INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

SEDO: Semantic Error Diagnostic Operation

by

Daniel C. B. Choi, B. Eng.

A thesis submitted to
the Faculty of Graduate Studies
in partial fulfillment of
the requirements for the degree of
Master of Engineering

Ottawa-Carleton Institute for Electrical
and Computer Engineering
Carleton University,
Ottawa, Ontario, Canada

August 20, 1998

© copyright
1998 Daniel C. B. Choi

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-48445-9

Canada

The undersigned recommend to
the Faculty of Graduate Studies
acceptance of the thesis

**“SEDO: Semantic Error
Diagnostic Operation”**

Submitted by **Daniel Choi, B.A.**
in partial fulfillment of the requirements for
the degree of Master of Engineering

Department Chair

Professor B. Pagurek, Thesis Supervisor

Carleton University
September 21, 1998

Abstract

Multi-agent networking systems introduce a lot of challenges such as the idea of agent collaboration and peer-to-peer human-like agent contract interaction. All these are to ensure the delivery of real-time multimedia services to customers. However, there are different kinds of errors existing in these systems and some of them cannot be located easily. An example is the unsatisfactory performance problem encountered in an application that can be detected only at the semantic level such that the source of problem is unknown to the system. This thesis demonstrates the structure of agents and their communication based on MANA, a multi-agent system developed by Mitel Corporation. Then it describes the simulation of an application domain and a fault scenario demonstrating semantic errors found in a multi-agent system. Finally, it introduces SEDO, a fault localization algorithm that has been applied to the fault scenario to show how it solved the semantic problem.

Acknowledgments

Special thanks to my advisor, Professor Bernie Pagurek, who offered me an opportunity to work in this project and given me help and valuable advises in this research. Thanks to Mitel Corporation and CITO that provided me equipment and financial support. Thanks to Tom Gray and Serguei Mankovski for I have learned a lot from working with them. Thanks to Daisy Fung for her help and the arrangement for me to join the Toronto conference. I was enlightened and that helped my research a lot. Thanks to my mother Lily Chu and Peggy Cheung for their love, encouragement and prayer. Finally I would like to give my thanks, praise and glory to the Lord my God who has blessed me so much that I could finish this thesis. His guidance is obvious to me throughout this project.

Table of Contents

Abstract	iii
Acknowledgments	iv
Acronyms & Glossary	vii
List of figures	viii
 Chapter 1 – Introduction	 1
1.1 Multi-Agent System	1
1.2 Problem of Multi-Agent System	3
1.3 SEDO and Its Contribution	6
 Chapter 2 – Agents in Multi-Agent System	 8
2.1 Agents and Objects	8
2.2 Agent Structure and Operation	9
2.3 Understanding between Agents and Contracts	12
 Chapter 3 – Micmac Software Testbed	 15
 Chapter 4 – Simulation of the Semantic Problem	 21
4.1 The Application Domain	21
4.2 A Fault Scenario in the Application Domain	22
4.3 Simulation	24
4.3.1 A More Specific Architecture of an Agent	25
4.3.2 Agent Communication in the Fault Scenario	32

Chapter 5 – SEDO: Semantic Error Diagnostic Operation 35

5.1 Introduction	35
5.2 A Simple Semantic Fault Model	37
5.3 The Diagnosis Model of SEDO	39
5.3.1 The Setup Connection Model for Application	42
5.3.1.1 The XOR Pattern	44
5.3.1.2 The AND Pattern	45
5.3.1.3 The ORDER Pattern	46
5.3.1.2 The LOOP Pattern	47
5.3.2 The Simplification of Setup Connection Model in SEDO Diagnosis Model	49
5.3.3 The Actions in Analyzer	54
5.3.3.1 Reduction of Simplified Tree	55
5.3.3.2 Transformation of Reduced List	56
5.3.3.3 Generation of Table from Transformed List	58
5.3.4 The SEDO Temporary Base (STB)	59
5.3.5 The Watcher Script	60
5.3.5.1 Performance Test and Verification of Test Results	61
5.3.5.2 The Use of Application Identity -- AppID	61
5.3.5.3 Localization of Faulty Resource and Agent	64
5.3.6 The SEDO Information Base (SIB) and SIB Monitor	66
5.3.7 The Tester Object in Application Agent Body	67
5.3.8 The Modified Broker Object in Agent Body	68
5.4 Simulation -- Applying SEDO to the Fault Scenario	72
5.5 SEDO and 'Trust' among Agents	76

Chapter 6 – Conclusion 79

6.1 Limitation	80
6.2 Future Works	81

References 83

Appendix

A -- Script 'analysis.clp'	86
B -- Function 'transform'	93
C -- Function 'genTable'	96

Acronyms & Glossary

API	: Application Programming Interface
AppID	: Application Identity
BDI	: Beliefs, Desires and Intentions
CLIPS	: C Language Integrated Production System
FSM	: Finite State Machine
GUI	: Graphical User Interface
JESS	: Java Expert System Shell
KQML	: Knowledge Query and Manipulation Language
MANA	: Multi-Agent Architecture for Networking Applications
MSC	: Message Sequence Chart
QuAL	: Quality Assurance Language
R&D	: Research and Development
SEDO	: Semantic Error Diagnostic Operation
SIB	: SEDO Information Base
STB	: SEDO Temporary Base

List of figures

Chapter 1

Figure 1-1:	The printer semantic fault example	4
Figure 1-2:	The data link semantic fault example	5

Chapter 2

Figure 2-1:	General agent architecture and inter-agent communication	10
Figure 2-2:	The structure of a simple agent in MANA	12
Figure 2-3:	Agents recognize each other by their understanding of one another	13
Figure 2-4:	Contracts are made based on the connection model in Figure 2-3	14

Chapter 3

Figure 3-1:	Agent communication via the Micmac testbed	16
Figure 3-2:	The model of the Micmac testbed	18
Figure 3-3:	Example of a tuple and a matched tuple template	19

Chapter 4

Figure 4-1:	A long distance service provided by a multi-agent system	22
Figure 4-2:	Finite state machine showing the behaviors of actor objects in agents	26
Figure 4-3:	The class hierarchical tree for agent objects	30
Figure 4-4:	Framework of the selected application domain	33
Figure 4-5:	Message sequence chart showing the messages exchanged between agents	34

Chapter 5

Figure 5-1:	A simple model of semantic error	38
Figure 5-2:	The diagnosis model of SEDO	40
Figure 5-3:	General physical connection model among agents/resources	43
Figure 5-4:	The XOR agent action	44
Figure 5-5:	The AND agent action	45
Figure 5-6:	The ORDER agent action	46
Figure 5-7:	The LOOP agent action	47
Figure 5-8:	An example of the setup connection model	49
Figure 5-9:	The simplified setup connection model from Figure 5-8	50
Figure 5-10:	Wrong interpretation of the simplified setup connection tree	53
Figure 5-11:	The reduced tree from Figure 5-9	55
Figure 5-12:	The transformed tree and list from Figure 5-11	58
Figure 5-13:	Table showing the arrangement of resource sets	59
Figure 5-14:	The structure of a STB	60
Figure 5-15:	Examples of applications used to generate the AppIDs	62
Figure 5-16:	Finite state machines showing the behaviors of the modified Script & the Watcher	65
Figure 5-17:	The SIB Monitor and SIB	66
Figure 5-18:	The behavior of the Tester object	68
Figure 5-19:	The behavior of the modified Broker object in the agent body	69
Figure 5-20:	The setup and operation of a new modem data transfer application	73
Figure 5-21:	The setup and operation of the 2nd modem data transfer application	74

Chapter 1

Introduction

1.1 Multi-Agent System

In a large firm, the project manager splits a large job into different parts and then dispatches them to different divisions. A division supervisor may then categorize the sub-task into smaller parts and give them to appropriate group leaders. A group leader may in turn select some capable workers who can do the job and so on. It is the idea of collaboration. The collaboration among the people involved in the project forms a hierarchy that describes their relationship and the level of responsibilities. Following this idea, new telecommunication systems have been designed to provide a variety of real-time conversational services in an open distributed environment through the collaboration between a set of software components called *agents*. Such multi-agent systems aim to adapt and evolve so that they become a potential solution to the problem of changing environment. One example of a multi-agent system is MANA (Multi-Agent Architecture for Networking

Applications) developed by Mitel Corporation [WGD95a] [WGD95b] [WGD95c] [G95a]. It follows the system architecture discussed in [AP96] & [H95] and the goal of such a system is to provide high reliability level for services from which some may be life-critical. Also the system needs to maintain its stability so that no system redesign is required when there is technological or service evolution. To achieve this purpose, a multi-agent system requires intelligence or learning mechanisms to update service information that was derived from the operation of the agents. This returning information helps to redefine agents and to reallocate resources to correct failures and to meet the requirements of the defined service more precisely.

