-loc ,(compute-var-loc))))
  (push p *p*) ;;;=>
  (push p (alpha-beta-p-mem node))
  p)

;;;
;;; find the var loc using the *r-env* and vars in *rule*
;;; Output : ((?x fld index)...)

(defun compute-var-loc (&aux (loc nil))
  (dolist (var (rule-vars *rule*))
    (dolist (binding *r-env*)
      (when (eq (third binding) var)
        (push `(,var ,(first binding) ,(second binding)) loc)
        (return t))))
  loc)

2.4 Reason.lsp
;;;(loose coupling)
(in-package :lrete)
(defvar inst-under-exe nil)

(defun start-reasoning (&aux (repeat t))
  (loop

```

```

(unless repeat
  (format t "~%*****")
  (format t "~%Reasoning terminated")
  (format t "~%To look for the data use - show-data ")
  (format t "~%To look for a specific data use - rfetch (...) ")
  (format t "~%Join operation      =~a" Joperation)
  (format t "~%Rule instantiated   =~a" instrule)
  (format t "~%Execute Normal Rule  =~a" ERule)
  (format t "~%Execute Contradiction Rule =~a" ECRule)
  (format t "~%Label Computation    =~a" Lcomp)
  (format t "~%*****")
  (return))
;;Find all applicable rule and store the instances in the conflict set
(loop
  (setf wme (pop (reps-queue *reps*)))
  (when (null wme) (return))
  (process-wme wme))
;;First of all : act all contradiction rules
(act-all-contradiction-rules (reps-conflict-set *reps*))
(setf repeat nil)
;; loop until a new datum is inserted or no instant in conflict set
(loop
  (setf inst (resolve-conflicts))
  (when (null inst) (return)) ; Empty conflict-set: finish resolution.
  (when (act inst) ;; When the act insert a new datum in queue-process it.
    (incf erule)
    (setf repeat t)
    (return)))
))

;;; Input :Conflict set
;;; Operation : act all contradiction rule
;;; Output :the remaining instance in the conflict set

(defun act-all-contradiction-rules (conflict-set &aux (noncont-rule nil))
  (dolist (inst conflict-set)
    (when (instance-contradiction inst)
      (act inst)
      (incf ecrule))
    (unless (instance-contradiction inst)
      (push inst noncont-rule)))
  (setf (reps-conflict-set *reps*) noncont-rule))

;;; You can put here any creteria you want
;;; I use a simple strategy (the last in first out)
;;; It will execute the first nonempty label instance
(defun resolve-conflicts ()
  (loop

```

```

(setf inst (pop(reps-conflict-set *reps*)))
(unless inst (return))
(setf label (have-new-env? inst))
(when label
  (setf (instance-label inst) label)
  (return-from resolve-conflicts inst)))
nil)

```

```

;;; Input   : Instance
;;; Operation : return the label if it exist or compute it
;;; Output  : label

```

```

(defun have-new-env? (inst &aux r)
  (setf A (mapcar #'(lambda (f)(datum-node f))
                 (instance-antecedents inst)))
  (find-new-envs nil (list (atms-empty-env *atms*) A))

```

```

;;; Input :Datum
;;; Insert any completely instantiated rule in conflict set.
(defun process-wme (wme)
  ;; Find all next-node of the correspond type-check node
  (setf tch (find-tcheck (get-pred-name wme)))
  (unless tch ;; No type checking for this predicate
    (push wme (reps-derived *reps*))
    (return-from process-wme t))
  (setf next-nodes (tcheck-node-next-nodes tch))
  ;;Next node :alpha or t-const
  (dolist (node next-nodes)
    (when (alpha-beta? node)
      (push wme (alpha-beta-datums node))
      (process-alpha node (list wme)))
    (when (t-const-node? node)
      (when (and
             (process-check1 node wme)
             (process-check2 node wme)
             (process-check3 node wme))
        (push wme (alpha-beta-datums (t-const-node-next-node node)))
        (process-alpha (t-const-node-next-node node)
                       (list wme)))))))

```

```

;;; Input : Predicate name
;;; Output : type checking node
(defun find-tcheck (p-name)
  (dolist (tch (rete-type-checking *rete*))
    (when (eq p-name (tcheck-node-name tch))
      (return-from find-tcheck tch)))
  nil)

```

```

;;; Input : t-const-node , datum

```

```

;;; Output : t , nil
;;; Check the constant values ((fld value)(fld value)...)
;;; Datum here is a single tuple.
(defun process-check1 (t-const datum)
  (dolist (check (t-const-node-check1 t-const))
    (unless (eq (fld-value (first check) 1 datum) (second check))
      (return-from process-check1 nil))))
  t)

```

```

;;; Input : t-const-node , datum
;;; Output : t , nil
;;; Check the fields equality ((fld1 fld2)(fld4 fld5)...)
;;; Datum here is a single tuple.
(defun process-check2 (t-const datum)
  (dolist (check (t-const-node-check2 t-const))
    (unless (eq (fld-value (first check) 1 datum)
                (fld-value (second check) 1 datum))
      (return-from process-check2 nil))))
  t)

```

```

;;; Input : t-const-node , datum
;;; Output : t , nil
;;; Check the test function ((fld1 fld2)(fld4 fld5)...)
;;; Datum here is a single tuple.
(defun process-check3 (t-const datum)
  (unless (t-const-node-check3 t-const)
    (return-from process-check3 t))
  (setf tuple (second(first(datum-fact datum))))
  (setf flds (reverse(extract-flds (list tuple))))
  (apply (t-const-node-check3 t-const)
    (mapcar #'(lambda (fld)
              (fld-value fld 1 datum))
            flds)))

```

```

;;; Input : The Field name, Index ,datum
;;; Output : The value of the field; if it exist
(defun fld-value (fld index datum)
  (setf tuples (datum-fact datum))
  (dolist (tuple tuples)
    (when (= (first tuple) index)
      (do ((pattern (second tuple) (rest pattern)))
          ((null pattern) nil)
        (when (eq (first pattern) fld)
          (return-from fld-value (second pattern)))))))

```

```

;;;Input : List of New Datums
;;;Process first Left and-node ,then right and-node ,
;;; finally look for p-mem if it exist.
(defun process-alpha (alpha new-datums)

```

```
(process-l-and alpha new-datums)
(process-r-and alpha new-datums)
(process-p-mem alpha new-datums))
```

```
;;; Join the new datums with fact stored in the right mem of the left and
;;; Process the out memory of the and-node
(defun process-l-and (alpha new-datums)
  (setf lands (alpha-beta-l-and alpha))
  (dolist (land lands)
    ;;; the following condition is used to prevent repeat work
    ;;; when the l and r mem for node AND node is the same and is
    ;;; traversed only in the right direction
    (unless (equal (and-node-r-mem land) (and-node-l-mem land))
      (setf new-joind-tuples
        (join new-datums (alpha-beta-datums (and-node-r-mem land)
          (and-node-o-mem land) (and-node-check1 land))))
      (insert-joined-tuple new-joind-tuples (and-node-o-mem land))
      (when new-joind-tuples
        (process-alpha (and-node-o-mem land) new-joind-tuples))))))
```

```
;;; Join the new datums with fact stored in the right mem of the right and
;;; Process the out memory of the and-node
(defun process-r-and (alpha new-datums)
  (setf rands (alpha-beta-r-and alpha))
  (dolist (rand rands)
    (setf new-joind-tuples
      (join (alpha-beta-datums (and-node-l-mem rand)) new-datums
        (and-node-o-mem rand) (and-node-check1 rand)))
    (insert-joined-tuple new-joind-tuples (and-node-o-mem rand))
    (when new-joind-tuples
      (process-alpha (and-node-o-mem Rand) new-joind-tuples))))))
```

```
;;; Check each new datums with inter-test function (if it exist)
;;; Store the good one in the conflict set
(defun process-p-mem (alpha new-datums)
  (unless (alpha-beta-p-mem alpha)
    (return-from process-p-mem t))
  (setf p-mems (alpha-beta-p-mem alpha))
  (dolist (p-mem p-mems)
    (dolist (datum new-datums)
      (when (process-inter-test p-mem datum)
        (update-conflict-set
          (p-mem-rule-name p-mem)
          (p-mem-contradiction p-mem)
          (p-mem-consequent p-mem)
          (generate-binding p-mem datum)
          (datum-datum-list datum) ))))))
```

```
(defun update-conflict-set (rname cont consequence binding antecedents)
```

```
(incf instrule)
(setf lambda-vars (extract-vars consequence))
(eval `(push (make-instance
  :rule-name ',rname
  :contradiction ',cont
  :consequence #(lambda ,lambda-vars ,@consequence)
  :binding ',binding
  :antecedents ',antecedents ;;list of datums
  :vars ',lambda-vars)
  (reps-conflict-set *reps*))))
```

```
;;; Input : p-mem , datum
;;; Output : t , nil
;;; Check the inter test function if it exist
(defun process-inter-test (p-mem datum)
  (unless (p-mem-inter-test p-mem) (return-from process-inter-test t))
  (when (p-mem-inter-test p-mem)
    (setf vals (extract-vals datum))
    (apply (p-mem-inter-test p-mem)
           vals))))
```

```
;;; Input : Datum
;;; Output : list of variable values
(defun extract-vals (datum &aux (vals nil) (temp nil))
  (dolist (tp (reverse(datum-fact datum)))
    (setf temp nil)
    (do ((tuple (second tp) (rest tuple)))
        ((null tuple) t)
      (when (field? (first tuple))
        (push (second tuple) temp))))
    (setf vals (append (reverse temp) vals)))
  vals)
```

```
;;; Input : p-mem node ,datum
;;; Output : ((?x . value)(..))
(defun generate-binding (p-mem datum &aux (binding nil))
  (dolist (loc (p-mem-var-loc p-mem))
    (push (cons (first loc)
                (fld-value (second loc) (third loc) datum))
          binding))
  binding)
```

```
;;; Input :right and left datums , out-mem of the and-node ,test
;;; Output : a list of datums (joined tuple test)
(defun join (ldatums rdatums alpha test &aux (result nil) )
  (dolist (ld ldatums)
    (dolist (rd rdatums)
      (incf joperation)
      (setf updated-r (inc-index rd))
```

```

(setf joined-tuple (append (datum-fact ld) updated-r))
(setf Ld1 (datum-datum-list ld))
(setf rd1 (datum-datum-list rd))
(setf d (make-datum
         :fact joined-tuple
         :datum-list (append Ld1 rd1)))
(when (process-andcheck d test)
      (push d result)))
result)

```

```

;;; Input : new-joined tuple
;;; Operation : push them at the datum for the alpha mem
(defun insert-joined-tuple (new-joind-tuples alpha)
  (setf (alpha-beta-datums alpha)
        (append new-joind-tuples (alpha-beta-datums alpha))))

```

```

;;; Input :Datums
;;; Output :fact in the datum after increment index
;;; increment the index in each tuple by one
(defun inc-index (datum &aux (result nil))
  (setf facts (datum-fact datum))
  (dolist (fact facts)
    (push (list (1+ (first fact)) (second fact)) result))
  (reverse result))

```

```

;;; Input : joined tuple , test ((fld1 index fld2 ind)(...))
;;; Output : t or nil
(defun process-andcheck (d tests)
  (dolist (test tests)
    (unless
      (eq
       (fld-value (first test) (second test) d)
       (fld-value (third test) (fourth test) d))
      (return-from process-andcheck nil)))
  t)

```

```

;;; Input :instance
;;; Output : t . if it insert a new data
(defun act (inst)
  (setf inst-under-exe inst)
  (when inst
    (apply (instance-consequence inst)
           (instantiate-variables (instance-vars inst)
                                 (instance-binding inst)))))

```

```

(defun instantiate-variables (consequent binding)
  (sublis binding consequent))

```



```

;;;Input : fact
;;;Insert in the queue if not exist
;;;We have four cases :
;;; 1- new just,new datum : propogate and repeat reasoning
;;; 2- new just,old datum : propogate
;;; 3- old just,new datum : repeat reasoning
;;; 4- old just,old datum : nothing

(defun assert! (fact &optional(inst inst-under-exe)
               &aux datum node )
  (setf datum (is-exist? (cons '1 (list fact))))
  (when datum
    (setf node (datum-node datum))
    (justify-node (instance-rule-name inst)
                  (instance-label inst)
                  nil;;; this nil will prevent atms recompute the label
                  node
                  (mapcar #'(lambda (f) (datum-node f))
                          (instance-antecedents inst)))
    (return-from assert! nil))
  (setf datum (create-datum (cons '1 (list fact)) nil ))
  (push datum (reps-queue *reps*))
  (setf node (datum-node datum))
  (justify-node (instance-rule-name inst)
                (instance-label inst)
                nil;;; this nil will prevent atms recompute the label
                node
                (mapcar #'(lambda (f) (datum-node f))
                        (instance-antecedents inst))))
  t)

```

```

;;; 1-Assert a new nogood
;;; 2-Justify the contradiction
;;; **This will not lead to new reasoning
(defun assert-nogood! (&optional(inst inst-under-exe))
  (justify-nogood-node
   (instance-rule-name inst)
   nil ;;; no label are computed
   t ;;; this nil will make atms compute the label
   (mapcar #'(lambda (f) (datum-node f))
            (instance-antecedents inst))))

```

;Copyright (c) 1986-1993 Kenneth D. Forbus, Johan de Kleer and Xerox Corporation. All Rights Reserved.

```

(defun quotize (pattern)
  (cond ((null pattern) nil)
        ((variable? pattern) pattern)
        ((not (listp pattern)) (list 'QUOTE pattern)))

```

```
(t `(cons ,(quotize (car pattern))
          ,(quotize (cdr pattern))))))
```

2.5 ATMS.lsp

```
(in-package :rete)
(defvar *atms* nil)
(setf *atms* nil)
```

```
;;; STRUCTURE
;;; 1- ATMS
;;; 2- NODE
;;; 3- JUSTIFICATION
;;; 4- ENVIRONMENT
```

```
(defstruct (atms (:print-function print-atms))
  (title nil)
  (nodes nil)      ;; all nodes
  (justs nil)      ;; all justification
  (good-env nil)   ;; good environment
  (nogood-env nil) ;; nogood environment
  (contra-node nil) ;; dummy contradiction node
  (empty-env nil) ;; {{{}} ;; hold in all environment
```

```
(defstruct (node (:print-function print-node))
  (datum nil)
  (label nil)
  (justs nil)      ;;justify this node
  (consequences nil) ;;this node belong to these justification
  (contradictory? nil)
  (assumption? nil))
```

```
(defstruct (just (:print-function print-just))
  (informant nil)
  (consequence nil)
  (antecedents nil))
```

```
(defstruct (env (:print-function print-env))
  (assumptions nil)
  (nodes nil))
```

```
;;; PRINT
;;; 1- SUMMARY DEF.
;;; 2- FULL DETAIL.
```

```
(defun print-atms (atms stream ignore)
  (format t "atms--a" (atms-title atms)))
```

```
(defun print-node (node stream ignore)
  (format t "node--a" (node-datum node)))
```

```

(defun print-just (just stream ignore)
  (format t "just-~a" (just-informant just)))

(defun print-env (env stream ignore)
  (format t "env-~a" (env-assumptions env)))

(defun print-full-atms ()
  (format t "~%Title:~a" (atms-title *atms*))
  (format t "~%Nodes:~a" (atms-nodes *atms*))
  (format t "~%Justs:~a" (atms-Justs *atms*))
  (format t "~%Good-env:~a" (atms-good-env *atms*))
  (format t "~%Nogood-env:~a" (atms-nogood-env *atms*))
  (format t "~%Contra-node:~a" (atms-contra-node *atms*))
  (format t "~%Empty-env:~a" (atms-empty-env *atms*)))

(defun print-full-node (nodes)
  (dolist (node nodes)
    (format t "~%Datum:~a" (node-datum node))
    (format t "~%Label:~a" (node-label node))
    (format t "~%Justs:~a" (node-justs node))
    (format t "~%Consequences:~a" (node-consequences node))
    (format t "~%Contradictory:~a" (node-contradictory? node))
    (format t "~%Assumption:~a" (node-assumption? node))))

(defun print-full-just (justs)
  (dolist (just justs)
    (format t "~%Informant:~a" (just-informant just))
    (format t "~%Antecedents:~a" (just-antecedents just))
    (format t "~%Consequence:~a" (just-consequence just))))

(defun print-full-env (envs)
  (dolist (env envs)
    (format t "~%Assumptions:~a" (env-assumptions env))
    (format t "~%nodes:~a" (env-nodes env))))

;;; CREATE ALL THE STRUCTURE
;;; 1- ATMS
;;; 2- CONTRADICTION NODE
;;; 3- EMPTY-ENV
;;; 4- ENVIRONMENT
;;; 5- ASSUMPTION NODE

(defun create-atms (title)
  (setq *atms* (make-atms :TITLE title))
  (create-contradiction-node)
  (create-empty-env)
  *atms*)

```

```

(defun create-node (datum &key assumptionp contradictoryp &aux node)
  (setq node
    (make-node
      :DATUM datum
      :ASSUMPTION? assumptionp
      :CONTRADICTION? contradictoryp) )
  (push node (atms-nodes *atms*))
  (when assumptionp
    (setq e (create-env (list node)))
    (push e (node-label node))
    (push node (env-nodes e)))
  node)

(defun create-contradiction-node ()
  (setf (atms-CONTRA-NODE *atms*)
    (create-node "The contradiction" :CONTRADICTION? t)))

(defun create-empty-env ()
  (setf (atms-empty-env *atms*) (create-env nil)))

(defun create-env (assumptions &aux e)
  (setq e (make-env :ASSUMPTIONS assumptions))
  (setq exist (lookup-env e (atms-good-env *Atms*)))
  (unless (or exist (nogood? e) )
    (setf (atms-good-env *Atms*) (cons e (atms-good-env *Atms*))))
  (if exist exist e) )

(defun assume-node (datum)
  (create-node datum :assumptionp t))

;;; PROVIDE THE JUSTIFICATION FOR
;;; 1- NODE
;;; 2- CONTRADICTION NODE

(defun eq-just (j1 j2)
  (and (equal (just-informant j1) (just-informant j2))
    (equal (just-antecedents j1) (just-antecedents j2))
    (equal (just-consequence j1) (just-consequence j2))))

;;; if recompute is true then ignore the label
;;; else use the input label

(defun justify-node (informant label recompute
  consequence antecedents &aux just)
  (setq just (make-just
    :INFORMANT informant
    :CONSEQUENCE consequence
    :ANTECEDENTS antecedents))

```

