

Coupling Expert Systems with Internet using Monitor Coupling Paradigm

By
Hussam Omar Saadeh

Supervisor
Dr. Khalil el Hindi

Co-Supervisor
Dr. Moneeb Qutaishat

تعتمد كلية الدراسات العليا
هذه النسخة من الرسالة
التوقيع: التاريخ: ٢٩/٧/٩٩

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

Faculty of Graduate Studies
University of Jordan

July 1999


١٢/٧/٩٩

This thesis was successfully defended and approved on 11-July-1999.

Examination Committee

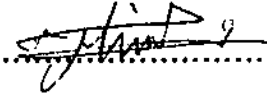
Signature

Dr. Khalil el Hindi / Chairman
Asst. Prof. of Artificial Intelligence



.....

Dr. Moneeb Qutaishat / Member
Asst. Prof. of Databases



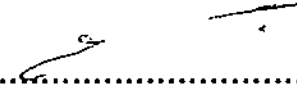
.....

Dr. Riad Jabri / Member
Assoc. Prof. of Compilers



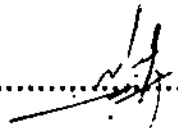
.....

Dr. Sami Sarhan / Member
Asst. Prof. of Computer Architecture



.....

Dr. Abdel-Raouf Al-Hallaq / Member
Asst. Prof. of Computer Network



.....

Acknowledgement

I would like to present the deepest acknowledgement to my supervisor Dr. Khalil el-Hindi on the efforts, guidance and support he gave me throughout this thesis. I do highly appreciate the trust he put on me by giving me the opportunity to work on extending the Monitor Coupling Paradigm over the Internet, thank you sir.

I would also like to express intensive thanks to my co-supervisor Dr. Monneb Qutaishat, besides his excellent hints on databases, he was very helpful in providing and facilitating the required resources for this research.

Of course, I am in debt to my family. They have provided a full support, a very healthy environment and constructive advises.

Many colleagues helped me in gathering information and gave very useful suggestions I would like to thank all folks, especial thanks to Tayseer Hasan, Khalid Waleed and Issa Qonbor.

Without the cooperation and understanding of the Arab Turnkey Systems – ATS, the company with which I work, it would be very difficult to continue my graduate study.

List of Contents

Examination Committee	ii
Acknowledgement	iii
List of Contents	iv
List of Figures	vii
List of Tables	viii
Abstract	ix
Chapter 1: Introduction	1
1.1 Coupling databases and expert systems	2
1.2 Extending the MCP for the Internet environment	4
1.3 Compromised approach for tightly coupling ATMS with Rete Network	6
1.4 Thesis Structure	6
Chapter 2: ATMS-based Reasoning Systems and Databases	7
2.1 Introduction	7
2.2 ATMS-based reasoning system	7
2.2.1 Production Rule System	8
2.2.2 The Rete Algorithm	9
2.2.3 Basic ATMS	11
2.2.4 Coupling ATMS with a Rete-based production system	15
2.2.4.1 ATMS loosely coupled production system	15
2.2.4.2 ATMS tightly coupled production system – the Morgue System	16
2.2.4.3 ATMS tightly coupled production system – the Hindi System	17
2.3 Expert systems and Databases coupling approaches	19
2.3.1 Loose coupling experts and databases	20
2.3.2 Tight coupling databases and databases	21
2.3.3 Monitor Coupling Paradigm	22
2.4 Summary	25

Chapter 3: The Network Computing and the Internet	26
3.1 Introduction	26
3.2 Communication Computing Models	27
3.2.1 Client/Server Model	28
3.2.2 Publish/Subscribe Model	29
3.3 Distributed Object (Component) Technology	31
3.3.1 OMG CORBA	32
3.3.2 Sun's Java	34
3.4 Oracle8i – database system for Internet	37
3.4.1 Java tightly integrated with Oracle8i	37
3.4.2 Oracle8i Advanced Queuing	38
3.5 Summary	40
Chapter 4: A Model for Extending MCP over the Internet	41
4.1 Introduction	41
4.2 Limitations of the original MCP for the Internet environment	42
4.3 The general architecture of the <i>i</i> MCP	43
4.3.1 Production rules in the <i>i</i> MCP	45
4.3.2 A three-tier/two-tier configuration approach	47
4.3.3 Main characteristics of the three-tier <i>i</i> MCP	49
4.4 The logic and knowledge structure of the ATMS-based ES	50
4.4.1 Knowledge representation	51
4.4.1.1 External knowledge representation design	52
4.4.1.2 Internal knowledge representation design	56
4.4.2 The inference engine design	58
4.4.2.1 The ATMS design	59
4.4.2.2 The Rete-based production system design	60
4.4.3 Generating of retrieve query commands	61
4.5 Middle-tier processing role	65
4.5.1 Middle-tier agent data structure	67
4.5.2 Middle-tier processing mechanism	68
4.6 Database notification mechanism	71
4.6.1 Publishing rules mechanism	72
4.6.2 Subscribing Rules Mechanism	72
4.7 Summery	73

Chapter 5: A New Compromised Improvement Approach to ATMS Tightly Coupled Production Systems	75
5.1 Introduction	75
5.2 Drawbacks of the Morgue and Hindi Systems	76
5.3 The new compromised approach	77
5.3.1 Discarding a tuple from the compromised system	78
5.3.2 Asserting a tuple to the compromised system	81
5.3.3 A summary of the three different approaches	82
5.4 An empirical study on the three approaches	83
5.4.1 Student Registration Guidance System (SRGS): a case study	84
5.4.2 Measuring the efficiency of the three systems as standalone systems	86
5.4.3 The efficiency of the three systems as a part of the iMCP	90
5.4.3.1 Retract an assumption due to deleting the corresponding tuple from the coupled database system	91
5.4.3.2 Assert an assumption when a new relevant tuple is inserted into the coupled database system	93
5.4.3.3 Retract followed by reassert an assumption when a tuple is discarded and then revived during a transient database update operation.....	94
5.4.3.4 Retract a tuple and assert another one due a database update operation	96
5.5 Summary	97
 Chapter 6: Conclusion and Future Work.....	99
6.1 Conclusion	99
6.2 Future Work	101
 References	104
 Appendices	108
Appendix A: Student Registration Guidance System - Knowledge Base	108
Appendix B: Experiments' Case Specific Data	122
 Abstract in Arabic	132

List of Tables

Table 4-1: Query filter expression for the example in section 4.3.1.1	63
Table 4-2: SQL statements for the example in section 4.3.1.1	70
Table 4-3: Values of the <i>DBChange</i> attributes	72
Table 5-1: Main cases of the match algorithm in the three approaches	83
Table 5-2: Time comparison of standalone Morgue-like systems	87
Table 5-3: Operations comparison of standalone Morgue-like systems	88
Table 5-4: Space comparison of standalone Morgue-like systems	89
Table 5-5: Time comparison of normal reasoning for Morgue-like systems	91
Table 5-6: Operations comparison of normal reasoning for Morgue-like systems	91
Table 5-7: Space comparison of normal reasoning for Morgue-like systems	91
Table 5-8: Time comparison for the retract operation in Morgue-like systems	92
Table 5-9: Space comparison for the retract operation in Morgue-like systems	92
Table 5-10: Time comparison for the assert operation in Morgue-like systems	94
Table 5-11: Operations comparison for the assert operation in Morgue-like systems ...	94
Table 5-12: Space comparison for the assert operation in Morgue-like systems	94
Table 5-13: Time comparison for the retract-reassert operation in Morgue-like systems	95
Table 5-14: Operations comparison for the retract-reassert operation in Morgue-like systems	96
Table 5-15: Time comparison for the update operation in Morgue-like systems	96
Table 5-16: Operations comparison for the update operation in Morgue-like systems ..	97
Table 5-17: Space comparison for the update operation in Morgue-like systems	97

List of Figures

Figure 1-1: Communication mechanism in three-tier <i>i</i> MCP.....	5
Figure 2-1: A Rete representation for $r(Y) \wedge s(X>10) \wedge s(X<20) \rightarrow z(X, Y)$	10
Figure 2-2: Database subnets using MCP	23
Figure 3-1: Two-tier client/server model	28
Figure 3-2: Three-tier client/server model	29
Figure 3-3: One-to-one and one-to-many publish/subscribe interaction	29
Figure 3-4: RPC communication mechanism	32
Figure 3-5: Dynamic Management Agent Architecture	35
<i>Figure 4-1: i</i> MCP basic architecture	44
Figure 4-2: 3-tier/2-tier configuration	47
Figure 4-3: Multi-type communication architectures for 3-tier <i>i</i> MCP	50
Figure 4-4: OMT diagram represents the basic data types of the rule interpreter.....	56
Figure 4-5 Alpha/Beta objects	57
Figure 4-6: Hypothetical example for coupling the Rete and DN of ATMS	58
Figure 4-7: The ATMS design	59
Figure 4-8: The Rete architecture	60
Figure 4-9: The Rete network for example in section 4.3.1.1	63
Figure 4-10: Query expression in Java Dynamic Management Agent	64
Figure 4-11: <i>DBTuple</i> structure	64
Figure 4-12: Middle-tier components' data structure	67
Figure 5-1: Illustrate the changes in the dependency network	78
Figure 5-2: Hypothetical example to illustrate the discard algorithm	79
Figure 5-3: Main planing steps of the SRGS	85

Abstract

Coupling Expert Systems with Internet using Monitor Coupling Paradigm

By

Hussam Omar Saadeh

Supervisor

Dr. Khalil el Hindi

Co-Supervisor

Dr. Moneeb Qutaishat

In the last few years, the network computing has rapidly evolved and widely used in the Internet and Intranet environments. Attractive new mission-critical and networked economy (or e-commerce) applications have emerged. These electronic systems create effective, reliable, secure, and transactional Internet business solutions, which empower employees, suppliers, and customers with services and products available on the Internet.

In this thesis, we realize the importance of such evolution. We have developed a reasoning system that can work efficiently with database systems over the Internet environment, gaining the advantages of available network frameworks, interfaces, standards, and services. We believe that, this new coupling will introduce new opportunities for effective mission-critical reasoning solutions (we call, e-reasoning). Electronic reasoning systems may adhere other coupled electronic solutions (e.g., e-commerce) for enhancement, consulting, and supporting decisions.

Our proposed paradigm is an extension of the Monitor Coupling Paradigm (MCP) (Hindi, 1994) for coupling ATMS-based expert system with an active database system. The MCP was originally introduced to maintain the data consistency between reasoning system and the coupled database system. The rule engine of the extended paradigm (that we call *i*MCP) is designed to be distributed over three-tier network computing architecture. Considering reliability, scalability, simplicity, and interoperability, the system network model interface is constructed using client/server, publish/subscribe, and distributed object technology.

The efficiency of the reasoning system is a key issue beyond the success for such paradigm. We propose a modified efficient tightly approach for coupling ATMS and Rete network based on the Morgue system and the Hindi system. In different dimensions, an empirical comparison study is proposed for such coupling system that proves its efficiency among others. The new compromised approach maintains the main advantages of other systems and avoids most of their drawbacks.

Chapter 1

Introduction

Internet has become one of the most rapidly adopted technology changes to date. Reaching a total base over 10 million users in under three years (Oracle, 1997, a). It is considered as an infrastructure for many 24-hour global electronic services over heterogeneous, distributed, and autonomous environments. Networked economy (or e-commerce) is the most sensitive and complex service that is being developed. Its importance requires developing and deploying large-scale applications with reliable, secure, and transactional features.

On the other hand, comprehensive information system has to integrate within its framework several powerful complementary technologies. Those technologies are collaborated efficiently to increase system's capability and meet future demands. Integrating databases with expert systems has been considered one of the most effective system's collaboration.

Brodie (1988) had expected that, "the future computing systems will require AI and DB technology to work together with other technologies. These systems will consist of a large number of heterogeneous, distributed agents that have varying capabilities to work cooperatively".

In this thesis, our target is coupling expert systems and databases with standard service-driven network agents. Such agents were developed and deployed as a part of the public Internet environment. The value of this integration can be noticed in self-service, e-commerce, and mission-critical distributed information systems. From such integration, a new class of collaborated systems or electronic reasoning systems (or e-

reasoning) are founded. Their importance gained from dealing with other coupled electronic mission-critical systems.

In electronic commerce systems, a planner or a consultant intelligent system may be required on the network to confidant and empower users on their desired choices. For example, in a stock mart electronic system, users sell and buy shares in more safe and care if an electronic reasoning system is available for advice. In electronic airline agency reservation system, a person can effectively buy a flight-ticket from his web browser, when an electronic planing reasoning system is available to generate a flight-route that minimizes cost, time, and some risk factors. Also, for universities' registration process, students can easily submit proper schedules from their home, using intelligent electronic registration guidance system.

1.1 Coupling databases and expert systems

Coupling expert systems and database systems has been extensively investigated by many researchers (Brodie, 1988; Fernandies et. al, 1992; Golshani, 1984; Hindi, 1994; Simt, 1984) to utilize the benefits of each. Reasoning engine extracts its data and/or general rules from vast, shared, persistent repository to generate sophisticated inferences.

Two main approaches have been available for coupling ESs and DBSs:

- Loose coupling approach: keeps the expert system and the database separate. Before reasoning starts, relevant data is retrieved from the database to the expert system memory space.
- Tight coupling approach: interferes the two systems with each other in bi-directional interface, so the capability for reasoning can be available to the database system or vise versa.

From an engineering viewpoint, loose coupling integration is the most suitable approach for distributed, heterogeneous, and autonomous environment. However, typical loose coupling approach lacks the handling of data consistency problem (Hindi, 1994). Due to a database update, the reasoning system that is loosely coupled with the database may work on out-of-date copy of data that could generate incorrect solutions. Neither read locks nor periodically polling can practically solve this problem (Hindi, 1994). Read locks on retrieved data prevent other users from modifying the data, while the reasoning system is being executed. Furthermore, this method cannot recognize new inserted relevant data. On the other hand, polling is expensive and requires reasoning from scratch in every time different data is polled.

Hindi (1994) has proposed a new solution to the data consistency problem by introducing the Monitor Coupling Paradigm (MCP). The approach is characterized by using active database system with Truth Maintenance System (TMS) based reasoning system. TMSs have been used to cache inferences in a dependency network form. The active database system maintains set of rules denoting and monitoring selected data. Those rules are automatically constructed and installed before reasoning startup. Using POSTGRES alert mechanism (POSTGRES, 1992), when a relevant DML event occurs, an active rule notifies external program to poll the available changes to the TMS-based reasoning system. The TMS can directly identify all related inferences and revise its beliefs accordingly.

For efficiency and to avoid unnecessary costly interaction between the two systems, Hindi (1994) has used the Rete network in generating queries and active rules statements. This maximizes the utilization of the best database functionality (join and select operations) and narrows the selection criteria and active rules affects as specific as possible.

1.2 Extending the MCP for the Internet environment

In order to meet Internet's challenges, developed systems must be adaptable, evolvable, scalable, and interoperable, meeting certain agreed upon standards, which are being used (Johnston, 1996; Oracle, 1997, a & b).

Our proposed system for extending the MCP for the Internet environment has to be in a form that satisfies the above characteristics. The system network architecture should include the best of open network standards, and web universal features.

Figure 1-1 shows that the functionality of the rule engine of our paradigm is distributed over three-tier network computing architecture:

- Data server: is a database system (e.g., Oracle8i) that maintains the data, set of a predefined general active rules (triggers), transactional queue(s) to retain the changes, and a rule engine to dispatch the changes to the subscribers.
- Application web server: is a middle-tier that embodies the logic and the knowledge of the reasoning system, and running a Java service-driven agent, which receives and sends requests and replies from and to the other two tiers.
- Universal thin client: is a universal web browser, which downloads the reasoning system logic and knowledge (as a Java *applet*) using the HTTP protocol.

The reasoning system requests for a set of initial application parameters, which they are part of its knowledge domain. e.g., student number in an electronic registration guidance system, and source and destination in a flight-route planning system. To retrieve remote data from the database, the system builds efficient and simple query statements each based on a single database relation. These queries are grouped and transmitted asynchronously to the middle-tier agent using an open distributed object technology (e.g., Java/RMI). The middle-tier agent sends each individual query to the database server via synchronous query service call (e.g., Java Database Connectivity (JDBC)) to evaluate and return results data. The returned data are then propagated to

the requester (reasoning system). The middle-tier agent retains an exact copy for each query statement with its subscribers (reasoning systems). The database system (e.g., Oracle8i) views the middle-tier as a proxy for all connected clients. Any captured change on the database is automatically forwarded, using transactional queues and a database rule engine, to the middle-tier in a publish/subscribe computing model. In the middle-tier, the arrived changes are passed into another rule engine for further filtering according to the available retained queries. Each connected reasoning system receives only relevant changes that is of interest and revises its beliefs accordingly using Assumption-based TMS (ATMS) labeling algorithm.

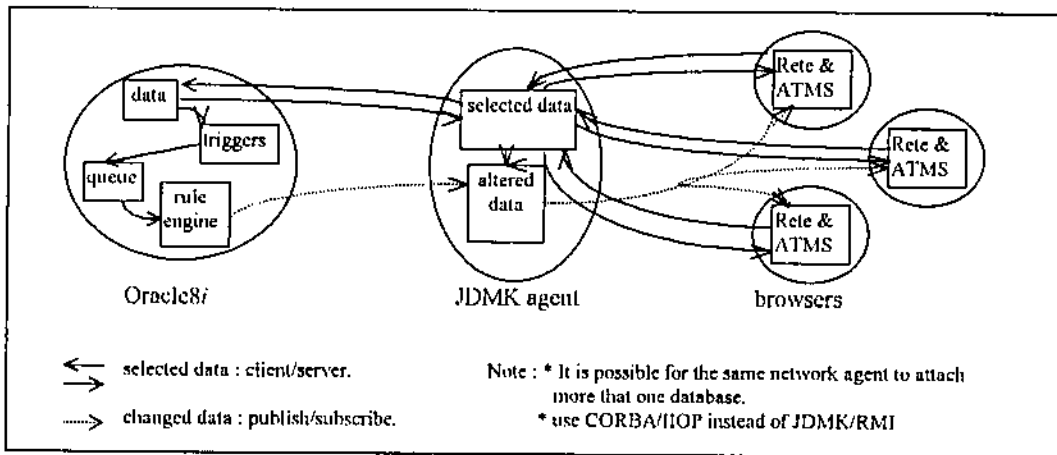


Figure 1-1: Communication mechanism in three-tier iMCP

The implementation of the expert system and the middle-tier agent was performed using Java language. Java development kit and Java dynamic management kit architectures had been adopted to develop and deploy the system. On the other hand, the data server functionality is only conceptually designed in Oracle DBMS, but not implemented. This is because Oracle8i, which has the necessary features for our paradigm, was recently released.

Many important issues can be discussed on this paradigm: efficiency, security, quality of services, and reliability of communication protocols. For security management, many policies can be identified to restrict data access for authorized clients. This is done through using proprietary database security features, standard services, secure protocols,

and other auditing tools. However, security is out of our scope in this thesis. Instead, we focus on building interoperable, portable, distributed architecture of the expert systems over the Internet.

1.3 Compromised approach for tightly coupling ATMS with Rete Network

A secondary but important goal in this thesis is improving the system efficiency using the Rete match algorithm in the inference engine, with the ATMS labeling algorithm. A modified tightly approach is implemented based on the Morgue system (Morgue and Chehire, 1991) and the Hindi system (Hindi, 1994).

Mahmoud (1997) had empirically proved that the Hindi system is more efficient than the Morgue system in terms of the time needed, but it requires more memory space than the Morgue system. In this thesis, we performed an empirical comparison study on the Hindi system, the Morgue system, and the compromised system. The results show that, the new compromised coupling approach is fair efficient for reserving space and reasoning runtime.

1.4 Thesis Structure

The rest of this thesis is organized as follows. In chapter 2, we review related literature on the ATMS-based reasoning system, and the coupling approaches with the DBS. In chapter 3, the network computing and Internet technologies are reviewed. Chapter 4 describes a proposed model for coupling expert system over the Internet using the MCP. Chapter 5 presents a compromised approach for tight coupling Rete network with ATMS dependency network. Finally, chapter 6 represents our conclusions and suggests some future work.

Chapter 2

ATMS-based Reasoning Systems and Databases

2.1 Introduction

In this chapter we review the conceptual design of the ATMS-based reasoning system that has the capability to revise its beliefs according to changes that have occurred on its assumptions. Based on this architecture, the mechanism of the Monitor Coupling Paradigm (MCP) (Hindi, 1994) for coupling reasoning systems and databases was presented. The MCP is characterized by keeping up-to-date data consistency between the database and the reasoning system in an asynchronous, loosely coupled manner. Where, there are strong demands for such coupling over the Internet in mission-critical applications.

The organization of this chapter is divided into two main sections: section 2.2 describes the architecture of the reasoning system as a Rete-based production rule system coupled with an ATMS label algorithm. Section 2.3 reviews the different approaches adopted to integrate DBs and ESs reaching to the original MCP.

2.2 ATMS-based reasoning system

Rule-based expert systems have been used since the 1970s (Davis et al., 1977) to represent complex knowledge and to formulate intelligent problem solving skills, for non-conventional problems, that may require large and adaptive human expertise.

Expert systems can be classified according to the nature of the adopted problem they intend to solve, including: planning, diagnosis, design, control, and monitoring. Typical expert system architecture has the following components. *User interface* for an

easy external accessing. *General knowledge* represented as a set of rules that formulate the problem solving. *Case specific data* that refers to a collection of facts describing a problem instance. *Explanation subsystem*, which produces how and why explanations. *Inference engine* interprets the knowledge, and controls the reasoning process.

In changing and adaptive reasoning environments, special caching mechanisms called Truth Maintenance Systems (TMSs), can integrate the expert system to record the inferences. Any modification of the data used to make these inferences automatically is propagated to the relevant ones and updates their status accordingly. Maintaining system truths during reasoning relieves the system from restarting the reasoning process from scratch every time some data changes.

In the following subsections, we discuss, how a production system architecture can be suitable representative for the rule-based interpreter, and how the Rete algorithm (Forgy, 1982) enhances its efficiency. Moreover, the basic ATMS (de Kleer, 1986, a) architecture is identified, that plays critical role in the success of the MCP. Also we describe, how the basic ATMS can be tightly coupled the production system in a way that increases the overall system efficiency.

2.2.1 Production Rule System

Production rule system is a pattern-directed computation model that addresses the importance of AI search algorithms for exploring effective solutions in modeling human problem solving.

A production system consists of a *working memory* (WM), a set of *production rules*, and a *recognize-act* interpreter. The *working memory* contains a set of assertions or facts that describe the current state of the reasoning process. Each *production rule* defines a small chunk of problem solving knowledge as condition/action pairs. The

recognize-act (match-select-act) cycle specifies the control structure of the production system. The system matches available rules to determine the applicable rules, and stores all applicable rules in a special agenda, called *conflict set*. Then, using one or more conflict resolution strategies (like FIFO, LIFO, highest priority, recency, refutation, specificity, ... etc.), a preferred instantiated rule is selected and fired. The previous steps are repeated, till the reasoning system has reached an empty agenda or a simple *halt* command is executed.

Production systems are characterized by a set of advantages that make it useful as an ideal tool for building expert systems. In addition to their simplicity, flexibility, tractability, and expandability, production system offers a range of opportunities for heuristic control of search. The system has the ability to use either data-driven (forward chaining) or goal-driven (backward chaining) search techniques, and the capability to employ different conflict resolution strategies within the select step of its recognize-act cycle. Also, the knowledge and the control are completely separated from each other. This increases the chances of building a domain-independent expert system shell. In this thesis, we use the term rule-based expert system and production system interchangeably.

2.2.2 The Rete Algorithm

491833

Conventional production systems spend a significant proportion of the processing time on the match step. To improve the performance of this repeatedly performed step, the Rete algorithm (Forgy, 1982) is widely used. *Sharing* and *state-saving* are the main system optimization issues that Rete address. Sharing common conditions across different productions rules reduces the number of test operations required to do the match. Unfortunately, this tuning has quite limited effects on the speedup of the system. As reported by Tambe and Rosenbloom (1992); the utilization of *shared* conditions has

improved time needed to perform the match step by a factor of 1.1 to 1.6. On the other hand, Rete *state-saving* property plays a critical key in the system optimization. Rete was motivated by observation, that firing a rule usually causes a limited number of changes in the status of the working memory. Thus, the result of the matching step performed in a cycle can be used, after a few modifications, in the next cycle.

The Rete algorithm compiles the production rules in a complex network structure, loads all initial working memory elements (WMEs) in the constructed network, and starts exploring new instantiation results. The rule interpreter uses predefined criteria for electing one of these results to be fired. Probably, due to firing a rule, the WM is modified by adding a new derived inference or by discarding an existing element.

Consider, for example, the following rule:

$$\text{if } r(Y) \wedge s(X > 10) \wedge s(X < 20) \text{ then } z(X, Y).$$

Where, r , s , and z are three predicates, and X & Y (capital letters) are variables. The left hand side of *then* is called the premise (or condition) and the right hand side is called the action (or consequent). Figure 2-1 below represents the Rete network for above rule.

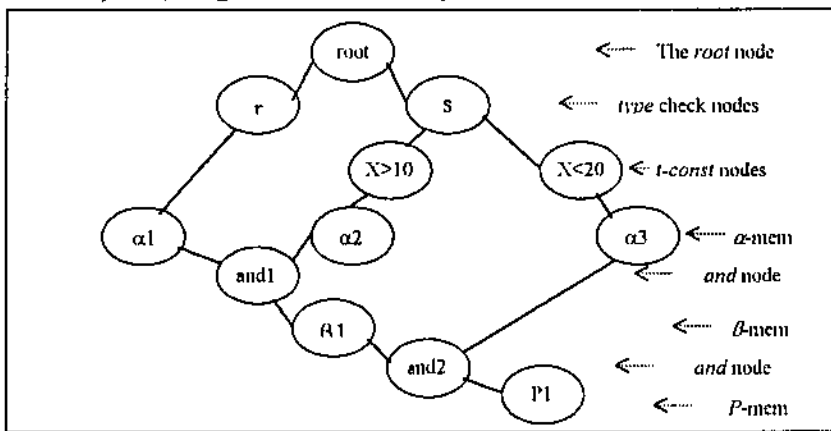


Figure 2-1: A Rete representation for: $r(Y) \wedge s(X > 10) \wedge s(X < 20) \rightarrow z(X, Y)$

The *root* node receives the working memory elements (WMEs or tuples) of s and r predicates, and propagates a copy of each one to its successors. A successor node (i.e., a *type check* node as in r and s nodes) tests the predicate's type for the arriving WMEs and passes all elements that have a proper type to its successors (mostly, *t-const* nodes) and

discards all the others. A *t-const* node (e.g., $X < 20$ and $X > 10$ nodes) forwards all WMEs that succeeded satisfy certain conditions into a special cache called α -memory node. An *and* node checks two WMEs (as an input) against a complex consistency condition, joins every pair of WMEs that satisfy the condition, and forwards a copy of the joined element to the following β -memory node. A β -memory node stores a copy of the arriving element and forwards it to the next node (which might be another *and* node). Every *P*-memory node represents a production rule. Any tuple that succeeds in reaching a *P*-node are cached as instantiation results in the *conflict set* to deduce new inferences (as in above example, z 's tuples).

Note that *shared* conditions are encoded once in Rete as a common path between two nodes. The *state saving* is realized by storing partial results of the match in the memory nodes to avoid re-performing the match from scratch.

It is worth mentioning that, there is a simple analogy between the operations performed by a Rete-based production system and a relational database system. *Type check* nodes correspond to the relations in the database. WMEs represent database records (or tuples). *T-const* nodes correspond to selection criteria. *And* nodes are no more than database join operation. The modification on the WM, as a result of a rule execution, is just a database insert, update, or delete operation.

2.2.3 Basic ATMS

Truth Maintenance Systems (TMSs) are useful for belief revision and as control mechanisms. They have been used in conjunction with a problem solver in adaptive reasoning systems to revise the current belief state due to environment changes without having to re-do the whole reasoning process every time some data changes.

TMSs accomplish this objective by recording all reasoning steps in form of dependencies. This property makes the system useful for database update, constraint satisfaction (avoiding re-evaluation of any constraint on the same data), and monitoring in dynamic environments (e.g., robots) (de Kleer, 1986, b).

Three main types of TMSs were developed: Justification-based TMS, Logical-based TMS, and Assumption-based TMS. In our work, we have adopted the Assumption-based TMS (ATMS). However, many of the developed techniques can be easily adapted to other TMSs. The ATMS can handle multiple-context for incremental systems on parallel. This issue is achieved by exploring all solutions simultaneously, dealing with inconsistent knowledge, and avoiding the need for backtracking (de Kleer, 1986, a).

The ATMS algorithm assigns a propositional atom node for each problem solver datum. The problem solver designates a subset of those nodes as assumptions that are presumed to be valid unless there is evidence to the contrary. A set of assumptions is called an environment. All inferences of the problem solver are registered in the ATMS as justifications. The justification has three parts: a consequent node, antecedent nodes that support the inference, and a problem solver description. Moreover, a node (n) is said to hold in the environment (E), if (n) can be derived from (E) and the current set of justifications (J), i.e. $E, J \vdash n$. Every ATMS node (n) is associated with a set of environments, that is called the label of (n). The node label should be consistent, sound, complete, and minimal. An environment with all its derivable ATMS nodes is called the ATMS context. Such that, $context(E) = \{ n \mid n \in N \supset (E, J \vdash n) \}$. ATMS dependency network is a network constructed using ATMS nodes as vertices and justifications as edges.

The ATMS node is the basic data structure in the ATMS. It has the following form: $\gamma_{\text{datum}}: \langle \text{datum}, \text{label}, \text{justification} \rangle$. Where *datum* is the problem solver datum, *label* is a set of environments in which the node holds, and *justification* is a list of inferences that support the node. According to the form of labels and justifications of the ATMS nodes, nodes are classified into:

- * A *Premise* node: has a justification with no antecedents, i.e., holds universally (always true). Premise's label is a single environment with no assumptions.

e.g. $\gamma_p: \langle p, \{\{\}\}, \{\{\}\} \rangle$.

- * An *Assumption* node: is a node whose label contains a singleton environment mentioning itself. In general, *Assumption* node is denoted in uppercase letters and can be justified. e.g. $\gamma_A: \langle A, \{\{A\}, \{B, C\}\}, \{(A), (d)\} \rangle$.

- * An *Assumed* node: mentions only assumptions in its justification slot. In other words, it is derived by immediate usage of one or more assumptions. The *assumed* node is preferable for defeasible problem-solving nodes. Assumption nodes are created using *assumption-assumed-node* pairs. Thus, no direct dealing with *assumptions* to prevent having ever justified *assumptions*. e.g. $\langle s, \{\{A\}, \{B\}\}, \{(A), (B)\} \rangle$.

- * A *derived* node: all other valid nodes are considered as *derived* nodes.

e.g. $\gamma_w: \langle w, \{\{A, B\}, \{C\}, \{E\}\}, \{(b), (c), (d)\} \rangle$

- * A *contradiction* node: is a special node type that can be proven from inconsistent environments. Inconsistent environments are also called *nogood* environments.

e.g. $\gamma_{\perp}: \langle \perp, \{\}, \{(A, B), (C, D), \dots\} \rangle$

Recall that, during reasoning process and label updating algorithm, any ATMS node can be dynamically converted from one type to another according to its label and

justifications list. However, reasoning system has to avoid such behavior to increase its performance.

The ATMS procedure performs three basic actions. Creating an ATMS node for a problem-solver datum, creating an *assumption* node, and adding a justification (inference) to the dependency network. In the last operation a significant processing time is consumed to propagate label-update through all relevant ATMS nodes to reflect their current status. This computation is carried out in five steps.