A defined service or application in the multi-agent system is mapped as a series of calls among agents to perform the service, and each agent is expected to specify the type, quantity and quality of service in order to provide for an overall application [G95b]. Since it is an open environment, the agent has no prior knowledge of each other. It is like a project in a large company that involves plenty of employees and external contractors. The only knowledge that the agent possesses is its requirements and capabilities to provide a specific type of service. The agent has to find other agents that are able to supply resources to fulfill each of its service requirements. A calling agent (*contracting agent*) will have to choose a suitable agent that can supply resources to complete the service. It will select the agent that has the best chance of success in performing a service based on past history. A called agent (*the contractor agent*) has no idea of the value a calling agent will place upon the performance of a service so it just tries its best to finish the task after accepting the job. A *contract* is said to be made when the contractor agent has promised the contracting agent that it will provide some kinds of resources for a period of time. During this period, the contracting agent provides job descriptions for the contractor agent and receives services from

it. The contracting agent put its trust on the contractor agent and only the *application agent* or the *top contracting agent* (the first calling agent that initiates the application, usually a helpdesk agent) has the knowledge of itself to determine whether the result is successful based on overall performance. With this capability, performance failures of applications are discovered and handled at the semantic state of the application.

1.2 Problem of Multi-Agent System

Due to the innovation of technology and increasing needs of clients, any external resource connected to a multi-agent system may alter without notifying the system. Usually there is no problem provided that both sides follow the agreements and understanding made earlier. Hence both sides can work on its own way as long as it can provide satisfactory services based on the requirements. Nevertheless, sometimes the agreements may not cover all areas, especially new ones brought by the changes. One example is the evolution of printers. In the old days people only needed to send a text data file to a dot-matrix printer and it would do the job soundly. Now most printers employ either ink-jet or laser technologies and there are a lot of options for users. A user can not simply send a text file to the laser printer because its default input format is postscript one. Instead, more details need to be specified before the printing job can occur or else the printer may not work properly. Suppose that the same printer job described above is done by a multi-agent system. The system contains a helpdesk agent, a printer server agent, a file server and some printers (resources) as shown in Figure 1-1. When the user wants to print a document, he/she simply specifies the name of the data file and selects the print option at the helpdesk. The helpdesk agent will pass the file name to the printer server agent that will then get the data file from the file server

and select an available printer for the job. Before postscript format has been introduced, the agreement between the printer server agent and the printers was simple: any printer that was not busy would be selected and that printer would get the data file and print the content. However, when the dot-matrix printers were later replaced by laser printers, potential problems existed. One problem was the lack of field which specified the file format in the agreement. The problem did not exist before because there was no file format called 'postscript' during the system setup. The printer server agent thinks that the printer has done a good job. However, the printer presumes that it got a postscript file since there has been no information specified in the agreement. The problem is discovered only after the printer output has been passed back to the helpdesk agent or the client.

Figure 1-1: The printer semantic fault example

Another example is the data link example described in [G95b]. It is shown in Figure 1-2. A data link with an error rate of 10^{-5} is highly suitable for voice but inadequate for video. Unfortunately, self-diagnostics on the link by its own agent is not able to identify that it is failing for its

video services but passing for the voice. Again, this failure can only be observed in the video application agent at semantic level.

Figure 1-2: The data link semantic fault example

This kind of failure is called a *semantic error*. There is no formal definition of a semantic error in a multi-agent system. In this project, a semantic error in an multi-agent system is defined as:

The fault that can only be discovered when the discrepancy of overall performance is detected by the application agent during the operation of an application.

Semantic level refers to the top level of the agent connection hierarchy of an application where the application agent resides. A semantic error may be caused by hardware failure that cannot be detected by underlying agents (i.e. at syntactic level) during the operation or a misunderstanding between agents in the previous application setup stage. The examples described above are only simple ones which may be encountered in a multi-agent service system. In more general

cases, semantic problems are complicated such that a faulty resource may lead to semantic errors on some applications but work fine for others.

1.3 SEDO and Its Contribution

There have been very few studies on semantic fault scenarios. Thus it is not possible to establish a general fault model of semantic error in a multi-agent based on limited knowledge and observation of known semantic errors. However, a simple model has also been created from some real examples encountered. *Semantic Error Diagnostic Operation (SEDO)* is a fault localization algorithm that has been invented by the author and is currently under the patent process initiated by Mitel Corporation. SEDO aims to solve the semantic problems that occurred in the simple fault model defined in this thesis. It will become a contribution to the network management of current and future multi-agent systems because there has been no fault diagnostic method that can automatically solve the semantic problems found in a multi-agent system such as MANA currently.

SEDO has been applied to an application domain, a long distance service, demonstrated on a simulated multi-agent environment and successfully located the resource of semantic error in some chosen fault scenarios. The simulation did not only demonstrate the operation of SEDO but also explained how the agents communicate with their own internal objects internally as well as other agents externally. The design of SEDO is to locate the semantic faulty resource of an application based on the setup connection tree information and then inform the appropriate agents so that they can avoid using the resources in the future. SEDO particularly focuses on several patterns of agent actions in order to analyze the logical relationship among agents in an application.

Then it establishes a setup connection tree model based on the patterns of agent actions, and transforms it into a new connection tree model that expresses the logical relationship between all available resources in the disjunctive normal form (d.n.f.). SEDO uses a table to record different set of resources and use them one by one in later resource selection process. Hence, the faulty resource can be located by induction. SEDO will also update the information of the semantic errors in a global database called the SEDO Information Base (SIB), where the system administrator can analyze the cause of semantic problems based on the detailed diagnostic information.

In this project, the agent architecture and operation will be discussed in Chapter 2 based on agent-based system MANA described in [WGD95a] [WGD95b] [WGD95c] [G95a]. Chapter 3 will give a brief introduction to the Micmac software testbed [B96] [D96] that has provided a media for the agent communication in the simulation. Chapter 4 will describe the application domain and fault scenario that have been selected in the simulation. Chapter 5 will explain how SEDO operates and how it has been applied to the fault scenario in the simulation.

Chapter 2

Agents in Multi-agent Systems

2.1 Agents and Objects

When compared with objects defined in software engineering, agents are more complicated and dynamic in both structure and behavior. Object behavior can be described as a fixed sequence of procedural steps that are followed by a service request. When a method in an object is invoked, all steps described in that method will be executed in sequence. Agent behavior is described in chunks that can be composed dynamically for each service request. It always checks for the conditions of the environment before starting an action in decision making. These conditions are the knowledge stored in the head of an agent that help the agent to determine whether it should take an action at that moment. They can be pre-conditions for the actions or history. An object maintains the current state in instance variables so that appropriate actions can be taken depending on the state. An agent, however, may rely on the information of the current state in addition to the information of a few past states that are stored *locally*. [DL97] and [LGD97] have a more detailed discussion on the difference between an object and an agent. Both agents and objects are entities.

An entity is generally a group of attributes without inserting a layer of functionality and it is distinct from the environment. An object is an entity that can affect environments. An agent is an object associated with a set of goals which intend to fulfill the needs of other agents. The BDI (Beliefs, Desires and Intentions) agent model introduced in [RG95] further defines an agent as an entity that has beliefs - the informative component of system state, desires - objectives or goals, and intentions - a course of planned actions to achieve the goals. An agent can be treated as a collection of objects which define and achieve the BDI. More discussion about the definition of agents can be found in [DK93][MF94][JW95][JW98].

2.2 Agent Structure and Operation

The idea of a multi-agent system is developed from the concept of 'contract net' as introduced in [S80]. An agent in a multi-agent system is similar to a node described in a contract net. A contract net node may act as a manager, a contractor, or both roles during the course of problem solving. The role of a manager is to monitor the execution of a task and process the results of its execution while the contractor is responsible for the actual execution of the task. Based on this idea, an agent is composed of two parts: head and body. The head of an agent monitors the arrival of request messages, verifies the result of peer-to-peer services and makes decision. The body of an agent is mainly responsible for the execution of a task and the selection of resources for assistance. The selected resources may be some local physical devices, software applications, a connected service-supply network or other agents. In many multi-agent systems, inter-agent communication is achieved by exchanging KQML (Knowledge Query and Manipulation Language) [GK94] messages. However, in this thesis, agent communication has been demonstrated

on the Micmac software testbed for simulation purpose. The general architecture of agents and their communication is demonstrated in Figure 2-1.

Figure 2-1: General agent architecture and inter-agent communication

Upon receiving a job request message, an agent has to first verify the nature of the job and to check whether the beliefs have been satisfied. To verify whether it has the capability to do the job itself, an agent has intelligence and memory to make appropriate decision. An agent is described as a hybrid system in [S98] that combines deliberative mechanisms in the agent head and reactive processes in the agent body. In MANA, an agent has a set of rules ([WGD95a] [WGD95b] [WGD95c] [G95a]) embedded in the head of an agent which is driven by rule-based expert systems. A rule in an expert system has the general form:

IF { A Set of Facts } EXISTS, THEN DO { A List of Actions }

A detailed example will be demonstrated in section 4.2.1.