```

;;; (atms-justs *atms*) ;;;
(unless (member just (node-justs consequence) :test #'eq-just)
  (push just (atms-justs *atms*))
  (push just (node-justs consequence))
  (dolist (node antecedents) (push just (node-consequences node)))
  (propagate just nil (list (atms-empty-env *atms*)) label recompute )
  (return-from justify-node t))
nil)

```

```

;;; if recompute is true then ignore the label
;;; else use the input label

```

```

(defun justify-nogood-node (informant label recompute nodes)
  (justify-node informant
    label recompute
    (atms-contra-node *atms*)
    nodes))

```

```

;;; HELPING FUNCTION

```

```

;;; Check if there exist an environment in the <env-list> Subset of <env>.

```

```

(defun env-subsumed-by (env env-list)
  (dolist (old-env env-list)
    (when (subsetp (env-assumptions old-env)
      (env-assumptions env))
      (return old-env))))

```

```

;;;Look for an environment in a list of environment

```

```

(defun lookup-env (e env-list)
  (dolist (env env-list)
    (when (equal-env? env e) (return-from lookup-env env)))
  nil)

```

```

(defun compare-env (e1 e2)
  (cond ((equal-env? e2 e1) :EQ)
        ((subset-env? e1 e2) :S12)
        ((subset-env? e2 e1) :S21)
        (t :NEQ)))

```

```

(defun equal-env? (e1 e2)
  (and (subsetp (env-assumptions e2)(env-assumptions e1))
    (subsetp (env-assumptions e1)(env-assumptions e2))))

```

```

(defun subset-env? (e1 e2)
  (if (subsetp (env-assumptions e1) (env-assumptions e2)) t nil))

```

```

(defun union-env (e1 e2)
  (create-env

```

```
(union (env-assumptions e1) (env-assumptions e2))))
```

```
;;; Propagate the label through the dependency
```

```
(defun propagate (just antecedent envs label recompute &aux new-envs)
  (if recompute
    (setq new-envs (find-new-envs antecedent envs (just-antecedents just)))
    (setq new-envs label))
  (setq consequence (just-consequence just))
  (unless new-envs (return-from propagate nil))
  (when (node-contradictory? consequence )
    (dolist (env new-envs) (new-nogood env))
    (return-from propagate nil))
  (setq new-envs (whose-still-new new-envs consequence))
  (unless new-envs (return-from propagate nil))
  (dolist (supported-just (node-consequences consequence ))
    (propagate supported-just consequence new-envs nil t )))
```

```
(defun whose-still-new (new-env node)
  (setq old-env (node-label node))
  (dolist (nenv new-env)
    (dolist (oenv old-env)
      (case (compare-env nenv oenv)
        (:S12 (setf (env-nodes oenv)
                    (delete node (env-nodes oenv):count 1))
              (setf (node-label node)
                    (delete oenv (node-label node):count 1))))
        ((:EQ :S21) (setf new-env
                          (delete nenv new-env :count 1))))))
  (dolist (nenv new-env)
    (push node (env-nodes nenv))
    (push nenv (node-label node)))
  new-env)
```

```
(defun find-new-envs (antecedent envs antecedents &aux new-envs)
  ;;;envs :new env come to the just from antecedent
  (incf lcomp)
  (drop-nogood
   (make-minimal
    (find-sound-complete-envs antecedent envs antecedents))))
```

```
(defun find-sound-complete-envs (antecedent envs antecedents)
  (dolist (node antecedents)
    (unless (eq node antecedent) ;; in order to process only the new environment
      (setq result nil)
      (dolist (env envs)
        (dolist (node-env (node-label node))
          (setq result (cons (union-env env node-env) result))))
      (setq envs result))))
```

```

envs)

(defun make-minimal (env-list &aux (result nil))
  (setq temp-list (copy-list env-list))
  (dolist (env env-list)
    (setq temp-list (delete env temp-list :count 1))
    (unless (env-subsumed-by env temp-list)
      (setq result (cons env result))))
  (setq temp-list (cons env temp-list)))
result)

(defun drop-nogood (env-list &aux (result nil))
  (setq temp-list (copy-list (atms-nogood-env *atms*)))
  (dolist (env env-list)
    (unless (env-subsumed-by env temp-list)
      (setq result (cons env result))))
  result)

(defun nogood? (env)
  (cond
    ((lookup-env env (atms-nogood-env *atms*)) t)
    ((env-subsumed-by env (atms-nogood-env *atms*)) t)
    ;; if it exist prev
    ;; if there exist e subset of env
    ;; else return nil
    (t nil)))

(defun new-nogood (cenv)
  (setf (atms-nogood-env *atms*)
        (cons cenv (atms-nogood-env *atms*)))
  (remove-env-from-labels cenv)
  (setf (env-nodes cenv) nil)
  (dolist (env (atms-good-env *atms*))
    (when (subset-env? cenv env)
      (remove-env-from-labels env)
      (setf (env-nodes env) nil)
      (setf (atms-good-env *atms*)
            (delete env (atms-good-env *atms*) :COUNT 1))))
  (setf (atms-nogood-env *atms*)
        (make-minimal (atms-nogood-env *atms*)))

(defun remove-env-from-labels (env)
  (dolist (node (env-nodes env))
    (setf (node-label node)
          (delete env (node-label node) :COUNT 1))))

```

Appendix 3

Morgue system Program

3.1 Rete.lsp

the same as before.

3.2 Def.lsp

```
;;; Morgue System
;;; Declaration Part
;;; 1- SPECIAL VARIABLES .
;;; 2- STRUCTURE USED .
;;; 3- PRINT FUNCTION .

(IN-PACKAGE :TRETE)

;;; RETE Production System
(defvar *REPS* nil)
(defvar *rule*)
(defvar *RETE* nil)

(defstruct (reps (:PREDICATE reps?)
               (:PRINT-FUNCTION print-reps))
  (title nil)
  (atms nil) ;Assumption beased truth maintenance system
  (rete nil) ;RETE network
  (alpha nil) ;Memory-node where all datum are stored there
  (derived nil) ;Datums that have no alpha are stored here
  (conflict-set nil) ;Conflict-set (instantiated rules)
  (queue nil)) ;Queue used conflict resolution.

(defun print-reps (reps st ignore)
  (format st "REPS:~A" (reps-title reps)))

(defun show-reps ()
  (format t "~%Title:~a" (reps-title *reps*))
  (format t "~%Atms:~a" (reps-atms *reps*))
  (format t "~%RETE:~a" (reps-rete *reps*))
  (format t "~%QUEUE:~a" (reps-queue *reps*))
  (format t "~%To show full atms Echo (print-full-atms)")
  (format t "~%To show All Data Echo (show-Data)")
  (format t "~%To show Conflict-set Echo (show-conflict-set)"))

;;; INSTANCE STRUCTURE
;;; Print Functions
```



```
;;; SHOW CONFLICT SET.
```

```
(defstruct (instance (:PREDICATE instance?)
  (:PRINT-FUNCTION print-instance))
  (rule-name nil)
  (contradiction nil) ;;weather its contradiction or not.
  (consequence nil) ;;un-instantiated consequence
  (vars nil) ;;variable in consequence
  (antecedents nil) ;;list of antecedents datums
  (label nil) ;;label of the antecedents
  (del nil) ;;tag used to detect the datum tha must be deleted
  (binding nil)) ;;assosiation list with variable binding
```

```
(defun print-instance (inst st ignore)
  (format st "~%Instance:-")
  (format st "~%Rule:~a" (instance-rule-name inst))
  (format st "~%Contradiction:~a" (instance-contradiction inst))
  (format st "~%Consequence:~a" (instance-consequence inst))
  (format st "~%Vars:~a" (instance-vars inst))
  (format st "~%Antecedents:~a" (instance-antecedents inst))
  (format st "~%Labels:~a" (instance-label inst))
  (format st "~%Deleted :~a" (instance-del inst))
  (format st "~%Binding:~a" (instance-binding inst)))
```

```
(defun show-conflict-set (&optional (conflict-set (reps-conflict-set *reps*)))
  (dolist (instance conflict-set)
    (print-instance instance t nil)))
```

```
;;; Temprrory structure used in rete builder
```

```
(defstruct (Rule) ;temp rule
  (name nil)
  (lhs nil)
  (test nil)
  (rhs nil)
  (flds nil)
  (vars nil)
  (contradiction nil))
```

```
;;;RETE NETWORK
```

```
;;; Print Functions
```

```
(defstruct (RETE (:PREDICATE rete?)
  (:PRINT-FUNCTION print-rete)) ;;; ROOT NODE
  (title 'Rete-nets)
  (type-checking nil) ;; Nodes to Check the predicate name
```

```
(defun print-rete (rete st ignore)
  (format st "RETE:~A" (rete-title rete)))
```

```

(defun show-rete ()
  (format t "~%Title:~a"          (rete-title *rete*))
  (format t "~%Type checking Nodes:~a" (rete-type-checking *rete*)))

;;;
;;;Type Checking Nodes
;;;
(defstruct (Tcheck-node (:PREDICATE tcheck-node?)
                      (:PRINT-FUNCTION print-Tcheck-node))
  (name nil)          ;; Name of the predicate
  (contradiction nil) ;; weather its a part of contradiction rule.
  (next-nodes nil))  ;; alpha-beta or t-const - list of nodes

(defun print-Tcheck-node (Tcheck-node st ignore)
  (format st "TCHECK:~A" (Tcheck-node-name Tcheck-node)))

(defun show-Tcheck-node (&optional (Tchecks (rete-type-checking *rete*)))
  (dolist (Tcheck Tchecks)
    (format t "~%Name:~a"      (Tcheck-node-name Tcheck))
    (format t "~%Contradiction:~a" (Tcheck-node-contradiction Tcheck))
    (format t "~%Next Nodes:~a"  (Tcheck-node-next-nodes Tcheck))))

;;;
;;; T-const nodes
;;;
(defstruct (T-const-node (:PREDICATE t-const-node?)
                       (:PRINT-FUNCTION print-T-const-node))
  (id 0)          ;; indexer
  (check1 nil)   ;; check the constant value ((field value)...)
  (check2 nil)   ;; check the field in one predicate ((field field)...)
  (check3 nil)   ;; intra-test-function name
  (contradiction nil) ;; weather its a part of contradiction rule.
  (next-node nil)) ;; alpha-beta -one node

(defun print-T-const-node (t-const-node st ignore)
  (format st "TC:~a" (t-const-node-id t-const-node))
  (if (t-const-node-check1 t-const-node) (format st " Const-test,"))
  (if (t-const-node-check2 t-const-node) (format st " Equal-var,"))
  (if (t-const-node-check3 t-const-node) (format st " Intra-Ext-fun,")))

(defun show-t-const-node (t-const-list)
  (dolist (t-const t-const-list)
    (format t "~%Id:~a"      (t-const-node-id t-const))
    (format t "~%Contradiction:~a" (t-const-node-contradiction t-const))
    (format t "~%Const-test:~a"  (t-const-node-check1 t-const))
    (format t "~%Equal-var:~a"   (t-const-node-check2 t-const))
    (format t "~%Intra-Ext-fun:~a" (t-const-node-check3 t-const)))

```

```
(format t "~%Next-node:~a" (t-const-node-next-node t-const))))
```

```
;;;
;;;
;;;
```

```
(defstruct (alpha-beta (:PREDICATE alpha-beta?)
  (:PRINT-FUNCTION print-alpha-beta))
  (id 0) ; indexer
  (Datums nil) ; List of facts
  (R-Ands nil) ; Right and node -list
  (L-Ands nil) ; Left and node -list
  (p-mem nil) ; P-mem node if it the next -list
  (contradiction nil) ; weather its a part of contradiction rule.
  (prev-node nil) ; pointer to the previous node -one element
```

```
(defun print-alpha-beta (alpha-beta st ignore)
  (format st "Alpha-beta:~A" (alpha-beta-id alpha-beta)))
```

```
(defun show-alpha-beta (mem-list)
  (dolist (mem mem-list)
    (format t "~%Id:~a" (alpha-beta-id mem))
    (format t "~%Contradiction:~a" (alpha-beta-contradiction mem))
    (format t "~%Datums:~a" (alpha-beta-datums mem))
    (format t "~%No of datums :~a" (length (alpha-beta-datums mem)))
    (format t "~%R-Ands:~a" (alpha-beta-R-ands mem))
    (format t "~%L-Ands:~a" (alpha-beta-L-ands mem))
    (format t "~%P-mems:~a" (alpha-beta-p-mem mem))
    (format t "~%Prev-node :~a" (alpha-beta-prev-node mem))
    (read))))
```

```
;;;
;;;
;;;
```

```
(defstruct (AND-node (:PREDICATE and-node?)
  (:PRINT-FUNCTION print-and-node))
  (id 0) ;; indexer
  (check1 nil) ;; check the var between predicates ((fld2 ind fld2 ind)..)
  (contradiction nil) ;; weather its a part of contradiction rule.
  (L-mem nil) ;; Left Memory Nodes -one element
  (R-mem nil) ;; Right Memory Nodes -one element
  (O-mem nil) ;; Output Memory Nodes -one element
```

```
(defun print-and-node (and-node st ignore)
  (format st "And:~A" (and-node-id and-node)))
```

```
(defun show-and-node (and-list)
  (dolist (and-node and-list)
    (format t "~%Id:~a"      (and-node-id and-node))
    (format t "~%Contradiction:~a" (and-node-contradiction and-node))
    (format t "~%Eq-var:~a"      (and-node-check1 and-node))
    (format t "~%Left mem :~a"   (and-node-L-mem and-node))
    (format t "~%Right mem :~a"  (and-node-R-mem and-node))
    (format t "~%Output mem :~a" (and-node-O-mem and-node))))
```

```
;;;
;;; P-memory nodes
;;;
```

```
(defstruct (P-mem (:PREDICATE p-mem?)
               (:PRINT-FUNCTION print-p-mem))
  (rule-name nil) ; the name of the rule corresponding to it
  (inter-test nil) ; the name of the inter test function
  (consequent nil) ; the consequent of the tule (un-instantiated)
  (contradiction nil); weather its contradiction or not.
  (var-loc nil) ; ((?x fld index)...)
)
```

```
(defun print-p-mem (p-mem st ignore)
  (format st "P-mem :~A" (p-mem-rule-name p-mem)))
```

```
(defun show-p-mem (p-mem-list)
  (dolist (p-mem p-mem-list)
    (format t "~%Rule Name :~a" (p-mem-rule-name p-mem))
    (format t "~%Contradiction:~a" (p-mem-contradiction p-mem))
    (format t "~%Test Name :~a" (p-mem-inter-test p-mem))
    (format t "~%Consequent:~a" (p-mem-consequent p-mem))
    (format t "~%Var-loc :~a" (p-mem-var-loc p-mem))))
```

3.3 Build.lsp

```
(in-package :tRETE)
;;; Special variable decleration

(defvar *dir* 'r) ; special variable to determine the direction of build rete net
(defvar *r-env* nil) ; place to hold the binding from right
(defvar *l-env* nil) ; place to hold the binding from left
(defvar *mem-id* 0) ; Counter for alpha-node
(defvar *con-id* 0) ; Counter for t-const-node
(defvar *and-id* 0) ; Counter for and-node
(setf *con-id* 0)
(setf *and-id* 0)
(setf *mem-id* 0)

;;; Variable Needed for debug and tree traverse
(defvar *tc* nil)
```

```

(setf *tc* nil)
(defvar *tch* nil)
(setf *tch* nil)
(defvar *al* nil)
(setf *al* nil)
(defvar *ad* nil)
(setf *ad* nil)
(defvar *p* nil)
(setf *p* nil)

... *****
;;;
;;; Main function
;;; *rule*-test is a test over all predicate
... *****
;;;

(defun create-rete ()
  (setf *r-env* nil)(setf *l-env* nil)(setf *dir* 'r)
  (create-p-mem (process-antecedents (decompose-ants))))

;;; Decompose each antecedent into a antecedent and a test
;;; Input: ((p1 ..):test #'f (p2 ..))
;;; Output: (((p1 ..) #'f) ((p2 ..) nil))

(defun decompose-ants (&optional (lhs (rule-lhs *rule*)) &aux nlhs)
  (do ((ants lhs (rest ants))
      (nlhs nil)
      ((endp ants) (reverse nlhs))
      (cond ((and (listp (first ants)) (eq (second ants) :test))
             (push (list (first ants) (third ants)) nlhs)
             (setf ants (rest(rest ants))))
          (t (push (list (first ants) nil) nlhs)))))

;;; Main function in build a rete for each rule
;;; Input : (((p1 ..) #'f) ((p2 ..) nil)) .
;;; Output: alpha-beta Memory Node .

(defun process-antecedents (lhs)
  (cond ((endp lhs) nil)
        (t (anded (process-antecedents (rest lhs))
                   (process (first lhs)) ))))

;;; Process one antecedent and create type-checking ,t-const node,alpha-beta
;;; Input :((p1 ..) #'f)
;;; Output: alpha-beta Memory Node .

(defun process (ant)
  (update1-l-r-env ant) ;; update r-env,l-env
  (setf check1 (compute-check1 ant)) ;; constant check
  (setf check2 (compute-check2 ant)) ;; var equality

```

```

(setf check3 (second ant))      ;; function name
;;If there is a test create t-const else create type-check-node only
(if (or check1 check2 check3)
    (alpha (t-const check1 check2 check3 (type-check ant)))
    (alpha (type-check ant))))

;;; Update the r-env ,l-env according to the new pattern
;;; Input :((p1 ..) #f) ,*dir*
;;; Output :((fld1 var index) ....)

(defun update1-l-r-env (ant)
  (cond ((eq *dir* 'r)
        (setf *dir* 'l)
        (setf *r-env* (generate-env ant)))
        ((eq *dir* 'l)
         (setf *l-env* (generate-env ant)))))

;;; Input :((p1 ..) #f)
;;; Output :((fld1 1 var1) ... )

(defun generate-env (ant &aux (env nil))
  (do ((ant1 (first ant) (rest ant1))
      (env nil))
      ((endp ant1) (reverse env))
    (when (and (field? (first ant1)) (variable? (second ant1)))
      (push `((first ant1) 1 ,(second ant1)) env))))

;;; Check the constant field in a predicate
;;; Input :((p1 ..) #f)
;;; Output :((fld val)...)