- First, a new label is computed for the consequent tuple of the current justification with respect to the label of each justification antecedent node. The label consists of all permutations of union one environment from a given antecedent with the others.
- Second, removing inconsistent and subsumed environments from the computed label.
- Third, adding the old label of the consequent tuple to the computed, and re-applying the second step again.
- Forth, if the node is γ_1 (i.e., inconsistent node), then, each computed environment is considered as a *nogood*. Traverse all environments to discard all these that are superset of the discovered *nogood* environments.
- Fifth, for non- γ_1 nodes, propagates label-update for each consequence node that has a justification, which mention the node whose label has changed.

The problem solver has to generate all inferences in terms of justifications, and pass them to the ATMS to reflect the current reasoning status. However, justifications must be precisely constructed otherwise unexpected ATMS errors may occur. If the problem solver fails to list all proper justification of a node, then the node's label will be either too general or too specific. de Kleer (1986, b) suggests that the problem solver must be encoded with a set of constraints and controls. These controls look like rules,

and are called *contradiction consumers* (or rules). Contradiction rules' task is to find out *nogood* environments in an early stage of the reasoning process to accelerate the overall process. A *nogood* environment represents a set of inconsistent assumptions that cannot be hold together. Note that a superset environment of a *nogood* environment is also *nogood*.

2.2.4 Coupling ATMS with a Rete-based production system

In this section, we will review several coupling architectures that integrate production system (as a problem solver) with the ATMS. This integration establishes a new flexible system that has the capability to revise its current beliefs in dynamic environments, and provides an interoperable interface to communicate effectively with other powerful sources of changing data such as database systems.

2.2.4.1 ATMS loosely coupled production system

This method which was introduced by Morgue and Chehire (1991), integrates a production system and ATMS in the select step (conflict resolution) of the inference engine. The production system transmits justifications and assumptions to the ATMS. The ATMS computes the label of the justification consequence. If the label is empty (has no environments), then, no more information is obtained. So the instantiated rule that belongs to the transmitted justification is discarded (not executed), and another one is selected to be executed.

In loose-coupling approach, contradiction rules have the highest priority for execution. Contradiction rule can find out the inconsistent (i.e., *nogood*) environments in an early stage to reduce unnecessary expensive operations. For example, suppose the system knowledge has the following two rules:

A: *if* $q(1) \wedge p(1)$ *then* \perp ,

and B: *if* $q(X) \wedge p(X) \wedge r(Y) \wedge w(Y)$ *then* $z(X, Y)$.

If rule B is executed before rule A , then the joined tuple " $q(I) \wedge p(I)$ " has to be matched with all elements in predicate r , and the partial results have to be matched with all elements in predicate w , and infer new derived facts about predicate z , that may also be a constituent of another rule, and so on. In a later stage, when rule A is executed, then we will discover that many of the above operations were performed unnecessarily, and their results must be discarded from the Rete and the ATMS.

In spite of its simplicity and modularity, this technique has two main drawbacks (Morgue and Chehire, 1991). A lot of work is done repeatedly by the ATMS labeling algorithm. Common patterns across different rules are not recognized by the ATMS. Thus, the ATMS does not benefit from the Rete *sharing* characteristic. Also, many unnecessary operations are still processed by the match operation, even if the labels of the partial results are empty. This is very clear from the previous example, when an expensive and unnecessary join operation in the Rete is performed on " $q(I) \wedge p(I)$ ".

2.2.4.2 ATMS tightly coupled production system – the Morgue System

To cope with the previous drawbacks, coupling production system and ATMS has to be at the match step instead of select step (Morgue and Chehire, 1991). The Rete match algorithm with some modifications is used. For partial joined results (that are located in β -memory nodes), new ATMS nodes and partial justifications are introduced. Furthermore, each stored tuple (simple or partial) in the Rete attaches a label. Due to rule firing and join operation, justifications and partial justifications are constructed. The label of the consequent tuple is computed. If it is empty, then, no further matching is done. Otherwise, the Rete propagates the tuple down, not only for joining, but also, for label computations.

e.g. *if* $p(X) \wedge q(X) \wedge r(X)$ *then* $w(X)$.

if $p(X) \wedge q(X) \wedge s(X)$ *then* $z(X)$.

The Rete replaces these two justifications by the following three partial ones:

$$p(X) \wedge q(X) \rightarrow N(X), \quad N(X) \wedge r(X) \rightarrow w(X), \quad N(X) \wedge s(X) \rightarrow z(X).$$

where, each partial joined tuple has an intermediate label located within its datum.

The Morgue system discards, during match algorithm, all tuples (facts) with empty labels, to reduce the cost of the join operations and labeling computations. Similarly, when a *nogood* environment is discovered, the system follows all relevant tuples for label-update, and removes all tuples that became with empty labels from the Rete memory nodes.

The main drawback of this approach is that, the system maintains labeling for more nodes than the loose coupling approach. In somehow, sharing the label for the common patterns (using the sharing property of the Rete algorithm) reduces this complexity. Moreover, this approach uses an extra bookkeeping space to optimize label updating and contradiction handling algorithms. First, each tuple (simple or joined) records the Rete memory nodes where it is stored. Second, each environment records all tuples (simple or joined), in the label of which it appears. Third, each tuple maintains the links to other tuples through partial justifications. In addition, contradiction rules still have the highest priority and are fired as soon as they have been instantiated.

2.2.4.3 ATMS tightly coupled production system – the Hindi System

Hindi (1994) noticed two main serious drawbacks in the Morgue system. Whenever a new environment is added to a tuple (which frequently happens, since ATMS applications require reasoning in multiple-context), the system has to re-join that tuple with all tuples that are located in the other input memory node. Because, if the output joined tuple has been discarded when its label was empty, then, the re-join operation is the only way to re-generate such tuples again, to check if their labels have become non-empty. The second drawback, re-generating previously discarded tuples, requires re-

performing the match on these tuples and all relevant ones from scratch, since discarding a tuple also discards all tuples that the tuple is a constituent of. Moreover, as a result of adding a new environment to a tuple, if that tuple is re-generated with non-empty label, then the system may re-perform an expensive re-join operation, and re-propagate the work of matching on the other output memory nodes, which already has been computed.

Hindi (1994) realized Morgue's drawbacks and designed a new improved tight coupling approach. Hindi's improvements are based on caching tuples with empty labels instead of discarding them. Tuples with empty labels are moved to a special inactive part of the Rete memory node, that is called the OUT-part. While, the active memory part, which is called IN-part stores all tuples with non-empty labels. The tuples in the IN-parts only are involved in match step. The Hindi system uses time stamp policy in the Rete match algorithm. The label is computed for each joined tuple. If the label is empty, then the tuple is placed in OUT-part after it is stamped with a special lowest unused time (e.g., -1). Later on, when it becomes with non-empty label, it will be joined with all opposite available in-tuples. On the other hand, the tuple with non-empty label is placed in the IN-part after it is stamped with the current system time. The Rete propagates the match as usual.

When a *nogood* environment is discovered, tuples with labels that have become empty are moved to the corresponding OUT-part re-stamped with the current system time. The system time is then advanced, so that, it is possible to identify all new tuples that are created while those empty-tuples are in OUT-part.

When adding a new environment (due to receiving a new justification), the tuple has only to follow the links of the dependency network of the ATMS to get the other related tuples for label-update; without having to re-join tuples. If an empty tuple becomes non-empty one, then, it is quietly moved to the IN-part. Then it is matched with other tuples

that were inserted while it was in the OUT-part (can be determined using time stamps), and following its old links for labeling the existing relevant tuples.

As noticed, the Hindi system does not perform an actual release operation on the discarded tuples. Thus, to reduce the memory requirement for the system, Mahmoud (1997) suggested that we could switch from the Hindi to Morgue systems once the memory required for the inactive parts become so large.

2.3 Expert systems and Databases coupling approaches

In this class of integration the expert system is used to perform intelligent reasoning processing on information being stored in, and retrieved from the database system. Many applications including office automation, statistical system, and military command and control, require this combination of technology (Smit, 1984). Approaches in this area are classified into two main categories. Tight coupling approaches and loose coupling approaches. Researchers in loose coupling approaches interest on constructing interoperable interface between two existing systems. While in tight coupling approaches, the interest has been concentrated on building single and efficient system that has the capability of storing and retrieving data, and deriving new inferences using shared and persistent rules.

Three main subjects are discussed below. Loose coupling approaches, tight coupling approaches, and monitor coupling approach which combines some of loose and tight features.

2.3.1 Loose coupling expert systems and databases

The reasoning system in this category accesses the database system using external database queries. Retrieved data is processed by the memory-based reasoning system without any database feedback. The connection between the two systems is temporarily established only to extract the relevant data either at run or compile time.

The efficiency of the loose coupling paradigm is proportional to the amount of data retrieved and number of times a database is accessed. However, most of the loose coupling techniques were concerned with a PROLOG interpreter and a relational database system (or PROLOG++ with object-oriented database system). PROLOG facts correspond to tuples in databases, predicates to relations, rules to views, assert/retract to insert/delete, and so on.

Hindi (1994) mentioned a set of drawbacks of using typical loose coupling approaches:

- They discard the implications of data consistency problem, which may make them produce incorrect solutions. The passive connection model between the two systems does not recognize the occurrence of any modification on the retrieved data in the database system. Neither read locks on the retrieved data nor periodically polling can solve this problem. To protect data against updates, using read locks on the retrieved data, for long runtime, prevents other uses from modifying the data, while the reasoning system is being executed. Furthermore, this method cannot recognize the occurrence of a new inserted relevant data. On the other hand, periodic polling for data for refreshment is expensive and requires reasoning from scratch in every time different data is polled.
- They lack the ability of handling knowledge persistence and sharing.
- They fail to process large retrieved data set with partitions.

2.3.2 Tight coupling expert systems and databases

The reasoning system in tight coupling approaches is an integral sub-component of the database system with a transparent interface. It is also possible to tightly couple the reasoning system with the database by extending the knowledge base system to support some DBMS features, which include concurrent control, persistency, transaction management... etc. In both cases data and rules are persistent and shared.

Tight coupled approaches are either use deductive rules that has the ability to derive new information, or active rules, which provide reactive behavior due to occurrence of a database events (e.g. insert, update, or delete).

In deductive relational database systems (DRDSs), the system is either constructed from two complete languages (e.g., PROLOG with RDBMS), or finely tuned formalism based on converging specifications of two particular engines (e.g. Datalog) (Fernandes, 1992). While deductive object-oriented database systems (DOODs) (for review see (Sampaio and Paton, 1997)), are build by extending relational deductive languages (e.g. CoceptBase, Logres+), integrating deductive with object-oriented imperative language (e.g. ROCK & ROLL), or reconstructing new object-oriented logic language (e.g. FLORID).

However, deductive database systems lack high level programming features, such as memory variables, call external procedures, IO functions, and control structure. Also, the research on DOOD model is not fully matured, only a small set of DOODs is released as commercial products (e.g., Validity) and many remain research projects. On the other hand, using active rules to simulate complex deductive rules affects the system performance.

2.3.3 Monitor Coupling Paradigm

Hindi (1994) has introduced a new paradigm to couple the database with the expert system, called the Monitor Coupling Paradigm (MCP). The MCP is characterized by using rules in both rule interpreter and database system in a loose communication model. The proposed paradigm addressed the data consistency problem, data partitioning, and systems' role distribution. With respect to the tight coupling approach, the system also offers sharing and persistency for some special rules.

The Monitor Coupling Paradigm relies on the capability of the active rules to monitor data against certain events and provide two-way communication. Not only the application can retrieve data, but also the database can send data to the running application. Also the system is characterized by integrating a TMS with a forward-chaining reasoning system. Before the system starts the reasoning process, all relevant data are retrieved from the database using external data queries. When a modification on the extracted data or an insertion of new relevant data take place, active rules notify the reasoning system about their occurrence. The TMS component of the reasoning system, in turns, receives those changes and tries to identify all relevant inferences and revise their states accordingly.

The functionality is distributed between the database system and the expert system in a way that increases the overall system performance. Not only the database role is focused on storing and retrieving data, but also the database participates in performing optimized join operation where possible, and emits all relevant changes to the expert system when they are occurred. The MCP uses the Rete network to generate the database query commands and the associated active rules. After the Rete network is constructed from the available rules, the system identifies all partitions (or subnets) in the

network that only deals with database predicates. Those subnets are automatically mapped into the corresponding database queries and active rules.

The MCP uses POSTGRES DBMS (POSTGRES, 1992) to generate database commands, since POSTGRES active rules define an alert mechanism (listen and notify command) that is necessary in the paradigm. Notify command on a given relation alerts the running program that is keeping an eye on that relation (through issuing listen command). A secondary relation is required in POSTGRES to retain the changes for each base relation.

e.g. *define rule r1 is on append to test1 do*
append test1a (i=new.i)
notify test1a

A relation *test1a* is a secondary relation for *test1* that has one attribute (*i*). When a new tuple is appended to *test1*, the system notifies running programs, which are currently monitoring the relation *test1a*.

It is worth to mention, that Hindi (1994) mechanism has defined also for non-monotonic reasoning systems (for *if...unless...then...* rules), which is out our scope in this thesis. However, to illustrate the MCP mechanism of monotonic case, consider for example, the following rules:

P1: *if db_q1(X>50, Y), db_q2(X, Z>25) then ...*

P2: *if db_q1(X>10, Y), q3(Y>20, Z) then ...*

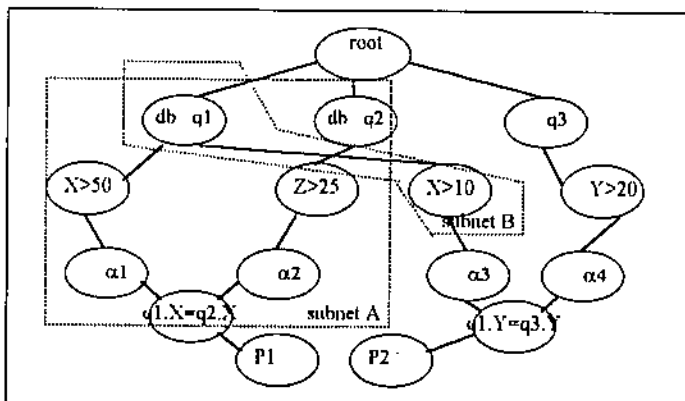


Figure 2-2: Database subnets using MCP

Two subnets are marked on the figure 2-2 as A & B. Subnets only deal with database predicates in a form that maximizes the database role in the select and join operations and minimizes the number of active rules and retrieve commands. The system generates the retrieve commands using POSTQUEL query language (which is an extension to the Quel query language). For example to extract subnet A:

```

retrieve unique (Tq1.all, Tq2.all)           // where "unique" tag used to discard
from Tq1 in q1, Tq2 in q2                 // any duplicate retrieved tuples
where Tq1.X > 50
    and Tq2.Z > 25
    and Tq1.X = Tq2.X

```

To generate the corresponding active rules, two secondary relations are required to alert the reasoning system with the changes. One for inserted tuples and the other for deleted tuples, where updated tuples are treated as deleting old values and inserting new ones. For each relation in a subnet, the system has to generate four rules: on-delete, on-append, on-replace for old values (equivalent to on-delete rule), and on-replace for new values (equivalent to on-append rule). For example, the following two rules monitor the tuples deleted from relation (*q1*):

```

for subnet A:      define rule deleted_q1_1 is on delete to q1
                   where current.X > 50 do
                   append deleted_P1 (current.all, q2.all)
                   where current.X = q2.X
                       and Tq2.Z > 25
                   notify deleted_P1

for subnet B:      define rule deleted_q1_2 is on delete to q1
                   where current.X > 10 do
                   append deleted_alpha3 (current.all)
                   notify deleted_apha3

```

2.4 Summary

The Rete-based production rule system is considered as an efficient, flexible architecture in modeling human problem solving. Tightly coupling the ATMS labeling algorithm with the Rete-based production rule system offers, to the produced system, the ability to revise its beliefs when some data changes, and the switching-free mechanism between inconsistent solutions in multiple-context problems.

On the other hand, different approaches could be effectively proposed to integrate the database system with the reasoning interpreter, to gain the benefits of each. The integration mainly has been performed either tightly as in deductive database systems, loosely using demand-driven communication model, or monitor relies on event-driven model. The last approach, as it was called the MCP, is characterized by coupling TMS-based reasoning system with active DBMS. In the MCP, the database system responsibility not only concentrated on the database selection, but also on publishing relevant changes to the listening external systems. In contrast, the ATMS-based reasoning system has to receive those changes and revise the system beliefs accordingly.

Chapter 3

The Network Computing and the Internet

3.1 Introduction

In this chapter, we review the evolution of the Internet computing technology that has become quietly matured for thin clients (i.e., browsers) to run reliable, secure, transactional, and dynamic programs. That evolution (or even revolution) permits reasoning systems (i.e., complex applications) to be developed, deployed, and integrated over the Internet environment. The Internet computing technology was started with limited capabilities such as accessing static pages, sending electronic mails, publishing news, and performing non-behavior controls. But today the technology is characterized with the ability to run behavior scripts and the use of the full power of programming (i.e., *Java applets*).

Java is an object-oriented, portable, networking language that was mainly designed for the Internet behavioral applications. Compiled *Java applets* have the capability to be downloaded from web servers and run over the Java Virtual Machine (Java VM) of the client's browser. However, creating efficient applications or reliable mission-critical systems requires developing different modules, libraries, and utilities over the available network with different interfaces. Also for completeness, those applications could collaborate with other existing legacy systems (Johnston et al., 1996).

The integration challenges for different components in distributed, heterogeneous environments are carefully addressed, managed, and controlled in distributed object technology. Distributed object technology combines object-oriented features with

distributed systems development. This technology intends to arrive to a seamless integration between application components by offering data encapsulation, polymorphism, and interoperable interfaces.

Using open standard TCP/IP-based distributed object protocols (such that, IIOP and RMI) and their services with already available protocols on the Internet (e.g., HTML/HTTP), increases the chances of offering behavioral interfaces on the Internet.

In the following sections, we describe related network issues for communication mechanisms, which are currently available either on the client side, web application server, or database system. Those issues are used in the next chapter in which we formulate the MCP for the Internet environment.

3.2 Communication Computing Models

Due to the evolution of high-speed and inexpensive personal computers, the systems architecture have been migrated from expensive, large, centralized mainframes to economic, reliable, expandable decentralized networked machines. Many communication models have been introduced to structure a meaningful connection from sender to receiver over networks. Client/server, publish/subscribe, workflow, peer-to-peer, and static and roaming agents are just examples of such communication models.

A communication model must guarantee the delivery of transferred messages in a meaningful format. Losing messages or receiving the same message twice might affect the semantic of messaging model. Protocols can transmit messages using either at-least once semantics, at-most once semantics, best-effort semantics (i.e., guarantee nothing), or exactly-once semantics. Where the last is the ultimate and could be costly to achieve.

In this thesis, we only concentrate on the architecture of client/server and publish/subscribe models. Illustrating their mechanisms, features, and variations.

3.2.1 Client/Server Model

Two cooperated processes on the same machine or on two different machines are connected through two-way communication. A request is initiated from a client process to be processed by a server process. The server offers set of services to the users (or clients). The client requests a given service from a certain server. During request processing, client can either block till the reply is returned in synchronous mode or continue executing in parallel with the request processing in asynchronous mode (see figure 3-1).

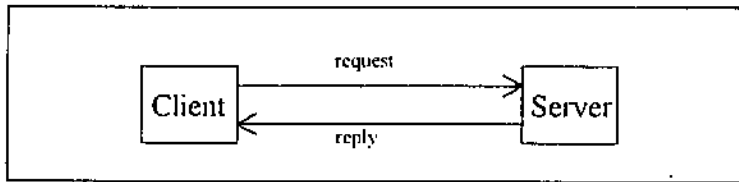


Figure 3-1: Two-tier client/server model

In order to send a message from client to server, the client has to know the service address on the server. The model can either use hardwire machine address and process identifier, logical process identifier, or an ASCII name (Tanenbaum, 1995).

Client/server model can be extended from a two-tier to a three-tier architecture. A three-tier architecture increases system scalability, reusability, manageability, and efficiency. Moreover, three-tier architecture decreases cost of development, deployment, and maintenance. As shown from figure 3-2, this network computing architecture, which has been adopted for the web applications, is composed of a data server as a back-end tier, an application server as the middle-tier, and a thin client as a front-end tier. Clients only own a simple set of programs, which mainly concentrate on building user interface. Most of the application logic is located in a centralized application server as services.

Clients request the application server for those services, which may in turn access the data that are stored and managed on the connected database server.

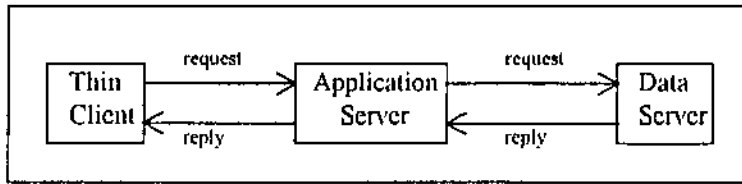


Figure 3-2: Three-tier client/server model

3.2.2 Publish/Subscribe Model

Unlike client/server architecture, publish/subscribe interaction is an event-driven model rather than a demand-driven. Publish/subscribe model transparently decouples the communication between processes without being directly connected to each other.

In the model, two types of entities are identified (see figure 3-3):

- The publisher: produces an event data as a message to a communication channel or information bus broker. Publisher responsibility is to create an event in an independent manner of where and when the event is dispatched.
- The subscriber: that receives (or consumes) event data from the associated channel or message brokers. It only consumes messages matching its interest. Subscriber is also unaware of how the received data is published.

The communication between publishers and subscribers is anonymous, no direct connection-oriented communication between them. On the other hand, the form of the interaction between applications in publish/subscribe model can take unicast or multicast communication models.

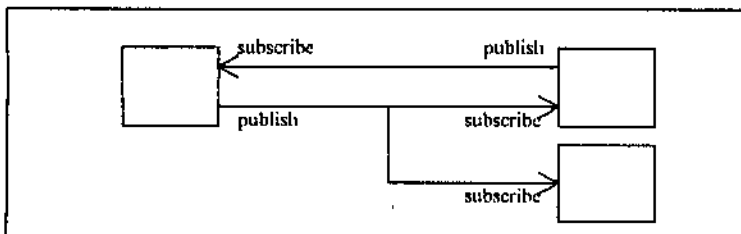


Figure 3-3: One-to-one and one-to-many publish/subscribe interaction

However, Messages in channels or information bus brokers could be further filtered using one of the following transparent addressing modes:

- **Subject-based addressing:** publishers associate with each produced message a subject name. Subscribers register their interest on a specific subject and receive only all messages associated with that subject.
- **Content-based addressing:** publishers produce structured messages (i.e., messages with attributes). Subscribers register their interest using filtering criteria. A dispatch engine has to evaluate each published message on the available filters to determine which message must be sent to which registered subscribers.

Pull and push models have been defined as applicable approaches to initiate event communication between producers and consumers. Pull model or synchronous notification, allows a consumer of event to periodically request the event or block until the event is arrived. While, push model or asynchronous notification, permits a producer of events to transfer the event automatically to the subscribers. Subscribers can register callback functions in the dispatch engine. When messages are produced, the engine immediately invokes the corresponding callback functions to be carried out on the interested subscribers.

Publish/subscribe model has been originally evolved using real world trading broker applications called Message-Oriented Middleware (MOM). TIB/Rendezvous (TIBCO, 1997) is one of most powerful communication tool that uses exactly-once reliable delivery services for subject-based publish/subscribe model.

3.3 Distributed Object (Component) Technology

It is widely believed that the next generation computing systems will consist of a set of distributed component-based systems. An application is a set of collaborated useful pieces (or components) that can be operated and accessed across different hardware, operating systems, networks, and languages. Bringing this technology to the Internet, enhances the nature of web-based and client/server network computing. Systems could be efficiently partitioned across the network in components fashion (or distributed objects) to utilize the network resources without being aware on the type of the running operating system and the available hardware architecture.

Distributed object technology has evolved through combining the procedural distributed technology with the object-oriented paradigm. The resulted technology (i.e., distributed object technology) enhances Remote Procedure Call (RPC) with data encapsulation, object polymorphism, and interoperable interfaces. However, objects could be large and complex components that were implemented using non-object oriented languages (e.g., legacy systems). In RPC, a client application calls remote procedures as if they were locally running, through hiding communication complexity. Remote procedures in the client side are represented as client stubs, which are linked with running local libraries. At runtime, client stubs are executed due to ordinary call from the client program. The client stub packs called procedure and its parameters into a binary message through marshaling operation, and sends that message as a request to the remote server. The server receives the message via another server stub, unpacks the calling statement, and dispatches the call to the actual server process. Execution result is returned back using the same manner (Tanenbaum, 1995). The mechanism of RPC is illustrated bellow in figure 3-4.

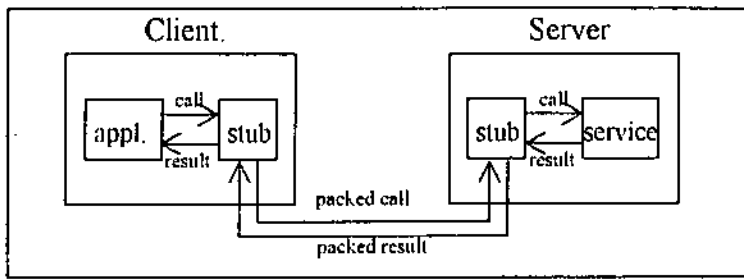


Figure 3-4: RPC communication mechanism

OMG CORBA, MS D/COM, OSF DCE, and Sun's Java are the most widely accepted distributed frameworks. The architecture of Sun's Java and OMG CORBA (OMG, 1998) are discussed in the following subsections. Where, CORBA is a specification, which can be implemented in Java, and both are a platform independent. Java is a portable language and CORBA is an interoperable specification.

3.3.1 OMG CORBA

Common Object Request Broker Architecture (CORBA), is an Object Management Group (OMG) specification that is supported by over 800 members (OMG, 1998). OMG was founded in 1989 to provide common object oriented guidelines for application development. OMG specification is based on a conceptual infrastructure called the Object Management Architecture (OMA). The OMA defines the framework by which the interoperability goal of matured components (application objects) can be achieved. CORBA reference model that defines the OMA, is divided into four main levels:

1. **ORB:** Object Request Broker enables object to transparently make requests and responses in distributed environments. i.e., the RPC mechanism for CORBA.

CORBA has defined a descriptive language to represent the external format of interfaces between objects (i.e., data types and method prototypes), called the Interface Definition Language (OMG IDL). OMG IDL obeys the same rules (i.e., syntax and semantic rules) of C++, and can be translated during implementation of application objects in a predefined mapping to Java, C++, C, COBOL, ADA, or

Smalltalk. Methods in CORBA are either executed in best-effort semantics, or exactly-once semantics. Also, method calls can run in synchronous, asynchronous, or deferred synchronous modes. Using OMG IDL, requests and replies are constructed in CORBA ORB using dynamic or static stubs. With no actual copy, clients identify remote objects through an opaque handle object called Object Reference.

To guarantee the interoperability between different vendor ORB's implementations, CORBA uses a set of APIs or internal conventions to support inter-ORB bridge that ensuring the semantics and contents from one ORB to another. Also, CORBA identifies a standard general transfer protocol to standardize data representation and message internal formats. Where, CORBA TCP/IP-based transfer protocol is called Internet Inter-ORB Protocol (IIOP).

2. **Object Services:** within the system framework, objects can extend their functionality by supporting collection of standard packages. These packages or services are considered as components with IDL-specific interfaces. They support naming bindings, persistency, transactions, objects life cycle, security, concurrency control, pull/push event models through powerful communication channels, externalization by recording the object status in a stream of data, and others. Also, an interoperable interface for a query service has been proposed that supports general manipulation operations: selection, insertion, update, and deletion. The query service is applied on a collection of objects using immediate evaluation or a more powerful management interface that increases the chance for optimization, monitoring, and frequent query execution.
3. **Common Facilities:** are collections of higher level services that many of applications may share. Also, standardization of the interface of those products increases their interoperability. Common Facilitatés can be classified into User Interface, Information

Management, Task Management, Services Management, and set of domain-specific market facilities. Specific market facilities support various market segments such as, health care, retailing, manufacturing, and financial systems.

4. Application Objects: are IDL-interfaces components specific to end-user applications. Application objects are based on extensive use and reuse of object services and common facilities, which increases application interoperability and portability.

3.3.2 Sun's Java

Distributed object technology in Java is modeled within the language itself. Java is a lightweight, simple distributed interface model compared to CORBA. However, CORBA interfaces and services are more powerful and reliable.

Java Remote Method Invocation (RMI) is a Java specific RPC mechanism. Java RMI interacts with two types of object parameters, remote and non-remote objects. Remote arguments are passed by reference, while non-remote objects are passed by copy. Packing parameters is done using *Serializable* protocol. Java *Serializable* protocol gets copy of objects' status (i.e., fields) in a binary serial message, writes the serial form on the available output stream with meta annotations, and restores that message into local proxy classes. Java downloads non-available proxy classes using annotations and predefined network loaders.

Each remote object is bounded with a name in server's special registry. Using URL-based name, clients can lookup for the reference to that remote object. Registry service is similar to Naming and Trader CORBA services.

CORBA can use object life cycle service to handle allocation and de-allocation of resources. On the other hand, Java RMI uses Distributed Garbage Collector (DGC) concept, which is similar to Java VM local Garbage Collector. DGC maintains a

reference to each remote object. To access remote objects, clients have to request for a lease (i.e., period of time). Client's RMI responsibility is to renew the lease before it has expired. When the used lease is expired, DGC assumes the client reference is no longer alive. DGC purges the remote object when no one is referencing it.

Since Java is a programming language, a set of built-in services is available as a part of the language itself. Java JDBC is a set of standard APIs available as SQL Query service. Java Listener/Event model supports publish/subscribe local event communication model.

On the other hand, many separate open standard services are written to be run on the Java VM. In addition to the Java portability, those services also increase the application interoperability. Java Dynamic Management Agent is one of a flexible, lightweight, service-driven network component model. The agent dynamically encapsulates, configures, and activates a set of needed services, and handles requests to and replies from the managed services with protocol-independent interfaces.

Dynamic agent architecture is composed of the following components (figure 3-5).

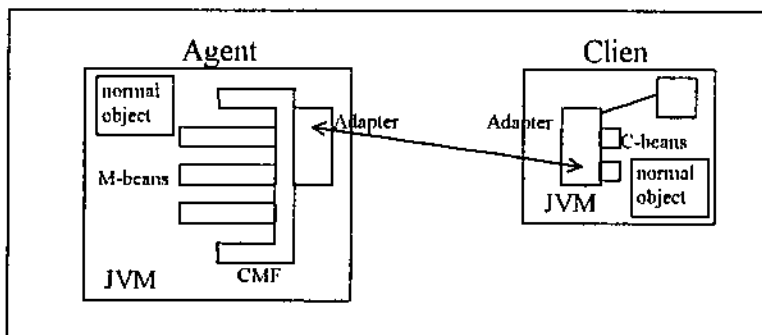


Figure 3-5: Dynamic Management Agent Architecture

- Core Management Framework (CMF): is the backbone object that registers all services and resource objects. CMF controls the accessibility between resource objects and services. The accessibility of objects can take a client/server or publish/subscribe models.

- **Services or resource objects (M-beans):** are managed objects that implemented in a form of components that conforms to Java Beans design patterns. The design pattern determines a simple standard form of objects' properties, actions, and events. Each managed object in CMF is assigned to a string name to uniquely identify it. Resources and services can be dynamically deployed and configured in the agent to be managed. The architecture supports a set of predefined services includes data repository, meta-data, filtering, relationship, library loader, bootstrap, launcher, scheduler, alarm clock, attributes' monitoring, discovery, and cascading agents.
- **Adapters:** are managed components, which provide a transparent communication between managed objects and external applications. They include RMI, HTTP/TCP, HTTP/UDP, HTTP/SSL, HTML, IIOP, and SNMP. Multiple-protocol support feature of the agent architecture has implemented the CMF, services, and resources as protocol-independent. The agent to be accessible it must include at least one adapter. Also, managed object cannot be accessed through adapters unless it was registered in the CMF.
- **A client-bean (C-bean):** is a stub object that represents a remote managed object in the client side. Client performs operations on c-bean which they are automatically propagated to m-beans. With the same concept, events that are emitted (published) by m-bean are directed through CMF to the corresponding c-bean using similar interface to the Java Listener/Event model. It is worth to mention, that the task of c-bean does not include parameters marshaling or any communication operations, which are the adapters responsibilities. C-beans are considered as a level for hiding the complexity of building requests on the corresponding m-beans through adapters' APIs. Therefore, without losing communication transparency, m-beans can be accessed directly through adapters without generating the corresponding c-beans.