The rules in a MANA agent head are packed into scripts. The definition, structure as well as reasoning with general expert system scripts have been described in [R88]. MANA uses the concept to implement scripts. Each script contains the description of a specific job and a set of rules which control the job behavior. The rules implement three main procedures of a script: template matching, beliefs checking and tasks execution. There is a template in each script that specifies the job nature. The called agent will first search for the appropriate template that matches the message received from the calling agent. Then it will check whether all beliefs, or some conditions, have been satisfied. If both a matched job is found and all beliefs are satisfied, the agent will select and perform the tasks that have already been specified by the rules. Any parameters and details of the tasks will be sent to the body as a job order.

Since the resource selection process of a job is the duty of the agent body, there may be some functional objects in the body like the Broker object that has the mechanism for selecting appropriate resources and some informant objects that provide useful information. In the body of a MANA agent, a Broker object waits for the order from the head and will start searching for appropriate resources once it gets and analyzes the order. The policy of selecting resources will be based on information such as cost parameters, usage rights or the current resource health information, etc. supplied by informant objects. The structure of a more specific agent, based on the architecture of a MANA agent [G95b], is shown in Figure 2-2.

The body of an agent may contain more functional objects. For example, a policy object may be created during the policy process that maintains the cost effectiveness of a service in the application. Some monitoring objects may be used to monitor the performance of the agents supplying the service if they have met their guarantees (by the local viewpoint of the agent). By doing so, they will attempt to identify resources that are not fulfilling their commitments. A log object may be used to log all the events happening during job execution, etc.

Figure 2-2: The structure of a simple agent in MANA

2.3 Understanding between Agents & Contracts

In a contract net, a contract is established by a process of local mutual selection based on a two-

way transfer of information, i.e. negotiation between two nodes. A contract describes the partnership between the contracting and contractor agents. In MANA, an agent can be treated as an active entity that achieves some of the services. During system setup, the nature of services which are embedded in the agents has been analyzed by a matching mechanism so that MANA is able to recognize the connection of agents for future collaboration [G95a]. This loose relationship existing between agents is called the '*understanding*' (or '*commitment*' as described in [G95a]). The matching mechanism assigns all possible agents which can provide services (by checking the templates residing in their scripts) for the agent based on the specification (format, content, etc.) of all predefined service requests. The information will then be stored in the agents' bodies. Hence an agent can use these information for later negotiation. A connection model is established from the understanding between agents and it is shown in Figure 2-3.

Figure 2-3: Agents recognize each other by their understanding of one another.

If an agent requires service, it first checks the resources connected by the understanding relationship. It will get all relative information of the resources from informant objects. Once a resource is selected, the agent starts negotiating with the resource by sending more detailed job requirements. If the resource accepts the job, a contract is said to be made that describes the relationship between two agents in the application domain. A contract may be just an abstract relationship between the agents or an object that contain specifications of a particular task. The idea of a contract is to provide a guarantee that a certain type, capacity and quality of service will be provided by one agent to another. However, if a misunderstanding exists between agents, fatal semantic errors will occur. Figure 2-4 demonstrates the establishment of contracts based on the connection (understanding) model shown in Figure 2-3.

Figure 2-4: Contracts are made based on the connection model in Figure 2-3.

Chapter 3

Agent Communication: The Micmac Software Testbed

In many multi-agent systems, agent communication is achieved by exchanging KQML messages as described in [GK94]. However, agent communication can also be achieved by other means. For example, the simulation described in this thesis has used the Micmac testbed, the blackboard system used in Mitel agent project, to simulate agent communication. Similar to KQML, the Micmac testbed supports the use of tuple, the data structure that can store various types of data. The fundamental difference between KQML and the Micmac testbed is that by using KQML the sender needs to specify the receiver address or otherwise broadcast the messages, but in Micmac testbed everyone can look into the messages in the pool of shared memory and determine whether it will pick up the information by using some matching mechanism (will be discussed later). The operation of SEDO is not influenced by what kind of agent communication method is adopted. This chapter will provide a quick overview for the Micmac testbed.

The Micmac testbed [B96] [D96] is a software testbed that consists of a set of software tools which together provide an environment for hands-on experimentation. It is specifically designed for the simulation of multi-agent interaction and negotiation protocols. By using the Micmac testbed, the model of inter-agent communication shown in Figure 2-1 has been changed to another model shown in Figure 3-1.

Figure 3-1: Agent Communication via the Micmac Testbed

The components of the Micmac tools are either issued from public sources, or developed by other technology research organization. It is now the possession of Mitel Corporation and has

been modified for Mitel R&D (Research & Development) purposes. The Micmac testbed was modified from *Sonia* [B96], an adaptation of the Linda coordination language. *Linda* is a machine-independent environment for implementing parallel codes and its introduction can be found in [M97]. Linda uses a different mechanism to create and coordinate multiple processes. If two processes need to communicate, no messages or share variables need to be exchanged. Instead, the data producing process will generate a data item called *tuple* and place it in a region of shared memory called *tuple space*. A tuple is an unordered set of fields represented as key-value associations. It is a general-purpose structure that can store all types of data. In the tuple space, any other process can access the tuple using what is commonly referred to as generative communication. Although Linda has been recognized for quite some time, it has mainly been applied in the area of parallel processing where the goal is to get maximum performance from processors. Sonia takes advantage of the expressiveness of Linda to support the inherent parallelism and need for coordination of workaday activities which are composed of people and software [B96]. Sonia has modified the tuple structure of Linda as well as the operation primitives. In Sonia, a tuple space is called *Agora* and a tuple is allowed to be part of another tuple so that complex structures can be expressed. Also, there is no process but only data posted or retrieved in an Agora. The Micmac testbed reuses most of the resources from Sonia and makes few modifications. The major one is the extension of the API (Application Programming Interface).

Currently an Agora server of the Micmac testbed manages the Agora which stores information as tuples. The API of Micmac testbed provides support for Smalltalk, C/C++, CLIPS and Java under both NT and Unix platforms. The model of the Micmac testbed is shown in Figure 3-2.

Figure3-2: The model of the Micmac testbed

There is only one simple set of primitives which triggers specific processing requests over tuple space content:

Post : adds a tuple to an Agora (tuple space), supports timeout option.

Pick : extracts a tuple from an Agora, supports timeout option.

Peek : read a tuple in an Agora, supports timeout option.

Cancel: terminate a pending pick/peek request.

Unlike *post* and *cancel* primitive, the *pick* and *peek* primitives remain pending at the server until an entry in the Agora matches the tuple template specified by the primitive or a cancel primitive specifying a matching tuple template is issued by another client. The matching mechanism applies the template matching rules explained in [M97] [B96]. This feature can be applied to the

matching mechanism of messages received and the templates embedded in agent scripts. Figure 3-3 demonstrates a simple tuple and its matched tuple template.

Figure3-3: Example of a tuple and a matched tuple template.

A template must have the same number of fields and field titles of a tuple for a successful matching. From the example above, there are three fields and their titles are callerID, destination and number respectively. The field content may be the same as the field content of the matched tuple (e.g. Ottawa) or anything indicated by the wildcard '*any*'. Any values like a number, a string, or even an embedded tuple can be matched by '*any*'.

Since the Micmac testbed offers a shared memory that can be accessed by any co-located clients as well as by remote or network-connected clients, it provides a channel for message transfer in both intra-agent and inter-agent communications. The head of an agent is capable of exchanging information with the body via the Agora, and any Agora can connect to other remote Agora so that messages can be exchanged among distant agents. In this project, the Micmac testbed has provided a Java-supported API for agents so that they can post and retrieve any Java object in the

Agora.

Chapter 4

Simulation of the Semantic Problem

4.1 The Application Domain

The application domain selected is a long distance service provided by a multi-agent system which is shown in Figure 4-1. In the service, agent S, the service resource selector suggests long distance service providers to a voice service helpdesk agent, V, which is responsible for supervising the services of all voice call applications. Agent V employs agent S as a service broker that connects to three resources. The resources are external agents (A1, A2 and A3) which belong to three different long distance service providers: AT&T, MCI and Sprint respectively. All these providers connect to the local telephone networks from different areas, say from regions A to E. Since each of these long distance companies has its own charging schemes, agent S has been set up with the knowledge of selecting the most appropriate one according to the time, destination of the receivers, price or quality based on past experience.

Figure 4-1: A long distance service provided by a multi-agent system

The long distance service worked fine for voice traffic. After a certain period of time, a new helpdesk agent (agent D) was added to the system to allow services for modem data transfer. Similar to agent V, it had understanding with agent S and it would make contract with agent S when a modem service was requested. This new service has worked properly and no problem was found during the operation of these new modem applications.