(defun compute-check1 (ant &aux (env nil) )
  (do ((ant1 (first ant) (rest ant1))
      (env nil))
      ((endp ant1) (reverse env))
    (when (and (field? (first ant1)) (not (variable? (second ant1))))
      (push `((first ant1) ,(second ant1)) env))))

;;; Check if two or more variable are equal
;;; Input :((p1 ..) #f)
;;; Output : (fld1 fld2)

(defun compute-check2 (ant &aux (env nil))
  (do ((ant1 (first ant) (rest ant1))
      (env nil))
      ((endp ant1) (reverse env))
    (when (field? (first ant1))
      (setf fld (first ant1))
      (setf var (second ant1))

```

```

(do ((rem (rest(rest ant1)) (rest rem) ))
    ((endp rem))
    (when (and (field? (first rem)) (eq (second rem) var))
        (push `(fld ,(first rem)) env)
        (return t )))))

;;; Create type checking node , If it exist return it
;;; Input :((p1 ..) #f)
;;; Output :Type Checking node

(defun type-check (ant)
  (setf tch (lookfor-tch ant))
  (if tch tch (create-type-checking ant)))

;;; Create alpha-mem node and associate it with the node
;;; Input : type-checking node or t-const or and node
;;; Output : Alpha-beta memory-node

;; (when (rule-contradiction *rule*)
;; (setf (alpha-beta-contradiction (t-const-node-next-node node)) t))

(defun alpha (node)
  (cond
    ((tcheck-node? node)
     ;; if it exist return it else create a new one
     (lookfor-alpha-beta node))
    ((and (t-const-node? node)
          (alpha-beta? (t-const-node-next-node node)))
     (t-const-node-next-node node))
    ((t-const-node? node)
     (setf *mem-id* (1+ *mem-id*))
     (setf (t-const-node-next-node node)
           (make-alpha-beta :id *mem-id* :prev-node node
                            :contradiction (rule-contradiction *rule*)))
     (push (t-const-node-next-node node) *al*) ;;=>test
     (push (t-const-node-next-node node) (reps-alpha *reps*))
     (t-const-node-next-node node))
    ((and (and-node? node) (and-node-o-mem node))
     (and-node-o-mem node))
    ((and-node? node)
     (setf *mem-id* (1+ *mem-id*))
     (setf (and-node-o-mem node)
           (make-alpha-beta :id *mem-id* :prev-node node
                            :contradiction (rule-contradiction *rule*)))
     (push (and-node-o-mem node) *al*) ;;=>test
     (and-node-o-mem node))))

;;; look if one of the node is a mem node ;; associated with type-checking

```

```

(defun lookfor-alpha-beta (tcheck-node)
  (dolist (node (tcheck-node-next-nodes tcheck-node))
    (when (alpha-beta? node)
      (return-from lookfor-alpha-beta node)))
  (create-alpha-beta tcheck-node))

;;; create memory node
;;; join it to the previous node

(defun create-alpha-beta (tcheck-node)
  (setf *mem-id* (1+ *mem-id*))
  (setf ab (make-alpha-beta :id *mem-id* :prev-node tcheck-node
                           :contradiction (rule-contradiction *rule*)))
  (push ab *al*) ;;;test ==>
  (setf (tcheck-node-next-nodes tcheck-node)
        (Rpush ab (tcheck-node-next-nodes tcheck-node)))
  (push ab (reps-alpha *reps*))
  ab)

;;; Look if a type-checking node for the ant exist previously

(defun lookfor-tch (ant)
  (setf p-name (first(first ant)))
  (dolist (tch (rete-type-checking *rete*))
    (when (eq p-name (tcheck-node-name tch))
      (return-from lookfor-tch tch) ) )

;;; Create a type checking structure and return it
;;; and join it to the root

(defun create-type-checking (ant)
  (setf tc (make-tcheck-node :name (first(first ant))
                           :contradiction (rule-contradiction *rule*)))
  (push tc *tch*) ;;;==>
  (push tc (rete-type-checking *rete*))
  tc)

;;; if there exist t-const return it else create a t-const node

(defun t-const (check1 check2 check3 tcheck-node)
  (setf t-const-node (lookfor-t-const check1 check2 check3 tcheck-node))
  (if t-const-node
      t-const-node
      ;; to make closure around the function
      (eval `(create-t-const ',check1 ',check2 ',check3 ',tcheck-node))))

;;; Look if a t-const node for the type-checking node. is exist previously
;;; must have the same condition

```



```
(defun lookfor-t-const (ck1 ck2 ck3 tcheck-node)
  (dolist (node (tcheck-node-next-nodes tcheck-node))
    (when (and (t-const-node? node)
              (t-const-equal? node ck1 ck2 ck3))
      (return-from lookfor-t-const node))))
```

```
;;; create t-const node structure and return it
;;; join it to the previous type-checking node
```

```
(defun create-t-const (ck1 ck2 ck3 tcheck-node)
  (setf *con-id* (1+ *con-id*))
  (setf tc (make-t-const-node
            :id *con-id*
            :contradiction (rule-contradiction *rule*)
            :check1 ck1
            :check2 ck2
            :check3 ck3))
  (setf (tcheck-node-next-nodes tcheck-node)
        (Rpush tc (tcheck-node-next-nodes tcheck-node)))
  (push tc *tc*) ;;;=>
  tc)
```

```
;;; check if the tests are in this t-const-node or not
```

```
(defun t-const-equal? (node ck1 ck2 ck3 )
  (and
   (equal (t-const-node-check3 node) ck3)
   (equal-ck (t-const-node-check2 node) ck2)
   (equal-ck (t-const-node-check1 node) ck1)))
```

```
(defun equal-ck (ck1 ck2)
  (unless (= (length ck1) (length ck2)) (return-from equal-ck nil))
  (dolist (item ck1)
    (unless (member item ck2 :test #'equal) (return-from equal-ck nil)))
  t)
```

```
;;; Main function to join between antecedents
;;; Create and-node , p memory node
;;; Input :two alpha-beta nodes
;;; Output :one alpha-beta
```

```
(defun anded (r-mem l-mem)
  (if (null r-mem) (return-from anded l-mem))
  (setf inter-test (update2-l-r-env)) ;; update r-env,l-env,dir,compute inter-test
  (setf Out-mem-and-node (lookfor-and-node inter-test l-mem r-mem))
  (if Out-mem-and-node
      Out-mem-and-node
      (alpha(create-and-node inter-test l-mem r-mem))))
```

```

;;; Input : *r-env* *l-env* *dir*
;;; Output : *r-env* after put l-env at the begining of the r-env
;;;         and increment the index of the old *r-env*
;;;         and return the inter-test

```

```

(defun update2-l-r-env (&aux (nrenv nil))
  (dolist (renv *r-env*)
    (push `,(first renv) ,(1+ (second renv)) ,(third renv) ) nrenv))
  (setf nrenv (reverse nrenv))
  (setf inter-test (compute-inter-test *l-env* nrenv))
  (dolist (lenv *l-env*)(push lenv nrenv))
  (setf *r-env* nrenv)
  (setf *l-env* nil)
  inter-test)

```

```

;;; Compute the inter test between the right and the left env
;;; Input : R-env L-env
;;; Output : Test

```

```

(defun compute-inter-test (l-env r-env &aux (test nil))
  (dolist (L l-env)
    (setf lvar (third L))
    (dolist (R r-env)
      (setf rvar (third R))
      (when (eq lvar rvar)
        (push `,(first l) ,(second l) ,(first R) ,(second R)) test))))
  test)

```

```

;;; create and-node and join it to the mem-nodes

```

```

(defun create-and-node (test l r)
  (setf *and-id* (1+ *and-id*))
  (setf and-node (make-and-node
    :id *and-id*
    :contradiction (rule-contradiction *rule*)
    :check1 test
    :l-mem l
    :r-mem r))
  (push and-node *ad*) ;;;=>
  (setf (alpha-beta-r-and) r)
  (Rpush and-node (alpha-beta-r-and) r))
  (setf (alpha-beta-l-and) l)
  (Rpush and-node (alpha-beta-l-and) l))
  and-node)

```

```

;;; look if there exist a previously created and node
;;; if yes return the output mem

```

```

(defun lookfor-and-node (test l r)
  (dolist (left-and (alpha-beta-l-and l))
    (when (and (member left-and (alpha-beta-r-and r) :test #'eq)
              (equal-and? left-and test))
      (return-from lookfor-and-node (and-node-o-mem left-and))))))

(defun equal-and? (anode test)
  (unless (= (length test) (length (and-node-check1 anode)))
    (return-from equal-and? nil))
  (dolist (item (and-node-check1 anode))
    (unless (member item test :test #'equal) (return-from equal-and? nil)))
  t)

;;; After finish build a rete nets for a rule you need to create a p_mem
;;; for this rule and associate it to the last node
;;; if there is a test in whole rule you must process this test
;;; Input :alpha-beta
;;; Output :p-mem

(defun create-p-mem (node &optional (inter-test-fun (rule-test *rule*)))
  (eval `(setf p (make-p-mem
    :rule-name ,(rule-name *rule*)
    :contradiction ,(rule-contradiction *rule*)
    :inter-test ,inter-test-fun
    :consequent ,(rule-rhs *rule*)
    :var-loc ,(compute-var-loc))))
  (push p *p*) ;;;=>
  (SETF (alpha-beta-p-mem node)
    (rpush p (alpha-beta-p-mem node)))
  p)

;;;
;;; find the var loc using the *r-env* and vars in *rule*
;;; Output : ((?x fld index)....)

(defun compute-var-loc (&aux (loc nil))
  (dolist (var (rule-vars *rule*))
    (dolist (binding *r-env*)
      (when (eq (third binding) var)
        (push `(,var ,(first binding) ,(second binding)) loc)
        (return t))))
  loc)

;;;Input :item ,node list
;;;Operation :insert the item in the node list at the begining if its
;;;      :belong to the contradiction node
;;;      :else at the end
;;; This help in reasoning : by give the contradiction more priority

```

;;; Over the normal rule

```
(defun Rpush (item node-list)
  (when (rule-contradiction *rule*)
    (setf node-list (append (list item) node-list)))
  (unless (rule-contradiction *rule*)
    (setf node-list (append node-list (list item)))))
  node-list)
```

3.4 Reason.lsp

;;; Morgue System

;;; Rete Reasoning System Coupled With Atms (tight coupling)

(in-package :Trete)

(defvar inst-under-exe nil)

(defvar *visited-datums*) ;; list to store all the datums that must be
(setf *visited-datums* nil) ;; traversed by the rete to delete the empty
;; label datums

(defvar *visited-mem*) ;; list to store all the mem that must be
(setf *visited-mem* nil) ;; visited to delete the empty label datums

(defun start-reasoning (&aux (repeat t))

(loop

(unless repeat

(format t "~%*****")

(format t "~%Reasoning terminated")

(format t "~%To look for the data use - show-data ")

(format t "~%To look for a specific data use - rfetch (...)")

(format t "~%Join operation =~a" Joperation)

(format t "~%Rule instantiated =~a" instrule)

(format t "~%Execute Normal Rule =~a" ERule)

(format t "~%Execute Contradiction Rule =~a" ECRule)

(format t "~%Label Computation =~a" Lcomp)

(format t "~%*****")

(return))

;;Find all applicable rule and store the instances in the conflict set

(loop

(setf wme (pop (reps-queue *reps*)))

(when (null wme) (return))

(process-wme wme))

;;Contradiction rule is fired as soon as they are instantiated

;;which is mean that those rule is not stored in conflict set

(setf repeat nil)

;;Loop until a new datum is inserted or no instant in conflict set.

(loop

(setf inst (simple-resolve-conflicts))

(when (null inst) (return)) ; Empty conflict-set: finish resolution.

(when (act inst) ;; When the act insert a new datum in queue-process it.

(incf erule)

```

      (setf repeat t)
      (return))))))

;;; Input :contradiction inst
;;; Operation : act contradiction rule
(defun act-contradiction-rule (inst)
  (incf ecrule)
  (act inst)
  (filter-rete *visited-datums*))

;;; You can put here any criteria you want ;
;;; I use a simple strategy (the last in first out) ;
(defun simple-resolve-conflicts ()
  (loop
    (setf inst (pop (reps-conflict-set *reps*)))
    (unless inst (return-from simple-resolve-conflicts nil))
    (unless (instance-del inst) (return)))
  (setf (instance-label inst)
    (The-label-of (first(instance-antecedents inst))))
  inst)

(defun the-label-of (datum)
  (datum-label datum))

;;; Input   : Antecedents-datums
;;; Operation : compute the label.
;;; Output  : label

(defun have-new-env? (Antecedents &aux r)
  (find-new-envs nil (list (atms-empty-env *atms*)) Antecedents))

;;; Input :Datum
;;; Insert any completely instantiated rule in conflict set.

(defun process-wme (wme &aux d)
  ;; Check the existence of the single tuple
  (setf d (is-exist? (first(datum-fact wme)))) ;; Search in all single-tuples
  ;;1- if it exist previously return it
  (when d
    ;; justify the tuple
    (when (datum-justs wme)
      (setf (just-consequence (first(datum-justs wme))) d))
    ;;compute the new label
    (setq new-envs (whose-still-new (datum-label wme) d))
    ;;if there is a new environmets
    (when new-envs (process-tcheck d nil))
    (return-from process-wme nil))
  ;;2- not exist previously

```

```
(unless d
  (unless (datum-assumption? wme)(dolist (env (datum-label wme)) (push wme (env-
datums env))))
  (push wme (atms-datums *atms*))
  (process-tcheck wme t)))
```

;;;input :the single tuple , flag used to insert or not insert the tuple

```
(defun process-tcheck (wme ins)
  (setf tch (find-tcheck (get-pred-name wme)))
  (unless tch ;; No type checking for this predicate
    (when ins (push wme (reps-derived *reps*)))
    (return-from process-tcheck t))
  ;; Find all next-node of the correspond type-check node
  (setf next-nodes (tcheck-node-next-nodes tch)) ;;Next node :alpha or t-const
  (dolist (node next-nodes)
    (when (alpha-beta? node)
      (when ins
        (push wme (alpha-beta-datums node))
        ;Register the alpha in wme
        (push node (datum-alpha-list wme)))
      (process-alpha node (list wme)))
    (when (t-const-node? node)
      (when (and
        ins
        (process-check1 node wme)
        (process-check2 node wme)
        (process-check3 node wme))
        (push wme (alpha-beta-datums (t-const-node-next-node node)))
        ;Register the alpha in wme
        (push (t-const-node-next-node node)(datum-alpha-list wme))
        (process-alpha (t-const-node-next-node node) (list wme)))
      (when (and (not ins)
        (member (t-const-node-next-node node) (datum-alpha-list wme)))
        (process-alpha (t-const-node-next-node node) (list wme)))
      )))
```

;;; Input : Predicate name

;;; Output : type checking node

```
(defun find-tcheck (p-name)
  (dolist (tch (rete-type-checking *rete*))
    (when (eq p-name (tcheck-node-name tch))
      (return-from find-tcheck tch)))
  nil)
```

;;; Input : t-const-node , datum

;;; Output : t , nil

;;; Check the constant values ((fld value)(fld value))

```

;;; Datum here is a single tuple.
(defun process-check1 (t-const datum)
  (dolist (check (t-const-node-check1 t-const))
    (unless (eq (fld-value (first check) 1 datum) (second check))
      (return-from process-check1 nil)))
  t)

```

```

;;; Input : t-const-node , datum
;;; Output : t , nil
;;; Check the fields equality ((fld1 fld2)(fld4 fld5)...)
;;; Datum here is a single tuple.
(defun process-check2 (t-const datum)
  (dolist (check (t-const-node-check2 t-const))
    (unless (eq (fld-value (first check) 1 datum)
                (fld-value (second check) 1 datum))
      (return-from process-check2 nil)))
  t)

```

```

;;; Input : t-const-node , datum
;;; Output : t , nil
;;; Check the test function ((fld1 fld2)(fld4 fld5)...)
;;; Datum here is a single tuple.
(defun process-check3 (t-const datum)
  (unless (t-const-node-check3 t-const)
    (return-from process-check3 t))
  (setf tuple (second(first(datum-fact datum))))
  (setf flds (reverse(extract-flds (list tuple))))
  (apply (t-const-node-check3 t-const)
    (mapcar #(lambda (fld)
              (fld-value fld 1 datum))
            flds)))

```

```

;;; Input : The Field name, Index , datum
;;; Output : The value of the field; if it exist
(defun fld-value (fld index datum)
  (setf tuples (datum-fact datum))
  (dolist (tuple tuples)
    (when (= (first tuple) index)
      (do ((pattern (second tuple) (rest pattern)))
          ((null pattern) nil)
        (when (eq (first pattern) fld)
          (return-from fld-value (second pattern)))))))

```

```

;;;Input : List of New Datums
;;;Process 1- P-mem first (if this belong to cont-rule then fire it)
;;; 2- process contradiction left-and
;;; 3- process contradiction right-and
;;; 4- process normal left-and

```

;;; 5- process normal right-and

```
(defun process-alpha (alpha new-datums)
  (process-p-mem alpha new-datums)
  (setf new-datums (rem-del-datum new-datums))
  (process-l-and alpha new-datums t) ;Contradiction
  (process-r-and alpha new-datums t) ;Contradiction
  (process-l-and alpha new-datums nil) ;Normal
  (process-r-and alpha new-datums nil) ;Normal
  )
```

;;; INPUT : a set of datums

;;; OUTPUT : only datums not set as deleted

```
(defun rem-del-datum (datums &aux (new-datums nil))
  (dolist (d datums)
    (unless (datum-del d)
      (push d new-datums)))
  new-datums)
```

;;; Join the new datums with fact stored in the right mem of the left and

;;; Process the out memory of the and-node

;;; cont : to determine weather you want to process contradiction and or not

```
(defun process-l-and (alpha new-datums cont)
  (setf lands (alpha-beta-l-ands alpha))
  (dolist (land lands)
    (when (equal (and-node-contradiction land) cont)
      ;;; the following condition is used to prevent repeat work
      ;;; when the l and r mem for node AND node is the same and is
      ;;; traversed only in the right direction
      (unless (equal (and-node-r-mem land) (and-node-l-mem land))
        (setf new-joind-tuples
              (join new-datums (alpha-beta-datums (and-node-r-mem land))
                    (and-node-o-mem land) (and-node-check1 land)))
        (when new-joind-tuples
          (process-alpha (and-node-o-mem land) new-joind-tuples))))))
```

;;; Join the new datums with fact stored in the right mem of the right and

;;; Process the out memory of the and-node

```
(defun process-r-and (alpha new-datums cont)
  (setf rands (alpha-beta-r-ands alpha))
  (dolist (rand rands)
    (when (equal (and-node-contradiction rand) cont)
      (setf new-joind-tuples
            (join (alpha-beta-datums (and-node-l-mem rand)) new-datums
                  (and-node-o-mem rand) (and-node-check1 rand)))
      (when new-joind-tuples
        (process-alpha (and-node-o-mem Rand) new-joind-tuples))))))
```



```

;;; Check each new datums with inter-test function (if it exist)
;;; Store the good one in the conflict set
;;; If the p-mem belong to contradiction rule then do not store it
;;; in the conflict set instead fire it

```