3.4 Oracle8i – database system for Internet

Oracle8i is designed specially to support database applications on the Internet. Oracle8i offers set of valuable features that increase performance and manageability of web-enabled applications. Oracle8i captures the importance of electronic commerce and mission-critical systems, by providing many opportunities for developing powerful procedure-based or component-based information systems for Internet and Intranet applications. For that purpose, Oracle engine has been integrated with various network technologies such as, messaging systems, Internet file system, and thin client support. The engine implements efficient Java VM and CORBA ORB, and supports HTML, XML, SSL, IIOP, ... protocols.

3.4.1 Java tightly integrated with Oracle8i

Java has being considered as a standard behavioral language for Internet. Oracle8i realized this fact by implementing a scalable, reliable Java VM within the database server. Oracle adopts Java as a choice for developing distributed applications based on three initiatives (Oracle, 1997, b):

- Implementing various optimized JDBC Oracle drivers through different network architectures.
- Supporting SQLJ as an open standard for embedding SQL in Java. SQLJ is a simple compile-time call-level library that enhances JDBC API with a static analysis and type checking. SQLJ is more convenient, concise, and more safe than JDBC calls. JSQL clauses are translated into equivalent JDBC API through pre-compilation step.
- Integrating a more scalable, reliable, efficient Java VM in the Oracle engine. This integration permits powerful implementation of stored procedures, functions, and

database triggers using Java programs. Java bytecode could be further translated to efficient running code using Java-through-C compiler.

3.4.2 Oracle8i Advanced Queuing

Message queuing is considered as the most powerful, reliable, and scalable model for coupling distributed, autonomous applications and multiple-vendor products in asynchronous loosely manner. The application of one side can issue a set of messages in persistent, transactional queues. Messaging brokers (i.e., MOM) or internal engine mechanisms provide the *guaranteed delivery* of those messages, to whom they may be of interest. Messages are delivered and placed into another queuing system or external listening program. The messaging communication model has the capability to preserve, track, document, correlate, and inquire messages. (Oracle, 1999, b).

Oracle8i Advanced Queuing (AQ) is designed to address messaging queuing technology with a high degree of reliability and scalability in developing mission-critical and message-based solutions. Oracle8i AQ offers structured payload (or data), priorities, ordering, window of execution, tracking, and transactional behavior for messages. Also, Oracle AQ supports content-based publish/subscribe model to multiple recipients, workflow communication model through schedule propagation, and asynchronous client/server model using messaging queues as reliable communication buffers.

Messages in Oracle AQ are either structured with an object type or non-structured (raw) in a binary data format. Queues are either persistent or non-persistent. Persistent queues are created over relational tables in which one or more queue with the same structure can be constructed within the same relation. Producing and consuming messages is handled using *enqueue*, and *dequeue* operations. The sequence of those operations has the transactional behavior so they are visible to public after committing the current transaction. However, using autonomous transaction model or non-persistent

queues, the effect of those operations can immediately take place. Also messages in a transaction, which are created in the same queue, may be grouped and then consumed as a unit.

Messages consist of properties and data. Correlation, priority, recipients, and identity are the main attributes of message properties. Where, the correlation is an optional user defined value which logically classifies messages into sub-categories. In contrast, structured or non-structured *payload* represents the message data. Recipients can be determined on queue or message level. A recipients-list may be constructed for each produced message while that message is being enqueued. On the other hand, queue level recipients, or queue subscribers are dynamically added, removed and altered as a default target of queued messages. A subscriber mainly is identified by a name and may be attached with a rule. A rule is a logical expression that represents the subscriber's interest in messages, and takes the power of where-clause conditions of SQL statement. Oracle rule engine matches all available rules on a given queue with the unprocessed published messages. All candidate messages are emitted to their subscribers using one of the following models:

- For non-persistent queues: messages content is dispatched using asynchronous notification mechanism. A callback function which already has been registered using OCI API (Oracle native calls) is asynchronously invoked to receive new available messages that match the subscription criteria.
- For persistent queues: the message properties is received using asynchronous notification mechanism, while the message content is retrieved using explicit *dequeue* operation.
- Extracting messages using one of appropriate pull models. This is accomplished, either using continuously polling model through *non-blocked dequeue* operation. Or

using *blocked dequeue* or *listen* operation. *Blocked dequeue* operation saves CPU and network resource because the current process remains blocked till a new message has become available. With the same manner, *Listen* operation has designed to listen on one or more queues in parallel, and returns the queue handle on which a new message is being available.

Database events (or triggers) represent the main significant source for publishing information. Using Oracle AQ rule-based subscription instead of callout procedures, enables notification to be accomplished in reliable and recoverable publish/subscribe model. Note that Oracle AQ interfaces are implemented in PL/SQL (Oracle procedural programming language), C/C++ precompiler, OCI API, and Java language.

3.5 Summary

The Internet computing has become quietly matured for developing and deploying full behavioral systems, transactional business solutions, and mission-critical applications. This is due to bringing the best of interoperable, reliable, and scalable network computing models to the Internet environment, such as, client/server, publish/subscribe, and distributed object technology. Those architectures increase the chances for building complex and effective reasoning systems over the Internet.

Currently, the research and the development of the database management systems (as in Oracle) have been focused on the Internet computing technologies. They have to support the best of client/server, publish/subscribe models. For interoperability, their services' interfaces, also, must obey one or more open standard distributed object technology.

Chapter 4

Extending the MCP for the Internet

4.1 Introduction

In this chapter, we propose a model for coupling reasoning systems and databases on the Internet based on the Monitor Coupling Paradigm (MCP). The proposed system, that we call *i*MCP (where, *i* denotes the Internet computing), is designed over a three-tier network computing architecture to increase the overall scalability and efficiency.

The MCP uses rules in both database and reasoning system, and embodies a TMS within the reasoning system framework to efficiently identify defected inferences, and revise system beliefs accordingly. The MCP maintains data consistency and to some extent distributes functionality in reasonable way between the loosely coupled systems. As such the system satisfies the strong demands for mission-critical Internet applications. Developing and deploying of Internet applications – which are highly distributed, autonomous and heterogeneous, require careful usage of open reliable standards to increase their availability, scalability, portability, and interoperability.

The *i*MCP is characterized by using asynchronous communication model whenever that is possible. Building asynchronous loosely integrated interfaces minimizes systems' dependencies and maximizes parallel utilization of local and remote resources.

This chapter presents the *i*MCP in details and the alternative architectures that might be considered. The chapter is organized as follows. Section 4.2 mentions the limitations that faced the original MCP for the Internet environment. Section 4.3 describes a general view of the proposed paradigm. In section 4.4, the knowledge and the logic of ATMS-based reasoning system are presented. Section 4.5 proposes the

architecture of the middle-tier application server in handling communications and service issues. Finally, the database notification mechanism is discussed in section 4.6.

4.2 Limitations of the original MCP for the Internet environment

The original MCP (Hindi, 1994) was developed to integrate an active database system with a TMS-based reasoning system. It was also restricted by the listen-notify mechanism of the POSTGRES (the active DBMS for which it was developed). Therefore, it is not entirely suitable for the Internet environment, due to the following reasons:

1. The original MCP generates the database retrieve commands and active rules in a way that makes the database system performs as many as possible of the select and join operations, that would otherwise be performed by the reasoning system. This is because the database system is more efficient in doing these operations. However, in the Internet environment, that makes the approach losses its scalability (i.e., the ability to efficiently manage a large number of concurrent connections). Recall that the process of generating rules, and retrieve commands must run once and before reasoning starts. But in many applications, new available constraint parameters are bounded at runtime for each problem instance, e.g., student number in an electronic registration system, and source and destination in a flight-route planning system. These parameters (or variables) must be recompiled in the database subnets (i.e., considered when the Rete is constructed) to restrict the retrieved data and the affect of the monitoring rules. Moreover, regenerating these rules for each user (or run) must be followed by creating private-secondary relations that are listening only for the relevant modifications according to their corresponding rules. Therefore, rules and

secondary relations are only being persistent, but not actually shared. Discarding those instance parameters, and dealing with one general copy of retrieve commands will affect the reasoning process performance due to extracting unnecessary data. Also, for heavily concurrent connections, dealing with many rules (each connection has its own set of rules) will degrade the database system performance.

2. In a highly distributed, heterogeneous, and autonomous environment, it is hard to determine the subnets that maximize the benefits of database select and join operations.

In addition, the POSTGRES alert mechanism in the MCP, suffers from the following drawbacks:

1. Using two secondary relations (or even one secondary relation) for each subnet in the problem instance could be considered as a system overhead.
2. Traditional secondary relations do not directly maintain the sequence of the modifications in the temporal order in which they occur. e.g., insert, delete, insert on a specific record, probably, transmitted as insert, insert, delete.
3. Listen and notify commands in POSTGRES are not fully compliant to the push technology (see section 3.2.2). The reasoning system has to periodically check for possible notifications.

4.3 The general architecture of the *i*MCP

In this section, we modify the original MCP for the Internet environment in a way that avoids the above drawbacks. We call the resulting paradigm, *i*MCP (Internet Monitor Coupling Paradigm). The *i*MCP is distributed among three-tier network computing architecture (see figure 4.1). The front-end tier is a Java-enabled Internet

browser used for easy development and deployment. The middle-tier agent is an application server that is assumed to be connected 24-hour a day to the public Internet. It also populates the TMS-based reasoning system. It provides open network agent services (e.g., Java Dynamic Management Agent) and interoperable database programming interface (e.g., Oracle / JDBC & AQ API). The back-end tier is a data server that supports both client/server and publish/subscribe network processing models (e.g., Oracle DBMS). The browser downloads the TMS-based reasoning system, as a Java *applet* program, using the HTTP protocol. The flown reasoning system runs on the browser's Java VM. It requests values for a set of initial parameters, such as student number in registration system, and then generates the required data retrieving commands. The system sends these queries to the middle-tier, asynchronously. The middle-tier, in turn, orders the database system to execute each query. The retrieved results are transmitted via the middle-tier agent to the reasoning system. Later on, any modification on the retrieved data (i.e., delete, update or insert relevant data in the database) is routed from the data server through the middle-tier to relevant clients (reasoning systems) using a set of active rules running over both database and middle-tier agents.

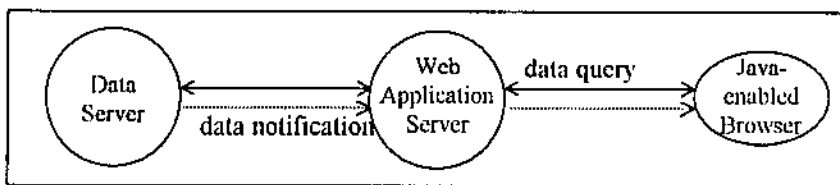


Figure 4-1: iMCP basic architecture

In general, during the execution of a client retrieving command, extracting subset of a database relation could contain repeated similar rows, since not all attributes of that relation's identifier may be selected. Consequently, deleting a row from the database does not necessary mean that the selected columns of that row are also deleted. Another copy of those non-unique selected columns may still exist in the database. To avoid emitting non-accurate alerts (modifications), the number of identical copies (or clones) of

each selected row must be computed with the retrieved data. When a row is deleted from the database, the reasoning system decrements the number of its clones by one. The reasoning system considers each database row with no clones as a deleted row. The ATMS treats the deleted rows as *nogood* environments, and therefore, all the beliefs that depend on their existence must be revised.

4.3.1 Production rules in the *iMCP*

Unlike the original MCP, in the *iMCP*, each active rule and retrieve command is based on a singleton predicate. The database system is only used to retrieve relevant data and alert the reasoning system for any new available modifications. For simplicity and scalability, the system tries to identify and restrict the amount of retrieved data without taking use of the optimized database join operation. The join is performed on the client side on the retrieved remote data only with reasoning meta-knowledge assistance, such as, a list of *nogood* environments. This is because, single-predicate selection criteria (no join) allows us to generate static general active rules that serve all problem instances. Consequently, this decreases systems' dependencies and increases the database scalability for concurrent handling of many clients' requests. Moreover, in distributed and autonomous database systems, complexity that may be raised in identifying subnets as in (Hindi, 1994), can be directly resolved if each subnet just contains a single remote predicate.

Production Rules in the *iMCP* are distributed among the three tiers as follows:

- In the database system, two kinds of rules are registered:
 - * Active – *Publishing* rules: are database triggers that are statically created for a particular database relation. During database operations on that relation, these rules are automatically activated to publish any relevant (to any client) data changes, as messages, using predefined transactional queues.

- * *Subscribing* rules: are dynamically assigned, by the middle-tier to automatically consume any relevant available queued messages. These queued messages are the relevant data changes, which were previously published due to firing *Publishing* rules. Using a database rule engine, available messages in a particular transactional queue are evaluated against the conditions of attached *Subscribing* rules. Matched messages are then dispatched to their subscribers (middle-tier agents).
- The middle-tier, in turn, maintains another type of rules, called *Selection* rules. They are used to determine where to send the retrieved data (i.e., to which reasoning system). These rules are implicitly registered to reflect the selection criteria for each connected client. Clients (reasoning systems) ask the middle-tier agent to run a set of selection query commands and send back the results. The middle-tier sends the selection query to the connected database to be evaluated there. In addition, a copy of each selection criteria is registered in the middle-tier as a *Selection* rule. Each *Selection* rule keeps a list of the clients that owns that rule. When the data arrive from the database queue(s) to the middle-tier using *Subscribing* rules, they are further dispatched to the corresponding clients using *Selection* rules
- The last rule type that is managed by the client, are the *Problem-Solving* rules that correspond to the expert system domain knowledge.

In the *iMCP*, for simplicity and modularity, the condition filters of *Publishing* and *Subscribing* rules are too loose and general. When at least one *Subscribing* rule is available on a relation, any modification on that relation is published as a message to the corresponding transactional queue(s). At transaction commit point, all these queued messages, are automatically dispatched to their subscribers or middle-tier agents.

Therefore, some of the arrived changes from the database possibly can be discarded by the middle-tier agent, because they are irrelevant to the available *Selection* rules. However, this messaging traffic problem, between the database server and the

middle-tier, may be substantially limited using message grouping option. The database system at the end of each transaction groups all messages in the same queue to be consumed as a unit with a single networked message. Moreover, in most cases, the data transferred between the data server and the middle-tier agent usually travel a short, fixed distance.

Another possible solution could use the compound content-based *Subscribing* rules. In this case, a *Subscribing* rule would not only restrict the relation type to retrieve the queued messages, but would ensure that the values of the message attributes are of interest to at least one client (reasoning system). In another words, *Subscribing* rule on a relation is a union of all clients' *Selection* rules. However, this approach to be effective, would require additional optimization step for test reduction. Moreover, deal with attributes' content, the system has to use the structured messages. That means, the modification messages of a certain relation would have to be queued in a queue table, which has the structure of the original relation. In spite of its static nature, this requires creation of different queues and queue tables with the number of the monitored relations. However, due to the performance issue and to save the network bandwidth, it is suggested to implement the architecture of the system messaging using the previous architecture. The transmitted message should be consumed by at least one available subscriber, so that no messages would be discarded.

4.3.2 A three-tier/two-tier configuration approach

Based on the content-based *Subscribing* rules, an alternative possible approach for the *iMCP* can be constructed, which is a tree-tier/two-tier architecture as shown in 4-2.

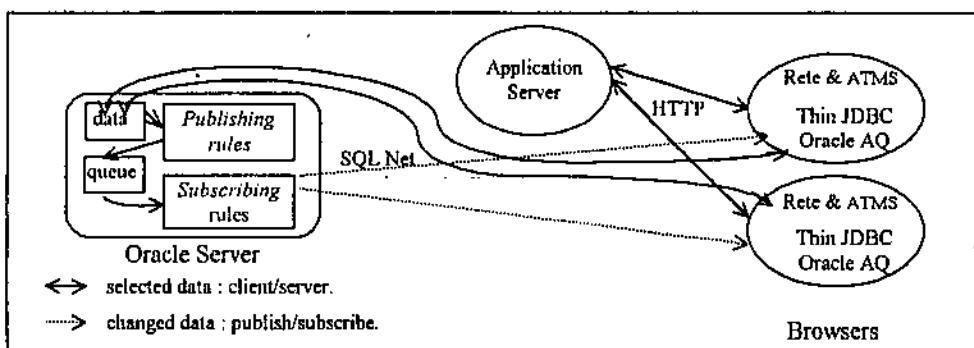


Figure 4-2: 3-tier/2-tier configuration

In this approach, transferred data need not pass through a middle-tier. Instead, it is transferred directly from data server to the browser that runs the reasoning system. The communication mechanism in the three-tier/two-tier configuration is described as follows. Using the HTTP communication protocol, the reasoning system, a thin JDBC driver, and an Oracle AQ API are downloaded to the client site from the middle-tier where they are stored. A direct connection is established between the client and the database server using JDBC APIs. Relevant data are retrieved through a simple implementation of TCP/IP Oracle transport layer (i.e., SQL*Net protocol). The client adds a *Subscriber* rule for each remote relation, which reflects its selection. *Selection* rules are themselves the *Subscribing* rules. Any modification on the database is published using database triggers to be dispatched directly to the connected clients when that modification matches clients' *Subscribing* rules. However, in spite of its simplicity, the configuration suffers from the following drawbacks:

- The system deployment wastes time and network resource. Before the system run, it has to download expert system logic and knowledge, Oracle Thin JDBC (around 150K in a compressed form), and Oracle AQ/Java API.
- The system lacks high scalability. To handle many concurrent connections, the system has to multiplex set of clients' connections into a single physical database connection. Using additional services, connection multiplexing in Oracle can be established either by using Oracle Connection Manager, or by managing pool of connections using Oracle Multi-Threaded Server's dispatcher (Oracle, 1997, b).
- As mentioned before, content-based subscribing on relation's attributes requires a creation of a queue and a queue table for each relation. However, this job statically takes place, which is independent from the clients' connections and subscribing rules.

- Currently, Oracle AQ/Java API does not support the structured message queuing (Oracle, 1999, a). Furthermore, programs to be downloaded from web servers and run on web browsers, they have to be written in Java. Till now, Oracle AQ APIs that support structured messaging are available only in C/C++ and PL/SQL interfaces.

Therefore, to easily develop and deploy the *iMCP* in a highly scalable environment, it has to be based on three-tier network architecture. The data server publishes data changes as non-structured messages in a set of predetermined relations. Using relation type *Publishing* and *Subscribing* rules and message grouping optimization, the middle-tier automatically receives relevant relations' modifications. Received modifications are routed to their clients using available *Selection* rules. In the client site, the ATMS responsibility is to receive modifications and to directly revise all relevant beliefs accordingly.

4.3.3 Main characteristics of the three-tier *iMCP*

The three-tier *iMCP* architecture, which we adopt, provides a high scalability, efficiency, interoperability, manageability, and flexibility. As shown in figure 4-3, the communication of the architecture can take many variations. A single database server may be attached to a single application server (middle-tier agent), which has the capability to serve many connected clients. A single database server may be connected to more than one application server to increase the systems' scalability and efficiency. A single application server can integrate one or more autonomous database servers using an open standard interface (e.g., JDBC). A single thin client has the capability to easily extract the data from multiple application servers, which use the same network agent architecture.

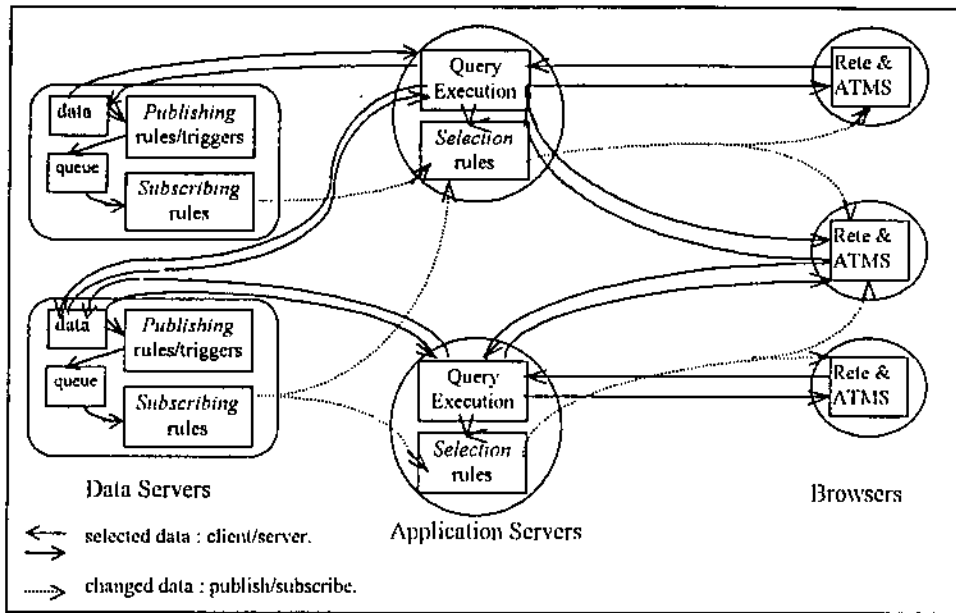


Figure 4-3: Multi-type communication architectures for 3-tier

Building the system using Monitor Coupling Paradigm (MCP) that is considered as a variation of loose coupling approaches, provides powerful reasoning capabilities that can only be limited by the language of which expert system is written. Unlike deductive database systems, no control restrictions are defined on the reasoning process to prevent generating unexpected massive data or performing non-polynomial computations on the data server. For the same goal, in three-tier architecture, reasoning that are being performed using the data and the middle servers have to be quiet limited to increase system scalability and efficiency. In contrast, most of heavily intelligent computations have to be processed on the client resource. Therefore, the electronic reasoning architecture is distinguished from other electronic technologies through using thin software deployment with fat computations.

4.4 The logic and knowledge structure of the ATMS-based ES

Hindi (1994) has mentioned two main properties for the reasoning system to be suitable for the MCP. The system has to be able to revise its beliefs when relevant data in

the database is changed or new data is inserted. Also, the system has to use incremental forward chaining control procedure to easily reexamine search paths of modified data and take into consideration any available new data. Where in backward chaining procedure, it is hard to identify or reexamine paths that are affected by modified data. Therefore, the most candidate suitable architecture for the proposed reasoning system is the ATMS tightly coupled Rete-based production system. In addition to the architecture's efficiency, the ATMS and the production system are both incremental in nature. Moreover, the TMSs have been designed originally, to provide a capability for the problem solver to maintain the truth of its beliefs in a changing environment.

In the iMCP, the reasoning system is defined in a structure that is based on PROLOG-like and Morgue-like systems. The syntax is based on PROLOG, while, the reasoning semantic is an ATMS-based forward chaining system, similar to Morgue's system (Morgue and Chehire, 1991). PROLOG's syntax can be directly and easily understood for non-procedural problem solving.

This section consists of three main subsections. Section 4.4.1 describes the internal and external system knowledge representation. Section 4.4.2 discusses the inference engine design that tightly couples Rete and ATMS algorithms. In 4.4.3, the mechanism for generating the retrieve database commands is presented.

4.4.1 Knowledge representation

We represent the design of the system knowledge base in terms of its external and internal representation. The external representation corresponds for the external knowledge format that needed by the problem solver. While, the internal representation describes the internal structure of the compiled knowledge that is constructed by the rule interpreter.

Moreover, the design of the external representation is consistent with the relational data model. In contrast, without contrary, the internal system representation is described using the OMT (Object Modeling Technique) as an object-oriented data modeling.

4.4.1.1 External knowledge representation design

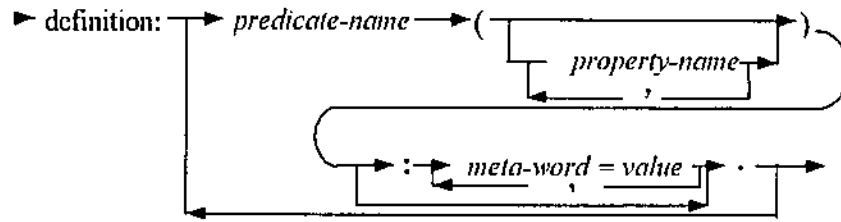
The external knowledge representation of the system is divided into two main sections. A *definition* (or header) section that lists the predicates' structure as a prototype, and a *knowledge* (or body) section, which consists of the general knowledge (or rules) and case specific data (or facts).

The system deals with two types of predicates: non-database (or local) predicates and database (or remote) predicates. The database predicate (or relation) is identified by an address of the middle server, which is responsible for retrieving the actual data from the corresponding database, and an optional condition filter that is applied as an additional variable selection constraint. The intersection result of that filter with the union of all rules' conditions that belong to that predicate reflects the actual system selection criteria on that predicate.

Moreover, the system distinguishes between two different production rule types: ordinary and contradictory rules. Ordinary rules are the normal rules which infer (or used to infer) new data. While, contradictory rules (or ATMS consumers), that have higher execution precedence than the ordinary rules, generate the inconsistent (*nogood*) environments in an early processing stage to save system resources. Priorities for firing ordinary matched rules may be explicitly managed using a special *priority* predicate and implicitly using the FIFO control procedure.

Using syntax diagram, a general format of the *Definition* section is described below:

General format:



Example:

definition:

```

student(stdNo, subjectNo, planYear):
    type=assumption,
    address="middle_tier_addr",
    filter= stdNo == Student_No,
           stdStatus != 0 .

sheetPlan(subjectNo, planYear, grpNo, crsNo, weight,
           prerequisiteCrsNo1, prerequisiteCrsNo2, parallelInd):
    type=promise,
    address="middle_tier_addr",
    filter= subjectNo == Subject_No,
           planYear == Plan_Year.

class(crsNo, classNo, dayFE, dayCat, fromHr, toHr):
    type=assumption,
    address="middle_tier_addr".

registeredCourse(stdNo, crsNo, grpNo, weight):
    type=promise,
    address="middle_tier_addr",
    filter= stdNo == Student_No.

classTryReg(stdNo, crsNo, grpNo, weight, classNo,
            dayOfFinalExam, dayCategory, fromHour, toHour,
            registerType): type=assumption.

alreadyRegistered ( ).

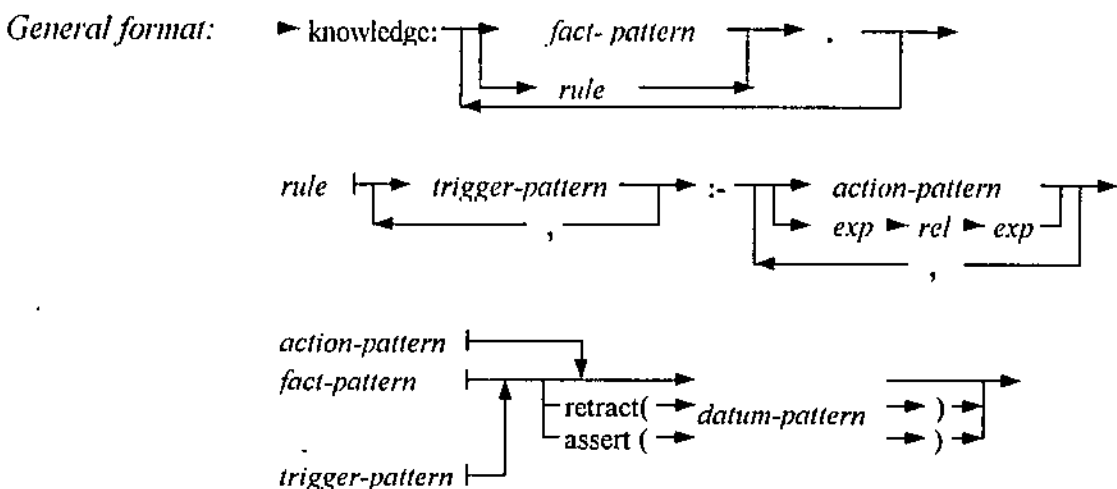
```

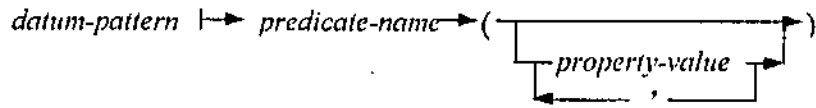
As can be noticed from the above example, the names of system predicates and properties start with small letters. While, variables start with capital letters (e.g., *Student_No*). Values for these definition variables are passed as arguments to the system when the reasoning process starts. Predicates with no properties correspond to contradictory rules (e.g. *alreadyRegistered*). *Type*, *address*, and *filter* are meta-words, which identify the properties of ordinary predicates. The *type* determines the ATMS node type of the existing or retrieved data. *Promise* and *assumption* are the only possible values for this keyword. Usually, *promise* type companions the historical and read-only

database relations, or non-database facts, while, *assumption* represents transactional and modifiable relations. As will be illustrated later, a *derived* type is automatically assigned at runtime for inferred data. The *address* keyword is only determined for remote database relations and refers to the middle-tier agent name. The optional *filter* clause may be used for the following purposes:

- Identifying extra common constraints on the extracted data, instead of rewriting the same conditions whenever that relation is used.
- Increasing reasoning efficiency, scalability, and generality. This is due to using unbounded variables that are substituted from the external environment without affecting the logic of the written rules, i.e., different problem instances can be served using the same general knowledge interface. Consequently, this gives the opportunity for compiling the knowledge base into a Rete network in advance for all clients.
- Providing the capability for imposing constraints (conditions) for non-selected columns (attributes) (as in, "*stdStatus* \neq 0" in *student* relation). In general, middle tier-server, may be far away from the client browser. Restricting the retrieved columns for only the needed ones could save the network bandwidth.

The *knowledge* section format is described using syntax diagrams as follows:





Example:

knowledge:

```
# note : the following facts correspond to database relations
#       and therefore, they have to exist inside the
#       database not in the knowledge section.
```

```
student(960001, 306, 1992).
```

```
registeredCourse(960001, 101100, 1, 3).
```

```
sheetPlan(306, 1992, 3, 302111, 1, 302101, 0, 1).
```

```
sheetPlan(306, 1992, 4, 306432, 3, 306331, 306325, 0).
```

```
class(301101, 1, 24, 135, 8.00, 9.00).
```

```
class(301101, 2, 24, 135, 8.00, 9.00).
```

```
class(302111, 1, 26, 135, 10.00, 11.00).
```

```
classTryReg(StdNo, CrsNo, GrpNo, Weight, ClassNo, DayFE,
             DayCat, FromHr, ToHr, 1):-
```

```
student(StdNo, SubjectNo, PlanYear).
```

```
sheetPlan(SubjectNo, PlanYear, GrpNo, CrsNo, Weight, 0, 0,
           ParallelInd),
```

```
class(CrsNo, ClassNo, DayFE, DayCat, FromHr, ToHr),
FromHr > 9.0.
```

```
classTryReg(StdNo, CrsNo, GrpNo, Weight, ClassNo, DayFE,
             DayCat, FromHr, ToHr, 1):-
```

```
student(StdNo, SubjectNo, PlanYear).
```

```
sheetPlan(SubjectNo, PlanYear, GrpNo, CrsNo, Weight,
           PreCrsNo, 0, ParallelInd); PreCrsNo > 0,
```

```
registeredCourse(StdNo, PreCrsNo, GrpNo2, Weight2),
```

```
class(CrsNo, ClassNo, DayFE, DayCat, FromHr, ToHr).
```

```
alreadyRegistered( ):-
```

```
classTryReg(StdNo, CrsNo, GrpNo1, Weight1,
```

```
ClassNo1, DayFE1, DayCat1, FromHr1, ToHr1, Type1),
Weight1 > 0,
```

```
registeredCourse(StdNo, CrsNo, GrpNo2, Weight2).
```

The system knowledge base is composed of facts and rules. As a common object type for both, the *datum* is a smallest chunk of data (or knowledge). In a datum, the *property-value* list consists of constants if that datum represents a *fact*. While it might contain variables (start with a capital letter), if it located as a constituent in a rule

(i.e., either in *action* or *trigger* part). Unlike PROLOG, the rule clause might be represented in a clause form not in a horn clause form.