4.2 A Fault Scenario in the Application Domain

Although the new modem service seemed to be working properly in the beginning, the introduction of new modem data transfer service to the system was not recognized by external agents (A1,

A2 and A3) from their local viewpoints because it happened after the system had been setup. These external agents cooperated with the system only through explicit guarantees in the agreements. The services on each side were not designed or tested together; instead, each side had been managing its own network and may alter at anytime without informing the other. Thus the long distance service providers presumed that all data streams received from the multi-agent system were voice traffic. The hazard is that any change may violate the understanding of the other side of contract. The fault scenario chosen for the application domain was that one of the long distance service provider MCI decided to use compression for voice on its trunks when it sent the voice to region B (says Washington) during a certain period of time. This external change was transparent to all voice services because it had been particularly designed for voice data. Nevertheless, the compression for voice is not suitable for modem data. As a result, an extremely low data transmission rate was encountered whenever modem data was sent to or received from region B. The reason is that with compression on trunks, compression of traffic must be done which assumes that the signal will have a speech-like distribution. Since the modem data distribution is completely different from voice, such a compression would cause the modem signal to be distorted upon receipt. In order to receive the signal with distortion, the modem at each side has to switch to a very low data transfer rate. Now all the agents are working properly and only agent D experienced failures occasionally.

This would have happened since presumably at the system setup time (when the details of agreement between S and A2 were defined), agent D was not existing and the use of modem data was not considered. A2 would have guaranteed service only for voice traffic and it was free to change service as long as it still maintained service quality in the ontology of its existing voice contract with S. In this case, A2 had no responsibility to inform S of its change because it would not affect its guarantee to S. D came into existence and was apparently well served by S and its supplier A2. In fact this was deceptive since the transmission requirements for data were not part of the ontology of the agreement between S and A2. Similar changes may happen at anytime because A2 would constantly be managing its network to optimize its service and to overcome faults. Some other changes may also be made when features like loaded/unloaded trunks or echo suppression, etc. are added to voice traffic.

4.3 Simulation

The selected application domain and fault scenario described in the previous section have been simulated on the Micmac testbed. The purpose of this simulation is to demonstrate agent communication and how a semantic error occurs in an application. In this simulation, the modem data transfer service had semantic problems whenever MCI (the problematic resource) had been used as the long-distance service provider. Voice call services and modem services employing other long-distance service providers (AT&T and Sprint) had no semantic problems.

4.3.1 A More Specific Architecture of an Agent

An agent used for the simulation was composed of a set of objects based on the ideas of MANA agents [G95a]. In the agent head, there were scripts each one of which specified an application. These scripts contained rules to determine what actions should be triggered when some conditions had been satisfied. To implement the scripts, JESS (Java Expert System Shell) introduced in [F98] had been used. JESS migrates CLIPS (C Language Integrated Production System) introduced in [R98], an expert system tool, to Java so that the rules and functions in CLIPS can be implemented by Java objects. In the agent body, there were some bidder objects, a Broker object, a policy object and some other data objects [G95b]. Bidder objects and Policy objects were informant objects that supplied information about the resources. The Policy object contained information about the understanding relationship between the resources and specific type of job requests so that the Broker object would know which resources could be employed according to the job nature. The bidder object provided the current health information of a resource for the Broker object so that resources could be selected accordingly. Some other data objects were used to store agent information like the agent ID, Agora information and diagnosis information, etc.

There were two actors among these objects: the object that represented a script in the agent head and the Broker object in the agent body. There may be a lot of script objects in the head of an agent in real multi-agent systems. However, there was only one script in an agent for this simula-

tion. The behavior of a script and the Broker object is shown in Figure 4-2(a) and (b) respectively.

Figure 4-2(a): Behavior of Script in the agent head

Figure 4-2(b): Behavior of Broker object in the agent body

Figure 4-2: Finite state machine showing the behaviors of actor objects in agents

Many helpdesk agents like the voice and modem service helpdesk agents (agent V and D) have script objects shown in Figure 4-2(a). They do not contain complicated logic or rules. Instead, they have interactive interfaces that enable communication between clients and the system. Other underlying agents may have more specific rules for making decisions. For example, the service resource broker agent S in the simulation had additional rules to determine the preferred long distance service provider based on the time, place, etc. The execution of these rules are embedded in the analysis actions between state 2 and state 3 of the finite state machine in the Figure 4-2(a). The followings are some rules in the script of agent S:

```
(defrule select-Sprint-for-HongKong
  (declare (salience 100))                ; the most economic (highest priority)
  ?day <- (day ? ?index)                  ; all days
  ?hour <- (hour ? ?index)                 ; anytime
  ?d <- (destination "Hong Kong" ?index)   ; destination = "Hong Kong"
=>
  (retract ?day) (retract ?hour) (retract ?d)
  (assert (preference Sprint ?index))       ; Set provider = Sprint
)

(defrule select-AT_T-for-HongKong
  ?day <- (day ? ?index)                  ; the 2nd best in price
  ?hour <- (hour ? ?index)                 ; all days
  ?d <- (destination "Hong Kong" ?index)   ; anytime
  ; destination = "Hong Kong"
=>
  (retract ?day) (retract ?hour) (retract ?d)
  (assert (preference AT_T ?index))        ; Set provider = AT&T
)

(defrule select-MCI-for-HongKong
  ?day <- (day ?day&:(or (eq ?day 0) (eq ?day 6)) ?index) ;weekend
  ?hour <- (hour ?hr&:(and (>= ?hr 9) (< ?hr 21)) ?index) ; from 9am to 9pm
  ?d <- (destination "Hong Kong" ?index)   ; destination = "Hong Kong"
=>
  (retract ?day) (retract ?hour) (retract ?d)
  (assert (preference MCI ?index))         ; Set provider = MCI
)
```

The purpose of these rules is to select a long distance service provider for a phone call to Hong Kong. Each of these rules has two sections: a list of matched facts and a list of actions. These two sections are separated by the arrow symbol '=>'. When all the facts indicated above the arrow have been matched, the actions listed beneath the arrow will be performed step by step. It appears that all rules will be triggered when the fact '(destination "HongKong")' has been asserted, but in fact, only one will be triggered. The decision making key is the conditions specified in the fact templates. For example, in the third rule, the fact template '(day ?day&~:(or (eq ?day 0) (eq ?day 6)) ?index)' will match a fact '(day x)' where x has a value of either 0 (represents Sunday) or 6 (represents Saturday) so that x can only be an integer value of 0 or 6. As a result, the rule will only be matched and fired during weekends. Sometimes conditions of more than one rule will be satisfied simultaneously. In this case, all of these rules will be triggered. To allow only one of them to be chosen, the '(retract)' statement will be used to remove the triggering facts so that only the first selected rule will capture the matched facts. The choice of these rules is unpredictable and the only way to assign priority for these rules is to add the '(declare (salience s))' statement in the matching-fact section of a rule. The first rule demonstrates the use of salience. The value of s can be set from -10000 to 10000 and the default value is 0. The higher the value of s, the higher the priority will be assigned to a rule. The rule with the highest priority will always go first. As a result, the first rule has a higher priority than the second one even though both require the same facts to be matched. The fact template '(destination "HongKong")' indicates the

destination must be “HongKong” in order to satisfy the rule. The conditions of these rules are set based on current date and time. Hence the appropriate long-distance service provider can be selected accordingly.

Figure 4-2(b) demonstrates the behavior of a Broker object in the simulation. The resource selection scheme in a Broker object may vary from agent to agent. For instance, helpdesk agents V and D simply select the first available resource from a list of resources but agent S would first pick up the preferred resource selected by the agent head and if it is unavailable, other resources will be considered. In both cases, the Broker object will verify whether the selected resource is operative before it sends detailed information to the resource. A resource is considered non-functional under the following circumstances:

- (i) A resource does not reply after timeout when the bidder object asks for health information.
- (ii) A resource returns a negative acknowledgment when the bidder object asks for health information.
- (iii) A resource does not reply after timeout when the Broker object waits for the result.
- (iv) A resource returns a failure message when the Broker object waits for the result.

In the first case, a resource has no response to the bidder object because the resource is completely broken down and it needs repairing. The second case implies that the resource is functional but it is under maintenance or for other reasons it simply replies with a negative acknowledgment. The third case indicates a resource which has just broken down after the broker

sent out a job request or it is extremely busy such that it delays in its reply. The last one happens only if the resource is an underlying agent and all its underlying resources have become unavailable at that moment. All these failure can be detected by the neighboring agent during the application setup stage and they are examples of syntactic errors. These errors may result in the failure of the application setup and hence no semantic problem will occur because the application has not yet come into operation. The occurrence of semantic error implies that the above errors may be encountered during application setup but they have been fixed or bypassed so that the application can start working eventually.

The class hierarchical trees for the objects defined in the simulation are shown in Figure 4-3.