```

(defun process-p-mem (alpha new-datums)
  (unless (alpha-beta-p-mem alpha)(return-from process-p-mem t))
  (setf p-mems (alpha-beta-p-mem alpha))
  (dolist (p-mem p-mems)
    (setf new-datums (rem-del-datum new-datums));some may be deleted
    (dolist (datum new-datums)
      (when (process-inter-test p-mem datum)
        (setf inst (is-inst-exist-prev? datum (p-mem-rule-name p-mem)))
        (unless inst
          (update-conflict-set
            (p-mem-rule-name p-mem)
            (p-mem-contradiction p-mem)
            (p-mem-consequent p-mem)
            (generate-binding p-mem datum)
            (list datum)))
          (when (p-mem-contradiction p-mem) ;;contradiction rule
            (decf instrule) ;; really its not stored in the conflict set
            (act-contradiction-rule (simple-resolve-conflicts)))
          ))))

```

```

;;; Input :datum and rule-name
;;; Output :If an instance is exist previously.
(defun is-inst-exist-prev? (datum Rname)
  (dolist (j (datum-consequences datum))
    (when (and (instance? (just-consequence j))
              (equal (instance-rule-name (just-consequence j)) rname))
      (return-from is-inst-exist-prev? (just-consequence j))))
  nil)

```

```

(defun update-conflict-set (rname cont consequence binding antecedents)
  (incf instrule)
  (setf lambda-vars (extract-vars consequence))
  (eval `(push (make-instance
    :rule-name ',rname
    :contradiction ',cont
    :consequence #'(lambda ,lambda-vars ,@consequence)
    :binding ',binding
    :antecedents ',antecedents ;;list of datums
    :vars ',lambda-vars)
    (reps-conflict-set *reps*)))
  (justify-inst 'inst (first(reps-conflict-set *reps*)) antecedents))

```

```

;;; Input : p-mem , datum
;;; Output : t , nil

```

```

;;; Check the inter test function if it exist
(defun process-inter-test (p-mem datum)
  (unless (p-mem-inter-test p-mem) (return-from process-inter-test t))
  (when (p-mem-inter-test p-mem)
    (setf vals (extract-vals datum))
    (apply (p-mem-inter-test p-mem)
            vals)))

```

```

;;; Input : Datum
;;; Output : list of variable values
(defun extract-vals (datum &aux (vals nil) (temp nil))
  (dolist (tp (reverse(datum-fact datum)))
    (setf temp nil)
    (do ((tuple (second tp) (rest tuple)))
        ((null tuple) t)
      (when (field? (first tuple))
        (push (second tuple) temp)))
    (setf vals (append (reverse temp) vals)))
  vals)

```

```

;;; Input : p-mem node , datum
;;; Output : ((?x . value)(..)..)
(defun generate-binding (p-mem datum &aux (binding nil))
  (dolist (loc (p-mem-var-loc p-mem))
    (push (cons (first loc)
                (fld-value (second loc) (third loc) datum))
          binding))
  binding)

```

```

;;; Input :right and left datums , out-mem of the and-node , test
;;; Output : a list of datums (joined tuple test)
(defun join (ldatums rdatums alpha test &aux (result nil) )
  (dolist (ld ldatums)
    (dolist (rd rdatums)
      (setf label (have-new-env? (list ld rd)))
      (setf jt (is-joined-tuple-exist-prev ld rd alpha))
      (when jt ;; there exist a joined tuple previously
        (setq new-envs (whose-still-new label jt)) ;; update jt label
        (when new-envs ;; there is a new environmets
          (push jt result)))
      (unless jt ;; no joint tuple created previosly
        (incf joperation)
        (setf updated-r (inc-index rd))
        (setf joined-tuple (append (datum-fact ld) updated-r))
        (setf d (make-datum :fact joined-tuple))
        (when (process-andcheck d test)
          (when label ;;it's label is not empty
            (process-new-joined-tuple d ld rd alpha label)
            (push d result))))))
  result)

```

```

;;; Input :left datum amd right datum and out-mem
;;; Output :If a joined tuple is exist previously return it
(defun is-joined-tuple-exist-prev (ld rd out-mem)
  (dolist (j (datum-consequences ld))
    (when (and (member out-mem (datum-alpha-list (just-consequence j)))
              (or (and (eq rd (first (just-antecedents j)))
                      (eq ld (second (just-antecedents j))))
                  (and (eq ld (first (just-antecedents j)))
                      (eq rd (second (just-antecedents j)))))))
      (return-from is-joined-tuple-exist-prev (just-consequence j) )))

```

nil)

```

;;; if it is not already exist do
;;; 1- establish the links between right and left datums with joined tuple
;;; 2- build a just without recompute the label
;;; 3- regist the out-mem in the joined tuple
(defun process-new-joined-tuple (d ld rd alpha label)
  (justify-datum 'PJ d (list ld rd))
  (push d (alpha-beta-datums alpha))
  (push alpha (datum-alpha-list d))
  (setf (datum-label d) label)
  (dolist (env label) (push d (env-datums env)))
  (push d (atms-datums *atms*))
  d)

```

```

;;; Input :Datums
;;; Output :fact in the datum after increment index
;;; increment the index in each tuple by one
(defun inc-index (datum &aux (result nil))
  (setf facts (datum-fact datum))
  (dolist (fact facts )
    (push (list (1+ (first fact)) (second fact)) result))
  (reverse result))

```

```

;;; Input : joined tuple , test ((fld1 index fld2 ind)(...))
;;; Output : t or nil
(defun process-andcheck (d tests)
  (dolist (test tests)
    (unless
      (eq
        (fld-value (first test) (second test) d)
        (fld-value (third test) (fourth test) d))
      (return-from process-andcheck nil)))
  t)

```

```

;;; Input :instance
;;; Output : t . if it insert a new data

```

```

(defun act (inst)
  ;; to break the connection between datum and the instance
  (setf justs (datum-consequences (first(instance-antecedents inst))))
  (dolist (j justs)
    (when (eq (just-consequence j) inst)
      (setf (datum-consequences (first(instance-antecedents inst)))
            (delete j (datum-consequences (first(instance-antecedents inst)))))))

  (setf inst-under-exe inst)
  (when inst
    (apply (instance-consequence inst)
           (instantiate-variables (instance-vars inst)
                                 (instance-binding inst)))))

(defun instantiate-variables (consequent binding)
  (sublis binding consequent))

;;;Input : fact
;;;Insert in the queue
;;;We have four cases :

(defun assert! (fact &optional(inst inst-under-exe)
               &aux datum node )
  ;;temp datum
  (setf datum (create-datum (cons '1 (list fact)) nil ))
  (justify-datum 'Inferred datum (instance-antecedents inst))
  (push datum (reps-queue *reps*))
  (setf (datum-label datum) (datum-label (first(instance-antecedents inst))))
  t)

;;; 1-Assert a new nogood
;;; 2-Justify the contradiction datum
(defun assert-nogood! (&optional(inst inst-under-exe)
                     datum)
  (setf datum (atms-contradatum datum *atms*))
  (justify-datum 'Cont datum (instance-antecedents inst))
  (setf *visited-datums* nil)
  (dolist (env (datum-label (first(instance-antecedents inst))))
    (new-nogood env)))

;;;Input :all the datums that become empty label because it's env become nogood
;;;Operation :traverse all the linked tuple and filter the rete
;;;Output :all the empty label tuple is removed from the rete
(defun filter-rete (datums)
  ;; record all mem and datums
  (setf *visited-mem* nil)
  (dolist (d datums)
    (process-datum d))
  ;; visit all mem and delete all marked datums
  (loop

```

```

(setf mem (pop *visited-mem*))
(unless mem (return))
(do ((d (alpha-beta-datums mem) (rest d)))
  ((null d)
   (when (datum-del (first d))
     (setf (atms-datums *Atms*)
           (delete (first d) (atms-datums *atms*) :test #'eq))
     ;;call a procedure to delete just
     (dolist (just (datum-justs (first d)))
       (dolist (d1 (just-antecedents just))
         (setf (datum-consequences d1) ;;to be sure only
               (delete just (datum-consequences d1) :test #'eq))))
     (rplaca d nil)))
   (setf (alpha-beta-datums mem) (remove nil (alpha-beta-datums mem))))
;;;some of the datums in reps-derived become empty label and must be deleted
(do ((d (reps-derived *reps*) (rest d)))
  ((null d)
   (unless (datum-label (first d))
     (setf (datum-del (first d)) t)
     (setf (atms-datums *Atms*)
           (delete (first d) (atms-datums *atms*) :test #'eq))
     ;;call a procedure to delete just
     (dolist (just (datum-justs (first d)))
       (dolist (d1 (just-antecedents just))
         (setf (datum-consequences d1) ;;to be sure only
               (delete just (datum-consequences d1) :test #'eq))))
     (rplaca d nil)))
   (setf (reps-derived *reps*) (remove nil (reps-derived *reps*))))

```

;;;Input :datum

;;;Operation :traverse the alpha and links until end

```

(defun process-datum (d)
  (dolist (alpha (datum-alpha-list d))
    (unless (datum-label d)
      (setf (datum-del d) t)
      (setf (atms-datums *Atms*)
            (delete d (atms-datums *atms*) :test #'eq))
      (unless (member alpha *visited-mem* :test #'equal)
        (push alpha *visited-mem*))
      (do ((justs (datum-consequences d) (rest justs)))
        ((null justs)
         (setf (datum-justs d) (remove nil (datum-justs d))))
        (cond ((instance? (just-consequence(first justs)))
               (setf (instance-del (just-consequence (first justs))) t)
               (rplaca justs nil))
              (t (process-datum (just-consequence(first justs))))))))))

```

;Copyright (c) 1986-1993 Kenneth D. Forbus, Johan de Kleer and Xerox

```
;Corporation. All Rights Reserved.
(defun quotize (pattern)
  (cond ((null pattern) nil)
        ((variable? pattern) pattern)
        ((not (listp pattern)) (list 'QUOTE pattern))
        (t `(cons ,(quotize (car pattern))
                   ,(quotize (cdr pattern))))))
```

3.5 ATMS.lsp

```
:: All definition and structure used in Atms .
:: All Function related to Atms operation.
:: In This Atms the nodes and the justification are integrated with
:: the datums in the rete Memory node ,
:: while the ATMS here is used only for environment manipulation
```

```
(In-package :Trete)
(defvar *atms* nil)
(setf *atms* nil)
```

```
;;; STRUCTURE
;;; 1- ATMS
;;; 2- ENVIRONMENT
;;; 3- DATUMS
```

```
(defstruct (atms (:print-function print-atms))
  (title nil)
  (datums nil)      ;; all datums
  (good-env nil)   ;; good environment
  (nogood-env nil) ;; nogood environment
  (contra-datum nil) ;; dummy contradiction datum
  (empty-env nil)  ;; {{}} ;; hold in all environment
```

```
(defun print-atms (atms stream ignore)
  (format t "atms~a" (atms-title atms)))
```

```
(defun print-full-atms ()
  (format t "~%Title:~a" (atms-title *atms*))
  (format t "~%Datums:~a" (atms-datums *atms*))
  (format t "~%Good-env:~a" (atms-good-env *atms*))
  (format t "~%Nogood-env:~a" (atms-nogood-env *atms*)))
```

```
;;; DATUM STRUCTURE <Fact>
;;; PRINT FUNCTIONS
```

```
(defstruct (datum (:PREDICATE datum?)
                  (:PRINT-FUNCTION print-datum))
  (fact nil)
  (assumption? nil) ;; Weather it is an assumption or not
  (alpha-list nil) ;; where the datum is stored
```

```
(justs nil)    ;;justify this DATUM
(consequences nil) ;;this DATUM belong to these justification
(label nil)    ;; Label of the datum
(del nil)      ;; tag used to detect the datum that must be deleted
```

```
(defun print-datum (datum st ignore)
  (format st "Datum:~a" (datum-fact datum)))
```

```
(defun show-datums (datums)
  (dolist (datum datums)
    (format t "~%-----")
    (format t "~%Datum: ~A ~%Assumption: ~A "
              (datum-fact datum)
              (datum-assumption? datum))
    (format t "~%Alpha-list :~a" (datum-alpha-list datum))
    (format t "~%Justs:~a" (DATUM-justs DATUM))
    (format t "~%Consequences:~a"(DATUM-consequences DATUM))
    (format t "~%label :~a" (datum-label datum))
    (format t "~%Deleted :~a" (datum-del datum))
    (read) ))
```

```
(defstruct (just (:print-function print-just))
  (informant nil)
  (consequence nil)
  (antecedents nil))
```

```
(defun print-full-just (justs)
  (dolist (just justs)
    (format t "~%Informant:~a" (just-informant just))
    (format t "~%Antecedents:~a" (just-antecedents just))
    (format t "~%Consequence:~a" (just-consequence just))))
```

```
(defun print-just (just stream ignore)
  (format t "~%just~a" (just-informant just))
  (format t "~%antecednts:~a" (just-antecedents just))
  (format t "~%Consequence:~a" (just-consequence just)))
```

```
;;;ENV structure and print function
```

```
;;;
```

```
(defstruct (env (:print-function print-env))
  (assumptions nil) ;; Those (single) datums in the environment
  (datums nil)     ;; A List of all datums have this environment
```

```
(defun print-env (env stream ignore)
  (format t "env~a" (env-assumptions env)))
```

```
(defun print-full-env (envs)
  (dolist (env envs)
    (format t "~%Assumptions:~a" (env-assumptions env)))
```

```

(format t "~%datums:~a" (env-datums env))))

;;; CREATE ALL THE STRUCTURE
;;; 1- ATMS
;;; 2- Datums and CONTRADICTION Datums
;;; 3- Environment and EMPTY-ENV .

(defun create-atms (title)
  (setq *atms* (make-atms :TITLE title))
  (create-contradiction-datum)
  (create-empty-env)
  *atms*)

;;; Input  : (1 (fact)) -- this is a tuple? (not a compound tuple)
;;;       : asn ;weather it's assumption or not
;;; Note   : The datum must not exist previously .
;;; Operation : create datum and process it is label
;;; Output  : Datum
(defun create-datum (wme asn)
  (setq datum (make-datum
                :fact (list wme)
                :assumption? asn))
  (when asn
    (setq e (create-env (list datum)))
    (push e (datum-label datum))
    (push datum (env-datums e)))
  datum)

;;; This datum is used only to justify the contradiction rules
(defun create-contradiction-datum ()
  (setf (atms-contradatum *atms*)
        (make-datum :fact '(1 '(FALSE)) )))

(defun create-empty-env ()
  (setf (atms-empty-env *atms*) (create-env nil)))

;;; Input  : Assumption involved in the environment
;;; Output : if the environment is exist previously return it
;;;       : else create a new one
(defun create-env (assumptions &aux e)
  (setq e (make-env :ASSUMPTIONS assumptions))
  (setq exist (lookup-env e (atms-good-env *Atms*)))
  (unless (or exist (nogood? e))
    (setf (atms-good-env *Atms*) (cons e (atms-good-env *Atms*))))
  (if exist exist e))

;;; Look for an environment in a list of environment
(defun lookup-env (e env-list)
  (dolist (env env-list)

```



```
(when (equal-env? env e) (return-from lookup-env env)))
nil)
```

```
;;; Two Environment are equal when the assumption in e1 is subset in e2
;;; and vice versa
```

```
(defun equal-env? (e1 e2)
  (and (subsetp (env-assumptions e2)(env-assumptions e1))
       (subsetp (env-assumptions e1)(env-assumptions e2))))
```

```
;;; HELPING FUNCTION
```

```
;;; Check if there exist an environment in the <env-list> Subset of <env>.
```

```
(defun env-subsumed-by (env env-list)
  (dolist (old-env env-list)
    (when (subsetp (env-assumptions old-env) (env-assumptions env))
      (return old-env))))
```

```
(defun compare-env (e1 e2)
  (cond ((equal-env? e2 e1) :EQ)
        ((subset-env? e1 e2) :S12)
        ((subset-env? e2 e1) :S21)
        (t :NEQ)))
```

```
(defun subset-env? (e1 e2)
  (if (subsetp (env-assumptions e1) (env-assumptions e2)) t nil))
```

```
;;; return the union of two environment
```

```
;;; if the union exist previously return it
```

```
(defun union-env (e1 e2)
  (create-env
   (union (env-assumptions e1) (env-assumptions e2))))
```

```
;;; Envs :new env come to the just from antecedent
```

```
(defun find-new-envs (antecedent envs antecedents &aux new-envs )
  (incf lcomp)
  (drop-nogood
   (make-minimal
    (find-sound-complete-envs antecedent envs antecedents))))
```

```
;;; Return all the env. result from combine the environments of
```

```
;;; the antecedents
```

```
(defun find-sound-complete-envs (antecedent envs antecedents)
  (dolist (datum antecedents)
    (unless (eq datum antecedent) ;; in order to process only the new environment
      (setq result nil)
      (dolist (env envs)
        (dolist (datum-env (datum-label datum))
          (setq result (cons (union-env env datum-env) result))))
      (setq envs result)))
  envs)
```

```

;;; No env. is subset of another
(defun make-minimal (env-list &aux (result nil))
  (setq temp-list (copy-list env-list))
  (dolist (env env-list)
    (setq temp-list (delete env temp-list :count 1))
    (unless (env-subsumed-by env temp-list)
      (setq result (cons env result))))
  (setq temp-list (cons env temp-list)))
result)

;;; delete all nogood environment
(defun drop-nogood (env-list &aux (result nil))
  (setq temp-list (copy-list (atms-nogood-env *atms*)))
  (dolist (env env-list)
    (unless (env-subsumed-by env temp-list)
      (setq result (cons env result))))
  result)

;;; Input :new-environments and the consequent of the justification
;;; operation :update the label of the consequence and return the new env
;;;          not exist in the old
(defun whose-still-new (new-env datum)
  (setq old-env (datum-label datum))
  (dolist (nenv new-env)
    (dolist (oenv old-env)
      (case (compare-env nenv oenv)
        (:S12 (setf (env-datums oenv)
                    (delete datum (env-datums oenv):count 1))
              (setf (datum-label datum)
                    (delete oenv (datum-label datum):count 1)))
        ((:EQ :S21) (setf new-env
                          (delete nenv new-env :count 1))))))
  (dolist (nenv new-env)
    (push datum (env-datums nenv))
    (push nenv (datum-label datum)))
  new-env)

;;; Input : an environment
;;; Output : return true if the environment is nogood
(defun nogood? (env)
  (cond
    ((lookup-env env (atms-nogood-env *atms*)) t)
     ;;if it exist prev
    ((env-subsumed-by env (atms-nogood-env *atms*)) t)
     ;;if there exist e subset of env
    (t nil))
     ;; else return nil

```

```

(defun new-nogood (cenv)
  (setf (atms-nogood-env *atms*)
        (cons cenv (atms-nogood-env *atms*)))
  (remove-env-from-labels cenv t)
  (setf (env-datums cenv) nil)
  (dolist (env (atms-good-env *atms*))
    (when (subset-env? cenv env)
      (remove-env-from-labels env)
      (setf (env-datums env) nil)
      (setf (atms-good-env *atms*)
            (delete env (atms-good-env *atms*) :COUNT 1))))
  (setf (atms-nogood-env *atms*)
        (make-minimal (atms-nogood-env *atms*))))

(defun remove-env-from-labels (env &optional (first nil))
  (dolist (datum (env-datums env))
    (setf (datum-label datum)
          (delete env (datum-label datum) :COUNT 1))
    (unless (datum-label datum)
      (when first
        (push datum *visited-datums*))))))

(defun eq-just (j1 j2)
  (and (equal (just-informant j1) (just-informant j2))
       (equal (just-antecedents j1) (just-antecedents j2))
       (equal (just-consequence j1) (just-consequence j2))))

(defun justify-DATUM (informant consequence antecedents &aux just)
  (setq just (make-just
              :INFORMANT informant
              :CONSEQUENCE consequence
              :ANTECEDENTS antecedents))

  (unless (member just (DATUM-justs consequence) :test #'eq-just)
    (push just (DATUM-justs consequence))
    (dolist (DATUM antecedents) (push just (DATUM-consequences DATUM)))
    (return-from justify-DATUM t))
  nil)

(defun justify-inst (informant consequence antecedents &aux just)
  (setq just (make-just
              :informant informant
              :CONSEQUENCE consequence
              :ANTECEDENTS antecedents))
  (dolist (datum antecedents) (push just (datum-consequences datum))))

```

Appendix 4

Hindi System Program

4.1 Rete.lsp

the same as before

4.2 Def.lsp

```
;;; Hindi System
;;; Declaration Part
;;; 1- SPECIAL VARIABLES .
;;; 2- STRUCTURE USED .
;;; 3- PRINT FUNCTION .