4.4.1.2 Internal knowledge representation design

The proposed reasoning system relies on set of basic data objects that are used to represent both the internal knowledge structure and the system inference engine architecture. Visually, figure 4-4 below presents a class hierarchy of the basic data types and their relationships with each other.

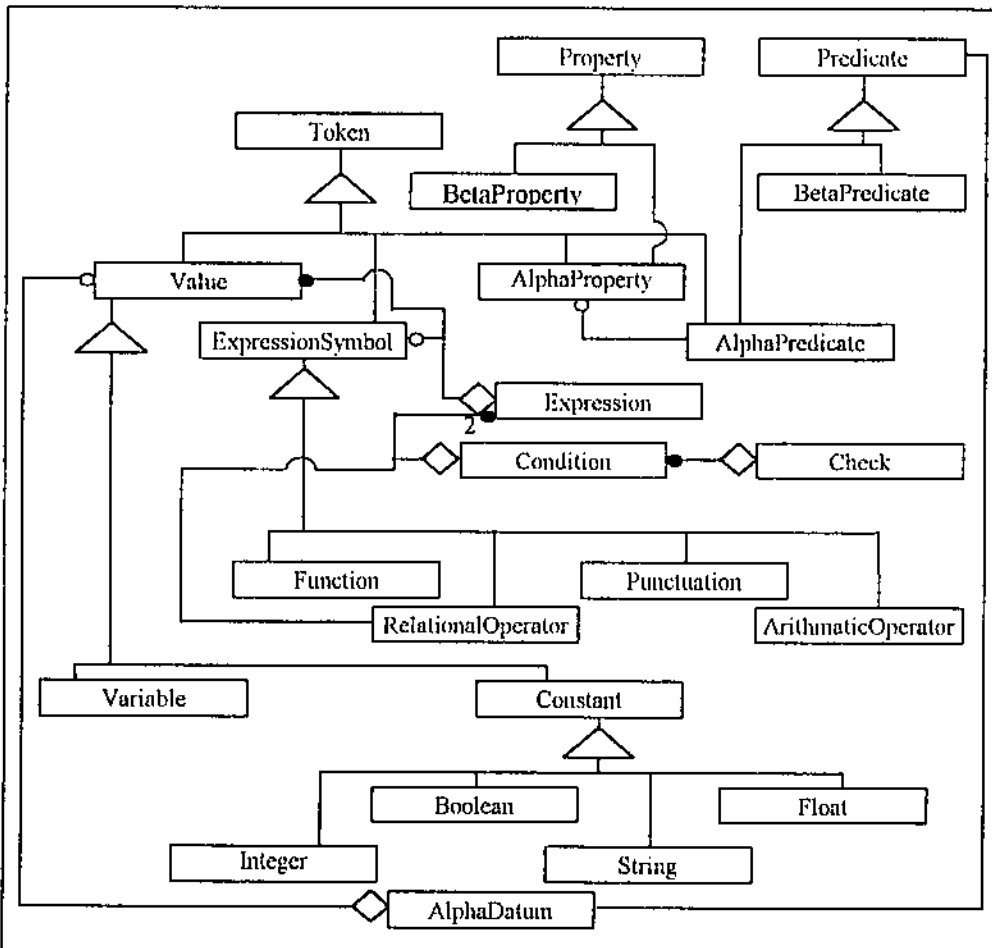


Figure 4-4: OMT diagram represents the basic data types of the rule interpreter

The rule interpreter parses the knowledge tokens into predicates, property names, property values, and expression symbols. The *Value* of a property could be either a *Constant* with *Integer*, *Float*, *String* or *Boolean* data type, or *Variable* with bounded domain. *Values*, *Functions*, *Braces*, and *Arithmetic operators* compose simple or nested

Expressions. Two *Expressions* with an appropriate relational operator in the middle form a *Condition*. A list of interrelated *Conditions* is called a *Check/Test* object.

In the internal knowledge design, the basic data structures are classified into two main types: *Alpha* and *Beta* objects. An *Alpha* structure denotes a singleton (or simple) object component. In contrast, a *Beta* structure refers to a joined (or complex) object data type.

Following the above convention, an *AlphaPredicate* is a simple object type that is characterized by a set of atomic entities called *AlphaProperties*, which reflect its internal structure. As mentioned before, an instance of that predicate is called an *AlphaDatum*. An *AlphaDatum* object is the common pattern to compose production rules (i.e., *AlphaTrigger/Action* objects) and specific case data (or *AlphaFacts*). Joining *AlphaFacts* for partial results in the Rete match algorithm composes *BetaFact* objects. *Alpha/BetaFacts* of the same predicate are located in a hash memory table called *Alpha/BetaData* object.

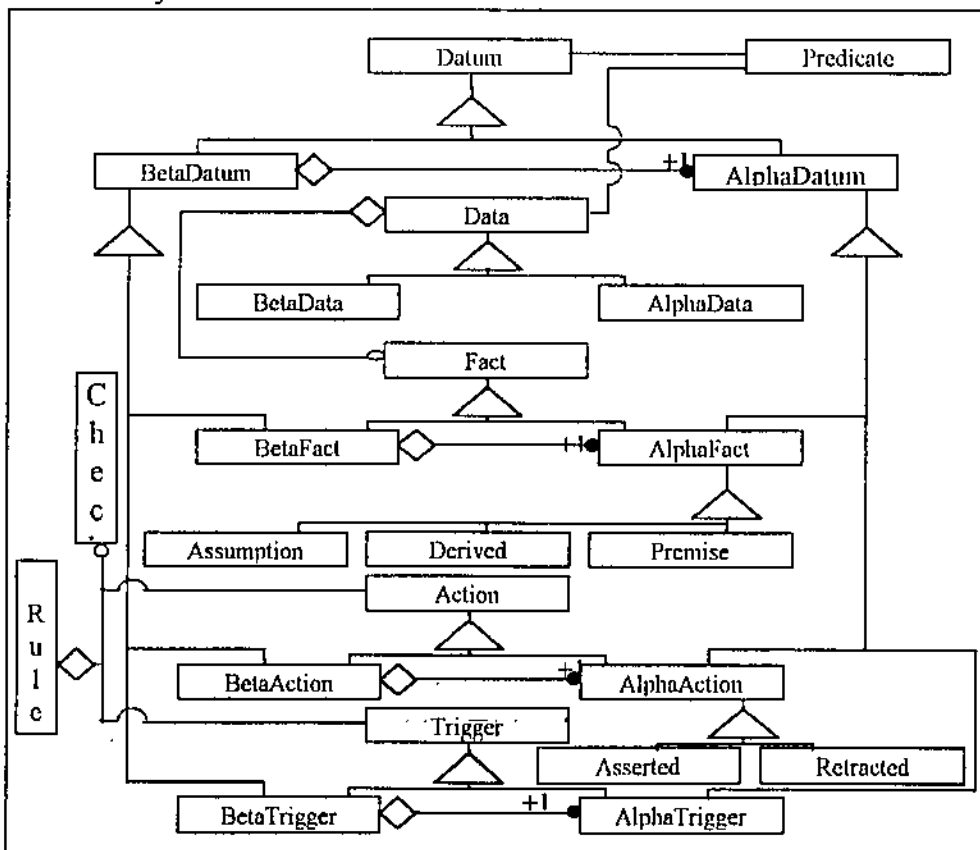


Figure 4-5: Alpha/Beta objects

Figure 4-5 above, summarizes *Alpha/Beta* objects that can be used to construct facts and complete production rules.

Consider, for example, $\text{trip}(\text{"Amman"}, \text{"Zarka"})$ as an *AlphaFact* that corresponds to a *trip AlphaPredicate*, which is located in *trip's AlphaData*. On the other hand, $\text{trip}(\text{"Amman"}, \text{"Zarka"}) \wedge \text{salesman}(\text{"Moh'd"}, \text{"Amman"})$ is a *BetaFact* that belongs to $\text{trip} \wedge \text{salesman}$ *BetaPredicate*, which is located in the *BetaData* of that predicate. While, $\text{trip}(\text{From}, \text{"Zarka"})$ is considered as an *AlphaTrigger* or *AlphaAction* if it corresponds to a premise or an action part, respectively. In the same manner, $\text{trip}(\text{"from"}, X) \wedge \text{salesman}(\text{"Moh"}, X)$ can be classified.

4.4.2 The inference engine

The *MCP* uses a Morgue-like approach for tightly coupling a forward-chaining Rete-based production rule system with the ATMS. The production system is considered a problem solver that generates intelligent inferences in the form of justifications. Justifications are passed to the ATMS labeling algorithm to maintain their truth and dependencies. In the tight coupling approach between the two modules, the Rete network of the production system interferes with the dependency network of the ATMS, and ATMS nodes are part of Rete data facts. This coupling is illustrated in figure 4-6 below. Where, normal lines represent Rete links between Rete nodes, and dotted lines correspond for links in the dependency network between ATMS nodes.

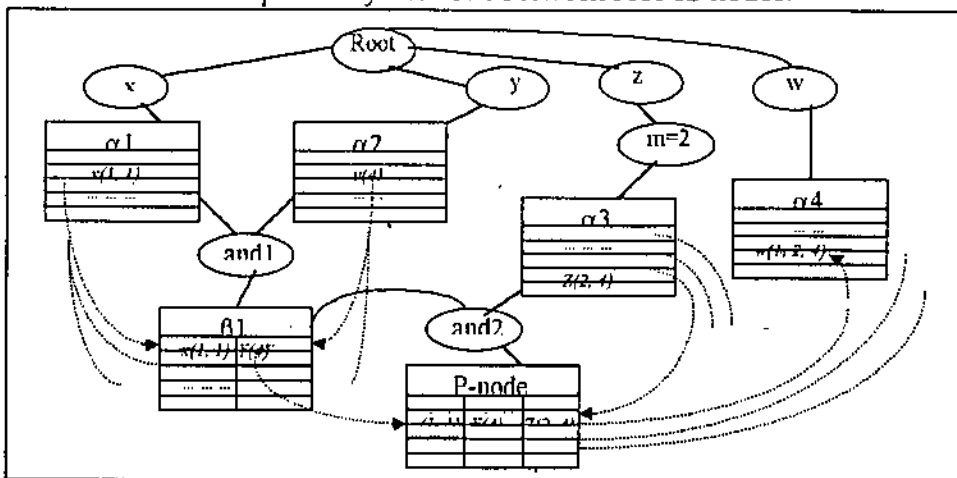


Figure 4-6: hypothetical example for coupling the Rete and DN of ATMS

4.4.2.1 The ATMS design

The ATMS maintains the truth of system beliefs and keeps them up to date to reflect current reasoning status when some assumptions are modified. Keeping ATMS nodes and links in the dependency network consistent with a minimum time and effort, is the goal of the ATMS labeling algorithm.

During ATMS labeling algorithm, type of nodes (i.e., premise, assumption, derived or assumed) might be automatically converted from one to another. However, for efficiency and simplicity reasons, it is preferable to avoid such situation by specifying a specific node type with each datum during datum lifetime. The system does not allow justification for assumptions and premises. Assumed nodes are considered as a part of derived nodes and are not recognized as a separate type. Assumed nodes in the *iMCP* do not have a significant useful usage, since no join operation is performed by the database system. The original MCP has used the assumed node to initially label the retrieved joined tuples from the database by its constituent tuples. If any of its constituents discovered as a *nogood* environment, then directly the system recognizes that joined retrieved tuple is a *nogood* environment.

Preventing dynamic swapping from one type to another, and assigning a static ATMS node type for each fact at its creation time, permits the type of an ATMS node to be considered, as a type of the tuple (fact) that is attached to that node. This is cleared from figure 4.5, where, *AlphaFact* object is classified into *Assumption*, *Premise*, or *Derived* object. In figure 4.7 below, the ATMS architecture of the *iMCP* is described.

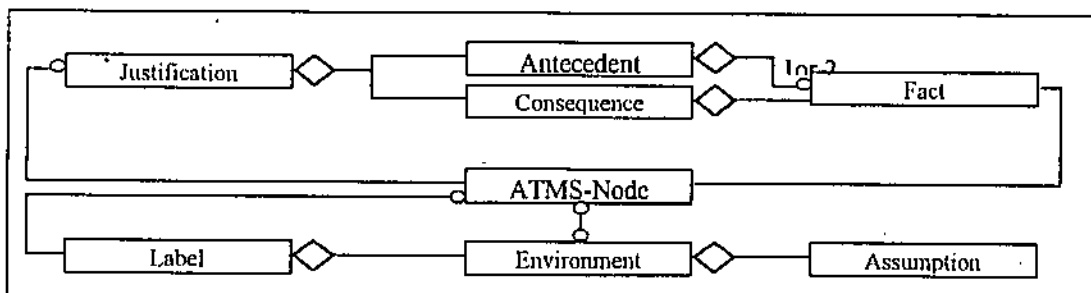


Figure 4-7: The ATMS design

Justifications (or inferences) are constructed in a form of *Consequence and Antecedents* facts. An ATMS node is a part of the *Fact* object (that is, *BetaFact*, *Assumption*, *Derived*, or *Premise* object). Moreover, the ATMS node has a *Label* and a set of *Justifications* that support that node. A *Label* is a set of *Environments* in which the ATMS node holds. Where, an *Environment* is a bit-vector of consistent *Assumptions*. It records all nodes in the label of which it appears and implements two main set operation methods: *union* and *subset test*, which are the basic operations of the ATMS labeling algorithm.

4.4.2.2 The Rete-based production system design

As discussed early in section 2.2.2, the Rete algorithm is used to optimize the match step of recognize-act-cycle by *saving* the matching results of the previous cycle to be available for the next one. Also common parts of condition elements are utilized (*shared*) to reduce the number of match test operations and minimize the cost of the labeling algorithm in Morgue-like tight coupling approaches.

The design of the Rete algorithm for the *iMCP* has concentrated on increasing the *sharing* benefits, especially during ATMS labeling algorithm.

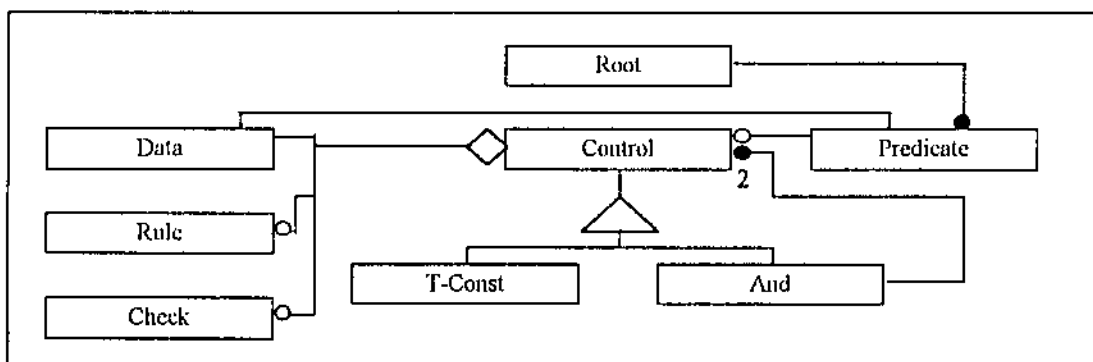


Figure 4-8: The Rete architecture

From figure 4-8, the basic data structure of the Rete implementation is the *Control* node that might be a *T-Const* node (corresponds for a simple data selection) or an *And* node (refers to a complex selection criteria). An *And* node has two input *Control* nodes.

The *Control* node, in turn, consists of *Data*, list of *Rules*, set on *NextAndNodes*, *Check*, and *Predicate* object. Therefore, the external structure of the Rete network consists only of interrelated *Control* nodes. The *Root* node is implemented as a hash memory table, which contains all system *Alpha/BetaPredicates*.

The *Data* for a *T-Const* node is an *AlphaData* object type with an *AlphaPredicate* type. On the other hand, the *Data* and the *Predicate* of an *And* node are of type *Beta* objects. As described, the *Predicate* structure also has a *Data* object, which is considered as a knowledge reference domain for any related *Control Data*. In other words, each fact in the *Data* object of any *Control* node must refer to an original reference in its *Predicate Data* object. Consequently, ATMS labels, which are part of fact objects, are shared for all *Control* nodes that have the same *Predicate* structure. The list of these *Control* nodes is maintained in the fact's *Predicate*. This list is also used in identifying all memory nodes in which a fact is stored, instead of recording a private subset list with each fact. That is, maintaining one main memory reference for each *Predicate* in Morgue-like tight coupling approaches increases the sharing benefits for labeling algorithm, reduces the overall system space, and improves searching algorithms to track facts in the Rete network.

4.4.3 Generating the retrieved commands

In the *iMCP*, client's browser downloads the reasoning system using HTTP protocol from the application server (i.e., the middle-tier agent). The reasoning system requests from the user to provide it with values for some special variables, which are located in filter clauses of the reasoning system knowledge. The system builds the corresponding Rete network and loads all initial local data according to its general knowledge and without considering these filters. Therefore, it is possible to persistently store the rules of the system as a compiled Rete network object instead of a text file in

the application server. If there is at least one remote relation, a new thread (i.e., light process) is activated to handle network communications and remote data retrieving. The original process starts the reasoning with available unprocessed data buffer and collaborates with the communication thread in a consumer-producer synchronization model. When no data is available and all remote data queries are executed, the system can halt.

The communication thread clusters all remote relations according to their middle-tier addresses. A connection is established between the client and each of the mentioned address. To form a retrieve query command, the communication thread has to construct for each remote relation the corresponding query command as a *Selection* rule using the following steps:

- 1- For the corresponding *type-check* node in the Rete network, disjunct (using OR) the check condition of each of its *t-const* node with the others. As a simple optimization, if an α -memory node is directly connected with that *type-check* node, then the step returns *true*.
- 2- If a filter clause has being attached with that predicate, then conjunct (using AND) the query expression of the previous step with the substituted filter expression.
- 3- Associating with each query expression, its predicate name and the selected properties with their orders in a form of *SelectionRule* object (its structure is described in the next section).

Let us consider the example listed in section 4.3.1.1, figure 4-9 shows the corresponding Rete network.

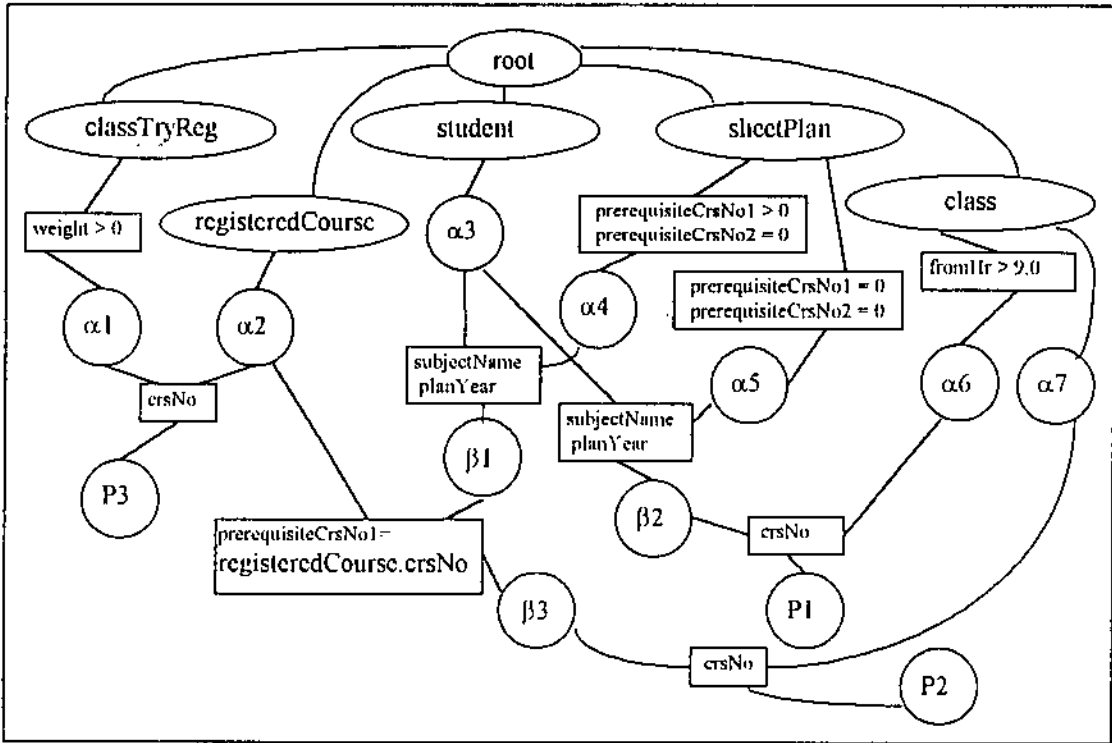


Figure 4-9: The Rete network for example in section 4.3.1.1

The system clusters all remote predicates in one group, since they all refer to one middle-tier agent. As described in figure 4-1, each remote predicate has the following query filter expression.

Where, *Student_No* = 960001, *Plan_Year* = 1992, and *Subject_No* = 306

Student	(stdNo = 960001 and stStatus != 0)
SheetPlan	(subjectNo = 306 and planYear = 1992) and ((prerequisiteCrsNo1 > 0 and prerequisiteCrsNo2 = 0) or (prerequisiteCrsNo1 = 0 and prerequisiteCrsNo2 = 0))
Class	
RegisteredCourse	(stdNo = 960001)

Table 4-1: Query filter expression for the example in section 4.3.1.1

It is worth mentioning that the constructed query expression must conform the structure of the query service of the used middle-tier agent, so that it can be easily understood and further processed. Figure 4.10 below describes the structure of the query expression of the Java Dynamic Management Agent, which we adopted in the implementation. Two new expression types are added to support binary/unary function expression and positional attribute (an indexed property), which are *FunctionValueExp* and *PositionalAttributeExp* respectively.

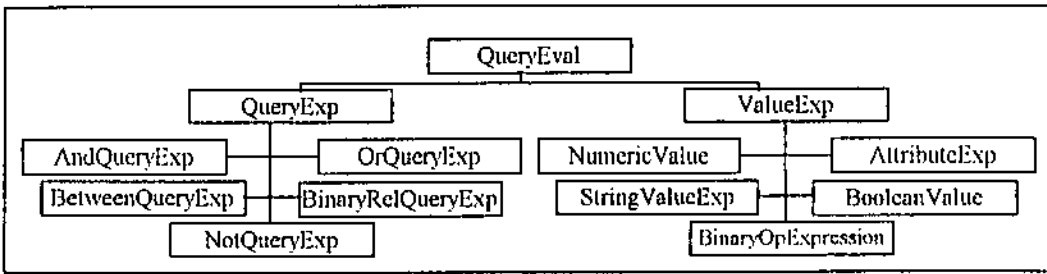


Figure 4-10: Query expression in Java Dynamic Management Agent

An asynchronous request to the middle-tier with the vector of constructed queries (i.e., *SelectionRules*) is issued. The communication thread registers itself as a subscriber (or active session) for any relevant incoming data to the middle-tier agent. Using the push model of the agent event service, the data that result from the query execution and the relevant modifications are automatically received while the reasoning procedure is running.

However, to save system resources and the network bandwidth, arrived data in the middle tier is buffered. The system periodically transmits the data to their subscribers (i.e., reasoning systems) in reliable message semantics. Each received element of the data has the following structure (visually presented in figure 4-11):

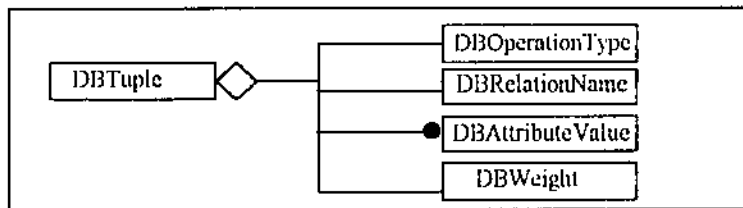


Figure 4-11: *DBTuple* structure

The data is received by the reasoning system in terms of a collection of *DBTuple* objects. The *DBTuple* is processed by the system as an assert or a retract command. The *DBTuple* object has an operation type, a relation name, ordered values, and a weight. The operation type can be either select, insert, or delete. Where the update operation is translated into delete followed by insert operation. The weight of *DBTuple* object in the select operation corresponds to the number of duplicates of each selected tuple from the database. On the other hand, for insert and delete operations, the weight equals one. This means that each selected tuple (*DBTuple*) from the database is mapped with its

duplicates into the corresponding *AlphaFact* object in the Rete network. Inserting a new clone of that tuple increment the number of cached duplicates by one. On the other hand, when that tuple is deleted from the database, the system decrements number of duplicates by one. Any cached fact in the Rete with no images (or clones) is treated by the reasoning system as a *nogood* environment.

Relying on exactly-once message semantics, the system counts the number of retrieved vectors with select operation to determine either to wait or to terminate the reasoning process when no new data is available for reasoning. The wait for new data only reasonable, if there are still queries to be evaluated.

4.5 Middle-tier processing role

In three-tier network computing architecture, the tasks of the middle-tier server includes:

- Maintaining the application logic into a single location for an easy development and deployment.
- Providing scalable, reliable, and interoperable open standard network services.
- Increasing applications availability and efficiency due to distributing systems' load and utilizing network resources.

In the *iMCP*, these functions of the middle-tier are utilized using both client/server and publish/subscribe message computing models. Evaluating database queries is the main client/server operation. On the other hand, receiving database notifications is automatically accomplished using publish/subscribe model.

The reasoning system asynchronously requests the middle-tier agent for evaluating a set of simple queries. The middle-tier agent responsibility is to tell the database system to execute the transmitted queries, statement by statement. Once the results are returned,

they are automatically propagated to their actual clients (i.e., reasoning systems). In addition, each query is considered as a *Selection* rule, which subscribes for the relevant arrived modifications to the middle-tier to immediately retransmitted to their original clients. Therefore, the middle-tier agent architecture should support:

- Open standard distributed object technology: to provide the mechanism for thin clients (i.e., browsers) to transparently communicate the middle-tier services in client/server and publish/subscribe communication models with interoperable and portable interfaces. The protocol used, also, must be supported by the browser as an open, reliable, and standard protocol, e.g., Java/RMI and CORBA/IIOP.
- Query Service: which formulates clients queries in predefined interoperable interface. That interface manages, analyzes, prepares, and executes database queries. Also, through the interface a simple rule engine can be constructed to assist what *Selection* rules do correspond to the current data modifications.
- Event Service: providing push-model or blocked pull-model event communication mechanism to receive the database modifications using the middle-tier *Subscribing* rules. The event service also enables the *iMCP* to reemit arrived modifications to their interested clients using the cached *Selection* rules.

A comprehensive architecture of these requirements could be implemented using either Java language or CORBA OMA architecture. The package of Java RMI, JDBC, Dynamic Management Agent, and Oracle AQ/Java API together includes the required functions. On the other hand, CORBA IIOP and ORP, CORBA Query and Event services, and Oracle AQ API in Java or C/C++ provide another effective alternate for constructing an infrastructure for the middle-tier network agent. Recall that, in the *iMCP* implementation, we have used the first network agent architecture (i.e., Java language) because of its simplicity and flexibility.

Using these cooperated services, two main components are built on the middle-tier agent. One for repository interface, which deals with database retrieving and notification mechanism. And the other is a rule engine manager, which receives the clients query commands, and passes the database notifications to their interested clients. In the following subsections, the architecture and behavioral processing of these components are abstractly described.

4.5.1 Middle-tier agent data structure

The internal architecture of the middle-tier agent is based on a set of elementary data structures, which are described in figure 4-12.

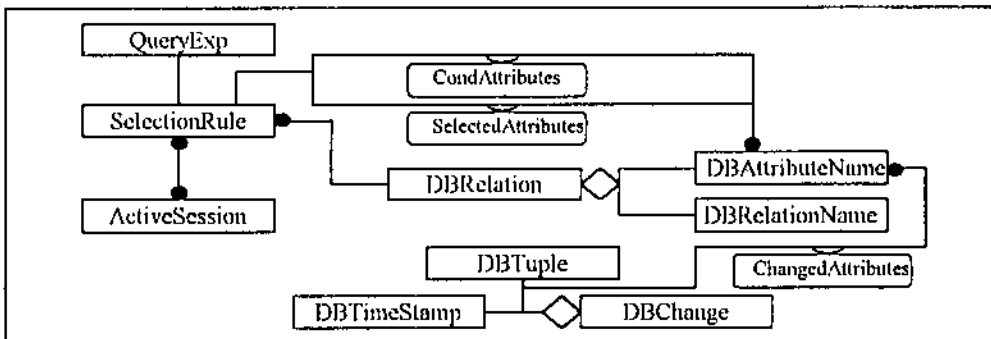


Figure 4-12: Middle-tier components' data structure

Connected clients are represented in the middle-tier as an *ActiveSession* object, which has been attached with a set of *SelectionRules*. Each *SelectionRule* belongs to a particular relation and has a query filter expression, selected attributes and conditional attributes (i.e., non-selected attributes, which is mentioned in the query filter expression). To validate the *SelectionRule*, the middle-tier caches meta-data (logical structure) for each used relation. The database changes are managed in the middle-tier in the form of *DBChange* entries. The *DBChange* consists of a time stamp, a tuple, and a list of changed attributes (for update operation). Where the time stamp is a serial number that informs the system on the temporal order of these changes. However, in Oracle DBMS, it can be directly generated from a sequence generator object. The sequence generator,

unlike tables, does not require any locking mechanism to guarantee its uniqueness and incremental growth behavior in multi-user environment.

Relying on these elements, the middle-tier components have the following main operations:

<i>Rule Engine Manager</i>	<ul style="list-style-type: none"> • createSession () return <i>Session</i> • dropSession (in <i>Session</i>) • doQueries (in <i>Session</i>, in <i>Vector_of_SelectionRules</i>) • dispatchRetrieved&ChangedData (in <i>Session</i>, in <i>Vector_of_DBTuples</i>)
<i>Repository Interface</i>	<ul style="list-style-type: none"> • executeQuery (in <i>SQL_TEXT</i>) return <i>Vector_of_DBTuples</i> • setActive (in <i>DBRelationName</i>) return <i>DBRelation</i> • setInactive (in <i>DBRelationName</i>) • receiveChanges (in <i>Vector_of_DBChange</i>)

4.5.2 Middle-tier processing mechanism

The architecture of the middle-tier agent consists of two main components. A rule engine that interacts with the client demands, and the repository interface that handles one or more database connections. The client asks the middle-tier rule engine to create an active session object that has the capability to asynchronously return the query execution and the data modifications. The middle-tier agent receives the client (or session) selection commands as a set of *SelectionRules*. It places them in a special job queue. Each job (i.e., session *SelectionRule*) is executed in the following ordered steps by a special dispatching thread:

- Using the middle-tier local buffers, it searches for a meta-data of the target relation, which that *SelectionRule* is based on (i.e., its attributes and their orders). If no entries available, the system calls the *setActive* function to return the required information and subscribe the database for any modification belongs to that target relation.

Subscribing is done by adding new *SubscribingRule* in the database system and listening for incoming relevant data.

- Using the relation meta-data, the system validates the requested *SelectionRule*.
- The system applies a transient database shared lock on that target relation to consistency executes the job.
- The system advances the database sequence generator, and then stamps that job (or session *SelectionRule*) with the new sequence value. Later on, when a new *DBChange* entry is being published by the database triggers, it would be stamped by that value, until a new job is executed.
- The corresponding SQL statement is executed for that *SelectionRule*, by appending a count function as a last selected attribute to retrieve the weight of each selected tuple, as shown in the following table.