Figure 4-3: The class hierarchical tree for agent objects

There are four groups of object classes: agent head, agent body, script functions and GUI (Graphical User Interface). In addition to Broker, Bidder and Policy objects which were introduced before, there are also more informant objects: AgentInfo, Contract and PhonePort in the agent body. An AgentInfo object basically stores the information of an agent and the Agora information from which the agent could establish connection to the Micmac testbed. A Contract object holds the information of an agreement between a resource and the agent itself. In this simulation, the contract information only specified the identity and type of the resource, the identity of the calling agent and the type of job relationship between them. A Port object has information that represents a physical port in the system. The objects in Script Functions group are the implementation of functions which have been called in the scripts. Since the scripts are written in CLIPS style, these special objects get the input arguments from the function call statements of the scripts, process the data by Java codes and then return the results back to those scripts. Although the functions can be implemented by some CLIPS commands like 'if then ...', the preferred approach is to hide the implementation details of the functions from the script. The reason is that the sequential logic of the functions can be implemented and executed more efficiently in Java while only the rules will be highlighted in the scripts. The GUI group contains classes that are used to build the user interface on windows. There are two GUIs and some agent monitor windows. The voice call helpdesk GUI provides some buttons and a display for users. The user can either press the buttons or type the number directly to place a phone call. There is also a choice box that provides

a list of areas so that the user need not to enter the area codes themselves. In addition, the GUI also contains miscellaneous functions like re-dial and memory storage. Similarly, the modem data transfer GUI has the functions to dial the number and select the area code. Agent monitors are used to trace agent activities so that agent interaction can be easily monitored on the windows.

4.3.2 Agent communication in the fault scenario

To simulate communication among agents, the framework of the selected application domain has been set up on the Micmac testbed as shown in Figure 4-4. The Micmac testbed has become the channel for agent communication. Messages are exchanged between the agent heads and the agent bodies via the Agora provided by the Micmac testbed. The long distance service provider agents shown in the figure are just dummy agents which do not contain any intelligence but are simply mechanisms to reply all incoming messages in this simulation. Among these dummy agents, A2 is the source of semantic problem which has a misunderstanding with agent S as described in the fault scenario. No error was detected during the setup of the long distance application until the client had started using the service. Figure 4-5 is the message sequence chart that shows the message exchange in the fault scenario.

Figure 4-4: Framework of the selected application domain

Agents V is not shown in Figure 4-5 for simplicity and it is not involved in the fault scenario.

There is no communication between agent V and agent D. In the successful scenario of a voice application, agent V has similar message sequences as agent D (just replace D by V in the figure) except the application is a successful one that did not encounter any semantic problem. Both A1 and A3 are healthy. If A2 is replaced by one of these healthy resources, similar sequence of messages will be shown on the schematic and there will be no semantic problem.

Figure 4-5: Message sequence chart showing the messages exchanged between agents

Chapter 5

SEDO

Semantic Error Diagnosis Operation

5.1 Introduction

From the definition of a semantic error stated in section 1.2, the nature of a semantic fault is that it can only be recognized by the application agent. It is different from syntactic errors which are usually low-level problems caused by hardware failure or resources which are too busy to respond to a request before the timeout. Syntactic errors may occur during the application setup or operation. However, they can be detected and handled by neighboring agents. In contrast, semantic errors are faults discovered during the operation of an application and may be caused by a misunderstanding between network components or hardware failure of resources which cannot be detected by underlying agents. The long-distance service shown in Figure 4-1 is an example of a misunderstanding between agents. Agent S has no mean to verify the result because it is just a

broker selecting suitable resources. The selection of resources depends on the understanding relationship between S and the resources. This relationship has already been specified during system setup. Hence agent S will not perform detailed tests to see which resource is capable of providing a completely satisfactory service before making contract with it. Instead, it will select the resources based on the understanding relationship, otherwise an application setup process will involve a lot of overheads which is inefficient in a busy real-time service system. In addition, the only one who knows the problem is the application agent (V or D) because it is the agent that established physical connection to the resources during operation. S only gets the port address of one of the resources and then passes it back to V or D. Performing local tests by S implies that S needs detailed information and mechanism to set up phone channel connected to the resources (A1, A2 or A3). It will then test the effective transmission rate by sending and receiving test data via the port like what the application agent does. However, this is not feasible and thus it is hard to locate the source semantic problem.

In order to solve the semantic problem, a fault localization algorithm SEDO (Semantic Errors Diagnosis Operation) has been introduced to locate the agent or resource which is suspected of causing semantic errors and hence will be avoided in the future. When an application comes into operation after a successful setup, the application agent will monitor the overall performance at the semantic level. It will issue performance tests on the operation and wait for the results. By examining the test results, it determines whether an application is a successful one or has encountered semantic problem. If the application is successfully set up in the beginning, SEDO will have no diagnostic actions. Otherwise, the fault localization mechanism will start to locate the semantic faulty resource. SEDO will pass diagnostic information to all underlying agents involved in the

application so that they can select an alternative resource when a similar application is set up next time. However, that agent can also be chosen for other types of applications. It will be rejected only by a specific type of application which encountered the semantic error (described above) before. The network administrator can also input some test cases with the same application nature to find out the erroneous resources. The faulty resource will not be located until a similar application has passed all the performance tests. Hence the resource chosen at the last time (when semantic problem still existed) which has just been replaced by the other one (the new resource that does not cause the semantic problem) is determined to be the faulty resource. The approach of SEDO is not to investigate why the faulty resource has a problem. Instead it allows the underlying agents to recognize the occurrence of semantic problem so that they can avoid the located faulty resources in future resource selection process. To fix the faulty resource physically, the system administrator can gather information from the SEDO Information Base (SIB) to figure out the repair strategy on the resource. This way, the system can remain in operation and at the same time the system administrator can analyze the problem.

In this chapter, a simple semantic fault model is established based on the definition of semantic error of a multi-agent system stated in section 1.2. Then in later sections the operation of the diagnosis model of SEDO will be discussed in details.

5.2 A Simple Semantic Fault Model

Defining a general model of semantic error is not ‘on-the-fly’. It is still an open issue because

there are very few studies and real examples about this topic. Here a simple model of semantic error has been created from only the limited examples mentioned in previous chapters and hence SEDO may not be a generic solution for some other multi-agent system.

Figure 5-1: A simple model of semantic error

A simple semantic fault model is shown in Figure 5-1. The model shows how semantic errors are detected by the application agent during the operation of an application. In this model, A_0 is the application agent that initiates an application, performs tests and determines whether the application has encountered semantic problem. There is at least one R that is the resource selected

by some underlying agents during application setup. Also there may be some A's which are the underlying agents involved in both the application setup and operation stages. The model presumes that A_0 has some means of testing the performance on the application output. For example, A_0 may have physical connection to a resource so that it can test the effective data transfer rate between the resource and itself. However, A_0 does not know details such as the location or identity of the resource since the connection between A_0 and the resource has been established by some underlying agents (which may not be involved in the operation) during application setup. The underlying agents and resources are all hidden in a 'black box' by the local view of A_0 .

5.3 The Diagnosis Model of SEDO

Although semantic errors are discovered during the operation of an application, the diagnostic actions of SEDO take place on all the underlying agents that have been involved in the application setup stage. SEDO needs to update the semantic fault information so that the underlying agents can select appropriate resources according to the information. Figure 5-2 is the diagnosis model of SEDO:

Figure 5-2: The diagnosis model of SEDO

A₀ is the application agent. In the head of **A₀**, there are analyzers, watchers, scripts and the SEDO Temporary Base (STB). The analyzers, watchers and STB are new components introduced

by SEDO and they exist only in the application agent. An analyzer gathers information of agent/resource connection during application setup and generates diagnostic information. A watcher defines and assigns an identity for the application (the AppID) in order to distinguish which application a piece of information belongs to. It also monitors the overall performance of an application by issuing performance tests and verifying the results so that it can provide diagnostic information for the script. Scripts inside an agent have been introduced in section 2.2 and their duties are simply to receive job requests and send orders to the body of the agent. STB is a local repository inside an application agent and its duty is to store temporary diagnostic information of applications. All these components except the STB are rule-based expert system scripts written in JESS (Java Expert System Shell).

SEDO has also added new components to the system. The SEDO Information Base (SIB) is a global database that stores detailed information of all semantic errors encountered in the applications. The SIB is managed by the SIB Monitor which provides input/output (I/O) interface as well as graphic user interface (GUI) for accessing the SIB.

A connection model of an application is the hierarchy formed by the connection of the agents and all available resources which may be selected for the application. There are two connection models: One represents the possible agent/resource connections of application setup and the other

one represents the actual physical agent/resource connections during the operation of an application. The operation connection model is an extraction of the setup connection model since the agents/resources connected in the model are those involved in the application setup. The setup connection model of agents/resources provides important information for the analyzer and it is established by the connection of agents/resources according to the pattern of agent actions.