(IN-PACKAGE :HRETE)

;;; RETE Production System
(defvar *REPS* nil)
(defvar *rule*)
(defvar *RETE* nil)

(defstruct (reps (:PREDICATE reps?)
                (:PRINT-FUNCTION print-reps))
  (title nil)
  (atms nil) ;Assumption beased truth maintenance system
  (rete nil) ;RETE network
  (alpha nil) ;Memory-node where all datum are stored there
  (derived nil) ;Datums that have no alpha are stored here
  (conflict-set nil) ;Conflict-set (instantiated rules)
  (queue nil)) ;Queue used conflict Resolution.

(defun print-reps (reps st ignore)
  (format st "REPS:~A" (reps-title reps)))

(defun show-reps ()
  (format t "~%Title:~a" (reps-title *reps*))
  (format t "~%Atms:~a" (reps-atms *reps*))
  (format t "~%RETE:~a" (reps-rete *reps*))
  (format t "~%QUEUE:~a" (reps-queue *reps*))
  (format t "~%To show full atms Echo (print-full-atms)")
  (format t "~%To show All Data Echo (show-Data)")
  (format t "~%To show Conflict-set Echo (show-conflict-set)"))

;;; INSTANCE STRUCTURE
;;; Print Functions
```

```
;;; SHOW CONFLICT SET.
```

```
(defstruct (instance (:PREDICATE instance?)
  (:PRINT-FUNCTION print-instance))
  (rule-name nil)
  (contradiction nil) ;;weather its contradiction or not.
  (consequence nil) ;;un-instantiated consequence
  (vars nil) ;;variable in consequence
  (antecedents nil) ;;list of antecedents datums
  (label nil) ;;label of the antecedents
  (del nil) ;;tag used to detect the datum tha must be deleted
  (binding nil)) ;;assosiation list with variable binding
```

```
(defun print-instance (inst st ignore)
  (format st "~%Instance:-")
  (format st "~%Rule:~a" (instance-rule-name inst))
  (format st "~%Contradiction:~a" (instance-contradiction inst))
  (format st "~%Consequence:~a" (instance-consequence inst))
  (format st "~%Vars:~a" (instance-vars inst))
  (format st "~%Antecedents:~a" (instance-antecedents inst))
  (format st "~%Labels:~a" (instance-label inst))
  (format st "~%Deleted :~a" (instance-del inst))
  (format st "~%Binding:~a" (instance-binding inst)))
```

```
(defun show-conflict-set (&optional (conflict-set (reps-conflict-set *reps*)))
  (dolist (instance conflict-set)
    (print-instance instance t nil)))
```

```
;;; Temporary structure used in rete builder
```

```
(defstruct (Rule) ;temp rule
  (name nil)
  (lhs nil)
  (test nil)
  (rhs nil)
  (flds nil)
  (vars nil)
  (contradiction nil))
```

```
;;;RETE NETWORK
```

```
;;; Print Functions
```

```
(defstruct (RETE (:PREDICATE rete?)
  (:PRINT-FUNCTION print-rete)) ;; ROOT NODE
  (title 'Rete-nets)
  (type-checking nil)) ;; Nodes to Check the predicate name
```

```
(defun print-rete (rete st ignore)
  (format st "RETE:~A" (rete-title rete)))
```

```

(defun show-rete ()
  (format t "~%Title:~a"      (rete-title *rete*))
  (format t "~%Type checking Nodes:~a" (rete-type-checking *rete*)))
;;;
;;;Type Checking Nodes
;;;
(defstruct (Tcheck-node (:PREDICATE tcheck-node?)
                      (:PRINT-FUNCTION print-Tcheck-node))
  (name nil)      ;; Name of the predicate
  (contradiction nil) ;; weather its a part of contradiction rule.
  (next-nodes nil)) ;; alpha-beta or t-const - list of nodes

(defun print-Tcheck-node (Tcheck-node st ignore)
  (format st "TCHECK:~A" (Tcheck-node-name Tcheck-node)))

(defun show-Tcheck-node (&optional (Tchecks (rete-type-checking *rete*)))
  (dolist (Tcheck Tchecks)
    (format t "~%Name:~a"      (Tcheck-node-name Tcheck))
    (format t "~%Contradiction:~a" (Tcheck-node-contradiction Tcheck))
    (format t "~%Next Nodes:~a" (Tcheck-node-next-nodes Tcheck)))
  )
;;;
;;; T-const nodes
;;;
(defstruct (T-const-node (:PREDICATE t-const-node?)
                       (:PRINT-FUNCTION print-T-const-node))
  (id 0)      ;; indexer
  (check1 nil) ;; check the constant value ((field value)...)
  (check2 nil) ;; check the field in one predicate ((field field)...)
  (check3 nil) ;; intra-test-function name
  (contradiction nil) ;; weather its a part of contradiction rule.
  (next-node nil)) ;; alpha-beta -one node

(defun print-T-const-node (t-const-node st ignore)
  (format st "TC:~a" (t-const-node-id t-const-node))
  (if (t-const-node-check1 t-const-node) (format st " Const-test,"))
  (if (t-const-node-check2 t-const-node) (format st " Equal-var,"))
  (if (t-const-node-check3 t-const-node) (format st " I intra-Ext-fun,")))

(defun show-t-const-node (t-const-list)
  (dolist (t-const t-const-list)
    (format t "~%Id:~a"      (t-const-node-id t-const))
    (format t "~%Contradiction:~a" (t-const-node-contradiction t-const))
    (format t "~%Const-test:~a" (t-const-node-check1 t-const))
    (format t "~%Equal-var:~a" (t-const-node-check2 t-const))
    (format t "~%Intra-Ext-fun:~a" (t-const-node-check3 t-const))
    (format t "~%Next-node:~a" (t-const-node-next-node t-const))))
  )
;;;

```

```
;;; Alph and beta memory nodes
```

```
;;;
```

```
(defstruct (alpha-beta (:PREDICATE alpha-beta?)
  (:PRINT-FUNCTION print-alpha-beta))
  (id 0) ; indexer
  (Datums nil) ; List of facts
  (R-Ands nil) ; Right and node -list
  (L-Ands nil) ; Left and node -list
  (p-mem nil) ; P-mem node if it the next -list
  (contradiction nil) ; weather its a part of contradiction rule.
  (prev-node nil)) ; pointer to the previous node -one element
```

```
(defun print-alpha-beta (alpha-beta st ignore)
  (format st "Alpha-beta:~A" (alpha-beta-id alpha-beta)))
```

```
(defun show-alpha-beta (mem-list)
  (dolist (mem mem-list)
    (setf in-d (E-in (alpha-beta-datums mem)))
    (setf out-d (E-out (alpha-beta-datums mem)))
    (format t "~%Id:~a" (alpha-beta-id mem))
    (format t "~%Contradiction:~a" (alpha-beta-contradiction mem))
    (format t "~%R-Ands:~a" (alpha-beta-R-ands mem))
    (format t "~%L-Ands:~a" (alpha-beta-L-ands mem))
    (format t "~%P-mems:~a" (alpha-beta-p-mem mem))
    (format t "~%Prev-node :~a" (alpha-beta-prev-node mem))
    (format t "~%In-Datums:~a" in-d)
    (format t "~%No of In datums :~a" (length in-d))
    (format t "~%Out-Datums:~a" out-d)
    (format t "~%No of out datums :~a" (length out-d))
    (read)))
```

```
;;;Extract the in-datums
```

```
(defun E-in (datums &aux (result nil))
  (dolist (d datums)
    (when (is-in d) (push d result)))
  result)
```

```
;;;Extract the out-datums
```

```
(defun E-out (datums &aux (result nil))
  (dolist (d datums)
    (when (is-out d) (push d result)))
  result)
```

```
;;;
```

```
;;; AND nodes
```

```
;;;
```

```
(defstruct (AND-node (:PREDICATE and-node?)
  (:PRINT-FUNCTION print-and-node))
  (id 0) ; indexer
  (check1 nil) ; check the var between predicates ((fld2 ind fld2 ind)..)
```

```
(contradiction nil) ;; weather its a part of contradiction rule.
(L-mem nil)        ;; Left Memory Nodes -one element
(R-mem nil)        ;; Right Memory Nodes -one element
(O-mem nil)        ;; Output Memory Nodes -one element
```

```
(defun print-and-node (and-node st ignore)
  (format st "And:~A" (and-node-id and-node)))
```

```
(defun show-and-node (and-list)
  (dolist (and-node and-list)
    (format t "~%Id:~a"      (and-node-id and-node))
    (format t "~%Contradiction:~a" (and-node-contradiction and-node))
    (format t "~%Eq-var:~a"      (and-node-check1 and-node))
    (format t "~%Left mem :~a"   (and-node-L-mem and-node))
    (format t "~%Right mem :~a"  (and-node-R-mem and-node))
    (format t "~%Output mem :~a" (and-node-O-mem and-node))))
```

```
...
;;;
;;; P-memory nodes
;;;
;;;
```

```
(defstruct (P-mem (:PREDICATE p-mem?)
  (:PRINT-FUNCTION print-p-mem))
  (rule-name nil) ; the name of the rule corresponding to it
  (inter-test nil) ; the name of the inter test function
  (consequent nil) ; the consequent of the tule (un-instantiated)
  (contradiction nil); weather its contradiction or not.
  (var-loc nil) ; ((?x fld index)...)
  (datums nil) ; justification tuple that can instantiate the rule
```

```
(defun print-p-mem (p-mem st ignore)
  (format st "P-mem :~A" (p-mem-rule-name p-mem)))
```

```
(defun show-p-mem (p-mem-list)
  (dolist (p-mem p-mem-list)
    (format t "~%Rule Name :~a" (p-mem-rule-name p-mem))
    (format t "~%Contradiction:~a" (p-mem-contradiction p-mem))
    (format t "~%Test Name :~a" (p-mem-inter-test p-mem))
    (format t "~%Consequent:~a" (p-mem-consequent p-mem))
    (format t "~%Var-loc :~a" (p-mem-var-loc p-mem))
    (format t "~%datums :~a" (p-mem-datums p-mem))
    (format t "~%# J-datums :~a" (length (p-mem-datums p-mem)))))
```

4.3 build.lsp

```
;;; Hindi System
(in-package :HRETE)
;;; Special variable decleration
```



```

(defvar *dir* 'r) ; special variable to determine the direction of build rete net
(defvar *r-env* nil) ; place to hold the binding from right
(defvar *l-env* nil) ; place to hold the binding from left
(defvar *mem-id* 0) ; Counter for alpha-node
(defvar *con-id* 0) ; Counter for t-const-node
(defvar *and-id* 0) ; Counter for and-node
(setf *con-id* 0)
(setf *and-id* 0)
(setf *mem-id* 0)

```

```

;;; Variable Needed for debug and tree traverse

```

```

(defvar *tc* nil)(setf *tc* nil)
(defvar *tch* nil)(setf *tch* nil)
(defvar *al* nil)(setf *al* nil)
(defvar *ad* nil)(setf *ad* nil)
(defvar *p* nil)(setf *p* nil)

```

```

... *****
;;;
;;; Main function
;;; *rule*-test is a test over all predicate

```

```

(defun create-rete ()
  (setf *r-env* nil)(setf *l-env* nil)(setf *dir* 'r)
  (create-p-mem (process-antecedents (decompose-ants))))

```

```

;;; Decompose each antecedent into a antecedent and a test
;;; Input: ((p1 ..):test #'f (p2 ..))
;;; Output: (((p1 ..) #'f) ((p2 ..) nil))

```

```

(defun decompose-ants (&optional (lhs (rule-lhs *rule*)) &aux nlhs)
  (do ((ants lhs (rest ants))
      (nlhs nil)
      ((endp ants) (reverse nlhs))
      (cond ((and (listp (first ants)) (eq (second ants) :test))
             (push (list (first ants) (third ants)) nlhs)
             (setf ants (rest(rest ants))))
          (t (push (list (first ants) nil) nlhs))))))

```

```

;;; Main function in build a rete for each rule
;;; Input : (((p1 ..) #'f) ((p2 ..) nil)) .
;;; Output: alpha-beta Memory Node .

```

```

(defun process-antecedents (lhs)
  (cond ((endp lhs) nil)
        (t (anded (process-antecedents (rest lhs))
                   (process (first lhs)) ))))

```

```

;;; Process one antecedent and create type-checking ,t-const node,alpha-beta
;;; Input :((p1 ..) #'f)

```

;;; Output: alpha-beta Memory Node .

```
(defun process (ant)
  (update1-l-r-env ant)      ;; update r-env,l-env
  (setf check1 (compute-check1 ant)) ;; constant check
  (setf check2 (compute-check2 ant)) ;; var equality
  (setf check3 (second ant))  ;; function name
  ;;If there is a test create t-const else create type-check-node only
  (if (or check1 check2 check3)
      (alpha (t-const check1 check2 check3 (type-check ant)))
      (alpha (type-check ant))))
```

;;; Update the r-env ,l-env according to the new pattern
 ;;; Input :((p1 ..) #f) ,*dir*
 ;;; Output :((fld1 var index))

```
(defun update1-l-r-env (ant)
  (cond ((eq *dir* 'r)
        (setf *dir* 'l)
        (setf *r-env* (generate-env ant)))
        ((eq *dir* 'l)
         (setf *l-env* (generate-env ant)))))
```

;;; Input :((p1 ..) #f) .
 ;;; Output :((fld1 l var1) ...)

```
(defun generate-env (ant &aux (env nil))
  (do ((ant1 (first ant) (rest ant1))
      (env nil))
      ((endp ant1) (reverse env))
      (when (and (field? (first ant1)) (variable? (second ant1)))
        (push `((first ant1) 1 ,(second ant1) ) env))))
```

;;; Check the constant field in a predicate
 ;;; Input :((p1 ..) #f)
 ;;; Output :((fld val)...)

```
(defun compute-check1 (ant &aux (env nil) )
  (do ((ant1 (first ant) (rest ant1))
      (env nil))
      ((endp ant1) (reverse env))
      (when (and (field? (first ant1)) (not (variable? (second ant1))))
        (push `((first ant1) ,(second ant1) ) env))))
```

;;; Check if two or more variable are equal
 ;;; Input :((p1 ..) #f)
 ;;; Output : (fld1 fld2)

```
(defun compute-check2 (ant &aux (env nil))
```

```

(do ((ant1 (first ant) (rest ant1))
    (env nil)
    ((endp ant1) (reverse env))
    (when (field? (first ant1))
      (setf fld (first ant1))
      (setf var (second ant1))
      (do ((rem (rest(rest ant1)) (rest rem) ))
          ((endp rem))
          (when (and (field? (first rem)) (eq (second rem) var))
            (push `(,fld ,(first rem)) env)
            (return t) )))))

```

```

;;; Create type checking node , If it exist return it
;;; Input :((p1 ..) #f)
;;; Output :Type Checking node

```

```

(defun type-check (ant)
  (setf tch (lookfor-tch ant))
  (if tch tch (create-type-checking ant)))

```

```

;;; Create alpha-mem node and associate it with the node
;;; Input : type-checking node or t-const or and node
;;; Output : Alpha-beta memory-node

```

```

;; (when (rule-contradiction *rule*)
;; (setf (alpha-beta-contradiction (t-const-node-next-node node)) t))

```

```

(defun alpha (node)
  (cond
    ((tcheck-node? node)
     ;; if it exist return it else create a new one
     (lookfor-alpha-beta node))
    ((and (t-const-node? node)
          (alpha-beta? (t-const-node-next-node node)))
     (t-const-node-next-node node))
    ((t-const-node? node)
     (setf *mem-id* (1+ *mem-id*))
     (setf (t-const-node-next-node node)
           (make-alpha-beta :id *mem-id* :prev-node node
                            :contradiction (rule-contradiction *rule*)))
     (push (t-const-node-next-node node) *al*) ;;=>test
     (push (t-const-node-next-node node) (reps-alpha *reps*))
     (t-const-node-next-node node))
    ((and (and-node? node) (and-node-o-mem node))
     (and-node-o-mem node))
    ((and-node? node)
     (setf *mem-id* (1+ *mem-id*))
     (setf (and-node-o-mem node)

```

```

(make-alpha-beta :id *mem-id* :prev-node node
                 :contradiction (rule-contradiction *rule*))
(push (and-node-o-mem node) *al*) ;;=>test
(and-node-o-mem node)))

```

;;; look if one of the node is a mem node ;; associated with type-checking

```

(defun lookfor-alpha-beta (tcheck-node)
  (dolist (node (tcheck-node-next-nodes tcheck-node))
    (when (alpha-beta? node)
      (return-from lookfor-alpha-beta node)))
  (create-alpha-beta tcheck-node))

```

;;; create memory node
 ;;; join it to the previous node