For example: *select* stdNo, subjectNo, planYear, count(*) studentCounts
from Student
where (stdNo = 960001 and stStatus != 0)
group by stdNo, subjectNo, planYear

select subjectNo, planYear, grpNo, crsNo, weight, prerequisiteCrsNo1,
prerequisiteCrsNo2, parallelInd, count(*) sheetPlanCounts
from SheetPlan
where (subjectNo = 306 and planYear = 1992)
and ((prerequisiteCrsNo1 > 0 and prerequisiteCrsNo2 = 0)
or (prerequisiteCrsNo1 = 0 and prerequisiteCrsNo2 = 0))
group by subjectNo, planYear, grpNo, crsNo, weight,
prerequisiteCrsNo1, prerequisiteCrsNo2, parallelInd

select crsNo, classNo, dayFE, dayCat, fromHr, toHr,
count(*) classCounts
from Class
group by crsNo, classNo, dayFE, dayCat, fromHr, toHr

```

select stdNo, crsNo, grpNo, weight, count(*) registeredCourseCounts
from RegisteredCourse
where (stdNo = 960001)
group by stdNo, crsNo, grpNo, weight

```

Table 4-2: SQL statements for the example in section 4.3.1.1

- At the end, the agent unlocks the target relation and returns back the result of execution to the actual client (reasoning system) that is transparently represented by the job's session object.

While one or more session is being active, the middle-tier agent automatically receives and buffers the arrived database modification entries in a form of *DBChange* objects. When the number of the buffered database changes exceeds a certain threshold or lasted for a small time window, the dispatcher thread automatically wakes up to transmit the changes according to their interested sessions in the following steps:

- For each buffered entry, the system assesses that entry against each available *SelectionRule* on the relation of that entry. Assessment includes matching entry values (tuple) to the rule filter expression. Also for entries with non-empty changed attributes list (i.e., obtained from update operation), at least one of the changed attributes must be either one of the selected or conditional *SelectionRule*'s attributes. Then, each session attached to these rules is candidates if the times stamp of the session *SelectionRule* is less or equal to that entry's time stamp.
- After classifying changes (entries) to their sessions, the systems sends all tuple of these changes by a session in a single networked message.

When a reasoning system decides to halt, it asks the middle-tier to drop its active session object and all related entries. As a result, any cached relation become with no *SelectionRules*, its meta-data and *SubscribingRules* are automatically purged from the middle-tier agent through calling *setInactive* function.

4.6 Database notification mechanism

In the *iMCP*, execution of the database queries and asynchronous notification of the data modifications are the main functionality of the DBMS. Therefore, the selected DBMS should have the mechanism of handling two-way communication (Hindi, 1994). From network computing viewpoint, the database not only retrieves data in client/server model, but also, automatically publishes data to the interested external applications in publish/subscribe model.

As stated in this chapter, to increase flexibility and scalability of the model, the database system has defined two types of rules in handling asynchronous notification: *Publishing* rules (or triggers) and *Subscribing* rules. *Publishing* rules task is to produce and capture data modifications with a certain format into transactional queues for later consuming. On the other hand, *Subscribing* rules responsibility is to express the interest of external applications (i.e., middle-tier agents) on the data they which to receive from the available queues. As a result of using publish/subscribe architecture, no direct connection is established between *Subscribing* and *Publishing* rules. *Publishing* rules are constructed in a general form without aware of how and who will consume the published data. In opposite, data is subscribed from broker transactional queues without care on what and who is beyond publishing those data.

In the next subsections, the mechanism of *Publishing* and *Subscribing* rules in the *iMCP* is conceptually described based on Oracle8i DBMS. Note that, in our study, the actual development of such mechanism was not implemented in Oracle interface, instead, a simple simulated Java API was developed. The first release of Oracle8i, was launched parallel with the working on this research.

4.6.1 Publishing rules mechanism

Using on-delete², on-insert, and on-update database active rules (or triggers), a copy of the data being processed is published in a form of *DBChange* object. These triggers are activated if at least one interested subscriber is available. The values of main attributes of the *DBChange* object are summarized below in table 4-3.

<i>Trigger event</i>	<i>Tuple event</i>	<i>Tuple values list</i>	<i>Changed attributes list</i>	<i>Time stamp</i>
Insert	Insert	new tuple values	null	current value of the DB sequence generator
Delete	Delete	old tuple values	null	
Update	Insert	new tuple values	ids of changed attributes	
	Delete	old tuple values	ids of changed attributes	

Table 4-3: Values of the *DBChange* attributes

Published *DBChange* objects are buffered in a non-structured (i.e., binary data type) queue. The non-structured queue relieves the system from creating a different table for each different relation structure. Using Oracle Java VM, the *DBChange* is transformed, before enqueued, in a binary message format using *Serializable* protocol. After the middle-tier agent dequeues that message, it directly restores the original *DBChange* object. Using Oracle persistent advanced queue(s), the system has the chance for grouping, guarantee delivering, and transactional visibility of messages (see section 3.4.2).

4.6.2 Subscribing rules mechanism

The middle-tier agent dynamically subscribes for all messages related to certain predefined relation types. Subscribing can be either done on a single queue for all

² In Oracle, we mean by the on-event trigger, either before, after, or instead of row level trigger.

interested relations or on multiple queues, in which each refers to a different relation. Where in both cases, a single queue table is needed.

In Oracle, subscribing on a single queue for all relations requires the system to build its *Subscribing* rules based on message correlation property (see section 3.4.2). Messages are published with a correlation property equals to the relation name of the *DBChange* object. Only one subscribing rule could be dynamically constructed to receive the relevant modifications.

e.g., *corrid* in ('*student*', '*sheetPlan*', '*class*', '*registeredCourse*')

Another possible architecture, let's the database system publishes each data relation in a non-structured private queue. All defined queues are constructed into a single queue table. *Subscribing* rules of the middle-tier agent are no more than subscribing for all data received on the queues, which refer to the target relations.

To receive the messages of the database modifications using Oracle DBMS, the middle-tier agent has either to initiate a blocked dequeue (or listen) command or to use asynchronous notification APIs. In the asynchronous notification mode, a callback function is registered from the middle-tier into the DBMS to be automatically executed on the middle-tier server when a new data is become visible in the requested queue(s).

4.7 Summary

In this chapter, a new class of systems integration was proposed for coupling expert systems with databases over the Internet. The model for the coupling mechanism is based on the MCP, called *iMCP*. The *iMCP* is asynchronous loosely coupling approach, which maintains data consistency between its components.

To make the *iMCP* suitable for the Internet environment, which is highly distributed, autonomous, and heterogeneous, it is designed³ to support open and interoperable network computing standards. The model has to use the best of client/server, publish/subscribe, and distributed object technology to run complex programs (such as a reasoning system) over thin deployed requirements (e.g., browser). The three-tier computing architecture is adopted to increase the system scalability and efficiency.

Communication architecture for retrieving database query commands is processed in client/server model, based on a single database relation (no join). In contrast, using rule-based publish/subscribe model, any database modification on relevant relations is automatically received by the middle-tier to be further routed to the actual interested clients (reasoning systems).

The proposed model is too flexible in which many efficient architecture variations could be implemented. A client could be connected to more than one middle-tier agent with an open interface, and the middle-tier agent might extract the needed data from one or more data server.

³ the system was implemented using Java development kit 1.2 and Java dynamic management kit 3.0. The functionality of the Oracle8i was simulated using Java APIs.

Chapter 5

A New Compromised Improvement Approach to ATMS Tightly Coupled Production System

5.1 Introduction

The efficiency of the reasoning system is an important issue in the MCP and *i*MCP, because the longer the reasoning process takes the more likely that some data will change and therefore, more data communication and belief revision will be needed. This was also realized by Hindi (1994), and developed an approach to couple the Rete network with the ATMS, that is more efficient than Morgue's approach in terms of the time needed to perform the match step. However, the Hindi (1994) method requires more memory than Morgue's approach.

In this chapter, we present a new modified approach and we present an empirical comparison between the three approaches: the Morgue approach, the Hindi approach, and our new modified approach. Our comparison study shows that the time efficiency of the new approach is close to that of the Hindi approach. On the other hand, the memory requirement is close to that of the Morgue approach. Therefore, the new approach can be considered as a good compromised between the two approaches, so, we will refer to it from now on as the compromised approach.

In section 5.2, the drawbacks of the Morgue and Hindi systems are reviewed. Section 5.3 represents the new compromised system. Section 5.4 proposes an empirical study between the three approaches.

5.2 Drawbacks of the Morgue and Hindi Systems

As discussed in section 2.6.2, the Morgue system for tight coupling ATMS with production rule system, removes tuples with empty labels from the Rete network. This reduces the cost of contradiction handling, label-update, and join operation. However, it has been noticed (Hindi, 1994) that adding a new environment to a tuple, which frequently happens in multiple-context problems, causes a crucial efficiency problem. In this situation, the Rete match algorithm not only need to perform environment union and subset tests, which are main operations of the ATMS label-update algorithm, but also need to perform an expensive join operation. Consider, that the updated tuple is located in a Rete memory node that is an input to a given *and* node. Then, re-generating discarded empty tuples to re-compute their labels again requires re-joining that tuple with all matching tuples in the other input memory node. These joined tuples may need further matching down the network. What makes this problem even worse is that a lot of the matching operations may have been performed before (when the label of the tuple was not empty for the first time).

Hindi (1994) noticed these implications of discarding tuples with empty labels, and developed an approach to avoid these drawbacks of the Morgue approach. The Hindi system divides each memory node in the Rete into two parts, one active part, called the IN-part, and another inactive part called the OUT-part. The IN-part of a memory node is used to store tuples with non-empty labels, while the OUT-part is used to cache tuples with empty labels. When the label of a tuple becomes empty, it is moved from the IN-part to the OUT-part of the same memory node. If the label of a tuple in the OUT-part becomes non-empty, it is moved to the IN-part and joined only with the new tuples in the

other input node (those that were inserted while it was in the OUT-part). Time stamps are used to determine these tuples.

However, due to caching all tuples with empty labels, an incremental growth of the system space requires extra memory space. In some applications, OUT-parts might become as a system overhead.

5.3 The new compromised approach

The new approach is a modification of the Morgue system. It aims at reducing the need to perform join operations when the label of a tuple is updated. Recall that, the Morgue system needs to join a tuple (in an input node of an *and* node) with the matching tuple every time a new environment is added to its label. This is to ensure that any tuple that might have been discarded from the output node (because its label becomes empty) will be generated.

The new compromised approach avoids the need to perform a re-join operation, when a new environment is added to a tuple with non-empty label. For this, some of the tuples with empty labels are cached in the system. To avoid involving the cached empty-tuple with any matching operation, its reference in the Rete memory node is discarded, while its reference in the dependency network of the ATMS is remained as *inactive-link*. When its label becomes non-empty, it is reactivated in the Rete memory node and matched. The method also avoids the high memory requirements of the Hindi system, because it does not cache all empty tuples. The new method caches (as *inactive-links*) only the empty tuple in the output node, but not the tuples that are linked to it down the network.

However, if the addition of an environment to a tuple makes the label of that tuple non-empty, then the new approach would need to perform a join operation as the Morgue system. Therefore, in applications those require the formation of a huge dependency network, it is expected that the Hindi system will be much better than the compromised approach. Especially when the system has enough resources to efficiently operate, and the data is frequently changed from the sources in which it locates.

Figure 5-1 illustrates an example of the new approach. X in the figure is a joined tuple of R and S . If as a result of some label-update operation, the label of X became empty, then X itself will be cached but not tuple Z , which is dependent on X . This eliminates the need to rejoin R and S when a new environment is added to the label of any of them, since X is not solely discarded from the system. A join operation will only be needed to join X with Y only if the label of X becomes non-empty. This join operation will be needed to regenerate Z . This is unlike the Hindi system, which would cache X and Z and would therefore, eliminate the need for a rejoin operation.

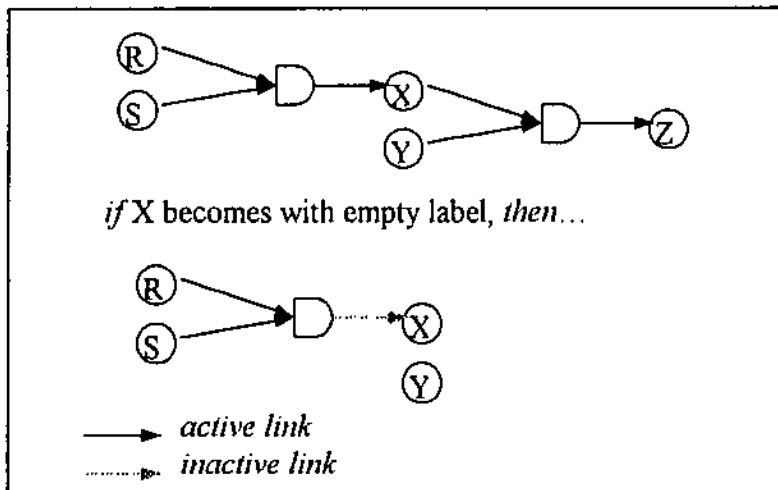


Figure 5-1: Illustrate the changes in the dependency network

5.3.1 Discarding a tuple from the compromised system

The system would discard a tuple that becomes with empty label either as a result of retracting an assumption or executing a contradictory rule. In general, to discard empty-tuples, the compromised system performs the following steps:

- The references to the empty tuples in the Rete memory nodes are purged.
- The links of the empty tuples in the dependency network of the ATMS that support other inferences are removed.
- The links from non-empty tuples that justify these empty tuples are remained as *inactive-links* in the dependency network.

To illustrate the algorithm, in more details, consider the following example in figure 5-2. Where, the normal lines represent the Rete network's links and the dotted lines correspond for the dependency network's links.

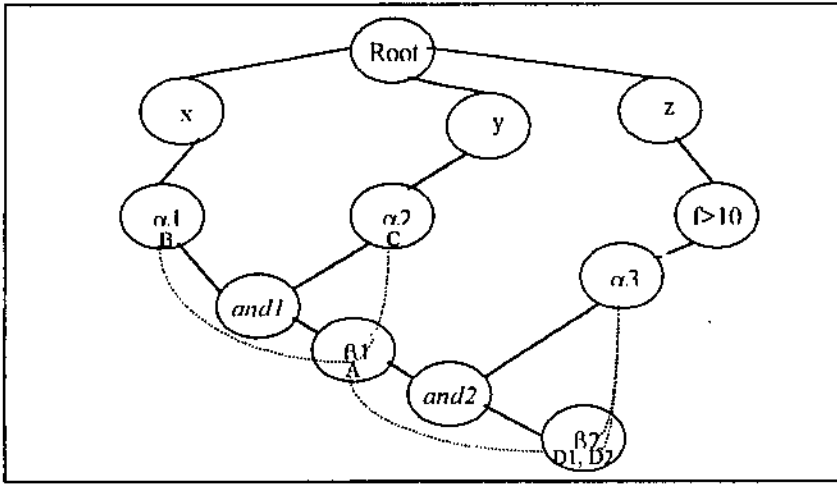


Figure 5-2: Hypothetical example to illustrate the discard algorithm

Assume that, a joined tuple A , that exists in $\beta 1$ memory node, has become with an empty label (for example, due to contradiction handling). The references to A and to any relevant tuple with empty label, which A is a constituent of (such that, $D1$ & $D2$ in $\beta 2$) are removed from the Rete memory nodes. Also, all links of A and their conjugates in $\alpha 3$ that support any tuple in $\beta 2$ -memory node are dropped. But, all links that support A itself are not affected (links from B & C , which they are non-empty tuples). With the same manner, the links of $D1$ & $D2$ are treated. As a result, using garbage collecting, $D1$ & $D2$ are automatically purged from the system, since they do not have any reference in

the Rete memory nodes or in the dependency network. But A remains alive, since it has *inactive-links* (from B & C) in the dependency network.

During adding a new environment to a label for a given tuple (e.g., B), the system does not need to re-perform the match from scratch (re-join B with all tuples in α_2) to re-generate possible discarded tuples (e.g., A) and compute their labels again. The system only has to follow the links in the dependency network of that tuple (e.g., B) to update the status of its successors in the dependency network (A and other tuples in β_1 where B is a constituent, somehow, as occurred in Hindi's approach). If that followed tuple was with non-empty label, then the system simply updates its label. Otherwise, if tuple becomes non-empty, the system reactivates that tuple in the corresponding output memory node. Also in this case, the system would re-perform the matching down the Rete to regenerate discarded tuples (e.g., D_1 & D_2). The Hindi system has solved this problem by caching all tuples with empty labels (A , D_1 & D_2) in inactive memory parts to reserve the match for possible reactivation.

However, we expect that the implication of the re-match propagation problem when a tuple with empty label becomes non-empty would have a quite limit. Since, inconsistent environments are discovered in an early stage of the reasoning process. Therefore in most times, the join operation (in the three approaches) is performed (not re-performed) on the relevant output memory nodes when a tuple becomes with non-empty label. Usually, during labeling, the system discovers that a given tuple (e.g., A) has an empty label before the match is propagated to the output memory nodes (e.g., β_2).

On the other hand, retracting of an assumption due to, for example, a delete operation in the coupled database system in the *iMCP* could affect the system performance. Retracting an assumption, which can be thought of discovering a new *nogood* environment that consists of the assumption itself, might cascade the discard

procedure into many consecutive Rete memory nodes. Later on, re-activating that assumption again could require an expensive re-matching. In contrast, caching all these inactive inferences (as in the Hindi system) would solve this problem, but with an extra memory space allocation.

5.3.2 Asserting a tuple to the compromised system

The reasoning system performs the needed match operation to assert a tuple into the corresponding Rete memory node. That tuple is asserted as a result of deriving a new inference, or creating a new assumption.

The compromised system deals with two main cases: the tuple being asserted, as a consequent of given justification, is new, or is already in the corresponding Rete memory node:

- In the two cases, the system has to compute the label of the matched tuple. If its label is empty or is a subset of its old label, then no new information is gained using the current justification. However, the system has to justify that tuple by creating a new *inactive-link* for possible re-activation due to adding a new environment to its constituents.
- When the tuple does not exist in the corresponding memory node, then:
 - If the current Rete node corresponds to the contradiction node (\perp), then the contradiction procedure of the ATMS is called.
 - Otherwise, the tuple is inserted in the corresponding memory node. If the memory node is of type *P*-memory, then the system instantiates any attached rule. Also, for each next *and* node, the system joins the tuple with matched tuples located in the other input node, and propagates the matching procedure down the Rete.

- If the label of the tuple is changed and the tuple is already in the Rete memory node, then using linked justifications to that tuple, the system propagates the matching down the Rete.

5.3.3 A summary of the three different approaches

The following table (5-1) highlights the differences among the three approaches: The Morgue system, The Hindi system, and the compromised system, in terms of seven main matching cases. Each case describes the appropriate action when the asserted tuple (initial, inferred, or joined) has arrived through the Rete network to a corresponding memory node in which it should be located.

	Case	Action		
		Morgue system	Compromised system	Hindi system
1	<ul style="list-style-type: none"> * A tuple does not exist in the arrived memory node. * It satisfies the node's condition. * Its label is empty. 	<ul style="list-style-type: none"> * Do nothing. 	<ul style="list-style-type: none"> * Justify that tuple. So, only new <i>inactive-link</i> is added into the dependency network. 	<ul style="list-style-type: none"> * Justify that tuple * Store it in the OUT-part memory location with time stamp = (-1).
2	<ul style="list-style-type: none"> * A tuple does not exist in that memory node. * It satisfies the node's condition. * Its label is not empty. 	<ul style="list-style-type: none"> * Store it in that memory node. * Justify it. * Propagate through the Rete for join and label-update. 	<ul style="list-style-type: none"> * As the Morgue system. 	<ul style="list-style-type: none"> * As the Morgue system. Where the tuple is placed in the IN-part of that memory node.
3	<ul style="list-style-type: none"> * A tuple exists in the IN-part memory location (for the Morgue system and the new system, memory nodes only consist of IN-parts). * Its label differs from the old label. 	<ul style="list-style-type: none"> * Update its label. * Justify it. * Propagate through the Rete for re-join and label-update. 	<ul style="list-style-type: none"> * Update its label. * Justify it. * Propagate through the Rete for label-update using dependency network's links (<i>active & inactive</i> links). 	<ul style="list-style-type: none"> * Update its label. * Justify it. * Propagate through the Rete using dependency network's links for label-update.
4	<ul style="list-style-type: none"> * A tuple exists in the IN-part memory location. * Its label is the same as the old label. 	<ul style="list-style-type: none"> * Justify it. 	<ul style="list-style-type: none"> * Justify it. 	<ul style="list-style-type: none"> * Justify it.

	Case	Action		
		Morgue system	Compromised system	Hindi system
5	<ul style="list-style-type: none"> * A tuple exists in the OUT-part memory location. * Its label is not empty. 	* NA.	* NA.	<ul style="list-style-type: none"> * Update its label. * Justify it. * Propagate through the Rete using the dependency network for label-update. * Propagate through the Rete for joining that tuple with all IN-facts in of the other input memory which they have time stamp \geq its time stamp.
6	<ul style="list-style-type: none"> * A tuple exists in the OUT-part. * Its label is empty. 	* NA.	* NA.	* Justify it.
7	<ul style="list-style-type: none"> * A tuple is \perp (i.e. corresponds to the contradiction node). * Its label is not empty 	<ul style="list-style-type: none"> * Consider each environment in its label as a <i>nogood</i> environment. * Remove from labels of all related tuples any superset environment of the <i>nogood</i> discovered environments. * Insert those tuples in a specialized agenda, if their labels become empty. * Traverse the agenda to delete the specified tuples from the Rete memory nodes and the dependency network. 	<ul style="list-style-type: none"> * Insert those tuples in a specialized agenda, if their labels become empty. * Traverse the agenda to delete the specified tuples from the Rete memory nodes. * Clear all links of these tuples except these links, which attached to non-empty tuples. 	<ul style="list-style-type: none"> * If any updated label become empty, move it with the current system time stamp to the corresponding OUT-part. * Advance the system current time.

Table 5-1: Main cases of the match algorithm in the three approaches

5.4 An empirical study of the three approaches

The following empirical study compares the three approaches under two different situations. First, we compare the Morgue system, the Hindi system, and the compromised system when they run as standalone program components, without any interaction with the external environment. All the knowledge (general or instance) is within the reasoning system shell. Second, we compare the efficiency of these systems when each one is a part of the *mMCP*. The reasoning system, in this case, takes in its

considerations any relevant modification on the retrieved data, with minimum effort, the system has to revise its beliefs and keep consistent (up-to-date) with the external database.

We are concentrated with three issues: time, memory requirement, and the number of required operations. The execution time and number of operations are proportional to each other. Together, they offer a reasonable justification for any produced result. Also, the number of conflict cycles that are required for whole reasoning process is computed as an assistant factor to describe the system general behavior. For example, the Morgue system requires more cycles to accomplish the whole process. That is because the Morgue system does not go through the dependency network for label propagation.

To clarify the differences among the three systems, we consider the results of the Morgue system as a baseline. As a case study, a *Student Registration Guidance System* (SRGS) is used in formulating all the needed experiments. That application offers the capability of building multiple-context problem that is required for ATMS-based reasoning systems. The experiments were performed using a PC Pentium 133 processor and a 16MB RAM, and implemented using Java 1.2 virtual machine.¹

5.4.1 Student Registration Guidance System (SRGS): a case study

Many planning applications have been accurately defined and formulated in terms of the ATMS labeling algorithm. The task of the planner is to find all possible actions and sequences to achieve some specific goal. Electronic planing systems could efficiently take the advantage of an integrated database by performing intelligent processing on the retrieved shared data. For example, in flight-route planner system, the

¹ However, increasing system specifications directly affects the computed results. For example, using 48MB decrease time required around the third.

planner has to generate all proper routes between two different locations. Data may include whether condition, flights, agencies and airlines information.

With the same manner, the objective of *Student Registration Guidance System* (SRGS) is to generate all semester schedules for a given student based on his/her record. The system directs students to the courses and classes in which they can register. The input data needed includes students' record, majors (subjects), courses, and classes basic information.

The reasoning system performs three main steps. First, the system tries to identify all possible classes that a student can register in, given his/her record, major plan, and a list of all available classes. These classes are elected and stored in *classTryReg* predicate. Second, the SRGS aggregates all applicable classes (i.e., the *classTryReg* data) according to their courses in *courseTryReg* predicate. In other words, all classes (in the *classTryReg*) that refer to the same course are grouped into a single tuple in the *courseTryReg*. Then, a set of constraints (and contradictory rules) is applied to ensure the validity of these courses. Third, the system tries to derive the consistent sets of valid courses to present all proper suggested schedules.

Figure 5-3 below illustrates the main reasoning steps in the SRGS.

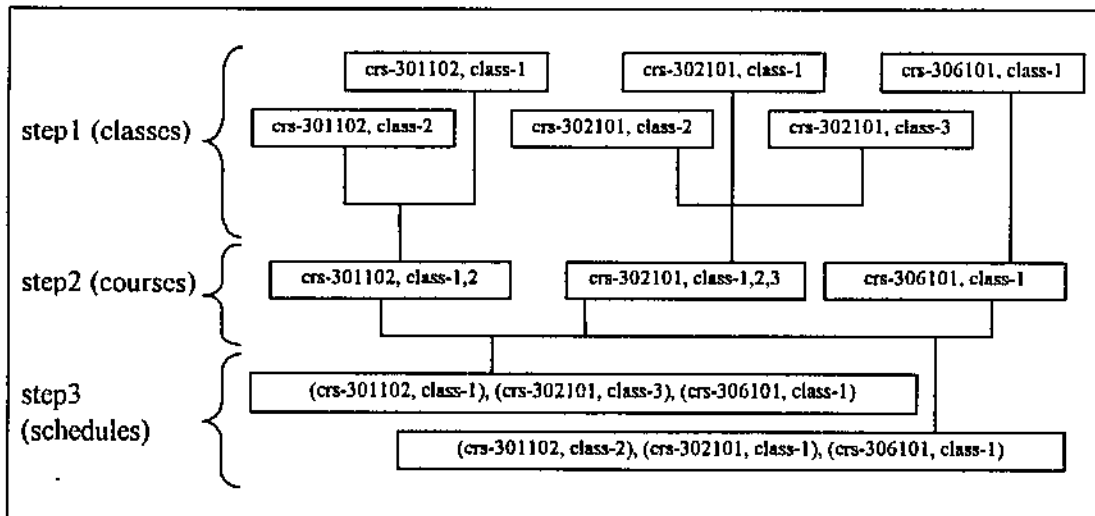


Figure 5-3: Main planning steps of the SRGS

Reasoning with the *classTryReg* is handled in a single-context search space. While, most of reasoning in the *courseTryReg* is processed in a multiple-context search space. Therefore, the reasoning behavior in the SRGS starts with a single-context state and possibly ends in a multiple-context state. A complete general knowledge of the system is listed in *Appendix A*.

5.4.2 Measuring the efficiency of the three systems as standalone systems

Comparisons in this subsection highlight the differences among the three systems during normal and complete reasoning process without any external interruptions. In this case, the system itself is considered as a closed world. Since all the specific case data and general knowledge are located within the expert system shell. Results in this subsection reflect the efficiency of the inference engine during reasoning in a multiple-context problem solving.

Experiments of this section are divided into two main groups (A & B). Group A has 10 different experiments for a first-year student, who suggests in each experiment, a different set of desirable courses. The goal is to generate all possible proper schedules based on the suggested set of courses. Group B contains 10 different second-year students. Considering their records, the system has to generate all possible schedules. The details (i.e., the case specific data) for experiments are presented in *Appendix B*.

Tables below represent the results in terms of the three approaches.

Experiment No.	# of Solutions	Time /s			Conflict Cycles		
		Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
A-01	31	144.158	132.430	126.152	285	279	279
02	22	71.924	70.763	62.901	284	278	278
03	2	69.300	67.658	59.276	282	277	277
04	5	112.121	107.865	98.411	282	278	278
05	0	93.234	86.805	77.622	281	277	277
06	72	61.858	54.398	50.793	284	283	283
07	0	71.373	58.244	57.353	279	275	275
08	36	47.489	42.622	40.138	276	275	275
09	77	80.706	75.930	74.898	283	280	280

Experiment No.	# of Solutions	Time /s			Conflict Cycles		
		Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
10	0	189.602	167.581	165.328	283	279	279
Total		941.765	864.296	812.872	2819	2781	2781
Percentage			%-8.23	%-13.69		%-13.48	%-13.48
B-01	302	215.560	174.421	177.936	456	383	383
02	1077	359.917	324.487	321.432	730	682	691
03	1994	582.948	457.067	442.796	882	608	605
04	109	114.575	99.773	93.585	363	351	353
05	15	101.526	77.291	68.498	332	316	316
06	914	281.765	256.909	247.846	657	500	500
07	3	38.365	38.095	36.903	288	288	288
08	456	253.865	249.709	237.442	433	426	432
09	119	169.354	150.316	134.924	374	357	357
10	14	131.069	120.543	112.952	311	309	313
Total		5003	1948.611	1874.314	4826	4220	4238
Percentage			%-13.35	%-16.66		%-12.56	%-12.19

Table 5-2: Time comparison of standalone Morgue-like systems

It is obvious from table 5-2 that the Hindi system is the quickest approach to fulfill the reasoning process. But, the new compromised system is not much worse than the Hindi system. While, the Morgue system is the slowest one. For long runtime processing, the difference between the Morgue system and the others increases as the chance of adding environments to existing tuples increases, while the difference remains nearly the same between the other two approaches.

Experiment No.	# of Join Operations			# of Evaluated Conditions			# of Labeling Operations		
	Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
A-01	18379	13701	13701	19291	14891	14863	1298	1295	1271
02	15939	12203	12203	16921	13370	13345	1152	1146	1125
03	16382	12770	12770	17382	13940	13911	1229	1224	1203
04	19518	14953	14953	20465	16153	16103	1436	1437	1408
05	18942	14772	14772	19920	15956	15915	1427	1424	1397
06	13115	12030	12005	14213	13224	13150	1077	1076	1054
07	14420	11062	11062	15411	12258	12196	1140	1138	1112
08	11525	10468	10448	12605	11648	11584	962	961	938
09	15885	14108	14085	16895	15288	15213	1252	1247	1221
10	22072	16383	16383	22912	17632	17507	1653	1652	1617
Total	166177	132450	132382	176015	144360	143787	12626	12600	12346
Percentage		%-20.30	%-20.37		%-17.98	%-18.31		%-0.21	%-2.22
B-01	35939	26952	26952	36405	28447	28328	3112	3067	3049
02	54330	45118	45118	55290	46829	46781	4334	4372	4352
03	75033	47274	47209	74584	49178	48849	6225	5924	5858
04	29909	25479	25479	30998	26892	26828	2156	2175	2127
05	26633	20294	20294	27373	21631	21596	1901	1934	1917
06	41913	30433	30433	42286	32062	31906	3954	3843	3828
07	15746	15108	15108	16945	16379	16371	1261	1273	1260
08	35352	31696	31693	36463	33120	33039	2923	2938	2900
09	38331	30017	30017	39172	31465	31371	2574	2586	2533
10	28996	24579	24579	29981	25901	25834	2159	2185	2153

Total	382182	296950	296882	389497	311904	310903	30599	30297	29977
Percentage		%-22.30	%-22.32		%-19.92	%-20.18		%-0.99	%-2.03

Table 5-3: Operations comparison of standalone Morgue-like systems

Table 5-3 shows statistical results for the main operations (the join, the evaluated conditions during rules' matching, and the label-update) needed during the match process for the three systems. The table supports the results presented in table 5-2, and shows the seriousness of the drawbacks of the Morgue system. The Morgue system performs an expensive join operation whenever a new environment is added to the label for an existing tuple. The Hindi system does not perform re-match operation during labeling propagation. While, the compromised approach may perform rematch operation only if an empty label become non-empty.