5.3.1 The Setup Connection Model of Application

A general physical agent/resource connection model represented by a tree hierarchy is shown in Figure 5-3. In the diagram, A_N s are agents that connect to the resources R_i ($i \geq 0$). M, P, T, K, S and Y are some integers greater than 0 ($T > P > M > 1$ and $Y > S > K > 1$). The value of N increases according to the position of an agent from top to bottom and left to right. For example, the application agent at the top level (level 0) is A_0 , the left most agent in the lower level (level 1) is A_1 and the following agent at the same level is A_2 and so on. Similarly, the value of i increases from left to right so that the left-most resource in the tree diagram will be R_0 , the next one at right will be R_1 and so on regardless of the level. An application may employ more than one agent or resource. A_0 has the role to initiate the application, issue performance tests and detect semantic error. In this model, A_0 is error-free. Other agents or resources, however, may cause semantic problem in later operation.

Figure 5-3: General physical connection model among agents/resources

Figure 5-3 is only a general physical connection model among agents and resources. However, it is not detailed enough to reflect the behavior of an application. Application behavior is represented by the complicated agent actions from which the setup connection model is established. Currently there is no defined agent/resource setup connection model for a multi-agent system. To define such a model is difficult because of the flexibility of the system configuration as

well as the dynamic agent actions and application behaviors in a multi-agent system. There may be undiscovered agent actions but currently SEDO can recognize four patterns of agent action: *XOR*, *AND*, *ORDER* and *LOOP*. The first three patterns have also been described by simple action language introduced in [B97] that used different terms to describe the patterns: Choice (XOR), Parallel (AND) and Sequences (ORDER). The agents involved during the application setup are connected together physically and logically by these actions. Hence the agent/resource connection model of application setup has been represented by a tree of agents and resources which are connected and related by these patterns.

5.3.1.1 The XOR Pattern

Figure 5-4: The XOR agent action. Only one of the underlying agents will be employed

The pattern of XOR agent action is demonstrated in Figure 5-4. During application setup, A gets information from the policy object and identifies the most suitable resources it can employ. In

this pattern agent A finds more than one agents $A_1, A_2, \dots$ and A_n which are all capable of doing the same job but only one of them will be chosen exclusively. A will then issue a job request j to that agent. Here the role of A is usually a broker which is responsible for selecting resources according to application requirements. The selector agent S in Figure 4-1 is an example of this pattern.

5.3.1.2 The AND Pattern

The AND agent action is demonstrated in Figure 5-5:

Figure 5-5: The AND agent action

In this pattern, agent A employs all of its underlying agents $A_1, A_2, \dots$ and A_n . The operation of a connection between A and one of these underlying agents may be parallel with one another. The job requests to the underlying agents may be the same or different from each other. For

example, an agent may request for printing jobs from all available printers to print more than one copy of the same document. On the other hand, an agent may gather the results of different job requests from its underlying agents and passes them back to its calling agent.

5.3.1.3 The ORDER Pattern

The ORDER agent action is demonstrated in Figure 5-6:

Figure 5-6: The ORDER agent action

Similar to the AND pattern, agent A employs all of its underlying agents A_1 , A_2 , ... and A_n . However, the operation of connection paths will run in order. The operation of path A-- A_2 will not start until path A-- A_1 has finished its operation and returned the result. Similarly, path A-- A_3 will start only when path A-- A_2 has finished its operation and returned the result, and vice versa. Again, the job requests to the underlying agents may be the same or different from each other. The

printer server agent (agent **S**) in Figure 1-1 is an example of this pattern. Agent **S** has to retrieve the whole data file from the file server **F** before it can ask for a printing job from the printers.

5.3.1.4 The LOOP Pattern

Agent actions in LOOP pattern is an interesting topic in network diagnosis. The pattern makes the behavior of agents and setup of an application more complicated because the sequence of actions are completely hidden from the topology of agent connection so that it seems to be 'invisible' by the physical connection viewpoint. Figure 5-7 shows the LOOP pattern formed by two agents.

Figure 5-7: The LOOP agent action

In fact, the 'loops' described in this pattern are not erroneous infinite loops. Instead, it is merely a course of different job requests between two agents. Suppose A has the capability to do jobs j_1 and j_2 while A_1 can do jobs j_3 and j_4 . Now the task of an application is to do j_1 . Provided that j_1 requires the outcome from j_3 and j_3 in turn needs the result from j_2 in order to finish the job. By the agent connection viewpoint, A will ask A_1 to provide service and A_1 will also ask A to provide service such that there is an apparent 'loop' between the two agents. However, by the job sequence viewpoint, it is merely a straight course of different job execution. It is a loop between agents physically but not between jobs logically.

In the application shown in Figure 5-8, the application agent A_0 requests for both jobs j_1 and j_2 from A_1 and A_2 respectively (the AND pattern) during the application setup. A_1 in turn requests for job j_3 from A_3 and so on. There is a LOOP pattern between A_1 and A_3 and an ORDER pattern between resource R_4 (j_7) and agent A_4 (j_8). Other agents are connected in XOR pattern. The schematic does not only show the physical connection among agents/resources but also reflect their logical relationship.

Figure 5-8 is an example of an agent/resource connection model during the application setup.

Figure 5-8: An example of the setup connection model

5.3.2 The Simplification of Setup Connection Model in SEDO Diagnosis Model

In the diagnosis model of SEDO, the setup connection model is simplified. The LOOP pattern has become visible as being represented by a sequence of jobs. In addition, there is no longer any differences between the AND pattern and the ORDER pattern in SEDO. The sequence of the agent actions is trivial in the diagnostic model since SEDO merely wants to know how A will connect to the underlying resources at the high level viewpoint so that all resources with the

ORDER pattern will be used eventually as in the AND pattern. As a result, only the XOR and the AND agent action patterns are highlighted in the simplified trees. In the case of OR (inclusive), SEDO will combine XOR and AND patterns in order to represent the relationship. For example, an agent may use resource A, or B, or both A and B, so it will be expressed as: A xor B xor (A and B).

Figure 5-9: The simplified setup connection model from Figure 5-8

Figure 5-9 is a simplified setup connection model from the original one in Figure 5-8. The loop between A₁ and A₃ in Figure 5-8 is converted to a sequence of jobs instead. The ORDER

pattern is then treated as the AND pattern. In the new tree, the small triangular symbols like the one between A_0 and A_1/A_2 are the *AND nodes* that represent the AND patterns among agents. They are labelled 'ANDx' (where x is a number). Similarly, the small circles labelled 'XORx' are the *XOR nodes* that represent the XOR pattern between agents.

The simplified tree information is supplied by the agents in the format of expert system facts and they become required input for the analyzer. In a rule-based expert system, facts are used as pre-conditions of rules and trigger rules. When all the facts required by a rule have been asserted into the expert system, the rule will be fired. The analyzer shown in the Figure 5-2 is basically a rule-based expert system that waits and receives the setup connection tree information and asserts it as facts. Then the analysis process will start after the analyzer has received all facts and the final result of the application. The following are some of the facts representing the tree in Figure 5-9:

(root A0)	: A0 is the root of tree, i.e. the application agent
(node Agent A1)	: A1 is an agent node
(node Rsc R1)	: R1 is a resource node
(node AND1 j1 j2)	: AND1 is a 'AND node' that requires both results of jobs j1 and j2
(node XOR1 j6 j6)	: XOR1 is a 'OR node' that requires a result of job j6 from either one of the two possible j6 connections
(connect A1 A3 j3)	: A1 connects to A3 and needs A3 to return job j3 result
(connect AND1 A1 j1)	: AND1 connects to A1 and needs A1 to return job j1 result

An agent knows which connecting agents or resources to choose in an application so that it can send information to the analyzer. However, an agent will do so only if the application has no 'bad record', i.e. a new application or a 'healthy' application which encountered semantic errors before but has been 'fixed'. During the setup of an application, the application agent checks for

diagnostic information stored in a local information base, the SEDO Temporary Base (STB), to see whether an application encountered semantic errors or not in the past. If the application had a semantic problem before and the source of problem has not yet been located, diagnostic information of the application should already exist in the STB. The application agent can pass the information to the underlying agents along with the job requests. Otherwise, the application agent merely passes a text string “none” to indicate that the application is a new one and thus the underlying agents will send connection information to the analyzer.