```

(defun create-alpha-beta (tcheck-node)
  (setf *mem-id* (1+ *mem-id*))
  (setf ab (make-alpha-beta :id *mem-id* :prev-node tcheck-node
                           :contradiction (rule-contradiction *rule*)))
  (push ab *al*) ;;test ==>
  (setf (tcheck-node-next-nodes tcheck-node)
        (Rpush ab (tcheck-node-next-nodes tcheck-node)))
  (push ab (reps-alpha *reps*))
  ab)

```

;;; Look if a type-checking node for the ant exist previosly

```

(defun lookfor-tch (ant)
  (setf p-name (first(first ant)))
  (dolist (tch (rete-type-checking *rete*))
    (when (eq p-name (tcheck-node-name tch))
      (return-from lookfor-tch tch) ) )

```

;;; Create a type checking structure and return it
 ;;; and join it to the root

```

(defun create-type-checking (ant)
  (setf tc (make-tcheck-node :name (first(first ant))
                           :contradiction (rule-contradiction *rule*)))
  (push tc *tch*) ;;=>
  (push tc (rete-type-checking *rete*))
  tc)

```

;;; if there exist t-const return it else create a t-const node

```

(defun t-const (check1 check2 check3 tcheck-node)
  (setf t-const-node (lookfor-t-const check1 check2 check3 tcheck-node))
  (if t-const-node
      t-const-node

```

```

;; to make closure around the function
(eval `(create-t-const ',check1 ',check2 ',check3 ',tcheck-node))))

;;; Look if a t-const node for the type-checking node. is exist previously
;;; must have the same condition

(defun lookfor-t-const (ck1 ck2 ck3 tcheck-node)
  (dolist (node (tcheck-node-next-nodes tcheck-node))
    (when (and (t-const-node? node)
              (t-const-equal? node ck1 ck2 ck3))
      (return-from lookfor-t-const node))))

;;; create t-const node structure and return it
;;; join it to the previous type-checking node

(defun create-t-const (ck1 ck2 ck3 tcheck-node)
  (setf *con-id* (1+ *con-id*))
  (setf tc (make-t-const-node
            :id *con-id*
            :contradiction (rule-contradiction *rule*)
            :check1 ck1
            :check2 ck2
            :check3 ck3))
  (setf (tcheck-node-next-nodes tcheck-node)
        (Rpush tc (tcheck-node-next-nodes tcheck-node)))
  (push tc *tc*) ;;;=>
tc)

;;; check if the tests are in this t-const-node or not
(defun t-const-equal? (node ck1 ck2 ck3 )
  (and
   (equal (t-const-node-check3 node) ck3)
   (equal-ck (t-const-node-check2 node) ck2)
   (equal-ck (t-const-node-check1 node) ck1)))

(defun equal-ck (ck1 ck2)
  (unless (= (length ck1) (length ck2)) (return-from equal-ck nil))
  (dolist (item ck1)
    (unless (member item ck2 :test #'equal) (return-from equal-ck nil)))
  t)

;;; Main function to join between antecedents
;;; Create and-node , p memory node
;;; Input :two alpha-beta nodes
;;; Output :one alpha-beta
(defun anded (r-mem l-mem)
  (if (null r-mem) (return-from anded l-mem))
  (setf inter-test (update2-l-r-env)) ;; update r-env,l-env,dir,compute inter-test
  (setf Out-mem-and-node (lookfor-and-node inter-test l-mem r-mem))

```

```
(if Out-mem-and-node
  Out-mem-and-node
  (alpha(create-and-node inter-test l-mem r-mem))))
```

```
;;; Input : *r-env* *l-env* *dir*
;;; Output : *r-env* after put l-env at the begining of the r-env
;;;         and increment the index of the old *r-env*
;;;         and return the inter-test
```

```
(defun update2-l-r-env (&aux (nrenv nil))
  (dolist (renv *r-env*)
    (push `((first renv) ,(1+ (second renv)) ,(third renv) ) nrenv))
  (setf nrenv (reverse nrenv))
  (setf inter-test (compute-inter-test *l-env* nrenv))
  (dolist (lenv *l-env*)(push lenv nrenv))
  (setf *r-env* nrenv)
  (setf *l-env* nil)
  inter-test)
```

```
;;; Compute the inter test between the right and the left env
;;; Input : R-env L-env
;;; Output : Test
```

```
(defun compute-inter-test (l-env r-env &aux (test nil))
  (dolist (L l-env)
    (setf lvar (third L))
    (dolist (R r-env)
      (setf rvar (third R))
      (when (eq lvar rvar)
        (push `((first l) ,(second l) ,(first R) ,(second R)) test))))
  test)
```

```
;;; create and-node and join it to the mem-nodes
```

```
(defun create-and-node (test l r)
  (setf *and-id* (1+ *and-id*))
  (setf and-node (make-and-node
    :id *and-id*
    :contradiction (rule-contradiction *rule*)
    :check1 test
    :l-mem l
    :r-mem r))
  (push and-node *ad*) ;;;=>
  (setf (alpha-beta-r-and r)
    (Rpush and-node (alpha-beta-r-and r)))
  (setf (alpha-beta-l-and l)
    (Rpush and-node (alpha-beta-l-and l)))
  and-node)
```

```
;;; look if there exist a previously created and node
;;; if yes return the output mem
```

```
(defun lookfor-and-node (test l r)
  (dolist (left-and (alpha-beta-l-and l))
    (when (and (member left-and (alpha-beta-r-and r) :test #'eq)
              (equal-and? left-and test))
      (return-from lookfor-and-node (and-node-o-mem left-and))))))
```

```
(defun equal-and? (anode test)
  (unless (= (length test) (length (and-node-check1 anode)))
    (return-from equal-and? nil))
  (dolist (item (and-node-check1 anode))
    (unless (member item test :test #'equal) (return-from equal-and? nil)))
  t)
```

```
;;; After finish build a rete nets for a rule you need to create a p_mem
;;; for this rule and associate it to the last node
;;; if there is a test in whole rule you must process this test
;;; Input :alpha-beta
;;; Output :p-mem
```

```
(defun create-p-mem (node &optional (inter-test-fun (rule-test *rule*)))
  (eval `(setf p (make-p-mem
    :rule-name ,(rule-name *rule*)
    :contradiction ,(rule-contradiction *rule*)
    :inter-test ,inter-test-fun
    :consequent ,(rule-rhs *rule*)
    :var-loc ,(compute-var-loc))))
  (push p *p*) ;;;=>
  (SETF (alpha-beta-p-mem node)
    (rpush p (alpha-beta-p-mem node)))
  p)
```

```
;;;
;;; find the var loc using the *r-env* and vars in *rule*
;;; Output : ((?x fld index)....)
```

```
(defun compute-var-loc (&aux (loc nil))
  (dolist (var (rule-vars *rule*))
    (dolist (binding *r-env*)
      (when (eq (third binding) var)
        (push `(,var ,(first binding) ,(second binding)) loc)
        (return t))))
  loc)
```

```
;;;Input :item ,node list
;;;Operation :insert the item in the node list at the begining if its
;;; :belong to the contradiction node
;;; :else at the end
```

```

;;; This help in reasoning : by give the contradiction more priority
;;; Over the normal rule
(defun Rpush (item node-list)
  (when (rule-contradiction *rule*)
    (setf node-list (append (list item) node-list)))
  (unless (rule-contradiction *rule*)
    (setf node-list (append node-list (list item)))))
  node-list)

```

4.4 Reason.lsp

```

;; Rete Reasoning System Coupled With Atms (tight coupling)
(in-package :Hrete)
(defvar inst-under-exe nil)
(defvar *visited-datums*) ; list to store all the datums that must be
(setf *visited-datums* nil) ; traversed by the rete to push out the empty
                          ; label datums
(defvar current-time 1) ; the time needed in time stamp
(setf current-time 1)
(defun start-reasoning (&aux (repeat t))
(loop
  (unless repeat
    (format t "~%*****")
    (format t "~%Reasoning terminated")
    (format t "~%To look for the data use - show-data ")
    (format t "~%To look for a specific data use - rfetch (...) ")
    (format t "~%Join operation      =~a" Joperation)
    (format t "~%Rule instantiated    =~a" instrule)
    (format t "~%Execute Normal Rule    =~a" ERule)
    (format t "~%Execute Contradiction Rule =~a" ECRule)
    (format t "~%Label Computation      =~a" Lcomp)
    (format t "~%*****")
    (return))
  ;;Find all applicable rule and store the instances in the conflict set
(loop
  (setf wme (pop (reps-queue *reps*)))
  (when (null wme) (return))
  (process-wme wme))
  ;;Contradiction rule is fired as soon as they are instantiated
  ;;which is mean that those rule is not stored in conflict set
  ;;Loop until a new datum is inserted or no instant in conflict set
  (setf repeat nil)
(loop
  (setf inst (simple-resolve-conflicts))
  (when (null inst) (return)) ; Empty conflict-set: finish resolution.
  (when (act inst) ;; When the act insert a new datum in queue process it.
    (incf erule)
    (setf repeat t)
    (return))))))

```



```

;;; Input :contradiction inst
;;; Operation : act contradiction rule
(defun act-contradiction-rule (inst)
  (incf ecrule)
  (act inst)
  (filter-rete *visited-datums*))

;;; You can put here any criteria you want
;;; I use a simple strategy (the last in first out)
;;; the instance must be in (in-part) of the mem node
(defun simple-resolve-conflicts ()
  (loop
    (setf inst (pop (reps-conflict-set *reps*)))
    (unless inst (return-from simple-resolve-conflicts nil))
    (when (is-inst-in inst) (return)))
  (setf (instance-label inst)
    (The-label-of (first(instance-antecedents inst))))
  inst)

;;; Input :Datum
;;; Insert any completely instantiated rule in conflict set.
(defun process-wme (wme)
  ;; Check the existence of the single tuple
  (setf d (is-exist? (first(datum-fact wme))))
  ;; 1- if it exist previously then
  (when d
    ;;1.1- if the datum d in the in-part of the memory
    (when (is-in d)
      ;; justify the tuple
      (when (datum-justs wme)
        (setf (just-consequence (first(datum-justs wme))) d))
      ;; compute the new label for d and find if there is a new env remain
      (setq new-envs (whose-still-new (datum-label wme) d))
      ;;if there is a new environmets
      (when new-envs ;;insert=no,join=no
        (setf (datum-jtime d) 0)
        (process-tcheck d nil))
      (return-from process-wme nil))
    ;;1.2 if the datum d in the out-part of the memory
    (when (is-out d)
      ;; if the datum have new label
      (when (datum-label wme)
        ;; move d in the in-part
        (setf d (push-in d (datum-stime d)(datum-stime d)))
        (setf (datum-label d) (datum-label wme))
        (dolist (env (datum-label d)) (push d (env-datums env)))
        ;; justify the tuple
        (when (datum-justs wme)
          (setf (just-consequence (first(datum-justs wme))) d))

```

```

    (process-tcheck d nil))
  (return-from process-wme nil)))
;;;2- not exist previously (New)
(unless d
  (unless (datum-assumption? wme) (dolist (env (datum-label wme)) (push wme (env-
datums env))))
  (push wme (atms-datums *atms*)))
;;;2.1 if the label is empty store the datum in the out-part with -1 as time-stamp
(unless (datum-label wme )
  (setf wme (push-out wme -1 0))
  (process-tcheck wme t)
  (return-from process-wme nil))
;;;2.2 if the label is not empty store the datum in the in-part with current as time-stamp
(when (datum-label wme)
  (setf wme (push-in wme current-time -1))
  (process-tcheck wme t))))

;;;input :the single tuple , flag used to insert or not insert the tuple
(defun process-tcheck (wme ins)
  (setf tch (find-tcheck (get-pred-name wme)))
  (unless tch ;; No type checking for this predicate
    (when ins (push wme (reps-derived *reps*)))
    (return-from process-TCHECK t))
  ;; Find all next-node of the correspond type-check node
  (setf next-nodes (tcheck-node-next-nodes tch)) ;;Next node :alpha or t-const
  (dolist (node next-nodes)
    (when (alpha-beta? node)
      (when ins
        (push wme (alpha-beta-datums node))
        ;Register the alpha in wme
        (push node (datum-alpha-list wme)))
      (process-alpha node (list wme)))
    (when (t-const-node? node)
      (when (and
        ins
        (process-check1 node wme)
        (process-check2 node wme)
        (process-check3 node wme))
        (push wme (alpha-beta-datums (t-const-node-next-node node)))
        ;Register the alpha in wme
        (push (t-const-node-next-node node)(datum-alpha-list wme))
        (process-alpha (t-const-node-next-node node) (list wme)))
      (when (and (not ins)
        (member (t-const-node-next-node node) (datum-alpha-list wme)))
        (process-alpha (t-const-node-next-node node) (list wme)))
      )))

;;; Input : Predicate name
;;; Output : type checking node

```

```
(defun find-tcheck (p-name)
  (dolist (tch (rete-type-checking *rete*))
    (when (eq p-name (tcheck-node-name tch))
      (return-from find-tcheck tch)))
  nil)
```

```
;;; Input : t-const-node , datum
;;; Output : t , nil
;;; Check the constant values ((fld value)(fld value))
;;; Datum here is a single tuple.
```

```
(defun process-check1 (t-const datum)
  (dolist (check (t-const-node-check1 t-const))
    (unless (eq (fld-value (first check) 1 datum) (second check))
      (return-from process-check1 nil)))
  t)
```

```
;;; Input : t-const-node , datum
;;; Output : t , nil
;;; Check the fields equality ((fld1 fld2)(fld4 fld5)...)
;;; Datum here is a single tuple.
```

```
(defun process-check2 (t-const datum)
  (dolist (check (t-const-node-check2 t-const))
    (unless (eq (fld-value (first check) 1 datum)
                (fld-value (second check) 1 datum))
      (return-from process-check2 nil)))
  t)
```

```
;;; Input : t-const-node , datum
;;; Output : t , nil
;;; Check the test function ((fld1 fld2)(fld4 fld5)...)
;;; Datum here is a single tuple.
```

```
(defun process-check3 (t-const datum)
  (unless (t-const-node-check3 t-const)
    (return-from process-check3 t))
  (setf tuple (second(first(datum-fact datum))))
  (setf flds (reverse(extract-flds (list tuple))))
  (apply (t-const-node-check3 t-const)
    (mapcar #'(lambda (fld)
                (fld-value fld 1 datum))
            flds)))
```

```
;;; Input : The Field name, Index , datum
;;; Output : The value of the field; if it exist
```

```
(defun fld-value (fld index datum)
  (setf tuples (datum-fact datum))
  (dolist (tuple tuples)
    (when (= (first tuple) index)
      (do ((pattern (second tuple) (rest pattern)))
          ((null pattern) nil)
```

```
(when (eq (first pattern) fld)
      (return-from fld-value (second pattern))))))
```

```
;;; Input : List of New Datums ,time used in join operation
;;; Process 1- P-mem first (if this belong to cont-rule then fire it)
;;; 2- process contradiction left-and
;;; 3- process contradiction right-and
;;; 4- process normal left-and
;;; 5- process normal right-and
```

```
(defun process-alpha (alpha new-datums)
  (process-p-mem alpha new-datums)
  (process-l-and alpha new-datums t) ;Contradiction
  (process-r-and alpha new-datums t) ;Contradiction
  (process-l-and alpha new-datums nil) ;Normal
  (process-r-and alpha new-datums nil) ;Normal
  )
```

```
;;; Join the new datums with fact stored in the right mem of the left and
;;; Process the out memory of the and-node
;;; cont : to determine weather you want to process contradiction and or not
(defun process-l-and (alpha new-datums cont)
```

```
  (setf lands (alpha-beta-l-ands alpha))
  (dolist (land lands)
    (when (equal (and-node-contradiction land) cont)
      ;;; the following condition is used to prevent repeat work
      ;;; when the l and r mem for node AND node is the same and is
      ;;; traversed only in the right direction
      (unless (equal (and-node-r-mem land) (and-node-l-mem land))
        (setf new-joind-tuples
              (ljoin new-datums (alpha-beta-datums (and-node-r-mem land)
                                                    (and-node-o-mem land) (and-node-check1 land) ))
              ;;; any new tuple is inserted in the out mem
              ;;; the old tuple is used to update the label
              (when new-joind-tuples
                (process-alpha (and-node-o-mem land) new-joind-tuples))))))
```

```
;;; Join the new datums with fact stored in the right mem of the right and
;;; Process the out memory of the and-node
```

```
(defun process-r-and (alpha new-datums cont)
  (setf rands (alpha-beta-r-ands alpha))
  (dolist (rand rands)
    (when (equal (and-node-contradiction rand) cont)
      (setf new-joind-tuples
            (rjoin (alpha-beta-datums (and-node-l-mem rand)) new-datums
                  (and-node-o-mem rand) (and-node-check1 rand) ))
            ;;; any new tuple is inserted in the out mem
            ;;; the old tuple is used to update the label
            (when new-joind-tuples
```

```
(process-alpha (and-node-o-mem Rand) new-joind-tuples))))))
```

```
;;; Check each new datums with inter-test function (if it exist)
;;; Store the good one in the conflict set
;;; If the p-mem belong to contradiction rule then do not store it
;;; in the conflict set instead fire it
;;; according to hindi system you must deal with in/out instance
```

```
(defun process-p-mem (alpha new-datums)
  (unless (alpha-beta-p-mem alpha)(return-from process-p-mem t))
  (setf p-mems (alpha-beta-p-mem alpha))
  (dolist (p-mem p-mems)
    (dolist (datum new-datums)
      (when (is-in datum)
        (cond ((member datum (p-mem-datums p-mem))
              (setf inst (is-inst-exist-prev? datum (p-mem-rule-name p-mem)))
              (when (and inst (instance-del inst))
                (push-inst-in inst))
              (unless inst
                (update-act p-mem datum)))
              ((process-inter-test p-mem datum)
              (push datum (p-mem-datums p-mem))
              (update-act p-mem datum)))
          ))))
```

```
;;; Input :datum and rule-name
;;; Output :If an instance is exist previously.
(defun is-inst-exist-prev? (datum Rname)
  (dolist (j (datum-consequences datum))
    (when (and (instance? (just-consequence j))
              (equal (instance-rule-name (just-consequence j)) rname))
      (return-from is-inst-exist-prev? (just-consequence j))))
  nil)
```

```
(defun update-act (p-mem datum)
  (update-conflict-set
   (p-mem-rule-name p-mem)
   (p-mem-contradiction p-mem)
   (p-mem-consequent p-mem)
   (generate-binding p-mem datum)
   (list datum))
  (when (p-mem-contradiction p-mem) ;;contradiction rule
    (defc instrule) ;; realy its not stored in the conflict set
    (act-contradiction-rule (simple-resolve-conflicts))))))
```