Also there is a large gap between the Morgue system and the other two systems with respect to the number to the evaluated conditions. This is because the Hindi system and compromised system avoid re-matching when that is possible. In addition, the Hindi system saves all empty tuples in the OUT memory parts. Moving them from OUT-part to IN-part locations does not need to re-evaluate the test conditions again.

However, the difference in operations between the Hindi system and the compromised system is relatively small. This supports our claim that discarding tuples with empty labels from the systems is not propagated to too many nodes down the Rete, because most of tuples with empty labels are discovered in the early stage of the reasoning process during contradiction handling procedure.

Thus, the previous behavior also justifies, why the difference in time between the Hindi system and the compromised system is greater in somehow than the difference in number of operations. This is because the discovering of the *nogood* environments has the highest priority than any other operation. Therefore, the reasoning requires many calls to the contradiction handling procedure. The contradiction handling procedure in

the Hindi system is quicker than the others. Since, discarding empty tuples are simply moving them to the OUT-part memory nodes without using an extra special agenda (as happen in the Morgue system and the compromised system).

As a general behavior for the three systems, the label computation process must be calculated for each matched tuple. Therefore, no significant differences among the three systems can be noticed.

Experiment No.	# of Tuples in the Rete Network			# of Links in the Dependency Network		
	Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
A-01	936	936	1325	1161	1571	1686
02	871	871	1229	1039	1379	1497
03	899	899	1316	1096	1554	1672
04	1002	1002	1480	1296	1882	1997
05	980	980	1497	1253	1917	2032
06	869	869	1228	1035	1347	1486
07	809	809	1211	929	1329	1462
08	790	790	1110	886	1126	1260
09	949	949	1335	1178	1582	1707
10	1046	1046	1573	1369	2061	2182
Total	9151	9151	13304	11242	15748	16981
Percentage		%0.0	%+45.38		%+40.08	%+51.05
B-01	1985	1985	2538	3069	3639	3791
02	3658	3658	4382	6136	6900	7170
03	3295	3295	4168	5436	6580	6756
04	1707	1707	2298	2549	3153	3308
05	1520	1520	2002	2252	2616	2794
06	2483	2483	3141	3968	4662	4891
07	1159	1159	1546	1553	1783	1933
08	2241	2241	3012	3528	4590	4722
09	1941	1941	2568	3022	3670	3843
10	1662	1662	2300	2511	3231	3385
Total	21651	21651	27955	34024	40824	42593
Percentage		%0.0	%+29.12		%+19.99	%+25.19

Table 5-4: Space comparison of standalone Morgue-like systems

From table 5-4, the allocated space for the Rete memory nodes in the Morgue system and the compromised system is identical. In contrast, the Hindi system uses additional special OUT memory parts to cache all empty tuples. On the other hand, the dependency network for the new approach keeps more entities (as *inactive-links*) than the original Morgue approach. Where, the Hindi system reserves all links for labeling without any reduction.

5.4.3 The efficiency of the three systems as a part of the *iMCP*

We discuss in this subsection, how the three systems can be integrated efficiently in a dynamic data environment using the *iMCP*. The approaches can take in their considerations any alerted data from the databases. The ATMS responsibility is to revise just all affected beliefs. The efficiency of the ATMS-based reasoning system is an important issue. The longer the reasoning system takes to run, the greater is the chance that it will conflict with a changing of retrieved data (Hindi, 1994).

Our objective is to study the efficiency of the reasoning system interface of the three systems with the coupled database, based on basic data manipulation operations: delete, update, and insert. The experiments, for this category, were performed on two students (C & D). The goal is to generate all possible schedules based on their records. Each student has (9) different experiments that are characterized as follows:

Experiment (1) A standalone reasoning (i.e., equivalent to the previous subsection).

Experiment (2)&(3) Experiment (1) has being alerted for retracting an assumption (for example, when a tuple is deleted from the coupled DBS), which serves in a single and multiple- context search space, respectively.

Experiment (4)&(5) Experiment (1) has being alerted for asserting a new assumption (for example, due to new relevant tuple has been inserted into the integrated DBS), which creates a new single and multiple-context space search, respectively.

Experiment (6)&(7) Experiment (1) has being alerted for retracting followed by re-asserting the same assumption (for example, due to transient changing on a given retrieved tuple) that supports a single and multiple-context search space, respectively.

Experiment (8)&(9) Experiment (1) has being alerted for retracting an assumption followed by asserting a new one (for example, due to updating selected attributes for a retrieved tuple, that causes removing the old status and inserting the new

one) for a single and multiple-context search space, respectively.

From experiment (2) to (9), we discuss only the efficiency of handling the alerted operation. Where, a full description of each experiment is listed in Appendix B.

Table 5-5, 5-6, and 5-7 below describe the time, the number of operations, and space required for experiment (1). The results' behavior is the same as the previous subsection, so no further explanation is needed for this case.

Experiment No.	# of Solutions	Time (s)			Conflict Cycles		
		Morgue system	Compromised system	Hindi System	Morgue system	Compromised system	Hindi system
C-01	12	141.754	112.902	112.652	338	336	348
D-01	9	87.336	78.243	77.094	343	341	343
Total		229.290	191.145	189.746	681	677	691
Percentage			%-16.64	%-17.25		%-0.59	%+1.47

Table 5-5: Time comparison of normal reasoning for Morgue-like systems

Experiment No.	# of Join Operations			# of Evaluated Conditions			# of Labeling Operations		
	Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
C-01	47179	38041	38023	47998	39520	39459	3208	3230	3193
D-01	41471	38862	38862	42802	40379	40336	2976	2995	2955
Total	88650	76903	76885	90800	79899	79795	6184	6225	6148
Percentage		%-13.26	%-13.27		%-12.01	%-12.12		%+0.66	%-0.58

Table 5-6: Operations comparison of normal reasoning for Morgue-like systems

Experiment No.	# of Tuples in the Rete Network			# of Links in the Dependency Network		
	Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
C-01	2557	2557	3206	4211	4759	4966
D-01	2745	2745	3319	4565	4921	5136
Total	5302	5302	6525	8776	9680	10102
Percentage		%0.0	%+23.07		%+10.30	%+15.11

Table 5-7: Space comparison of normal reasoning for Morgue-like systems

5.4.3.1 Retract an assumption due to deleting the corresponding tuple from the coupled database system

Experiment (2): for a single-context search space.

e.g. from SRGS: drop a single-class course from the list of open classes.

It is obvious from tables 5-8, and 5-9 below that the Hindi system is the best in terms of the time required to perform the retract operation. It just moves all empty tuples to the corresponding OUT-part memory locations. While, the other systems need a

special agenda to record all discarded tuples and traverse that agenda again to remove them from the system. In contrast, in the Hindi system, no actual space releasing is happened, because all space still in-use for OUT-part memory locations.

Experiment (3): for a multiple-context search space.

e.g. from SRGS: close a class from multiple-class course.

This situation has the same behavior of experiment (2).

Note that, no join operation, condition evaluation, and labeling computation is required in retracting assumptions (so, we omit operations comparison table). Retracting an assumption is equivalent to discovering a new *nogood* environment. Thus, only contradiction handling operation is needed, which is not presented in our study, since the three coupling approaches execute contradictory rules as soon as they have been instantiated.

Experiment No.	# of Solutions	Time (s)			Conflict Cycles		
		Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
C-02	6	149.846	118.491	114.635	339	337	349
D-02	3	87.542	78.974	77.752	344	342	344
Total		237.388	197.465	192.387	683	679	693
Difference		8.098	6.320	2.461	2	2	2
Percentage			%-21.96	%-69.91		%0.0	%0.0
C-03	8	146.841	117.469	114.875	339	337	349
D-03	6	87.831	78.832	77.963	344	342	344
Total		234.672	196.301	192.838	683	679	693
Difference		5.382	5.156	3.092	2	2	2
Percentage			%-4.20	%-42.55		%0.0	%0.0

Table 5-8: Time comparison for retract operation in Morgue-like systems

Experiment No.	# of Tuples in the Rete Network			# of Links in the Dependency Network		
	Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
C-02	2252	2252	3206	3617	3937	4966
D-02	2597	2597	3319	4285	4559	5136
Total	4849	4849	6525	7902	8496	10102
Difference	-453	-453	0	-874	-1184	0
Percentage		%0.0	%+100.0		%-35.47	%+100.0
C-03	2540	2540	3206	4183	4727	4966
D-03	2728	2728	3319	4537	4889	5136
Total	5268	5268	6525	8720	9616	10102
Difference	-34	-34	0	-56	-74	0
Percentage		%0.0	%+100.0		%-32.14	%+100.0

Table 5-9: Space comparison for retract operation in Morgue-like systems

5.4.3.2 Assert an assumption when a new relevant tuple is inserted into the coupled database system

Experiment (4): for a single-context search space.

e.g. from SRGS: submit a new class for non-existing course.

Tables 5-10 and 5-11 show that the time and operations required for the new system and the Hindi system are close to each other. Also, the Morgue system is not faraway from them. That is cleared since, creating a new single-context search space has a small chance for adding an environment to an existing tuple. In this case, the match procedure for the three approaches requires in most times join (not re-join) operation.

Table 5-12 shows that the allocated space for the assert statement is relatively large, since most of search space had created in this context is new. Also we can notice that around half of entries, in the Hindi system and the compromised system, for that operation belong to the discarded tuples. This is because we assert the new tuple at the end of the reasoning process. Where, the list of the *nogood* environments is mostly discovered. So in this case, any asserted tuple, in the Rete network, has a high probability to be a new and with empty label. Where, a new tuple with empty label is neglected in the Morgue system, cached in an OUT-part memory location in the Hindi system, and maintained as *inactive-link* in the compromised system.

Experiment (5): for a multiple-context search space.

e.g. from SRGS: open a new class for an existing course.

In this case, the chance of adding an environment to already exist tuple is increased. That is the main bottleneck of the Morgue system. It would require expensive join operation during the label-update propagation. While the space required for the three systems during executing the assert statement is quite limited, since the match operation has been performed on already existed search space.

Experiment No.	# of Solutions	Time (s)			Conflict Cycles		
		Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
C-04	18	182.483	150.597	146.381	345	343	367
D-04	15	101.896	90.570	90.801	350	348	356
Total		284.379	241.167	237.182	695	691	723
Difference		55.089	50.022	47.436	14	14	32
Percentage			%-9.20	%-13.89		%0.0	%+128.6
D-05	18	182.783	138.409	137.137	345	343	351
D-05	15	98.902	82.338	82.909	350	345	346
Total		281.685	220.747	220.046	695	688	697
Difference		52.395	29.602	30.300	14	11	6
Percentage			%-43.50	%-42.17		%-21.43	%-57.14

Table 5-10: Time comparison for the assert operation in Morgue-like systems

Experiment No.	# of Join Operations			# of Evaluated Conditions			# of Labeling Operations		
	Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
C-04	53179	43747	43729	53994	45232	45180	3750	3782	3751
D-04	44112	41395	41395	45451	42920	42880	3274	3293	3253
Total	97291	85142	85124	99445	88152	88060	7024	7075	7004
Difference	8641	8239	8239	8645	8253	8265	840	850	856
Percentage		%-4.65	%-4.65		%-4.53	%-4.40		%+1.19	%+1.90
C-05	52724	38709	38699	53256	40316	40148	3627	3651	3613
D-05	43885	39385	39387	45092	40957	40874	3165	3183	3142
Total	96609	78094	78086	98348	81273	81022	6792	6834	6755
Difference	7959	1191	1201	7548	1374	1227	608	609	607
Percentage		%-85.03	%-84.91		%-81.80	%-83.74		%+0.16	%-0.16

Table 5-11: Operations comparison for the assert operation in Morgue-like systems

Experiment No.	# of Tuples in the Rete Network			# of Links in the Dependency Network		
	Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
C-04	2862	2862	3761	4805	5823	6042
D-04	2893	2893	3624	4845	5497	5718
Total	5755	5755	7385	9650	11320	11760
Difference	453	453	860	874	1640	1658
Percentage		%0.0	%+89.85		%+87.64	%+89.70
C-05	2579	2579	3233	4247	4805	5012
D-05	2761	2761	3337	4589	4949	5164
Total	5340	5340	6570	8836	9754	10176
Difference	38	38	45	60	74	74
Percentage		%0.0	%+18.42		%+23.33	%+23.33

Table 5-12: Space comparison for the assert operation in Morgue-like systems

5.4.3.3 Retract followed by reassert an assumption when a tuple is discarded and then revived during a transient database update operation

Experiment (6): for a single-context search space.

e.g. from SRGS: close and immediately re-open a single-class course.

In this case, the Hindi system is much better than the compromised and the Morgue systems. This situation would require expensive re-match propagating down the

Rete. All retracted tuples in the Hindi system have been cached in the OUT-part memory locations. So re-activating them requires only moving them to the IN-part memory locations with label-update propagation. But in the other two systems, the re-match is a must to regenerate the discarded tuples.

Experiment (7): for a multiple-context search space.

e.g. from SRGS: close and immediately re-open a class from multiple-class course.

This case reduces the size of the re-match propagation problem that raised in the previous experiment, since ATMS problem solving mostly applied on multiple-context reasoning. Retracting a context does not mean discarding all of its inferences. Many inferences might also be supported by other valid contexts. Later on, re-activating the discarded context requires mainly labeling (not re-join) propagation. However, the Morgue system does not recognize the difference between this case and the previous one. It has to re-perform the match again. The results in table 5-14 illustrate that behavior.

Note that, no new space is allocated for this operation.

Experiment No.	# of Solutions	Time (s)			Conflict Cycles		
		Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
C-06	12	173.129	144.428	127.243	346	344	350
D-06	9	96.819	87.486	81.837	351	349	345
Total		269.948	231.914	209.08	697	693	695
Difference		40.658	40.769	19.334	16	16	4
Percentage			%+0.27	%-52.45		%0.0	%-0.75
C-07	12	206.237	131.970	127.023	353	342	350
D-07	9	107.044	83.170	80.616	356	347	345
Total		313.281	215.140	207.639	709	689	695
Difference		83.991	23.995	17.893	28	12	4
Percentage			%-71.43	%-78.70		%-57.14	%-85.71

Table 5-13: Time comparison for the retract-reassert operation in Morgue-like systems

Experiment No.	# of Join Operations			# of Evaluated Conditions			# of Labeling Operations		
	Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
C-06	52570	43159	38209	53385	44644	39659	3625	3657	3612
D-06	43825	41117	38973	45164	42642	40459	3163	3182	3141
Total	96395	84276	77182	98549	87286	80118	6788	6839	6753
Difference	7745	7373	297	7749	7387	323	604	614	605
Percentage		%-4.80	%-94.17		%-4.67	%-95.83		%+1.66	%+0.17

C-07	60317	38696	38040	60378	40291	39502	3812	3738	3699
D-07	46396	39490	38882	47451	41079	40382	3249	3223	3178
Total	106713	78186	706922	107829	81370	79884	7061	6961	6877
Difference	18063	1283	37	17029	1471	89	877	736	729
Percentage		%-92.90	%-99.80		%-90.36	%-99.48		%-16.08	%-16.88

Table 5-14: Operations comparison for the retract-reassert operation in Morgue-like systems

5.4.3.4 Retract a tuple and assert another one due to a database update operation

Experiment (8): for a single-context search space.

e.g. from SRGS: update the lecture time of a single-class course.

From time and operations viewpoint, the systems' behavior for this case is similar to that in experiment (6). The Morgue system and the compromised system discard all tuples with empty labels due to the retract operation, and assert new entries from scratch for the asserted one. In the Hindi system, the retracted data are cached in OUT-part memory locations. During asserting the new tuple, the system has a great chance to reuse the common tuples again. For example, the inferences of the *courseTryReg* (not the *classTryReg*) predicate could be reused, since they are common for the old and the new reasoning states.

Experiment (9): for a multiple-context search space.

e.g. from SRGS: update the final-date of a class from multiple-class course.

Table 5-15 and 5-16 show that the compromised system is as efficient as the Hindi system, while the Morgue system is faraway (similar to the experiment 7).

Experiment No.	# of Solutions	Time (s)			Conflict Cycles		
		Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
C-08	12	175.202	144.188	127.603	346	344	352
D-08	9	94.095	88.037	82.098	351	349	347
Total		269.297	232.225	209.701	697	693	699
Difference		40.007	41.080	19.955	16	16	8
Percentage			%+2.68	%-50.13		%0.0	%-50.0
C-09	12	181.290	126.642	122.837	352	342	354
D-09	9	95.787	81.697	83.370	355	347	349
Total		277.077	208.339	206.207	707	689	703
Difference		47.787	17.194	16.461	26	12	12
Percentage			%-64.02	%-65.55		%-53.85	%-53.85

Table 5-15: Time comparison for the update operation in Morgue-like systems

Experiment No.	# of Join Operations			# of Evaluated Conditions			# of Labeling Operations		
	Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
C-08	52570	43159	38825	53385	44644	40275	3625	3657	3612
D-08	43854	41132	39471	45192	42656	40958	3164	3183	3142
Total	96424	84291	78296	98577	87300	81233	6789	6840	6754
Difference	7774	7388	1411	7777	7401	1438	605	615	606
Percentage		%-4.97	%-81.85		%-4.83	%-81.51		%+1.65	%+0.17
C-09	54203	38696	38661	54643	40242	40118	3526	3504	3465
D-09	44359	39438	39438	45553	41000	40933	3127	3122	3080
Total	98562	78134	78099	100196	81242	81051	6653	6626	6545
Difference	9912	1231	1214	9396	1343	1256	469	401	397
Percentage		%-87.58	%-87.75		%-85.71	%-86.63		%-14.50	%-15.35

Table 5-16: Operations comparison for the update operation in Morgue-like systems

Experiment No.	# of Tuples in the Rete Network			# of Links in the Dependency Network		
	Morgue System	Compromised System	Hindi System	Morgue System	Compromised System	Hindi System
C-08	2557	2557	3232	4211	4759	5010
D-08	2746	2746	3337	4567	4923	5164
Total	5303	5303	6569	8778	9682	10174
Difference	1	1	44	2	2	72
Percentage		%0.0	%+4300.		%0.0	%+3500.
C-09	2555	2555	3226	4205	4757	5000
D-09	2743	2743	3338	4559	4919	5168
Total	5298	5298	6564	8764	9676	10168
Difference	-4	-4	39	-12	-4	66
Percentage		%0.0	%+1075.		%-66.66	%+650.

Table 5-17: Space comparison for the update operation in Morgue-like systems

5.5 Summary

We propose a new compromised approach for the original Morgue system (Morgue and Chehire, 1991) that can compete the Hindi system (1994) efficiency in many cases. The system is motivated by the observation that the Hindi system requires an extra space allocation, which during long runtime reasoning could be considered as a system overhead. However, in those applications require the formation of a huge dependency network (and operate with enough space), it is expected that the Hindi system will be much better than the compromised approach, because it caches all retracted tuples.

The approach is based on solving expensive re-join operation due to adding a new environment to a label of an existing tuple. This is done without caching the discarded tuples in the Rete memory nodes (as in the Hindi system), and by maintaining some of them (as *inactive-links*) in the dependency network of the ATMS.

The new approach neglects the re-match propagation problem due to re-activating discarded tuples again. Since, discovering *nogood* environments has the highest reasoning priority. Then, the tuples with empty labels might be explored before the join operation is propagated down the Rete.

The presented experiments emphasize our improvement's bases. The efficiency of the Hindi system and the compromised system are close to each other. While, the amount of the space allocated for our approach is close to the Morgue system. On the other hand, the comparison results for the system when it is embedded in the iMCP record the same percentage of improvement among the other systems, except for reasoning in a single-context search space. The system efficiency might be decreased to be close to the Morgue system, because the system has to re-perform the match down the Rete when a tuple become non-empty.

Chapter 6

Conclusion and Future Work

In the previous two chapters, we extend the MCP for coupling expert systems with database systems over the Internet, and present a new compromised tightly coupling mechanism between the ATMS labeling algorithm and the Rete network. In this chapter we present our conclusion and suggested future work.

6.1 Conclusion

- Electronic reasoning systems (or e-reasoning systems) are a new class of electronic mission-critical systems that have the capability to enhance and support other coupled networked services. For example, a user of a critical e-commerce system will be more safe and confident when he/she is instructed by a consultant e-reasoning system. This reliable reasoning system provides additional support for his/her decisions.
- Using three-tier network computing architecture empowers the constructed paradigm with the scalability, availability, efficiency, interoperability, and manageability. Also the three-tier architecture is the most natural network structure to loosely couple the three different agents (the database system, the service-driven network agent, and the reasoning system). Moreover, the three-tier electronic reasoning architecture is characterized of using thin client (e.g., browser) with fat computations.
- Electronic reasoning systems could be efficiently proposed in terms of the MCP (Hindi, 1994), to take use of its advantages. The MCP maintains the data

consistency of the integrated systems in asynchronous loosely coupled manner. Furthermore, it tries to distribute the processing across the network resources in a best possible utilization.

- The implementing of *iMCP*, which corresponds to the extended MCP for the Internet, has combined the best of the network computing models to efficiently function. The database query commands are executed in a client/server model. The data modifications are notified through a publish/subscribe model. Although, communications are transparently established between different tiers using an open standard distributed object technology.
- Adopting an open standard distributed object technology (e.g., CORBA, D/COM, DCE, and Java) increases the system interoperability, portability, and reliability. Consequently, different architectures could be directly constructed within the paradigm framework. A single or multiple databases (even from different data models or vendors) could be directly coupled. A single or multiple application servers (middle-tier agent) for the same database system may be needed. A single or multiple connections for a client could be constructed for retrieving data of remote predicates from different locations.
- Processing data from the database based on a single relation, and reliable queuing messaging system, increases the paradigm scalability, generality, and simplicity.
- The system efficiency could be achieved by reducing the interaction between the coupled systems. Therefore the approach addressed the following points:
 - The interchanged data between the three tiers might be restricted using additional dynamic filters in the system knowledge domain.
 - Each of the middle-tier agent and the reasoning system should consume only the data in which it interests.

- Using reliable delivery, the transferred group of data has to be emitted as a single networked message.
 - Asynchronous communication model must be established whenever possible.
- In the *iMCP*, the ATMS tightly coupled with the Rete network is one good candidate structure for building the reasoning system architecture. Incremental nature of that architecture provides the capability of the system to consider new available data. Consequently, the system can asynchronously work without any information about the database speed in executing or partitioning queries, and receiving data notifications. Second, the system has capability to revise its beliefs according to the arrived data to keep the reasoning up to date with the integrated database system.
 - Our proposed compromised system in coupling the Rete network with the ATMS labeling algorithm could increase the overall *iMCP* efficiency. In many applications, the formulated reasoning system has reserved the running space and time.
 - In a single-context search space reasoning, the Hindi system could be much better than the compromised system. While, in a multiple-context reasoning, the efficiency of the compromised system is close to the Hindi system.

6.2 Future Work

- Hindi (1994) had introduced the MCP with monotonic and non-monotonic reasoning capabilities. However, for simplicity, the *iMCP* can only perform

monotonic reasoning. Providing the *iMCP* with non-monotonic reasoning capabilities remains an issue for future work.

- Unlike Hindi (1994), in the *iMCP*, join operation is only performed on the client site. This is due to increasing the system scalability and simplicity. Moreover, in autonomous, heterogeneous, and distributed databases, it is hard to identify database subnets in the Rete network that each corresponds for a single logical database system. However, if the system only deals with one logical database, or the system can resolve ambiguity in determining the maximum database subnet, which belong to a single database system, then, it is possible to utilize the database join operation. In this case, we suggest executing database queries based on database joined subnets, while for scalability reasons, data notifications have to be still based on a singleton relation. This requires building of a SQL statement (which may contain join operation) that differs from the active *Selection* rule. On the other hand, the constructed paradigm must never permit dealing with non-unique data. The *count-column* strategy cannot resolve the redundancy in the joined selected retrieved data.
- The middle-tier agent's (or application server) main task is to handle the communications services between the database and the reasoning system in a scalable, reliable form. We suggest constructing a global shared inference engine that has the capability to reason the common inferences that is similar to the *blackboard* architecture for problem solving. The specific instance rules would be processed in the client's site, while the general common rules would be manipulated by the global inference engine. Cooperation must take place to solve the clients' demands on data. However, the design objective of the *blackboard* architecture is to solve a single problem over parallel machine architecture. In

contrast, the objective of the global inference engine is to solve different problem instances that share some of the general rules.

- In general, the efficiency of the distributed systems can be simply measured in terms of the amount of interaction between its distributed components. For the Internet applications this is even made more crucial because saving network bandwidth is the key requirement of a success of any electronic critical system. Therefore, we suggest that we have to build a general open interface for the reasoning systems (e.g., for ATMS coupled Rete network) similar to the ODBC and JDBC for the databases. The Java language has to adopt such interface with an implementation within its specification. This does not only reduce the amount of the logic (i.e., the *applet*) and the knowledge of the reasoning system to be downloaded, but also increases the application interoperability and portability.

References

- Brodie, M. 1988. *Future intelligent information systems: AI and database technologies working together. In Reading in AI and databases.* Morgan Kaufmann Publisher.
- Ceri, S. and P. Fraternali. 1997. *Designing Database Applications with Objects and Rules. The IDEA Methodology.* First edition. Addison-Wesely.
- Davis, R., B. Buchanan, and E. Shartiliffe. 1977. *Production Rules as a Representation for a Knowledge-Based Consultation Program.* Artificial Intelligence, 8(1): 15-45.
- a- de Kleer, J. 1986. *An Assumption-based TMS.* Artificial Intelligence, 28: 127-162.
- b- de Kleer, J. 1986. *Problem Solving with the ATMS.* Artificial Intelligence, 28: 197-224.
- Dresslar, O. 1990. *Problem Solving with the NM-ATMS.* In *Proceedings of European Conference on (ECAI)* : 253-358.
- Fernandes, A., N. Paton, M. Williams, and A. Bowles. 1992. *Approches to Deductive Object-Oriented Databases.* Information and Software Technology, 34(12): 787-803.
- Forgy, C. 1982. *A Fast Algorithm For the Many pattern/many Object Pattern Match Problem.* Artificial Intelligence, 19:17-37.
- Golshani, F. 1984. *Specification and Design of Expert Database Systems. In Expert Database Systems, Proceeding From the First-International Workshop:* 369-381.
- Gwertzman, J. and M. Seltzer. 1996. *World-Wide Web Cache Consistency.* On

<http://www.eess.harvard.edu/vino/web/usenix.html>.

- Hindi, K. 1994. *Integration Truth Maintenance System with Active Database Systems for Next Generation Cooperative Systems*. Ph.D. Thesis, Department of Computer Science, University of Exter, UK.
- Hindi, K. and B. Lings. 1994. *Using Truth Maintenance Systems To Solve The Data Consistency Problem*. In *Proceeding of the second International Conference on Cooperative Information Systems: CoolS-94*, University of Toronto Press, Canada.
- Johnston, R., V. Wolfe, M. Steele. 1996. *Real-time CORBA, Distributed Object Technology*. On <http://www.omg.org/docs/orbos/97-02-22.txt>.
- Mahmoud, K. 1997. *A Study of Efficiency of TMS-based Production Rule Systems*. Master Thesis, University of Jordan, Jordan.
- Morgue, G. and T. Chehire. 1991. *Efficiency of Production Systems when coupled with an assumption based Truth Maintenance System*. In *Proceedings of Ninth National Conference on Artificial Intelligence, AAAI*, 268-274.
- Ohta, Y. and K. Inoue. 1990. *A Forward-Chaining Multiple Context Reasoner And Its Application to Logic Design*. In *IEEE second International Conference on Tools for Artificial Intelligence*, 386-392.
- OMG Inc. *CORBAfacilities: Common Facilities Architecture*. <http://www.omg.org>. 1995.
- OMG Inc. *CORBAservices: Common Object Services Specifications*. <http://www.omg.org>. 1997.
- OMG Inc. *The Common Object Request Broker: Architecture and Specification*. <http://www.omg.org>. 1998.

- a- Oracle Corp. *Network Computing Architecture, an Oracle technical white paper.* <http://www.oracle.com>. 1997.
- b- Oracle Corp. *Oracle's JDBC Driver Accessing the Oracle RDBMS for Java,, an Oracle technical white paper.* <http://www.oracle.com>. 1997.
- a- Oracle Corp. *Capturing Electronic Commerce Opportunities, an Oracle business white paper.* <http://www.oracle.com>. 1998.
- b- Oracle Corp. *Oracle8i Advanced Queuing, Database-Integrated Message Queuing, an Oracle technical white paper.* <http://www.oracle.com>. 1998.
- c- Oracle Corp. *Oracle8i Features for Java, features overview.* <Http://www.oracle.com>. 1998.
- a- Oracle Corp. *Oracle8i Application Developer's Guide – Advanced Queuing.* <Http://www.oracle.com>. 1999.
- b- Oracle Corp. *Oracle8i Application Developer's Guide – Fundamentals.* <Http://www.oracle.com>. 1999.
- POSTGRES Group. 1992. *The POSTGRES reference manual version 4*. EECS Dept. University of California, Berkeley.
- Resnick, R. 1996. *Bring Distributed Objects to the World Wide Web*. On <http://www.interlog.com/~resnick/javacorba.html>.
- Roger, J. and M. Atkinson. 1997. *Distributed Objects*. On <http://bumetb.bu.edu/~bmce/cs776/projects/fall97/rogers/rogers.html>.
- Sampaio, P. and N. Paton. 1997. *Deductive Object-Oriented Database Systems: A Survey*. E-mail:[sampaio,norm]@cs.man.ac.uk.

- Smit, J. 1984. *Expert Database Systems: A Database Prospective In Expert Database Sysems, Proceeding From the First-International Workshop*: 3-15.
- a- Sun Microsystems, Inc. *Java Development Kit Documentation jdk1.2*.
<http://www.sun.com/software>. 1998.
- b- Sun Microsystems, Inc. *Java Dynamic Management Kit Documentation dm3.0*.
<http://www.sun.com/software>. 1998.
- Tambe, M. and P.Rosonbloom. 1992. *Uni-Rete: Specializing the Rete Match Algorithm for the Unique Attribute Representation*. On :
<http://www.isi.edu/soar/tambe/papers/92/unirete.ps>.
- Tanenbaum, A. 1995. *Distributed Operation Systems*. First edition. Prentice-Hall. USA.
- TIBCO Software Inc. *TIB/Connect for Oracle8 AQ, white paper*.
<http://www.tibco.com>. 1998.
- TIBCO Software Inc. *TIB/Rendezvous, white paper*. <http://www.tibco.com>. 1997.
- Worrell, K. 1994. *Invalidation in Large Scale Network Object Caches*. Master Thesis. University of Colorado.

Appendix A

Student Registration Guidance System - Knowledge Base

In this appendix, we list the general knowledge and the case specific data of the *Student Registration Guidance System (SRGS)*, which is described in section 4.4.1.

- 1) Definition : *dayOfWeek*(dayNo, dayCategory): type=premise.

Description: define lecture's days, which take the following forms:
sat-mon-wed, sun-tue, sat, sun, mon, tue, or wed.

Knowledge: *dayOfWeek*(1, 135). *dayOfWeek*(3, 135). *dayOfWeek*(5, 135).
dayOfWeek(2, 24). *dayOfWeek*(4, 24).
dayOfWeek(1, 1). *dayOfWeek*(2, 2). *dayOfWeek*(3, 3).
dayOfWeek(4, 4). *dayOfWeek*(5, 5).

- 2) Definition : *courseRegisterType*(registerType): type=assumption.

Description: determine the course registration type:

1. Regular registration.
2. Parallel with another one.
3. Alternate for a given course.

Knowledge: *courseRegisterType*(1). *courseRegisterType*(2). *courseRegisterType*(3).

- 3) Definition : *class*(crsNo, classNo, dayOfFinalExam, dayCategory,
fromHour, toHour): type=assumption,
address="middle_tier_addr".

Description: list all submitted classes' information.