Consider the case of a LOOP pattern from the previous example, the following are some facts of the setup tree connection information from the left subtree in Figure 5-9:

Fact 1 =>	(connect AND1 A1 j1)	; AND1 connects to A1
Fact 2 =>	(connect A1 A3 j3)	; A1 connects to A3
Fact 3 =>	(connect A3 AND2 link)	; A3 connect to AND2
Fact 4 =>	(connect AND2 A1 j4)	; AND2 connects to A1
Fact 5 =>	(connect A1 R1 j5)	; A1 connects to R1

The facts above are listed in order according to the figure. However, the analyzer has no knowledge about how they actually connect to each other upon receiving these facts. The facts may not be received in the order shown above at the analyzer side such that the analyzer may have a wrong interpretation on the connection geometry, for instance, fact 5 was received prior to facts 3 and 4:

Fact 1 =>	(connect AND1 A1 j1)	; AND1 connects to A1
Fact 2 =>	(connect A1 A3 j3)	; A1 connects to A3
Fact 5 =>	(connect A1 R1 j5)	; A1 connects to R1
Fact 3 =>	(connect A3 AND2 link)	; A3 connect to AND2
Fact 4 =>	(connect AND2 A1 j4)	; AND2 connects to A1

As a result, the analyzer will misunderstand and generate incorrect diagnostic information as shown below:

Figure 5-10: Wrong interpretation of the simplified setup connection tree

The misinterpreted information may lead to unexpected results in SEDO. In order to avoid such problem, there is a list called the *CallList* which is passed along with a request message. The purpose of this list is to record which agent has been called during the chain of job requests. If the same agent is employed again just like A_1 in the example, the underlying agents will know from the list that the agent has been previously called. Thus other agents will simply add an index num-

ber after the original ID of that agent to make it look like another new agent. If the original ID of the agent is A_1 , the new ID representing the agent in the facts will be $A_1##01$ in the 2nd time it is being called and then $A_1##02$, $A_1##03$, etc.. In the previous example, the list of facts to be sent to the analyzer become:

Fact 1 => (connect AND1 A1 j1)	; AND1 connects to A1
Fact 2 => (connect A1 A3 j3)	; A1 connects to A3
Fact 3 => (connect A3 AND2 link)	; A3 connect to AND2
Fact 4 => (connect AND2 A1##01 j4)	; AND2 connects to A1 (2nd time)
Fact 5 => (connect A1##01 R1 j5)	; A1 (2nd time) connects to R1

Hence the problem has been solved. The index added to the ID can also be used as a counter to check for a potential erroneous infinite loop.

5.3.3 The Actions in Analyzer

The main job of an analyzer is to accept the simplified setup connection tree information from the underlying agents and then generate diagnostic information to help locate the source of a semantic problem. An analyzer object is created and associated with a watcher when the watcher is being initialized during system setup. The analyzer starts waiting for a message that contains a piece of information called the AppID. The AppID is an identity to distinguish different type of application and will be discussed in more details in section 5.3.5.2. The location of the analyzer is hidden from other agents; however the agents send the AppID along with tree information so that the analyzer can retrieve correct tree information by matching the same AppID. The analyzer only accepts the setup tree information based on the same AppID that has been assigned to it so that tree information from different applications will not confuse the analyzer. After receiving the simplified setup connection tree information, the next step an analyzer will do is to wait for the final

result of the application. The watcher in the application agent will verify the application to see if a semantic error was found. If notification of a semantic error was received, the analyzer starts the analysis process.

There are three procedures in the analysis process: reduction, transformation and table generation.

5.3.3.1 Reduction of Simplified Tree

The reduction process reduces the simplified tree so that only the logical relationship between the application agent and the underlying resources are highlighted. As a result, all the underlying agent nodes present in the simplified tree will be eliminated. In addition, any two relational nodes (AND or XOR) with the same pattern connected in sequence will merge together. The reduced tree of Figure 5-9 is shown in Figure 5-11.

Figure 5-11: The reduced tree from Figure 5-9

The reduced tree represented by facts is understood only by the expert system. In order to allow the transformation function (will be discussed in the next section) to work on it later, it will be translated into a logical expression. For example, the AND pattern of two nodes N_1 and N_2 will be represented as $(\wedge (N_1) (N_2))$ where ' $\wedge$ ' is the AND symbol. Similarly, the XOR symbol is ' $\vee$ '. The logical expression f derived from the reduced tree in Figure 5-11 is:

$$f = (\wedge (R_1) (\vee (R_2) (R_3)) (R_4) (\vee (R_5) (R_6) (R_7)))$$

The reduction process is operated by a set of rules in the script 'analysis.clp' which is shown in the Appendix.

5.3.3.2 Transformation of Reduced List

The next procedure is the transformation of the reduced list. Since the transformation function is written in Java, it does not accept facts as input arguments. Thus the reduced tree described in the previous section has been translated into a logical expression. The purpose of transformation is to transform the logical expression of the reduced tree into the disjunctive normal form (d.n.f.). There are a lot of references describing d.n.f. and one of them is [G89]. Basically, the d.n.f. is a representation of f as the disjunction of fundamental conjunctions except the disjunction is exclusive:

$$f = (\vee (rs_0) (rs_1) \dots (rs_m)) \quad ; rs = \text{a set of resource}$$

and each resource set (rs) is a fundamental conjunction of resources (rsc):

$$rs_i = (\wedge (rsc_a) (rsc_b) \dots (rsc_n))$$

As a result, the transformed tree represented by the new expression has only the application agent remaining at the top level that connects to all available resource sets. There are only two levels in the tree hierarchy where level 0 (the top level) is the application agent and level 1 (the bottom level) is a number of resource sets which are related by the XOR pattern (exclusive disjunction). The resources within the same set are related by the AND pattern (conjunction). The transformed tree and list of Figure 5-11 is shown in Figure 5-12.

The transformation function is a Java recursive function that searches and transforms the tree by applying the following rules:

Rule 1: Conversion from AND to XOR

$$\begin{aligned} & (\wedge (| (rsc_x) (subtree_1)) (subtree_2)) \\ \Rightarrow & (| (\wedge (rsc_x) (subtree_2)) (\wedge (subtree_1) (subtree_2))) \end{aligned}$$

Rule 2: Merge of XORs

$$\begin{aligned} & (| (| (subtree_1) \dots (subtree_m)) (subtree_n)) \\ \Rightarrow & (| (subtree_1) \dots (subtree_m) (subtree_n)) \end{aligned}$$

A ‘subtree’ may be a resource or an expression. The second rule is applied at all levels except level 0 (the application agent). By applying the two rules, a transformed tree is expressed in the d.n.f.. The interface of the function, ‘(transformed-list <reduced-list>)', is provided as a function call in the script ‘analysis.clp’ and the actual implementation is embedded in the function ‘transform’ of the class ‘TransformListFunction’. All of these are shown in the Appendix.

The Transformed Tree:

The Transformed list:

(| (^ (R₁)(R₂)(R₄)(R₅)) (^ (R₁)(R₃)(R₄)(R₅)) (^ (R₁)(R₂)(R₄)(R₆))
 (^ (R₁)(R₃)(R₄)(R₆)) (^ (R₁)(R₂)(R₄)(R₇)) (^ (R₁)(R₃)(R₄)(R₇)))

Figure 5-12: The transformed tree and list from Figure 5-11

5.3.3.3 Generation of Table from Transformed List

The final step of the analysis process is table generation. The analyzer uses the transformed list to generate a table for diagnostic purpose. Each set of resources shown in the transformed tree is a possible combination of the resources which can be selected during application setup. A table is generated from this information in which each resource set is inserted as a row of the table. These rows are then arranged in an order such that there is only one different resource between

two neighboring rows. Each resource set in Figure 5-12 is mapped into the table as shown in Figure 5-13.

Row (Set #)	Resources
1	R1, R2, R4, R5
2	R1, R3, R4, R5
3	R1, R3, R4, R6
4	R1, R2, R4, R6
5	R1, R2, R4, R7
6	R1, R3, R4, R7

Figure 5-13: Table showing the arrangement of resource sets

This table will be stored as a piece of diagnostic information in the STB for future resource selection strategy. The interface for the function that generates a table is:

'(generate-table-from <transformed-list> <initial-rsc-set> <AppID> <reduced-list>)'

in the 'analysis.clp' script. The function is 'genTable' in class 'GenerateTableFunction' which is shown in the Appendix.

5.3.4 The SEDO Temporary Base (STB)

The SEDO Temporary Base (STB) is a local information repository that only resides in the head of an application agent. Upon receiving the notification of a semantic error, the analyzer starts the analysis process. Then it will store the simplified tree information as well as the table

into the STB. The watcher can retrieve these information from the repository and perform fault localization process.

A STB stores information as key-value pairs like a hashtable. It uses the application identity (AppID) as a key and the STBRecord objects as the values associated with the key. AppID is generated by the watcher that specifies a group of applications with the same nature. The level of likeness can be adjusted in the watcher. The generation of AppIDs will be explained in the next section. A STBRecord contains some fields to store information like the AppID, the simplified tree, the table of an application as well as a table pointer initialized to 0 (pointing to the first row of table initially). Figure 5-14 is the structure of a STB.

AppID	STBRecords
V01-S01-002	<div> AppID: V01-S01-002 Simplified Tree: [(root A1),] Resource Table: [(^ (R1)(R2)), (^ (R2)(R3)) ...] TablePointer: 0 </div>
M02-S03-001	
.....	