```
(defun update-conflict-set (rname cont consequence binding antecedents)
  (incf instrule)
  (setf lambda-vars (extract-vars consequence)))
```

```
(eval `(push (make-instance
  :rule-name ',rname
  :contradiction ',cont
  :consequence #'(lambda ,lambda-vars ,@consequence)
  :binding ',binding
  :antecedents ',antecedents ;;list of datums
  :vars ',lambda-vars)
  (reps-conflict-set *reps*)))
(justify-inst 'inst (first(reps-conflict-set *reps*)) antecedents))
```

```
;;; Input : p-mem , datum
;;; Output : t , nil
;;; Check the inter test function if it exist
(defun process-inter-test (p-mem datum)
  (unless (p-mem-inter-test p-mem) (return-from process-inter-test t))
  (when (p-mem-inter-test p-mem)
    (setf vals (extract-vals datum))
    (apply (p-mem-inter-test p-mem)
           vals)))
```

```
;;; Input : Datum
;;; Output : list of variable values
(defun extract-vals (datum &aux (vals nil) (temp nil))
  (dolist (tp (reverse(datum-fact datum)))
    (setf temp nil)
    (do ((tuple (second tp) (rest tuple)))
        ((null tuple) t)
      (when (field? (first tuple))
        (push (second tuple) temp)))
    (setf vals (append (reverse temp) vals)))
  vals)
```

```
;;; Input : p-mem node ,datum
;;; Output : ((?x . value)(..)..)
(defun generate-binding (p-mem datum &aux (binding nil))
  (dolist (loc (p-mem-var-loc p-mem))
    (push (cons (first loc)
               (fld-value (second loc) (third loc) datum))
          binding))
  binding)
```

```
;;; Input : Right and left datums , out-mem of the and-node ,test
;;; Output : A list of datums (joined tuple test)
(defun ljoin (ldatums rdatums alpha test &aux (result nil) )
  (dolist (ld ldatums)
    (when (is-in ld)
      (dolist (rd rdatums)
        (when (is-in rd)
          (cond
```

```

      ((=(datum-jtime ld) 0)
       (setf result (append (update-label ld rd alpha) result)))
      ((=(datum-jtime ld) -1)
       (setf result (append (join ld rd alpha test) result)))
      (t
       (setf result (append (join ld rd alpha test 'l) result))))))
result)

```

;; Input :right and left datums , out-mem of the and-node ,test

;; Output : a list of datums (joined tuple test)

```
(defun rjoin (ldatums rdatums alpha test &aux (result nil) )
```

```
(dolist (ld ldatums)
```

```
(when (is-in ld)
```

```
(dolist (rd rdatums)
```

```
(when (is-in rd)
```

```
(cond
```

```
((=(datum-jtime rd) 0)
```

```
(setf result (append (update-label ld rd alpha) result)))
```

```
((=(datum-jtime rd) -1)
```

```
(setf result (append (join ld rd alpha test) result)))
```

```
(t
```

```
(setf result (append (join ld rd alpha test 'r) result))))))
result)

```

```
(defun join (ld rd alpha test &optional (dir nil) &aux (result nil))
```

```
;1- check if the joined tuple exist previously in the out mem
```

```
(setf jt (is-joined-tuple-exist-prev ld rd alpha))
```

```
;2- if it exist then
```

```
(when jt
```

```
;2.1 : exist and in
```

```
(when (is-in jt)
```

```
;3.1 : update the label of the old one and find the remaining env.
```

```
(setf label (have-new-env? (list ld rd)))
```

```
(setq new-envs (whose-still-new label jt))
```

```
;3.1.1 :exist ,in , new-env
```

```
(when new-envs
```

```
(setf (datum-jtime jt) 0) ;no need to do join operation
```

```
(push jt result))
```

```
(return-from join result))
```

```
;2.2 :exist and out
```

```
(when (is-out jt)
```

```
;3.2.1 :exist ,out , new-env
```

```
(setf label (have-new-env? (list ld rd)))
```

```
(when label
```

```
;push the joined tuple in the in part
```

```
(setf (datum-label jt) label)
```

```
(dolist (env (datum-label jt)) (push jt (env-datums env)))
```

```
(setf jt (push-in jt (datum-stime jt)(datum-stime jt)))
```

```
(push jt result))
```

```

(return-from join result)))

;4- if it does not exist
(unless jt
  (when (or (and (eq dir 'r) (<= (datum-stime rd)(datum-stime ld)))
            (and (eq dir 'l) (<= (datum-stime ld)(datum-stime rd)))
            (not dir))
    (incf joperation)
    (setf updated-r (inc-index rd))
    (setf joined-tuple (append (datum-fact ld) updated-r))
    (setf d (make-datum :fact joined-tuple))
    (when (process-andcheck d test)
      ;4.1 it's label is not empty
      (setf label (have-new-env? (list ld rd)))
      (when label
        (setf d (push-in d current-time -1))
        (process-new-joined-tuple d ld rd alpha label)
        (push d result))
      ;4.2 it's label is empty
      (unless label
        (setf d (push-out d -1 0))
        (process-new-joined-tuple d ld rd alpha label)
        ;;No need to use this datum in label update or join (push d result);
        )))
result)

(defun update-label (ld rd alpha &aux (result nil))
;1- check if the joined tuple exist previously in the out mem
(setf jt (is-joined-tuple-exist-prev ld rd alpha))
;2- if it exist then
(when jt
  ;3- :exist and in
  (when (is-in jt)
    ;update the label of the old one and find the remaining env.
    ;compute the label (inc)
    (setf label (have-new-env? (list ld rd)))
    (setq new-envs (whose-still-new label jt))
    ;3.1.1 :exist ,in , new-env
    (when new-envs
      (setf (datum-jtime jt) 0)
      ;no need to do join operation
      (push jt result))
    (return-from update-label result))
  ;3.2 :exist and out
  (when (is-out jt)
    ;3.2.1 :exist ,out , new-env
    (setf label (have-new-env? (list ld rd)))
    (when label
      ;push the joined tuple in the out part

```



```

      (setf (datum-label jt) label)
      (dolist (env (datum-label jt)) (push jt (env-datums env)))
      (setf jt (push-in jt (datum-stime jt)(datum-stime jt)))
      (push jt result))))
;4- if it does not exist
;No join operation is needed here
result)

;;; Input :left datum amd right datum and out-mem
;;; Output :If a joined tuple is exist previously return it
(defun is-joined-tuple-exist-prev (ld rd out-mem)
  (dolist (j (datum-consequences ld))
    (when (and (member out-mem (datum-alpha-list (just-consequence j)))
               (or (and (eq rd (first (just-antecedents j)))
                       (eq ld (second (just-antecedents j))))
                   (and (eq ld (first (just-antecedents j)))
                       (eq rd (second (just-antecedents j))))))
      (return-from is-joined-tuple-exist-prev (just-consequence j) )))
  nil)
;;; if it is not already exist do
;;; 1- establish the links between right and left datums with joined tuple
;;; 2- build a just without recompute the label
;;; 3- regist the out-mem in the joined tuple
(defun process-new-joined-tuple (d ld rd alpha label)
  (justify-datum 'PJ d (list ld rd))
  (push d (alpha-beta-datums alpha))
  (push alpha (datum-alpha-list d))
  (setf (datum-label d) label)
  (dolist (env label) (push d (env-datums env)))
  (push d (atms-datums *atms*))
  d)

;;; Input :Datums
;;; Output :fact in the datum after increment index
;;; increment the index in each tuple by one
(defun inc-index (datum &aux (result nil))
  (setf facts (datum-fact datum))
  (dolist (fact facts)
    (push (list (1+ (first fact)) (second fact)) result))
  (reverse result))

;;; Input : joined tuple , test ((fld1 index fld2 ind)(...))
;;; Output : t or nil
(defun process-andcheck (d tests)
  (dolist (test tests)
    (unless
      (eq
        (fld-value (first test) (second test) d)

```

```

    (fld-value (third test) (fourth test) d))
  (return-from process-andcheck nil)))
t)

```

```

;;; Input :instance
;;; Output : t . if it insert a new data
(defun act (inst)
  ;; to break the connection between datum and the instance
  (setf justs (datum-consequences (first(instance-antecedents inst))))
  (dolist (j justs)
    (when (eq (just-consequence j) inst)
      (setf (datum-consequences (first(instance-antecedents inst)))
            (delete j (datum-consequences (first(instance-antecedents inst))))))
    (setf inst-under-exe inst)
    (when inst
      (apply (instance-consequence inst)
              (instantiate-variables (instance-vars inst)
                                    (instance-binding inst))))))

```

```

(defun instantiate-variables (consequent binding)
  (sublis binding consequent))

```

```

;;;Input : fact
;;;Insert in the queue
;;;We have four cases :

```

```

(defun assert! (fact &optional(inst inst-under-exe)
               &aux datum node )
  ;;temp datum
  (setf datum (create-datum (cons '1 (list fact)) nil ))
  (justify-datum 'Inferred datum (instance-antecedents inst))
  (setf (datum-label datum) (datum-label (first(instance-antecedents inst))))
  (push datum (reps-queue *reps*))
t)

```

```

;;; 1-Assert a new nogood
;;; 2-Justify the contradiction datum
(defun assert-nogood! (&optional(inst inst-under-exe)
                      (setf datum (atms-contradatum *atms*))
                      (justify-datum 'Cont datum (instance-antecedents inst))
                      (setf *visited-datums* nil)
                      (dolist (env (datum-label (first(instance-antecedents inst))))
                        (new-nogood env)))

```

```

;;;Input :all the datums that become empty label because it's env become nogood
;;;Operation :traverse all the linked tuple and filter the rete
;;;Output :all the empty label tuple is removed to the out mem of the rete
(defun filter-rete (datums)
  (dolist (d datums)(process-datum d))

```

```

;;;some of the datums in reps-derived
;;;become empty label and must be pushed out
(do ((d (reps-derived *reps*) (rest d)))
    ((null d)
     (unless (datum-label (first d))
              (when (is-in (first d));after remove nogood env and before push it to the out.
                    (setf (first d) (push-out (first d) current-time 0))
                    (incf current-time))))))

```

```

;;;Input :datum
;;;Operation :traverse the alpha and links until end
(defun process-datum (d)
  (unless (eq d (atms-contradatum *atms*))
          (unless (datum-label d)
                (when (is-in d) ;after remove nogood env and before push it to the out.
                      (setf d (push-out d current-time 0))
                      (incf current-time)
                      (do ((justs (datum-consequences d) (rest justs)))
                          ((null justs)
                           (cond ((instance? (just-consequence (first justs)))
                                   (setf (just-consequence (first justs))
                                         (push-inst-out (just-consequence(first justs))))
                                   (t (process-datum (just-consequence(first justs))))))))))

```

```

;;; Input :instance
;;; Operation : move the inst to the out part
(defun push-inst-out (inst)
  (setf (instance-del inst) t)
  inst)

```

```

;;; Input :instance
;;; Operation : move the inst to the in part
(defun push-inst-in (inst)
  (setf (instance-del inst) nil)
  inst)

```

```

;;; Input :instance
;;; Output :return t if it in (in -mem)
(defun is-inst-in (inst)
  (not(instance-del inst)))

```

```

;;;Input : the datum
;;;Output : the label of the datum
(defun the-label-of (datum)
  (datum-label datum))

```

```

;;; Input   : Antecedents-datums
;;; Operation : compute the label.
;;; Output  : label

```

```
(defun have-new-env? (Antecedents &aux r)
  (find-new-envs nil (list (atms-empty-env *atms*)) Antecedents))
```

```
;;; Input :Datum , time stamp
;;; Operation : move the datum to the out part and stamp datum with time
(defun push-out (datum st jt)
  (setf (datum-del datum) t)
  (setf (datum-jtime datum) jt)
  (setf (datum-stime datum) st)
  datum)
```

```
;;; Input :Datum , time stamp
;;; Operation : move the datum to the in part and stamp datum with time
(defun push-in (datum st jt)
  (setf (datum-del datum) nil)
  (setf (datum-jtime datum) jt)
  (setf (datum-stime datum) st)
  datum)
```

```
;;; Input :datum
;;; Output :return t if it in (in -mem)
(defun is-in (wme)
  (not(datum-del wme)))
```

```
;;; Input :datum
;;; Output :return t if it in (out -mem)
(defun is-out (wme)
  (datum-del wme))
```

```
;Copyright (c) 1986-1993 Kenneth D. Forbus, Johan de Kleer and Xerox
;Corporation. All Rights Reserved.
```

```
(defun quotize (pattern)
  (cond ((null pattern) nil)
        ((variable? pattern) pattern)
        ((not (listp pattern)) (list 'QUOTE pattern))
        (t `(cons ,(quotize (car pattern))
                   ,(quotize (cdr pattern))))))
```

4.5 ATMS.lsp

```
;;; Hindi System
;; All definition and structure used in Atms .
;; All Function related to Atms operation.
;; In This Atms the nodes and the justification are integrated with
;; the datums in the rete Memory node ,
;; while the ATMS here is used only for environment manipulation
```

```
(In-package :Hrete)
(defvar *atms* nil)
(setf *atms* nil)
```

```
;;; STRUCTURE
;;; 1- ATMS
;;; 2- ENVIRONMENT
;;; 3- DATUMS
;;; 4- Just
```

```
(defstruct (atms (:print-function print-atms))
  (title nil)
  (datums nil)      ;; all datums
  (good-env nil)    ;; good environment
  (nogood-env nil)  ;; nogood environment
  (contra-datum nil) ;; dummy contradiction datum
  (empty-env nil)) ;; {{{}} ;; hold in all environment
```

```
(defun print-atms (atms stream ignore)
  (format t "atms-~a" (atms-title atms)))
```

```
(defun print-full-atms ()
  (format t "~%Title:~a" (atms-title *atms*))
  (format t "~%Datums:~a" (atms-datums *atms*))
  (format t "~%Good-env:~a" (atms-good-env *atms*))
  (format t "~%Nogood-env:~a" (atms-nogood-env *atms*)))
```

```
;;; DATUM STRUCTURE <Fact>
;;; PRINT FUNCTIONS
```

```
(defstruct (datum (:PREDICATE datum?)
                  (:PRINT-FUNCTION print-datum))
  (fact nil)
  (assumption? nil) ;; Weather it is an assumption or not
  (alpha-list nil) ;; where the datum is stored
  (justs nil)      ;; justify this datum
  (consequences nil) ;; this datum belong to these justification
  (label nil)      ;; Label of the datum
  (del nil)        ;; tag used to detect the datum that in out mem
  (Stime nil)      ;; time stamp when its moved to in part
  (jtime nil))    ;; this time is used to determine with which you need to join
                  ;; -1 :join with all datums
                  ;; 0 :no join is needed
                  ;; else :join according to stime
```

```
(defun print-datum (datum st ignore)
  (format st "Datum:~a" (datum-fact datum)))
```

```
(defun show-datums (datums)
```

```
(dolist (datum datums)
  (format t "~%-----")
  (format t "~%Datum: ~A ~%Assumption: ~A "
    (datum-fact datum)
    (datum-assumption? datum))
  (format t "~%Alpha-list :~a" (datum-alpha-list datum))
  (format t "~%Justs:~a" (datum-justs datum))
  (format t "~%Consequences:~a"(datum-consequences datum))
  (format t "~%label :~a" (datum-label datum))
  (format t "~%Out :~a" (datum-del datum))
  (format t "~%time stamp :~a" (datum-Stime datum))
  (format t "~%join timep :~a" (datum-jtime datum))
  (read)))
```

```
;;;Justification structure
```

```
(defstruct (just (:print-function print-just))
  (informant nil)
  (consequence nil)
  (antecedents nil))
```

```
(defun print-just (just stream ignore)
  (format t "~%just~a" (just-informant just))
  (format t "~%antecednts:~a" (just-antecedents just))
  (format t "~%Consequence:~a" (just-consequence just)))
```

```
(defun print-full-just (justs)
  (dolist (just justs)
    (format t "~%Informant:~a" (just-informant just))
    (format t "~%Antecedents:~a" (just-antecedents just))
    (format t "~%Consequence:~a" (just-consequence just))))
```

```
;;;ENV structure and print function
```

```
;;;
;;;
(defstruct (env (:print-function print-env))
  (assumptions nil) ;;; Those (single) datums in the environment
  (datums nil) ;;; A List of all datums have this environment)
```

```
(defun print-env (env stream ignore)
  (format t "env~a" (env-assumptions env)))
```

```
(defun print-full-env (envs)
  (dolist (env envs)
    (format t "~%Assumptions:~a" (env-assumptions env))
    (format t "~%datums:~a" (env-datums env))))
```

```
;;; CREATE ALL THE STRUCTURE
```

```
;;; 1- ATMS
;;; 2- Datums and CONTRADICTION Datums
;;; 3- Environment and EMPTY-ENV
```

```

(defun create-atms (title)
  (setq *atms* (make-atms :TITLE title))
  (create-contradiction-datum)
  (create-empty-env)
  *atms*)

;;; Input   : (1 (fact)) -- this is a tuple? (not a compound tuple)
;;;        : asn ;weather it's assumption or not
;;; Note    : The datum must not exist previously
;;; Operation : create datum and process it is label
;;; Output  : Datum
(defun create-datum (wme asn)
  (setq datum (make-datum
                :fact (list wme)
                :assumption? asn))
  (when asn
    (setq e (create-env (list datum)))
    (push e (datum-label datum))
    (push datum (env-datums e)))
  datum)

;;; This datum is used only to justify the contradiction rules
(defun create-contradiction-datum ()
  (setf (atms-contradatum *atms*)
        (make-datum :fact '(1 '(FALSE)) )))

(defun create-empty-env ()
  (setf (atms-empty-env *atms*) (create-env nil)))

;;; Input : Assumption envolved in the environment
;;; Output : if the environment is exist previously return it
;;;         else create a new one
(defun create-env (assumptions &aux e)
  (setq e (make-env :ASSUMPTIONS assumptions))
  (setq exist (lookup-env e (atms-good-env *Atms*)))
  (unless (or exist (nogood? e))
    (setf (atms-good-env *Atms*) (cons e (atms-good-env *Atms*))))
  (if exist exist e))

;;; Look for an environment in a list of environment
(defun lookup-env (e env-list)
  (dolist (env env-list)
    (when (equal-env? env e) (return-from lookup-env env)))
  nil)

;;; Two Environment are equal when the assumption in e1 is subset in e2
;;; and vice versa
(defun equal-env? (e1 e2)

```

```
(and (subsetp (env-assumptions e2)(env-assumptions e1))
     (subsetp (env-assumptions e1)(env-assumptions e2))))
```