Knowledge: # *University prerequisites*
class(101100, 1, 23, 135, 8.00, 9.00). *class*(101100, 2, 23, 135, 8.00, 9.00).
class(101100, 3, 23, 24, 11.00, 12.30). *class*(102100, 1, 22, 135, 8.00, 9.00).
class(102100, 2, 22, 24, 9.30, 11.00). *class*(102100, 3, 22, 24, 11.00, 12.30).
class(102100, 4, 22, 24, 12.30, 14.00). *class*(103100, 1, 22, 24, 9.30, 11.00).
class(103100, 2, 22, 24, 9.30, 11.00). *class*(103100, 3, 22, 135, 13.00, 14.00).
class(204100, 1, 25, 135, 8.00, 9.00). *class*(204100, 2, 24, 135, 10.00, 11.00).
class(204100, 3, 27, 135, 9.30, 11.00).
Mathematics courses
class(301101, 1, 24, 135, 8.00, 9.00). *class*(301101, 2, 24, 135, 8.00, 9.00).
class(301101, 3, 24, 135, 9.00, 10.00). *class*(301101, 4, 24, 135, 9.00, 10.00).
class(301101, 5, 24, 135, 14.00, 15.00). *class*(301102, 1, 22, 135, 8.00, 9.00).
class(301102, 2, 22, 135, 11.00, 12.00). *class*(301102, 3, 22, 24, 8.00, 9.30).
class(301201, 1, 25, 135, 8.00, 9.00). *class*(301201, 2, 23, 135, 11.00, 12.00).

class(301203, 1, 25, 135, 11.00, 12.00). *class*(301203, 2, 27, 24, 8.00, 9.30).
class(301319, 1, 25, 135, 13.00, 14.00). *class*(301329, 1, 25, 135, 13.00, 14.00).
class(301329, 2, 25, 24, 11.00, 12.30). *class*(301101, 1, 24, 135, 8.00, 9.00).
class(301101, 2, 24, 135, 8.00, 9.00). *class*(301339, 1, 26, 135, 10.00, 11.00).
class(301339, 2, 26, 24, 11.00, 12.30).
 # *Physics courses*
class(302101, 1, 22, 135, 8.00, 9.00). *class*(302101, 2, 22, 135, 11.00, 12.00).
class(302101, 3, 22, 24, 12.30, 14.00). *class*(302111, 1, 20, 1, 14.00, 17.00).
class(302111, 2, 20, 3, 14.00, 17.00). *class*(302102, 1, 24, 135, 9.00, 10.00).
class(302102, 2, 24, 135, 10.00, 11.00). *class*(302112, 1, 20, 5, 14.00, 17.00).
class(302329, 1, 26, 24, 8.00, 9.30). *class*(302329, 2, 25, 135, 14.00, 15.00).
 # *Chemistry courses*
class(303101, 1, 24, 135, 12.00, 13.00). *class*(303101, 2, 24, 24, 11.00, 12.30).
class(303102, 1, 26, 135, 11.00, 12.00). *class*(303102, 2, 26, 135, 13.00, 14.00).
class(303102, 3, 26, 135, 9.00, 10.00). *class*(303106, 1, 19, 2, 14.00, 17.00).
 # *Biology courses*
class(304101, 1, 25, 135, 10.00, 11.00). *class*(304101, 2, 25, 135, 11.00, 12.00).
 # *Geology courses*
class(305101, 1, 28, 24, 9.30, 11.00). *class*(305101, 2, 28, 24, 11.00, 12.30).
class(305111, 1, 20, 2, 14.00, 17.00). *class*(305102, 1, 24, 135, 9.00, 10.00).
class(305102, 2, 23, 24, 11.00, 12.30). *class*(305112, 1, 20, 4, 14.00, 17.00).
 # *Computer courses*
class(306101, 1, 25, 135, 8.00, 9.00). *class*(306101, 2, 27, 135, 9.00, 10.00).
class(306101, 3, 24, 135, 10.00, 11.00). *class*(306101, 4, 24, 24, 8.00, 9.30).
class(306101, 5, 27, 135, 9.00, 10.00). *class*(306101, 6, 24, 24, 8.00, 9.30).
class(306101, 7, 24, 135, 10.00, 11.00). *class*(306101, 8, 25, 135, 8.00, 9.00).
class(306102, 1, 25, 135, 8.00, 9.00). *class*(306102, 2, 27, 135, 9.00, 10.00).
class(306102, 3, 24, 135, 10.00, 11.00). *class*(306102, 4, 24, 24, 8.00, 9.30).
class(306104, 1, 26, 24, 9.30, 11.00). *class*(306104, 2, 23, 135, 11.00, 12.00).
class(306104, 3, 26, 24, 9.30, 11.00). *class*(306104, 4, 23, 135, 11.00, 12.00).
class(306105, 1, 26, 135, 12.00, 13.00). *class*(306105, 2, 23, 24, 11.00, 12.30).
class(306105, 3, 23, 24, 11.00, 12.30). *class*(306107, 1, 27, 135, 9.00, 10.00).
class(306111, 1, 24, 1, 10.00, 11.00). *class*(306111, 2, 24, 3, 10.00, 11.00).
class(306111, 3, 24, 5, 10.00, 11.00). *class*(306211, 1, 23, 135, 10.00, 11.00).
class(306211, 2, 23, 135, 10.00, 11.00). *class*(306212, 1, 24, 135, 10.00, 11.00).
class(306215, 1, 26, 24, 9.30, 11.00). *class*(306221, 1, 25, 135, 8.00, 9.00).
class(306221, 2, 26, 24, 9.30, 11.00). *class*(306323, 1, 25, 135, 13.00, 14.00).
class(306325, 1, 23, 135, 11.00, 12.00). *class*(306325, 2, 28, 24, 12.30, 14.00).
class(306331, 1, 27, 135, 9.00, 10.00). *class*(306333, 1, 27, 135, 9.00, 10.00).
class(306333, 1, 28, 24, 12.30, 14.00). *class*(306333, 2, 24, 135, 10.00, 11.00).
class(306339, 1, 26, 135, 10.00, 11.00). *class*(306343, 1, 25, 135, 8.00, 9.00).
class(306420, 1, 23, 24, 11.00, 12.30). *class*(306431, 1, 23, 24, 11.00, 12.30).
class(306432, 1, 24, 24, 8.00, 9.30). *class*(306433, 1, 27, 135, 9.00, 10.00).
class(306434, 1, 23, 135, 11.00, 12.00). *class*(306435, 1, 25, 135, 13.00, 14.00).
class(306437, 1, 26, 24, 9.30, 11.00). *class*(306443, 1, 25, 135, 8.00, 9.00).
class(306499, 1, 0, 0, 0.00, 0.00). *class*(306499, 2, 0, 0, 0.00, 0.00).
class(306499, 3, 0, 0, 0.00, 0.00).

- 4) Definition : *groupsPlan*(subjectNo, planYear, grpNo, totalWeights):
 type=premise.
 address="middle_tier_addr",
 filter= subjectNo == Subject_No.
 planYear == Plan_Year.

Description: divide each major's (or subject's) sheet plan into its main groups.

Knowledge: note: below, we list only the groups of the subject (306/1992):
groupsPlan(306, 1992, 1, 9). *groupsPlan*(306, 1992, 2, 15).
groupsPlan(306, 1992, 3, 30). *groupsPlan*(306, 1992, 4, 51).
groupsPlan(306, 1992, 5, 9). *groupsPlan*(306, 1992, 6, 9).
groupsPlan(306, 1992, 7, 12).

- 5) Definition : *sheetPlan*(subjectNo, planYear, grpNo, crsNo, weight,
 prerequisiteCrsNo1, prerequisiteCrsNo2, parallelInd):
 type=premise.
 address="middle_tier_addr",
 filter= subjectNo == Subject_No.
 planYear == Plan_Year.

Description: define all applicable courses and their prerequisites in each main group.

Knowledge: note: we list only the courses included in each group of the subject (0306):
groupsPlan(306, 1992, 1, 9).
sheetPlan(306, 1992, 1, 101100, 3, 0, 0, 0).
sheetPlan(306, 1992, 1, 102100, 3, 0, 0, 0).
sheetPlan(306, 1992, 1, 103100, 3, 0, 0, 0).
groupsPlan(306, 1992, 2, 15).
sheetPlan(306, 1992, 1, 101100, 3, 0, 0, 0).
sheetPlan(306, 1992, 1, 102100, 3, 0, 0, 0).
sheetPlan(306, 1992, 1, 103100, 3, 0, 0, 0).
sheetPlan(306, 1992, 2, 100, 3, 0, 0, 0).
groupsPlan(306, 1992, 3, 30).
sheetPlan(306, 1992, 3, 301101, 3, 0, 0, 0).
sheetPlan(306, 1992, 3, 301102, 3, 0301101, 0, 0).
sheetPlan(306, 1992, 3, 301151, 3, 0, 0, 0).
sheetPlan(306, 1992, 3, 302101, 3, 0, 0, 0).
sheetPlan(306, 1992, 3, 302102, 3, 302101, 0, 0).
sheetPlan(306, 1992, 3, 302111, 1, 302101, 0, 1).
sheetPlan(306, 1992, 3, 302112, 1, 302102, 302111, 1).
sheetPlan(306, 1992, 3, 303101, 3, 0, 0, 0).
sheetPlan(306, 1992, 3, 303102, 3, 303101, 0, 0).
sheetPlan(306, 1992, 3, 303106, 2, 303102, 0, 1).
sheetPlan(306, 1992, 3, 304101, 3, 0, 0, 0).
sheetPlan(306, 1992, 3, 304102, 3, 304101, 0, 0).
sheetPlan(306, 1992, 3, 304103, 1, 304101, 0, 1).
sheetPlan(306, 1992, 3, 304104, 1, 304102, 304103, 1).
sheetPlan(306, 1992, 3, 305101, 3, 0, 0, 0).
sheetPlan(306, 1992, 3, 305102, 3, 305101, 0, 0).
sheetPlan(306, 1992, 3, 305111, 1, 305101, 0, 1).
sheetPlan(306, 1992, 3, 305112, 1, 305102, 305111, 1).
sheetPlan(306, 1992, 3, 306101, 3, 0, 0, 0).

sheetPlan(306, 1992, 3, 306102, 3, 306101, 0, 0).
groupsPlan(306, 1992, 4, 51).
sheetPlan(306, 1992, 4, 306111, 1, 0, 0, 0).
sheetPlan(306, 1992, 4, 306211, 3, 306101, 0, 0).
sheetPlan(306, 1992, 4, 306212, 3, 306211, 0, 0).
sheetPlan(306, 1992, 4, 306221, 3, 0, 0, 0).
sheetPlan(306, 1992, 4, 306222, 3, 306211, 306221, 0).
sheetPlan(306, 1992, 4, 306323, 3, 306222, 0, 0).
sheetPlan(306, 1992, 4, 306325, 3, 306222, 0, 0).
sheetPlan(306, 1992, 4, 306331, 3, 306212, 0, 0).
sheetPlan(306, 1992, 4, 306333, 3, 306331, 0, 0).
sheetPlan(306, 1992, 4, 306335, 3, 306221, 0, 0).
sheetPlan(306, 1992, 4, 306341, 3, 306211, 301241, 0).
sheetPlan(306, 1992, 4, 306343, 3, 306341, 0, 0).
sheetPlan(306, 1992, 4, 306431, 3, 306331, 306323, 0).
sheetPlan(306, 1992, 4, 306432, 3, 306331, 306325, 0).
sheetPlan(306, 1992, 4, 306434, 3, 306325, 306333, 0).
sheetPlan(306, 1992, 4, 306433, 3, 306325, 306431, 0).
sheetPlan(306, 1992, 4, 306442, 3, 306331, 0, 0).
sheetPlan(306, 1992, 4, 306499, 2, 306222, 306331, 0).
groupsPlan(306, 1992, 5, 9).
sheetPlan(306, 1992, 5, 306214, 3, 306101, 0, 0).
sheetPlan(306, 1992, 5, 306215, 3, 306101, 0, 0).
sheetPlan(306, 1992, 5, 306326, 3, 306325, 0, 0).
sheetPlan(306, 1992, 5, 306345, 3, 301241, 0306331, 0).
sheetPlan(306, 1992, 5, 306346, 3, 301241, 0, 0).
groupsPlan(306, 1992, 6, 9).
sheetPlan(306, 1992, 6, 306420, 3, 306325, 0, 0).
sheetPlan(306, 1992, 6, 306421, 3, 306325, 0, 0).
sheetPlan(306, 1992, 6, 306435, 3, 306331, 0, 0).
sheetPlan(306, 1992, 6, 306436, 3, 306432, 0, 0).
sheetPlan(306, 1992, 6, 306437, 3, 306331, 0, 0).
sheetPlan(306, 1992, 6, 306438, 3, 306335, 0, 0).
sheetPlan(306, 1992, 6, 306441, 3, 306341, 0, 0).
sheetPlan(306, 1992, 6, 306443, 3, 306343, 0, 0).
sheetPlan(306, 1992, 6, 306445, 3, 0, 0, 0).
sheetPlan(306, 1992, 6, 306490, 3, 0, 0, 0).
groupsPlan(306, 1992, 7, 12).
sheetPlan(306, 1992, 7, 301201, 3, 301102, 0, 0).
sheetPlan(306, 1992, 7, 301203, 3, 301101, 0, 0).
sheetPlan(306, 1992, 7, 301231, 3, 0, 0, 0).
sheetPlan(306, 1992, 7, 301241, 3, 0, 0, 0).

- 6) Definition : *student*(stdNo, subjectNo, planYear): type=assumption,
 address="middle_tier_addr",
 filter= stdNo == Student_No,
 stdStatus != 0 .

Description: define student basic information.

Knowledge: e.g., *student*(950101, 0306, 1992).

- 7) Definition : *registeredGroups*(stdNo, grpNo, weight):

type=promise,
address="middle_tier_addr",
filter= stdNo == Student_No.

Description: present student's registered groups' summaries.

Knowledge: e.g., *registeredGroups*(950101, 1, 6).
registeredGroups(950101, 2, 3). *registeredGroups*(950101, 3, 6).
registeredGroups(950101, 4, 6). *registeredGroups*(950101, 5, 3).
registeredGroups(950101, 6, 0). *registeredGroups*(950101, 7, 0).

- 8) Definition : *registeredCourse*(stdNo, crsNo, grpNo, weight):

type=promise,
address="middle_tier_addr",
filter= stdNo == Student_No.

Description: present student's registered courses (i.e., group's details).

Knowledge: e.g., *registeredCourse*(950101, 101100, 1, 3).
registeredCourse(950101, 103100, 1, 3).
registeredCourse(950101, 303100, 2, 3).
registeredCourse(950101, 301101, 3, 3).
registeredCourse(950101, 302101, 3, 3).
registeredCourse(950101, 306101, 4, 3).
registeredCourse(950101, 306211, 4, 3).
registeredCourse(950101, 306214, 5, 3).

- 9) Definition : *alternativeCourse*(stdNo, crsNo, alternativeCrsNo):

type=assumption,
address="middle_tier_addr",
filter= stdNo == Student_No.

Description: determine student's alternate courses (exist for special circumstances).

Knowledge: e.g., *alternativeCourse*(960555, 306445, 306443).

- 10) Definition : *parallelReg*(stdNo, crsNo, prerequisiteCrsNo).

Description: generate all courses that must be taken in parallel with their prerequisites.

Knowledge: see 11-E, F, G, & H

- 11) Definition : *classTryReg*(stdNo, crsNo, grpNo, weight, classNo, dayOfFinalExam, dayCategory, fromHour, toHour, registerType): type=assumption.

Description: determine all candidate classes for each student that he/she can register in.

- A) university elective courses that end with 100.
- B) courses without prerequisite.
- C) courses with one prior prerequisite.
- D) courses with two prior prerequisites.
- E) courses with one parallel prerequisite.
- F) courses with one prior prerequisite and another parallel prerequisite.
- G) complementary to (E).
- H) courses with two parallel prerequisites.
- I) alternate courses.

Knowledge: A) *classTryReg*(StdNo, CrsNo, GrpNo, Weight, ClassNo, DayFE, DayCat, FromHr, ToHr, 1) :-

```

student(StdNo, SubjectNo, PlanYear),
sheetPlan(SubjectNo, PlanYear, GrpNo, 100, Weight, 0, 0,
ParallelInd),
class(CrsNo, ClassNo, DayFE, DayCat, FromHr, ToHr),
mod(CrsNo, 1000) == 100, floor(CrsNo/100000) != 1.

```

B) *classTryReg*(StdNo, CrsNo, GrpNo, Weight, ClassNo, DayFE, DayCat, FromHr, ToHr, 1) :-

```

student(StdNo, SubjectNo, PlanYear),
sheetPlan(SubjectNo, PlanYear, GrpNo, CrsNo, Weight, 0, 0,
ParallelInd),
class(CrsNo, ClassNo, DayFE, DayCat, FromHr, ToHr).

```

C) *classTryReg*(StdNo, CrsNo, GrpNo, Weight, ClassNo, DayFE, DayCat, FromHr, ToHr, 1) :-

```

student(StdNo, SubjectNo, PlanYear),
sheetPlan(SubjectNo, PlanYear, GrpNo, CrsNo, Weight,
PreCrsNo, 0, ParallelInd), PreCrsNo != 0,
registeredCourse(StdNo, PreCrsNo, GrpNo2, Weight2),
class(CrsNo, ClassNo, DayFE, DayCat, FromHr, ToHr).

```

D) *classTryReg*(StdNo, CrsNo, GrpNo, Weight, ClassNo, DayFE, DayCat, FromHr, ToHr, 1) :-

```

student(StdNo, SubjectNo, PlanYear),
registeredCourse(StdNo, PreCrsNo1, GrpNo2, Weight2),
registeredCourse(StdNo, PreCrsNo2, GrpNo3, Weight3),
sheetPlan(SubjectNo, PlanYear, GrpNo, CrsNo, Weight,
PreCrsNo1, PreCrsNo2, ParallelInd), PreCrsNo2 != 0,
class(CrsNo, ClassNo, DayFE, DayCat, FromHr, ToHr).

```

E) *classTryReg*(StdNo, CrsNo, GrpNo, Weight, ClassNo, DayFE, DayCat, FromHr, ToHr, 2),

```

parallelReg(StdNo, CrsNo, PreCrsNo1) :-
student(StdNo, SubjectNo, PlanYear),
sheetPlan(SubjectNo, PlanYear, GrpNo, CrsNo, Weight,
PreCrsNo1, 0, 1), PreCrsNo1 != 0,
classTryReg(StdNo, PreCrsNo1, GrpNo2, Weight2,
ClassNo2, DayFE2, DayCat2, FromHr2, ToHr2, Type2),
class(CrsNo, ClassNo, DayFE, DayCat, FromHr, ToHr).

```

- F) *classTryReg*(StdNo, CrsNo, GrpNo, Weight, ClassNo, DayFE, DayCat, FromHr, ToHr, 2),
parallelReg(StdNo, CrsNo, PreCrsNo2) :-
student(StdNo, SubjectNo, PlanYear),
registeredCourse(StdNo, PreCrsNo1, GrpNo2, Weight2),
sheetPlan(SubjectNo, PlanYear, GrpNo, CrsNo, Weight, PreCrsNo1, PreCrsNo2, 1), PreCrsNo2 != 0,
classTryReg(StdNo, PreCrsNo2, GrpNo3, Weight3, ClassNo3, DayFE3, DayCat3, FromHr3, ToHr3, Type3),
class(CrsNo, ClassNo, DayFE, DayCat, FromHr, ToHr).
- G) *classTryReg*(StdNo, CrsNo, GrpNo, Weight, ClassNo, DayFE, DayCat, FromHr, ToHr, 2),
parallelReg(StdNo, CrsNo, PreCrsNo1) :-
student(StdNo, SubjectNo, PlanYear),
registeredCourse(StdNo, PreCrsNo2, GrpNo3, Weight3),
sheetPlan(SubjectNo, PlanYear, GrpNo, CrsNo, Weight, PreCrsNo1, PreCrsNo2, 1), PreCrsNo2 != 0,
classTryReg(StdNo, PreCrsNo1, GrpNo2, Weight2, ClassNo2, DayFE2, DayCat2, FromHr2, ToHr2, Type2),
class(CrsNo, ClassNo, DayFE, DayCat, FromHr, ToHr).
- H) *classTryReg*(StdNo, CrsNo, GrpNo, Weight, ClassNo, DayFE, DayCat, FromHr, ToHr, 2),
parallelReg(StdNo, CrsNo, PreCrsNo1),
parallelReg(StdNo, CrsNo, PreCrsNo2) :-
student(StdNo, SubjectNo, PlanYear),
sheetPlan(SubjectNo, PlanYear, GrpNo, CrsNo, Weight, PreCrsNo1, PreCrsNo2, 1), PreCrsNo2 != 0,
classTryReg(StdNo, PreCrsNo1, GrpNo2, Weight2, ClassNo2, DayFE2, DayCat2, FromHr2, ToHr2, Type2),
classTryReg(StdNo, PreCrsNo2, GrpNo3, Weight3, ClassNo3, DayFE3, DayCat3, FromHr3, ToHr3, Type3),
class(CrsNo, ClassNo, DayFE, DayCat, FromHr, ToHr).
- I) *classTryReg*(StdNo, AlternativeCrsNo, GrpNo, Weight, ClassNo, DayFE, DayCat, FromHr, ToHr, 3) :-
student(StdNo, SubjectNo, PlanYear),
alternativeCourse(StdNo, CrsNo, AlternativeCrsNo),
sheetPlan(SubjectNo, PlanYear, GrpNo, CrsNo, Weight, PreCrsNo1, PreCrsNo2, ParallelInd),
class(AlternativeCrsNo, ClassNo, DayFE, DayCat, FromHr, ToHr).

12) Definition : *sugSchedule*(stdNo, crsNo1, crsNo2, crsNo3, crsNo4, crsNo5, crsNo6, crsNo7): type=promise.

Description: an optional relation that narrows the search in the listed courses only.

Knowledge: sec 13.

13) Definition : *suggestedCourseNotFound()*.

Description: check for existence of all candidate classes in the suggested courses list. This check is useless if relation 12 does not exist in student case data.

Knowledge: *suggestedCourseNotFound()* :-

classTryReg(StdNo, CrsNo, GrpNo, Weight,
ClassNo, DayFE, DayCat, FromHr, ToHr, Type),
Weight > 0,
sugSchedule(StdNo, CrsNo1, CrsNo2, CrsNo3, CrsNo4,
CrsNo5, CrsNo6, CrsNo7),
sign(abs(CrsNo - CrsNo1)) + sign(abs(CrsNo - CrsNo2)) +
sign(abs(CrsNo - CrsNo3)) + sign(abs(CrsNo - CrsNo4)) +
sign(abs(CrsNo - CrsNo5)) + sign(abs(CrsNo - CrsNo6)) +
sign(abs(CrsNo - CrsNo7)) = = 7.

14) Definition : *alreadyReg()*.

Description: check if the selected course (or alternate course) is already registered.

Knowledge: A) *alreadyReg()* :-

classTryReg(StdNo, CrsNo, GrpNo1, Weight1,
ClassNo1, DayFE1, DayCat1, FromHr1, ToHr1, Type1),
Weight1 > 0,
registeredCourse(StdNo, CrsNo, GrpNo2, Weight2).

B) *alreadyReg()* :-

alternativeCourse(StdNo, CrsNo, AlternativeCrsNo),
registeredCourse(StdNo, CrsNo, GrpNo2, Weight2).

15) Definition : *sameCourse()*.

Description: prohibit any two equivalent classes to be in the same schedule. If they are:

- A) from the same course.
- B) alternatives to each others.
- C) alternatives to another common course.

Knowledge: A) *sameCourse()* :-

classTryReg(StdNo, CrsNo, GrpNo1, Weight1,
ClassNo1, DayFE1, DayCat1, FromHr1, ToHr1, Type1),
classTryReg(StdNo, CrsNo, GrpNo2, Weight2,
ClassNo2, DayFE2, DayCat2, FromHr2, ToHr2, Type2),
ClassNo1 < ClassNo2.

B) *sameCourse()* :-

alternativeCourse(StdNo, CrsNo, AlternativeCrsNo),
classTryReg(StdNo, CrsNo, GrpNo2, Weight2,
ClassNo2, DayFE2, DayCat2, FromHr2, ToHr2, Type2),
Type2 < 3.

C) *sameCourse()* :-

alternativeCourse(StdNo, CrsNo, AlternativeCrsNo1),
alternativeCourse(StdNo, CrsNo, AlternativeCrsNo2),
 alternativeCrsNo1 < AlternativeCrsNo2.

16) Definition : *conflictTime()*.

Description: guarantee that no time conflict in the same schedule.

Knowledge: *conflictTime()* :-

classTryReg(StdNo, CrsNo1, GrpNo1, Weight1,
 ClassNo1, DayFE1, DayCat1, FromHr1, ToHr1, Type1),
dayOfWeek(DayNo, DayCat1),
dayOfWeek(DayNo, DayCat2),
classTryReg(StdNo, CrsNo2, GrpNo2, Weight2,
 ClassNo2, DayFE2, DayCat2, FromHr2, ToHr2, Type2),
 CrsNo1 < CrsNo2, FromHr1 < ToHr2,
 FromHr2 < ToHr1.

17) Definition : *threeFinalExamsInTheSameDay()*.

Description: ensure that no three classes their final exams are in the same day.

Knowledge: *threeFinalExamsInTheSameDay()* :-

classTryReg(StdNo, CrsNo1, GrpNo1, Weight1,
 ClassNo1, DayFE, DayCat1, FromHr1, ToHr1, Type1),
classTryReg(StdNo, CrsNo2, GrpNo2, Weight2,
 ClassNo2, DayFE, DayCat2, FromHr2, ToHr2, Type2),
classTryReg(StdNo, CrsNo3, GrpNo3, Weight3,
 ClassNo3, DayFE, DayCat3, FromHr3, ToHr3, Type3),
 CrsNo1 < CrsNo2,
 CrsNo2 < CrsNo3.

18) Definition : *classHighOverhead()*.

Description: refuse any schedule that obligates the student to :

- A) come at 8:00 every morning.
- B) leave afternoon every day.
- C) get three successive classes without rest.

Knowledge: A) *classHighOverhead()* :-

classTryReg(StdNo, CrsNo1, GrpNo1, Weight1,
 ClassNo1, DayFE1, 135, 8.00, ToHr1, Type1),
classTryReg(StdNo, CrsNo2, GrpNo2, Weight2,
 ClassNo2, DayFE2, 24, 8.00, ToHr2, Type2),
 CrsNo1 != CrsNo2.

B) *classHighOverhead()* :-

classTryReg(StdNo, CrsNo1, GrpNo1, Weight1,

ClassNo1, DayFE1, 135, FromHr1, ToHr1, Type1),
classTryReg(StdNo, CrsNo2, GrpNo2, Weight2,
 ClassNo2, DayFE2, 24, FromHr2, ToHr2, Type2),
 CrsNo1 != CrsNo2, ToHr1 > 13, ToHr2 > 13.

C) *classHighOverhead*() :-
classTryReg(StdNo, CrsNo1, GrpNo1, Weight1,
 ClassNo1, DayFE1, DayCat1, FromHr1, ToHr1, Type1).
dayOfWeek(DayNo, DayCat1),
dayOfWeek(DayNo, DayCat2),
classTryReg(StdNo, CrsNo2, GrpNo2, Weight2,
 ClassNo2, DayFE2, DayCat2, ToHr1, ToHr2, Type2).
dayOfWeek(DayNo, DayCat3),
classTryReg(StdNo, CrsNo3, GrpNo3, Weight3,
 ClassNo3, DayFE3, DayCat3, ToHr2, ToHr3, Type3).

19) Definition : *courseTryReg*(stdNo, crsNo, grpNo, weight).

Description: generate all courses that can be registered, based on *classTryReg* relation.

- A) define a null course (0).
- B) determine regular courses.
- C) determine substituted courses.

Note that, this relation generates multiple-context search spaces.

Knowledge: A) *courseTryReg*(StdNo, 0, 0, 0) :-
student(StdNo, SubjectNo, PlanYear).

B) *courseTryReg*(StdNo, CrsNo, GrpNo, Weight) :-
classTryReg(StdNo, CrsNo, GrpNo, Weight,
 ClassNo, DayFE, DayCat, FromHr, ToHr, Type).
registerType(Type), Type < 3.

C) *courseTryReg*(StdNo, CrsNo, GrpNo, Weight) :-
classTryReg(StdNo, AlternativeCrsNo, GrpNo, Weight,
 ClassNo, DayFE, DayCat, FromHr, ToHr, 3).
registerType(3).
alternativeCourse(StdNo, CrsNo, AlternativeCrsNo).

20) Definition : *groupOverflow*().

Description: discard any schedule that causes an overflow in one of main subject's groups.

This is due to trying to register:

- A) a course into a certain group.
- B) two courses into a certain group.
- C) three courses into a certain group.
- D) four courses into a given group.
- E) five courses into a given group.
- F) six courses into a given group.
- G) seven courses into a given group.

Knowledge: A) *groupOverflow*():-
student(StdNo, SubjectNo, PlanYear),

groupsPlan(SubjectNo, PlanYear, GrpNo, PlanTotal),
registeredGroups(StdNo, GrpNo, StudentTotal),
courseTryReg(StdNo, CrsNo, GrpNo, Weight),
 StudentTotal \geq PlanTotal.

B) *groupOverflow*() :-

student(StdNo, SubjectNo, PlanYear),
groupsPlan(SubjectNo, PlanYear, GrpNo, PlanTotal),
registeredGroups(StdNo, GrpNo, StudentTotal),
courseTryReg(StdNo, CrsNo1, GrpNo, Weight1),
courseTryReg(StdNo, CrsNo2, GrpNo, Weight2),
 CrsNo1 < CrsNo2,
 StudentTotal + Weight1 + Weight2 -
 min(Weight1, Weight2) \geq PlanTotal.

C) *groupOverflow*() :-

student(StdNo, SubjectNo, PlanYear),
groupsPlan(SubjectNo, PlanYear, GrpNo, PlanTotal),
registeredGroups(StdNo, GrpNo, StudentTotal),
courseTryReg(StdNo, CrsNo1, GrpNo, Weight1),
courseTryReg(StdNo, CrsNo2, GrpNo, Weight2),
courseTryReg(StdNo, CrsNo3, GrpNo, Weight3),
 CrsNo1 < CrsNo2, CrsNo2 < CrsNo3,
 StudentTotal + Weight1 + Weight2 + Weight3 -
 min(min(Weight1, Weight2), Weight3) \geq PlanTotal.

D) *groupOverflow*() :-

student(StdNo, SubjectNo, PlanYear),
groupsPlan(SubjectNo, PlanYear, GrpNo, PlanTotal),
registeredGroups(StdNo, GrpNo, StudentTotal),
courseTryReg(StdNo, CrsNo1, GrpNo, Weight1),
courseTryReg(StdNo, CrsNo2, GrpNo, Weight2),
courseTryReg(StdNo, CrsNo3, GrpNo, Weight3),
courseTryReg(StdNo, CrsNo4, GrpNo, Weight4),
 CrsNo1 < CrsNo2, CrsNo2 < CrsNo3, CrsNo3 < CrsNo4,
 StudentTotal + Weight1 + Weight2 + Weight3 + Weight4 -
 min(min(Weight1, Weight2), min(Weight3, Weight4)) \geq
 PlanTotal.

491833

E) *groupOverflow*() :-

student(StdNo, SubjectNo, PlanYear),
groupsPlan(SubjectNo, PlanYear, GrpNo, PlanTotal),
registeredGroups(StdNo, GrpNo, StudentTotal),
courseTryReg(StdNo, CrsNo1, GrpNo, Weight1),
courseTryReg(StdNo, CrsNo2, GrpNo, Weight2),
courseTryReg(StdNo, CrsNo3, GrpNo, Weight3),
courseTryReg(StdNo, CrsNo4, GrpNo, Weight4),
courseTryReg(StdNo, CrsNo5, GrpNo, Weight5),
 CrsNo1 < CrsNo2, CrsNo2 < CrsNo3,
 CrsNo3 < CrsNo4, CrsNo4 < CrsNo5,

StudentTotal + Weight1 + Weight2 + Weight3 + Weight4 +
 Weight5 -

$\min(\min(\text{Weight1}, \text{Weight2}),$
 $\min(\min(\text{Weight3}, \text{Weight4}), \text{Weight5})) \geq \text{PlanTotal}.$

F) *groupOverflow*():-

student(StdNo, SubjectNo, PlanYear),
groupsPlan(SubjectNo, PlanYear, GrpNo, PlanTotal),
registeredGroups(StdNo, GrpNo, StudentTotal),
courseTryReg(StdNo, CrsNo1, GrpNo, Weight1),
courseTryReg(StdNo, CrsNo2, GrpNo, Weight2),
courseTryReg(StdNo, CrsNo3, GrpNo, Weight3),
courseTryReg(StdNo, CrsNo4, GrpNo, Weight4),
courseTryReg(StdNo, CrsNo5, GrpNo, Weight5),
courseTryReg(StdNo, CrsNo6, GrpNo, Weight6),
CrsNo1 < CrsNo2, CrsNo2 < CrsNo3, CrsNo3 < CrsNo4,
CrsNo4 < CrsNo5, CrsNo5 < CrsNo6,
StudentTotal + Weight1 + Weight2 + Weight3 + Weight4 +
Weight5 + Weight6 -
 $\min(\min(\min(\text{Weight1}, \text{Weight2}),$
 $\min(\text{Weight3}, \text{Weight4})),$
 $\min(\text{Weight5}, \text{Weight6})) \geq \text{PlanTotal}.$

G) *GroupOverflow*():-

student(StdNo, SubjectNo, PlanYear),
groupsPlan(SubjectNo, PlanYear, GrpNo, PlanTotal),
registeredGroups(StdNo, GrpNo, StudentTotal),
courseTryReg(StdNo, CrsNo1, GrpNo, Weight1),
courseTryReg(StdNo, CrsNo2, GrpNo, Weight2),
courseTryReg(StdNo, CrsNo3, GrpNo, Weight3),
courseTryReg(StdNo, CrsNo4, GrpNo, Weight4),
courseTryReg(StdNo, CrsNo5, GrpNo, Weight5),
courseTryReg(StdNo, CrsNo6, GrpNo, Weight6),
courseTryReg(StdNo, CrsNo7, GrpNo, Weight7),
CrsNo1 < CrsNo2, CrsNo2 < CrsNo3, CrsNo3 < CrsNo4,
CrsNo4 < CrsNo5, CrsNo5 < CrsNo6, CrsNo6 < CrsNo7,
StudentTotal + Weight1 + Weight2 + Weight3 + Weight4 +
Weight5 + Weight6 + Weight7 -
 $\min(\min(\min(\text{Weight1}, \text{Weight2}),$
 $\min(\text{Weight3}, \text{Weight4})),$
 $\min(\min(\text{Weight5}, \text{Weight6}),$
 $\text{Weight7})) \geq \text{PlanTotal}.$

21) Definition : *courseOverhead*().

Description: refuse any schedule that includes:

- A) three courses from other collages.
- B) five courses from the student's major.

Knowledge: A) *courseOverhead*():-

student(StdNo, SubjectNo, PlanYear),
courseTryReg(StdNo, CrsNo1, GrpNo1, Weight1), Weight1 > 0,
 $\text{floor}(\text{CrsNo1}/100000) \neq \text{floor}(\text{SubjectNo}/100),$

courseTryReg(StdNo, CrsNo2, GrpNo2, Weight2), Weight2>0,
 floor(CrsNo2/100000) != floor(SubjectNo/100),
courseTryReg(StdNo, CrsNo3, GrpNo3, Weight3), Weight3>0,
 floor(CrsNo3/100000) != floor(SubjectNo/100),
 CrsNo1 < CrsNo2, CrsNo2 < CrsNo3.

B) *courseOverhead*():-

student(StdNo, SubjectNo, PlanYear),
courseTryReg(StdNo, CrsNo1, GrpNo1, Weight1).
 floor(CrsNo1/1000) == SubjectNo,
courseTryReg(StdNo, CrsNo2, GrpNo2, Weight2).
 floor(CrsNo2/1000) == SubjectNo,
courseTryReg(StdNo, CrsNo3, GrpNo3, Weight3).
 floor(CrsNo3/1000) == SubjectNo,
courseTryReg(StdNo, CrsNo4, GrpNo4, Weight4).
 floor(CrsNo4/1000) == SubjectNo,
courseTryReg(StdNo, CrsNo5, GrpNo5, Weight5).
 floor(CrsNo5/1000) == SubjectNo.
 CrsNo1 < CrsNo2, CrsNo2 < CrsNo3, CrsNo3 < CrsNo4,
 CrsNo4 < CrsNo5.

22) Definition : *regSchedule*(stdNo, crsNo1, crsNo2, crsNo3, crsNo4, crsNo5, crsNo6, crsNo7).

Description: generate schedules in term of courses. where the schedule label determine the actual structure in terms of classes.

Knowledge: *regSchedule*(StdNo, CrsNo1, CrsNo2, CrsNo3, CrsNo4, CrsNo5, CrsNo6, CrsNo7) :-

courseTryReg(StdNo, CrsNo1, GrpNo1, Weight1),
courseTryReg(StdNo, CrsNo2, GrpNo2, Weight2),
courseTryReg(StdNo, CrsNo3, GrpNo3, Weight3),
courseTryReg(StdNo, CrsNo4, GrpNo4, Weight4),
courseTryReg(StdNo, CrsNo5, GrpNo5, Weight5),
courseTryReg(StdNo, CrsNo6, GrpNo6, Weight6),
courseTryReg(StdNo, CrsNo7, GrpNo7, Weight7),
 Weight1 > 0, Weight2 > 0, Weight3 > 0,
 Weight4 > 0, Weight5 > 0,
 CrsNo1+(1-sign(CrsNo1))*9999001 <
 CrsNo2+(1-sign(CrsNo2))*9999002 ,
 CrsNo2+(1-sign(CrsNo2))*9999002 <
 CrsNo3+(1-sign(CrsNo3))*9999003 ,
 CrsNo3+(1-sign(CrsNo3))*9999003 <
 CrsNo4+(1-sign(CrsNo4))*9999004 ,
 CrsNo4+(1-sign(CrsNo4))*9999004 <
 CrsNo5+(1-sign(CrsNo5))*9999005 ,
 CrsNo5+(1-sign(CrsNo5))*9999005 <
 CrsNo6+(1-sign(CrsNo6))*9999006 ,
 CrsNo6+(1-sign(CrsNo6))*9999006 <
 CrsNo7+(1-sign(CrsNo7))*9999007 ,
 Weight1 + Weight2 + Weight3 +
 Weight4 + Weight5 + Weight6 + Weight7 >= 15,
 Weight1 + Weight2 + Weight3 +

$$\text{Weight4} + \text{Weight5} + \text{Weight6} + \text{Weight7} \leq 18.$$

23) Definition : *parallelCourseNotFound()*.

Description: discard any schedule if it includes a course without its parallel prerequisite.

Knowledge: *parallelCourseNotFound()*:-

```
parallelReg(StdNo, CrsNo, PreCrsNo),
regSchedule(StdNo, CrsNo1, CrsNo2, CrsNo3, CrsNo4,
             CrsNo5, CrsNo6, CrsNo7),
sign(abs(CrsNo - CrsNo1)) + sign(abs(CrsNo - CrsNo2)) +
sign(abs(CrsNo - CrsNo3)) + sign(abs(CrsNo - CrsNo4)) +
sign(abs(CrsNo - CrsNo5)) + sign(abs(CrsNo - CrsNo6)) +
sign(abs(CrsNo - CrsNo7)) == 6, # CrsNo belongs to courses
sign(abs(PreCrsNo - CrsNo1)) + sign(abs(PreCrsNo - CrsNo2)) +
sign(abs(PreCrsNo - CrsNo3)) + sign(abs(PreCrsNo - CrsNo4)) +
sign(abs(PreCrsNo - CrsNo5)) + sign(abs(PreCrsNo - CrsNo6)) +
sign(abs(PreCrsNo - CrsNo7)) == 7. # PreCrsNo not found
```

Appendix B

Experiments' Case Specific Data

Experiment A

student(980001, 306, 1992).

registeredGroups(980001, 1, 3).

registeredCourse(980001, 102100, 1, 3).

registeredGroups(980001, 2, 0).

registeredGroups(980001, 3, 9).

registeredCourse(980001, 304101, 3, 3).

registeredCourse(980001, 305101, 3, 3).

registeredCourse(980001, 306101, 3, 3).

registeredGroups(980001, 4, 0).

registeredGroups(980001, 5, 0).

registeredGroups(980001, 6, 0).

registeredGroups(980001, 7, 0).

01 - sugSchedule(980001, 101100, 301101, 302101, 302111, 303101, 306102, 0).

02 - sugSchedule(980001, 101100, 204100, 301101, 302101, 302111, 303101, 0).

03 - sugSchedule(980001, 101100, 204100, 301101, 302101, 302111, 306211, 0).

04 - sugSchedule(980001, 204100, 301101, 302101, 302111, 305102, 306102, 0).

05 - sugSchedule(980001, 204100, 301101, 302101, 302111, 306102, 306211, 0).

06 - sugSchedule(980001, 103100, 301101, 302101, 305102, 305111, 204100, 0).

07 - sugSchedule(980001, 301101, 302101, 302111, 304102, 306102, 306111, 0).

08 - sugSchedule(980001, 101100, 103100, 301101, 302101, 305102, 0, 0).

09 - sugSchedule(980001, 101100, 103100, 204100, 301101, 302101, 305102, 0).

10 - sugSchedule(980001, 204100, 301101, 302101, 302111, 305111, 306111,
306102).

Experiment B-01

student(970001, 306, 1992).

registeredGroups(970001, 1, 9).

registeredCourse(970001, 101100, 1, 3).

registeredCourse(970001, 102100, 1, 3).

registeredCourse(970001, 103100, 1, 3).

registeredGroups(970001, 2, 3).

registeredCourse(970001, 204100, 2, 3).

registeredGroups(970001, 3, 20).

registeredCourse(970001, 301101, 3, 3).

registeredCourse(970001, 301102, 3, 3).

registeredCourse(970001, 302101, 3, 3).

registeredCourse(970001, 302102, 3, 3).

registeredCourse(970001, 302111, 3, 1).

registeredCourse(970001, 302112, 3, 1).

registeredCourse(970001, 306101, 3, 3).

registeredCourse(970001, 306102, 3, 3).
registeredGroups(970001, 4, 10).
registeredCourse(970001, 306111, 4, 1).
registeredCourse(970001, 306211, 4, 3).
registeredCourse(970001, 306221, 4, 3).
registeredCourse(970001, 306222, 4, 3).
registeredGroups(970001, 5, 6).
registeredCourse(970001, 306214, 5, 3).
registeredCourse(970001, 306215, 5, 3).
registeredGroups(970001, 6, 0).
registeredGroups(970001, 7, 3).
registeredCourse(970001, 301241, 7, 3).
alternativeCourse(970001, 306323, 301319).
alternativeCourse(970001, 306323, 301329).
alternativeCourse(970001, 306323, 301339).

Experiment B-02

student(970002, 306, 1992).
registeredGroups(970002, 1, 6).
registeredCourse(970002, 101100, 1, 3).
registeredCourse(970002, 103100, 1, 3).
registeredGroups(970002, 2, 3).
registeredCourse(970002, 202100, 2, 3).
registeredGroups(970002, 3, 25).
registeredCourse(970002, 301101, 3, 3).
registeredCourse(970002, 301102, 3, 3).
registeredCourse(970002, 302101, 3, 3).
registeredCourse(970002, 302102, 3, 3).
registeredCourse(970002, 302111, 3, 1).
registeredCourse(970002, 303101, 3, 3).
registeredCourse(970002, 304101, 3, 3).
registeredCourse(970002, 306101, 3, 3).
registeredCourse(970002, 306102, 3, 3).
registeredGroups(970002, 4, 10).
registeredCourse(970002, 306111, 4, 1).
registeredCourse(970002, 306211, 4, 3).
registeredCourse(970002, 306221, 4, 3).
registeredCourse(970002, 306222, 4, 3).
registeredGroups(970002, 5, 3).
registeredCourse(970002, 306214, 5, 3).
registeredGroups(970002, 6, 0).
registeredGroups(970002, 7, 6).
registeredCourse(970002, 301203, 7, 3).
registeredCourse(970002, 301241, 7, 3).

Experiment B-03

student(970003, 306, 1992).

registeredGroups(970003, 1, 6).
 registeredCourse(970003, 101100, 1, 3).
 registeredCourse(970003, 103100, 1, 3).
registeredGroups(970003, 2, 3).
 registeredCourse(970003, 204100, 2, 3).
registeredGroups(970003, 3, 28).
 registeredCourse(970003, 301101, 3, 3).
 registeredCourse(970003, 301102, 3, 3).
 registeredCourse(970003, 302101, 3, 3).
 registeredCourse(970003, 302102, 3, 3).
 registeredCourse(970003, 302111, 3, 1).
 registeredCourse(970003, 303101, 3, 3).
 registeredCourse(970003, 304101, 3, 3).
 registeredCourse(970003, 304102, 3, 3).
 registeredCourse(970003, 306101, 3, 3).
 registeredCourse(970003, 306102, 3, 3).
registeredGroups(970003, 4, 16).
 registeredCourse(970003, 306111, 4, 1).
 registeredCourse(970003, 306211, 4, 3).
 registeredCourse(970003, 306212, 4, 3).
 registeredCourse(970003, 306221, 4, 3).
 registeredCourse(970003, 306222, 4, 3).
 registeredCourse(970003, 306331, 4, 3).
registeredGroups(970003, 5, 0).
registeredGroups(970003, 6, 0).
registeredGroups(970003, 7, 6).
 registeredCourse(970003, 301203, 7, 3).
 registeredCourse(970003, 301241, 7, 3).

Experiment B-04

student(970004, 306, 1992).
registeredGroups(970004, 1, 6).
 registeredCourse(970004, 101100, 1, 3).
 registeredCourse(970004, 102100, 1, 3).
registeredGroups(970004, 2, 3).
 registeredCourse(970004, 204100, 2, 3).
registeredGroups(970004, 3, 28).
 registeredCourse(970004, 301101, 3, 3).
 registeredCourse(970004, 301102, 3, 3).
 registeredCourse(970004, 302101, 3, 3).
 registeredCourse(970004, 302102, 3, 3).
 registeredCourse(970004, 302111, 3, 1).
 registeredCourse(970004, 303101, 3, 3).
 registeredCourse(970004, 304101, 3, 3).
 registeredCourse(970004, 305101, 3, 3).
 registeredCourse(970004, 306101, 3, 3).
 registeredCourse(970004, 306102, 3, 3).
registeredGroups(970004, 4, 10).

registeredCourse(970004, 306111, 4, 1).
registeredCourse(970004, 306211, 4, 3).
registeredCourse(970004, 306221, 4, 3).
registeredCourse(970004, 306222, 4, 3).
registeredGroups(970004, 5, 3).
registeredCourse(970004, 306215, 5, 3).
registeredGroups(970004, 6, 0).
registeredGroups(970004, 7, 3).
registeredCourse(970004, 301241, 7, 3).

Experiment B-05

student(970005, 306, 1992).
registeredGroups(970005, 1, 9).
registeredCourse(970005, 101100, 1, 3).
registeredCourse(970005, 102100, 1, 3).
registeredCourse(970005, 103100, 1, 3).
registeredGroups(970005, 2, 3).
registeredCourse(970005, 204100, 2, 3).
registeredGroups(970005, 3, 24).
registeredCourse(970005, 301101, 3, 3).
registeredCourse(970005, 301102, 3, 3).
registeredCourse(970005, 302101, 3, 3).
registeredCourse(970005, 302102, 3, 3).
registeredCourse(970005, 304101, 3, 3).
registeredCourse(970005, 305101, 3, 3).
registeredCourse(970005, 306101, 3, 3).
registeredCourse(970005, 306102, 3, 3).
registeredGroups(970005, 4, 10).
registeredCourse(970005, 306111, 4, 1).
registeredCourse(970005, 306211, 4, 3).
registeredCourse(970005, 306221, 4, 3).
registeredCourse(970005, 306222, 4, 3).
registeredGroups(970005, 5, 3).
registeredCourse(970005, 306215, 5, 3).
registeredGroups(970005, 6, 0).
registeredGroups(970005, 7, 3).
registeredCourse(970005, 301201, 7, 3).

Experiment B-06

student(970006, 306, 1992).
registeredGroups(970006, 1, 9).
registeredCourse(970006, 101100, 1, 3).
registeredCourse(970006, 102100, 1, 3).
registeredCourse(970006, 103100, 1, 3).
registeredGroups(970006, 2, 0).
registeredGroups(970006, 3, 20).
registeredCourse(970006, 301101, 3, 3).

registeredCourse(970006, 301102, 3, 3).
registeredCourse(970006, 302101, 3, 3).
registeredCourse(970006, 302102, 3, 3).
registeredCourse(970006, 302111, 3, 1).
registeredCourse(970006, 302112, 3, 1).
registeredCourse(970006, 306101, 3, 3).
registeredCourse(970006, 306102, 3, 3).
registeredGroups(970006, 4, 13).
registeredCourse(970006, 306111, 4, 1).
registeredCourse(970006, 306211, 4, 3).
registeredCourse(970006, 306212, 4, 3).
registeredCourse(970006, 306221, 4, 3).
registeredCourse(970006, 306222, 4, 3).
registeredGroups(970006, 5, 0).
registeredGroups(970006, 6, 0).
registeredGroups(970006, 7, 3).
registeredCourse(970006, 301201, 7, 3).
alternativeCourse(970006, 306323, 301319).
alternativeCourse(970006, 306323, 302329).

Experiment B-07

student(970007, 306, 1992).
registeredGroups(970007, 1, 6).
registeredCourse(970007, 102100, 1, 3).
registeredCourse(970007, 103100, 1, 3).
registeredGroups(970007, 2, 3).
registeredCourse(970007, 204100, 2, 3).
registeredGroups(970007, 3, 23).
registeredCourse(970007, 301101, 3, 3).
registeredCourse(970007, 302101, 3, 3).
registeredCourse(970007, 302102, 3, 3).
registeredCourse(970007, 302111, 3, 1).
registeredCourse(970007, 302112, 3, 1).
registeredCourse(970007, 305101, 3, 3).
registeredCourse(970007, 305102, 3, 3).
registeredCourse(970007, 306101, 3, 3).
registeredCourse(970007, 306102, 3, 3).
registeredGroups(970007, 4, 10).
registeredCourse(970007, 306111, 4, 1).
registeredCourse(970007, 306211, 4, 3).
registeredCourse(970007, 306212, 4, 3).
registeredCourse(970007, 306221, 4, 3).
registeredGroups(970007, 5, 6).
registeredCourse(970007, 306214, 5, 3).
registeredCourse(970007, 306215, 5, 3).
registeredGroups(970007, 6, 0).
registeredGroups(970007, 7, 3).
registeredCourse(970007, 301203, 7, 3).

Experiment B-08

student(970008, 306, 1992).
registeredGroups(970008, 1, 3).
 registeredCourse(970008, 101100, 1, 3).
registeredGroups(970008, 2, 0).
registeredGroups(970008, 3, 28).
 registeredCourse(970008, 301101, 3, 3).
 registeredCourse(970008, 301102, 3, 3).
 registeredCourse(970008, 302101, 3, 3).
 registeredCourse(970008, 302102, 3, 3).
 registeredCourse(970008, 302111, 3, 1).
 registeredCourse(970008, 304101, 3, 3).
 registeredCourse(970008, 304102, 3, 3).
 registeredCourse(970008, 305101, 3, 3).
 registeredCourse(970008, 306101, 3, 3).
 registeredCourse(970008, 306102, 3, 3).
registeredGroups(970008, 4, 10).
 registeredCourse(970008, 306111, 4, 1).
 registeredCourse(970008, 306211, 4, 3).
 registeredCourse(970008, 306212, 4, 3).
 registeredCourse(970008, 306221, 4, 3).
registeredGroups(970008, 5, 0).
registeredGroups(970008, 6, 0).
registeredGroups(970008, 7, 3).
 registeredCourse(970008, 301241, 7, 3).

Experiment B-09

student(970009, 306, 1992).
registeredGroups(970009, 1, 6).
 registeredCourse(970009, 101100, 1, 3).
 registeredCourse(970009, 102100, 1, 3).
registeredGroups(970009, 2, 0).
registeredGroups(970009, 3, 27).
 registeredCourse(970009, 301101, 3, 3).
 registeredCourse(970009, 301102, 3, 3).
 registeredCourse(970009, 302101, 3, 3).
 registeredCourse(970009, 303101, 3, 3).
 registeredCourse(970009, 303102, 3, 3).
 registeredCourse(970009, 305101, 3, 3).
 registeredCourse(970009, 305102, 3, 3).
 registeredCourse(970009, 306101, 3, 3).
 registeredCourse(970009, 306102, 3, 3).
registeredGroups(970009, 4, 13).
 registeredCourse(970009, 306111, 4, 1).
 registeredCourse(970009, 306211, 4, 3).
 registeredCourse(970009, 306212, 4, 3).

registeredCourse(970009, 306221, 4, 3).
registeredCourse(970009, 306222, 4, 3).
registeredGroups(970009, 5, 0).
registeredGroups(970009, 6, 0).
registeredGroups(970009, 7, 6).
registeredCourse(970009, 301201, 7, 3).
registeredCourse(970009, 301203, 7, 3).

Experiment B-10

student(970010, 306, 1992).
registeredGroups(970010, 1, 6).
registeredCourse(970010, 102100, 1, 3).
registeredCourse(970010, 103100, 1, 3).
registeredGroups(970010, 2, 3).
registeredCourse(970010, 204100, 2, 3).
registeredGroups(970010, 3, 25).
registeredCourse(970010, 301101, 3, 3).
registeredCourse(970010, 301102, 3, 3).
registeredCourse(970010, 302101, 3, 3).
registeredCourse(970010, 302111, 3, 1).
registeredCourse(970010, 304101, 3, 3).
registeredCourse(970010, 304102, 3, 3).
registeredCourse(970010, 305101, 3, 3).
registeredCourse(970010, 306101, 3, 3).
registeredCourse(970010, 306102, 3, 3).
registeredGroups(970010, 4, 9).
registeredCourse(970010, 306211, 4, 3).
registeredCourse(970010, 306221, 4, 3).
registeredCourse(970010, 306222, 4, 3).
registeredGroups(970010, 5, 3).
registeredCourse(970010, 306215, 5, 3).
registeredGroups(970010, 6, 0).
registeredGroups(970010, 7, 6).
registeredCourse(970010, 301201, 7, 3).
registeredCourse(970010, 301203, 7, 3).

Experiment C

student(960001, 306, 1992).
registeredGroups(960001, 1, 9).
registeredCourse(960001, 101100, 1, 3).
registeredCourse(960001, 102100, 1, 3).
registeredCourse(960001, 103100, 1, 3).
registeredGroups(960001, 2, 9).
registeredCourse(960001, 105100, 2, 3).
registeredCourse(960001, 106100, 2, 3).
registeredCourse(960001, 204100, 2, 3).
registeredGroups(960001, 3, 27).

registeredCourse(960001, 301101, 3, 3).
registeredCourse(960001, 301102, 3, 3).
registeredCourse(960001, 301151, 3, 3).
registeredCourse(960001, 302101, 3, 3).
registeredCourse(960001, 302102, 3, 3).
registeredCourse(960001, 303101, 3, 3).
registeredCourse(960001, 303102, 3, 3).
registeredCourse(960001, 306101, 3, 3).
registeredCourse(960001, 306102, 3, 3).
registeredGroups(960001, 4, 25).
registeredCourse(960001, 306111, 4, 1).
registeredCourse(960001, 306211, 4, 3).
registeredCourse(960001, 306212, 4, 3).
registeredCourse(960001, 306221, 4, 3).
registeredCourse(960001, 306222, 4, 3).
registeredCourse(960001, 306323, 4, 3).
registeredCourse(960001, 306325, 4, 3).
registeredCourse(960001, 306331, 4, 3).
registeredCourse(960001, 306333, 4, 3).
registeredGroups(960001, 5, 9).
registeredCourse(960001, 306214, 5, 3).
registeredCourse(960001, 306215, 5, 3).
registeredCourse(960001, 306338, 5, 3).
registeredGroups(960001, 6, 3).
registeredCourse(960001, 306490, 6, 3).
registeredGroups(960001, 7, 12).
registeredCourse(960001, 301201, 7, 3).
registeredCourse(960001, 301203, 7, 3).
registeredCourse(960001, 301231, 7, 3).
registeredCourse(960001, 301241, 7, 3).

Alerted operations

2 - *retract*(*class*(306437, 1, 26, 24, 9.30, 11.00)).
 3 - *retract*(*class*(302111, 2, 20, 3, 14.00, 17.00)).
 4 - *assert*(*class*(306445, 1, 26, 24, 9.30, 11.00)).
 5 - *assert*(*class*(306437, 2, 26, 24, 9.30, 11.00)).
 6 - *retract*(*class*(306437, 1, 26, 24, 9.30, 11.00)).
 assert(*class*(306437, 1, 26, 24, 9.30, 11.00)).
 7 - *retract*(*class*(302111, 2, 20, 3, 14.00, 17.00)).
 assert(*class*(302111, 2, 20, 3, 14.00, 17.00)).
 8 - *retract*(*class*(306437, 1, 26, 24, 9.30, 11.00)).
 assert(*class*(306437, 1, 27, 24, 9.30, 11.00)).
 9 - *retract*(*class*(302111, 2, 20, 3, 14.00, 17.00)).
 assert(*class*(302111, 2, 20, 5, 14.00, 17.00)).

Experiment D

student(950001, 306, 1992).
registeredGroups(950001, 1, 9).

registeredCourse(950001, 101100, 1, 3).
registeredCourse(950001, 102100, 1, 3).
registeredCourse(950001, 103100, 1, 3).
registeredGroups(950001, 2, 15).
registeredCourse(950001, 105100, 2, 3).
registeredCourse(950001, 106100, 2, 3).
registeredCourse(950001, 201100, 2, 3).
registeredCourse(950001, 202100, 2, 3).
registeredCourse(950001, 204100, 2, 3).
registeredGroups(950001, 3, 27).
registeredCourse(950001, 301101, 3, 3).
registeredCourse(950001, 301102, 3, 3).
registeredCourse(950001, 301151, 3, 3).
registeredCourse(950001, 302101, 3, 3).
registeredCourse(950001, 302102, 3, 3).
registeredCourse(950001, 303101, 3, 3).
registeredCourse(950001, 303102, 3, 3).
registeredCourse(950001, 306101, 3, 3).
registeredCourse(950001, 306102, 3, 3).
registeredGroups(950001, 4, 45).
registeredCourse(950001, 306111, 4, 1).
registeredCourse(950001, 306211, 4, 3).
registeredCourse(950001, 306212, 4, 3).
registeredCourse(950001, 306221, 4, 3).
registeredCourse(950001, 306222, 4, 3).
registeredCourse(950001, 306323, 4, 3).
registeredCourse(950001, 306325, 4, 3).
registeredCourse(950001, 306331, 4, 3).
registeredCourse(950001, 306333, 4, 3).
registeredCourse(950001, 306335, 4, 3).
registeredCourse(950001, 306341, 4, 3).
registeredCourse(950001, 306431, 4, 3).
registeredCourse(950001, 306432, 4, 3).
registeredCourse(950001, 306434, 4, 3).
registeredCourse(950001, 306442, 4, 3).
registeredCourse(950001, 306499, 4, 2).
registeredGroups(950001, 5, 9).
registeredCourse(950001, 306214, 5, 3).
registeredCourse(950001, 306215, 5, 3).
registeredCourse(950001, 306338, 5, 3).
registeredGroups(950001, 6, 3).
registeredCourse(950001, 306490, 6, 3).
registeredGroups(950001, 7, 12).
registeredCourse(950001, 301201, 7, 3).
registeredCourse(950001, 301203, 7, 3).
registeredCourse(950001, 301231, 7, 3).
registeredCourse(950001, 301241, 7, 3).

Alerted operations

- 2 - *retract(class(306437, 1, 26, 24, 9.30, 11.00)).*
- 3 - *retract(class(302111, 2, 20, 3, 14.00, 17.00)).*
- 4 - *assert(class(306445, 1, 26, 24, 9.30, 11.00)).*
- 5 - *assert(class(306437, 2, 26, 24, 9.30, 11.00)).*
- 6 - *retract(class(306437, 1, 26, 24, 9.30, 11.00)).*
assert(class(306437, 1, 26, 24, 9.30, 11.00)).
- 7 - *retract(class(302111, 2, 20, 3, 14.00, 17.00)).*
assert(class(302111, 2, 20, 3, 14.00, 17.00)).
- 8 - *retract(class(306437, 1, 26, 24, 9.30, 11.00)).*
assert(class(306437, 1, 27, 24, 9.30, 11.00)).
- 9 - *retract(class(302111, 2, 20, 3, 14.00, 17.00)).*
assert(class(302111, 2, 20, 5, 14.00, 17.00)).

ملخص

ربط الأنظمة الخبيرة بالشبكة المعلوماتية باستخدام مبدأ الربط بالمراقبة

إعداد

حسام عمر سعادة

إشراف

د. خليل المهدي

مساعد إشراف

د. منيب قطيشات

في الآونة الأخيرة، طرأ تطور ملحوظ واستخدام مميز، لعلم حساب الشبكات في بيئة الشبكة المعلوماتية العالمية والمحلية (Internet/Intranet). فقد تم استحداث تطبيقات جديدة في المهام الحرجة وللتجارة الإلكترونية مميزت بالكفاءة والمتانة والسرية. حيث أن هذه التطبيقات الإلكترونية تزود الموظفين والموردين والزبائن بخدمات قيمة ومعلومات متوفرة على الشبكة المعلوماتية.

في هذه الرسالة، مدركين أهمية هذا التطور، فقد تم بحث إمكانية عمل الأنظمة الخبيرة بكفاءة مع قواعد البيانات على الشبكة المعلوماتية، حاصلين على الميزات المتوفرة من خلال استخدام المعايير والخدمات والأطر في هذا المجال. حيث أننا على يقين أن هذا الربط سيسحدث فرصاً جديدة لحلّول الإلكترونية استنتاجية في المهام الحرجة (أسبيناها، e-reasoning). إن الأنظمة الإلكترونية الاستنتاجية قد تلازم حلول إلكترونية أخرى (مثل، e-commerce) للتحيز، والاستشارة، ودعم القرارات.

إن المبدأ المقترح توسيع مبدأ المراقبة (هندي، 1994) لربط الأنظمة الخبيرة المبنية على نظام صيانة الحقائق مع قواعد البيانات النشطة. إن الهدف الأساسي من وراء مبدأ المراقبة هو حفظ تماسك البيانات بين النظام الاستنتاجي وقاعدة البيانات. إن تصميم مولد القواعد للمبدأ الجديد مبني على أساس المراقبة وموزع على هيكلية شبكة مكونة من ثلاثة أجزاء. للسهولة والقدرة على التوسع والارتباط مع التطبيقات الأخرى، إن هيكلية المبدأ المقترح تستخدم مبدأ حساب الشبكات الخادم/المخدوم والمشارك/الناشر وتقنية العنصر الموزع.

الكفاءة هي الركيزة الأساسية خلف نجاح هذا المبدأ. لذلك، نحن نعرض طريقة ربط وثيقة لربط نظام صيانة الحقائق مع شبكة قواعد الإنتاج معتمدة على نظام Morguc ونظام Hindi. بمقاييس مختلفة، تم إجراء تجارب عملية تثبت كفاءة النظام المقترح بين النظامين الآخرين. حيث أن الطريقة الجديدة تحقق أهم ميزات النظامين الآخرين وتجنب أغلب سيئاتهم.