```
;;; HELPING FUNCTION
```

```
;;; Check if there exist an environment in the <env-list> Subset of <env>.
```

```
(defun env-subsumed-by (env env-list)
  (dolist (old-env env-list)
    (when (subsetp (env-assumptions old-env) (env-assumptions env))
      (return old-env))))
```

```
(defun compare-env (e1 e2)
  (cond ((equal-env? e2 e1) :EQ)
        ((subset-env? e1 e2) :S12)
        ((subset-env? e2 e1) :S21)
        (t :NEQ)))
```

```
(defun subset-env? (e1 e2)
  (if (subsetp (env-assumptions e1) (env-assumptions e2)) t nil))
```

```
;;; return the union of two environment
;;; if the union exist previously return it
```

```
(defun union-env (e1 e2)
  (create-env (union (env-assumptions e1) (env-assumptions e2))))
```

```
;;; Envs :new env come to the just from antecedent
```

```
(defun find-new-envs (antecedent envs antecedents &aux new-envs)
  (incf lcomp)
  (drop-nogood
   (make-minimal
    (find-sound-complete-envs antecedent envs antecedents))))
```

```
;;; Return all the env. result from combine the environments of
;;; the antecedents
```

```
(defun find-sound-complete-envs (antecedent envs antecedents)
  (dolist (datum antecedents)
    (unless (eq datum antecedent) ;; in order to process only the new environment
      (setq result nil)
      (dolist (env envs)
        (dolist (datum-env (datum-label datum))
          (setq result (cons (union-env env datum-env) result))))
      (setq envs result)))
  envs)
```

```
;;; No env. is subset of another
```

```
(defun make-minimal (env-list &aux (result nil))
  (setq temp-list (copy-list env-list))
```



```
(dolist (env env-list)
  (setq temp-list (delete env temp-list :count 1))
  (unless (env-subsumed-by env temp-list)
    (setq result (cons env result)))
  (setq temp-list (cons env temp-list)))
result)
```

;;; delete all nogood environment

```
(defun drop-nogood (env-list &aux (result nil))
  (setq temp-list (copy-list (atms-nogood-env *atms*)))
  (dolist (env env-list)
    (unless (env-subsumed-by env temp-list)
      (setq result (cons env result))))
  result)
```

;;; Input :new-environments and the consequent of the justification
 ;;; operation :update the label of the consequence and return the new env
 ;;; not exist in the old

```
(defun whose-still-new (new-env datum)
  (setq old-env (datum-label datum))
  (dolist (nenv new-env)
    (dolist (oenv old-env)
      (case (compare-env nenv oenv)
        (:S12 (setf (env-datums oenv)
                    (delete datum (env-datums oenv):count 1))
              (setf (datum-label datum)
                    (delete oenv (datum-label datum):count 1)))
        ((:EQ :S21) (setf new-env
                          (delete nenv new-env :count 1))))))
  (dolist (nenv new-env)
    (push datum (env-datums nenv))
    (push nenv (datum-label datum)))
  new-env)
```

;;; Input : an environment
 ;;; Output : return true if the environment is nogood

```
(defun nogood? (env)
  (cond ;;if it exist prev
        ((lookup-env env (atms-nogood-env *atms*)) t)
         ;;if there exist e subset of env
        ((env-subsumed-by env (atms-nogood-env *atms*)) t)
         ;; else return nil
        (t nil)))
```

;;; Input : Cenv is a new nogood datum

;; operation :remove this env from the all labels

```
(defun new-nogood (cenv)
  (setf (atms-nogood-env *atms*)
        (cons cenv (atms-nogood-env *atms*)))
  (remove-env-from-labels cenv t)
  (setf (env-datums cenv) nil)
  (dolist (env (atms-good-env *atms*))
    (when (subset-env? cenv env)
      (remove-env-from-labels env)
      (setf (env-datums env) nil)
      (setf (atms-good-env *atms*)
            (delete env (atms-good-env *atms*) :COUNT 1))))
  (setf (atms-nogood-env *atms*)
        (make-minimal (atms-nogood-env *atms*))))
```

```
(defun remove-env-from-labels (env &optional (first nil))
  (dolist (datum (env-datums env))
    (setf (datum-label datum)
          (delete env (datum-label datum) :COUNT 1))
    (unless (datum-label datum)
      (when first
        (push datum *visited-datums*))))))
```

;; Input :two justification
 ;; Output :T if they are equal

```
(defun eq-just (j1 j2)
  (and (equal (just-informant j1) (just-informant j2))
        (equal (just-antecedents j1) (just-antecedents j2))
        (equal (just-consequence j1) (just-consequence j2))))

(defun justify-datum (informant consequence antecedents &aux just)
  (setq just (make-just
              :INFORMANT informant
              :CONSEQUENCE consequence
              :ANTECEDENTS antecedents))
  (unless (member just (datum-justs consequence) :test #'eq-just)
    (push just (datum-justs consequence))
    (dolist (datum antecedents) (push just (datum-consequences datum))))
  (return-from justify-DATUM t))
nil)

(defun justify-inst (informant consequence antecedents &aux just)
  (setq just (make-just
              :informant informant
              :CONSEQUENCE consequence
              :ANTECEDENTS antecedents))
  (dolist (datum antecedents) (push just (datum-consequences datum))))
```

Appendix 5

Rules

5.1 Queen problem

```
(defun queens-not-ok (r1 c1 r2 c2)
  (and (or (= r1 r2)
           (= c1 c2)
           (= (abs (- r1 r2))
              (abs (- c1 c2))))
        (or (not (= r1 r2))
            (not (= c1 c2)))))
```

;;; We can use initial-working-memory to write the clauses

;;; But since it's a large set we prefer to write a function to generate it

```
(defun generate-initial-working-memory (n &aux (result nil))
  (dotimes (r n)
    (dotimes (c n)
      (push `(q !x ,(1+ r) !y ,(1+ c)) result)))
  (set-queue (reverse result)))
```

(Contradiction-rule C1

```
((q !x ?r1 !y ?c1 ) (q !x ?r2 !y ?c2 )) :test #'queens-not-ok
=> (rassert-nogood!))
```

(Rule R1

```
((q !x 1 !y ?c1 ) (q !x 2 !y ?c2 ) (q !x 3 !y ?c3 ) (q !x 4 !y ?c4 ))
=> (rassert! (loc !c1 ?c1 !c2 ?c2 !c3 ?c3 !c4 ?c4 )))
```

(generate-initial-working-memory 4)

5.2 Constraint satisfaction problem

```
(defun c-a-not-ok (vr1 c vr2 a) (not(> (+ a c) 4)))
(defun b-c-not-ok (vr1 b vr2 c) (not(< (+ b c) 5)))
(defun b-c-a-not-ok (vr1 b vr2 c vr3 a) (not(< (+ a c b) 9)))
```

(Contradiction-rule C1

```
((ass !var c !val ?v2 ) (ass !var a !val ?v3 )) :test #'c-a-not-ok
=> (rassert-nogood!))
```

(Contradiction-rule C2

```
((ass !var b !val ?v1 ) (ass !var c !val ?v2 )) :test #'b-c-not-ok
=> (rassert-nogood!))
```

(Contradiction-rule C3

```
((ass !var b !val ?v1 ) (ass !var c !val ?v2 )
 (ass !var a !val ?v3 )) :test #'b-c-a-not-ok
=> (rassert-nogood!))
```

(Rule R1

```
((ass !var b !val ?v1 ) (ass !var c !val ?v2 ) (ass !var a !val ?v3 ))
=> (rassert! (sol !a ?v3 !b ?v1 !c ?v2 )))
```

(initial-working-memory

```
(ass !var a !val 3 ) (ass !var a !val 5 ) (ass !var b !val 2 )
(ass !var b !val 3 ) (ass !var c !val 1 ) (ass !var c !val 3 )
(ass !var c !val 5 ) )
```

5.3 Student Registration system

```
(defun diff-courses (&rest Courses &aux (cou nil))
```

```
(loop
  (push (first courses) Cou)
  (pop courses)(pop courses)(pop courses)
  (when (null courses) (return t)))
(do ((C cou (Rest C)))
  ((null c) t)
  (when (member (first c) (rest c))
    (return-from diff-courses nil)))
t)
```

```
(defun g3-2 (&rest C &aux (lst nil)(cou nil))
```

```
(push (nth 2 c) lst)(push (nth 5 c) lst)(push (nth 8 c) lst)
(push (nth 11 c) lst)(push (nth 14 c) lst)(push (nth 17 c) lst)
(push (nth 20 c) lst)
;;; different courses
(loop
  (push (first C) Cou)
  (pop C)(pop C)(pop C)
  (when (null C) (return t)))
(do ((C1 C (Rest C1)))
  ((null c1) t)
  (when (member (first c1) (rest c1))
    (return-from g3-2 nil)))
(not (and (member '1 lst)
  (member '2 lst)
  (member '3 lst))))
```

(Contradiction-Rule G2-1

```
((TryReg !N ?x1 !G-1 2 !G-2 ?z)
(TryReg !N ?x2 !G-1 2 !G-2 ?z)) :test #'diff-courses
=> (rassert-nogood!))
```

(Contradiction-Rule G2-2

```
((TryReg !N ?x1 !G-1 2 !G-2 ?z1) (TryReg !N ?x2 !G-1 2 !G-2 ?z2)
(TryReg !N ?x3 !G-1 2 !G-2 ?z3) (TryReg !N ?x4 !G-1 2 !G-2 ?z4)
(TryReg !N ?x5 !G-1 2 !G-2 ?z5)) :test #'diff-courses
=> (rassert-nogood!))
```

```

(Contradiction-Rule G5-1
  ((TryReg !N ?x1 !G-1 5 !G-2 ?z) (TryReg !N ?x2 !G-1 5 !G-2 ?z)
   (TryReg !N ?x3 !G-1 5 !G-2 ?z)) :test #diff-courses
  ==> (rassert-nogood!))
(Contradiction-Rule G5-2
  ((TryReg !N ?x1 !G-1 5 !G-2 ?z1) (TryReg !N ?x2 !G-1 5 !G-2 ?z2)
   (TryReg !N ?x3 !G-1 5 !G-2 ?z3) (TryReg !N ?x4 !G-1 5 !G-2 ?z4)
   (TryReg !N ?x5 !G-1 5 !G-2 ?z5)
   (TryReg !N ?x6 !G-1 5 !G-2 ?z6)) :test #diff-courses
  ==> (rassert-nogood!))
(Contradiction-Rule G3-1
  ((TryReg !N ?x1 !G-1 3 !G-2 ?z) (TryReg !N ?x2 !G-1 3 !G-2 ?z)
   (TryReg !N ?x3 !G-1 3 !G-2 ?z) (TryReg !N ?x4 !G-1 3 !G-2 ?z)
   (TryReg !N ?x5 !G-1 3 !G-2 ?z)) :test #diff-courses
  ==> (rassert-nogood!))
(Contradiction-Rule G3-2
  ((TryReg !N ?x1 !G-1 3 !G-2 ?z1) (TryReg !N ?x2 !G-1 3 !G-2 ?z2)
   (TryReg !N ?x3 !G-1 3 !G-2 ?z3) (TryReg !N ?x4 !G-1 3 !G-2 ?z4)
   (TryReg !N ?x5 !G-1 3 !G-2 ?z5) (TryReg !N ?x6 !G-1 3 !G-2 ?z6)
   (TryReg !N ?x7 !G-1 3 !G-2 ?z7)) :test #G3-2
  ==> (rassert-nogood!))
(Contradiction-Rule G3-3
  ((TryReg !N ?x1 !G-1 3 !G-2 ?z1) (TryReg !N ?x2 !G-1 3 !G-2 ?z2)
   (TryReg !N ?x3 !G-1 3 !G-2 ?z3) (TryReg !N ?x4 !G-1 3 !G-2 ?z4)
   (TryReg !N ?x5 !G-1 3 !G-2 ?z5) (TryReg !N ?x6 !G-1 3 !G-2 ?z6)
   (TryReg !N ?x7 !G-1 3 !G-2 ?z7) (TryReg !N ?x8 !G-1 3 !G-2 ?z8)) :test #diff- courses
  ==> (rassert-nogood!))

(Rule R1
  ((Regist !Cou ?x) (Course !No ?x !G1 ?y !G2 ?z))
  ==> (rassert! (TryReg !N ?x !G-1 ?y !G-2 ?z)))

(Rule R2
  ((TryReg !N ?x1 !G-1 ?y1 !G-2 ?z1) (TryReg !N ?x2 !G-1 ?y2 !G-2 ?z2)
   (TryReg !N ?x3 !G-1 ?y3 !G-2 ?z3) (TryReg !N ?x4 !G-1 ?y4 !G-2 ?z4)
   (TryReg !N ?x5 !G-1 ?y5 !G-2 ?z5) (TryReg !N ?x6 !G-1 ?y6 !G-2 ?z6)
   (TryReg !N ?x7 !G-1 ?y7 !G-2 ?z7) (TryReg !N ?x8 !G-1 ?y8 !G-2 ?z8))
  ==> (rassert! (YouCanReg !Na ?x1)))

(initial-working-memory
  (course !No 1100 !G1 1 !G2 1)(course !No 2100 !G1 1 !G2 1)
  (course !No 3100 !G1 1 !G2 1)(course !No 4100 !G1 2 !G2 1)
  (course !No 5100 !G1 2 !G2 1)(course !No 8100 !G1 2 !G2 2)
  (course !No 9100 !G1 2 !G2 2)(course !No 10100 !G1 2 !G2 2)
  (course !No 11100 !G1 2 !G2 2)(course !No 14100 !G1 2 !G2 3)
  (course !No 16100 !G1 2 !G2 3)(course !No 17100 !G1 2 !G2 4)
  (course !No 18100 !G1 2 !G2 4)(course !No 19100 !G1 2 !G2 4)
  (course !No 1101 !G1 3 !G2 1) (course !No 1102 !G1 3 !G2 1)
  (course !No 1103 !G1 3 !G2 1) (course !No 2101 !G1 3 !G2 1)

```

(course !No 2102 !G1 3 !G2 1) (course !No 3101 !G1 3 !G2 1)
 (course !No 3102 !G1 3 !G2 1) (course !No 2103 !G1 3 !G2 1)
 (course !No 1104 !G1 3 !G2 1) (course !No 4101 !G1 3 !G2 2)
 (course !No 4102 !G1 3 !G2 2) (course !No 5101 !G1 3 !G2 2)
 (course !No 5102 !G1 3 !G2 2) (course !No 3103 !G1 3 !G2 2)
 (course !No 6101 !G1 3 !G2 2) (course !No 6102 !G1 3 !G2 2)
 (course !No 7101 !G1 3 !G2 3) (course !No 7102 !G1 3 !G2 3)
 (course !No 8101 !G1 3 !G2 3) (course !No 8102 !G1 3 !G2 3)
 (course !No 9101 !G1 3 !G2 3) (course !No 9102 !G1 3 !G2 3)
 (course !No 10101 !G1 3 !G2 3) (course !No 10102 !G1 3 !G2 3)
 (course !No 211 !G1 4 !G2 1) (course !No 212 !G1 4 !G2 1)
 (course !No 213 !G1 4 !G2 1) (course !No 214 !G1 4 !G2 1)
 (course !No 215 !G1 4 !G2 1) (course !No 221 !G1 4 !G2 1)
 (course !No 252 !G1 4 !G2 1) (course !No 311 !G1 4 !G2 1)
 (course !No 312 !G1 4 !G2 1) (course !No 313 !G1 4 !G2 1)
 (course !No 322 !G1 4 !G2 1) (course !No 331 !G1 4 !G2 1)
 (course !No 334 !G1 4 !G2 1) (course !No 341 !G1 4 !G2)
 (course !No 422 !G1 4 !G2 1) (course !No 323 !G1 4 !G2 1)
 (course !No 424 !G1 4 !G2 1) (course !No 432 !G1 4 !G2 1)
 (course !No 433 !G1 4 !G2 1) (course !No 441 !G1 4 !G2 1)
 (course !No 442 !G1 4 !G2 1) (course !No 314 !G1 5 !G2 1)
 (course !No 315 !G1 5 !G2 1) (course !No 411 !G1 5 !G2 1)
 (course !No 412 !G1 5 !G2 1) (course !No 321 !G1 5 !G2 2)
 (course !No 323 !G1 5 !G2 2) (course !No 426 !G1 5 !G2 2)
 (course !No 434 !G1 5 !G2 3) (course !No 333 !G1 5 !G2 3)
 (course !No 431 !G1 5 !G2 3) (course !No 305104 !G1 6 !G2 1)
 (course !No 105221 !G1 6 !G2 1) (course !No 105451 !G1 6 !G2 1)
 (course !No 133222 !G1 6 !G2 1))

ملخص

دراسة كفاءة قواعد الإنتاج المعتمدة على أنظمة صيانة الحقائق

إعداد

خالد وليد محمود

المشرف

الدكتور رياض جبيري

المشرف المشارك

الدكتور خليل الهندي

تعتبر أنظمة صيانة الحقائق من الأنظمة التي لعبت دورا مهما في عالم الذكاء الاصطناعي، حيث قدمت الحلول للعديد من المشاكل، ومن أبرز هذه المشاكل: حذف حقيقة معينة، إعطاء تفسيرات للاستنتاجات، العمل في بيئة غير مستقرة. ويعتبر نظام صيانة الحقائق المعتمد على الفرضيات ATMS أحد أهم أنواع أنظمة صيانة الحقائق التي تدعم الحلول المتعددة، والتي يمكن دمجها مع أنظمة صيانة الحقائق للوصول إلى الحل الأمثل عن طريق فحص كل الحلول الممكنة.

وهناك عدة طرق لدمج أنظمة صيانة الحقائق مع قواعد الإنتاج، وهذه الرسالة تهدف إلى إجراء دراسة عملية لثلاثة طرق خاصة بالدمج. وهذه هي: الطريقة المنفصلة؛ وفيها يكون كل من نظام صيانة الحقائق وقواعد الإنتاج منفصلين تماما، وطريقتين معتمدين على أساس الطريقة المدججة؛ وفيها تدمج وظائف نظام صيانة الحقائق مع وظائف قواعد الإنتاج. وهذه طريقتين هما نظام Morgue ونظام

478599

.Hindi

وفي إطار هذه الرسالة قمت بتوضيح الحسنيات والسيئات لجميع الطرق، واستخراج مقارنات عملية بين الطرق الثلاث لتوضيح الفرق بينهما. وقد تبين نتيجة الدراسة، باستخدام أربع تطبيقات مختلفة، أن الطرق المدججة بنوعها أكثر كفاءة من الطريقة المنفصلة. وأن نظام Hindi على الأقل هو بكفاءة نظام Morgue، وفي كثير من الأحيان يتفوق عليه.