Figure 5-14: The structure of a STB

5.3.5 The Watcher Script

A watcher is a rule-based expert system script written in JESS. It only resides in the head of an application agent. Its job is to generate an application identity (AppID), issue performance tests to

verify the occurrence of semantic problems, trigger the analysis process in the analyzer and send diagnostic information to the underlying agents in order to locate the source of problem.

5.3.5.1 Performance Tests and Verification of Test Results

To issue performance tests, the application agent has tailored methods to check all the required quality of service of an application so that it can figure out whether a requirement has been satisfied or not. The watcher simply specifies the test conditions and sends out test requests. A performance test is described by a keyword, a conditional symbol and a threshold. A keyword is usually the name of quality such as 'TransferRate' or 'Bandwidth', etc.. A conditional symbol is either '>' (greater than), '<' (less than) or '=' (equals). The threshold is a numerical value. For example, a performance test for the modem data transfer service may be 'TransferRate>14.39' and the Tester object in the body will return a result 'pass' if the modem data transfer rate is greater than 14.39 kbit/sec. There may be more than one performance test for an application. If *not* all the results received from the Tester object are 'pass', a semantic error is said to be detected by the watcher. As a result, the watcher sends notification to the analyzer to start the analysis process. Diagnostic information can then be generated and put into the STB.

5.3.5.2 The Use of Application Identity -- AppID

AppID is the key to retrieve diagnostic information. All the applications which have the same nature and properties will share a unique AppID. AppID can also be used as the message identity to distinguish the messages from different applications. An AppID is generated by combining the agent identity (agentID), the script object identity (scriptID) and a number that represents a row in the AppID table storing detailed description of the application. The level of details can be

adjusted in the watcher so that different applications can be categorized accordingly. In a long-distance call service, the area code can be used as part of the information to generate the AppID so that all long-distance service to the same area will have the same AppID. For instances, there are seven phone call applications and their details are shown in Figure 5-15.

Application	HelpDesk Agent ID	Script ID involved	Category	Type	Area Code	Phone #
Voice Call 1	VoiceCallAgent_1	001	VoiceCall	long_distance	416	123-4567
Voice Call 2	VoiceCallAgent_1	001	VoiceCall	long_distance	416	234-5678
Voice Call 3	VoiceCallAgent_1	001	VoiceCall	long_distance	852	345-6789
Voice Call 4	VoiceCallAgent_1	001	VoiceCall	local	---	456-7890
Modem Call 1	ModemCallAgent_1	001	ModemCall	long_distance	416	567-8901
Modem Call 2	ModemCallAgent_1	001	ModemCall	long_distance	416	678-9012

Figure 5-15: Examples of applications used to generate the AppIDs

There are four voice call applications and two modem data transfer applications. Five of them are applications using long distance phone service and one is a local call application. The identities of the agents responsible for voice call services and modem data transfer services are 'VoiceCallAgent_1' and 'ModemCallAgent_1' respectively. Both agents have their own script objects for different type of applications. In the example above, the voice agent has the script with ID '001' for all phone call applications. Similarly, the modem agent has its own scripts. A function in the watcher uses these information to generate the AppIDs. The level of details or particularity can be adjusted so that an AppID can be used to represent an unique application or a group

of similar applications. First of all, the details of the first application in the table can be expressed as an information string as follow:

'VoiceCallAgent_1-001-Type^long_distance-AreaCode^416-PhoneNum^1234567'

Similarly, the details of the next application can be represented by:

'VoiceCallAgent_1-001-Type^long_distance-AreaCode^416-PhoneNum^2345678'

However, in practice the detail level is reduced so that they can be used to describe a group of applications but not just one unique application. A better representation of the first application hence becomes:

'VoiceCallAgent_1-001-Type^long_distance-AreaCode^416' or

'VoiceCallAgent_1-001-Type^long_distance'

The first string implies that all applications for the long distance voice phone call service with an area code 416 share the same AppID. The other string is more general for all long distance voice phone call services regardless of the area codes or the phone numbers.

A local AppID table in the watcher records these string as the data and assigns an ID for each of these string. The ID becomes the AppID of the group of applications represented by the string and each AppID has the following form:

AgentID-ScriptID-N

where N is an integer greater than or equal to 0 that represents the row number of the AppID table.

The AppIDs generated from the applications in the previous examples hence become 'VoiceCallAgent_1-001-0' and 'VoiceCallAgent_1-001-1' respectively. The local AppID table is a hashtable that uses the AppID as a key to store the corresponding details of an application. There are two pieces of information in each row of the AppID table: an AppID and the details of an application expressed as a string.

5.3.5.3 Localization of Faulty Resource and Agent

The finite state machines in Figure 5-16 demonstrate the behavior of a modified script and a watcher respectively in the application agent. The script has been simplified to have only three states in the figure. All the results or responses from the body are no longer received by the script but instead filtered by the watcher. If an application was successful (passed all the performance tests) the first time, the watcher will do nothing but wait for another job request. If semantic error was found in an application, the watcher will notify the analyzer to start analysis and generate the table as discussed in section 5.3.3.3. Hence, next time when an application with the same AppID is set up, the watcher will choose the next set of resources from the table stored in the STB and send the information to the underlying agents. The underlying agents can then choose resources accordingly based on the information provided. The diagnostic actions will continue until an application with the same AppID is found to be successful or no more resources are available from the table. The system administrator can also apply a set of test applications (have the same AppID) to locate the fault. If all available resources under an agent have been tested but the application still has semantic problem, but the problem is solved when the resources of another agent are employed; SEDO concludes that the first agent is the faulty agent and will update the information in the same way.

Figure 5-16(a) Behavior of the modified Script in the agent head

Figure 5-16(b) Behavior of the Watcher in the agent head

Figure 5-16: Finite state machines showing the behaviors of the modified Script & the Watcher

Once an application has been proven successful, and the STB contains the diagnostic information of the application, the watcher will send the notification to the analyzer. The analyzer will then determine which resource has been replaced in this instance and consider it to be the source of the semantic error. If all underlying agents and resources attaching to an agent failed the application, the agent itself would be considered the faulty resource instead of the underlying agents / resources. The analyzer will update a badlist that holds a set of faulty resources of the application as well as update the SIB by sending information to the SIB Monitor. Finally, it will clear the STB record of the application in order to save local memory space. By passing the badlist along with job request messages, faulty resources can be avoided in the future.

5.3.6 The SEDO Information Base (SIB) and SIB Monitor

Figure 5-17: The SIB Monitor and SIB

The SEDO Information Base (SIB) is a global information base that stores the information of all semantic errors encountered in the whole multi-agent system. The analyzer described earlier

updates the SIB when the faulty resource is located. The system administrator can then use an interface, the SIB Monitor, to get information such as which performance test failed, the list of faulty resources, the simplified setup connection tree information (expressed in facts) and the detailed description of the application. The administrator can clear the SIB records at anytime to reset the system into the original state. Figure 5-17 is the outlook of the SIB Monitor.

Similar to STBs, the SIB uses key-value pair to store information. It also uses the AppIDs as the keys and a SIBRecord object associated with each key. A SIBRecord contains the details of an application, setup connection tree, resource table, a list of resources suspected of causing semantic errors and a list of failed performance tests.

5.3.7 The Tester Object in Application Agent Body

In the body of an application agent, a Tester object is created once a Broker object has passed all the results to the head. It waits for the request from the watcher and starts testing the quality of service when it is asked to do so. If an application is successfully set up and the result is returned by underlying agents, the watcher will issue performance test requests to check the overall performance. The Tester object will start testing the quality of service upon receiving the test request. Once the tests have finished and the results have been passed back to the watcher, it will be destroyed by the Broker object. When the test request arrives, the Tester object compares the requirement with the value measured, and then returns the results to the watcher. In the long distance service example, after the helpdesk agent connects to the phone port, a Tester object is created to measure the data transfer rate or the error rate. The behavior of a Tester object is shown in Figure 5-18:

Figure 5-18: The behavior of the Tester object

There may be varied ways of performing the performance tests and capturing the results. The Quality Assurance Language (QuAL) introduced in [F94] is one of the methods to verify the end-to-end quality of service requirement in a Tester object in the voice/data transfer example. In the printer example demonstrated in Figure 1-1, A device may be connected to or internally built into each printer so that the image of printout could be captured, returned to the caller and compared to the original one (e.g. applying OCR -- Optical Character Recognition technology). The assumption is that the application could always have means to test the outcome.

5.3.8 The Modified Broker Object in Agent Body

Figure 5-19 shows the modified Broker object in the agent body. The resource selection is more dynamic in that it depends on the job order received from the head. If the job order message contains diagnostic information, the broker will select the resource accordingly. There are three

pieces of diagnostic information included in each job request (order) message: *SedoInfo*, *BadList* and *CallList*.

(* The use of *SedoInfo* will be discussed in more details later)

Figure 5-19: The behavior of the modified Broker object in the agent body

SedoInfo is a set of resources selected from the table which is generated by the analyzer in the application. If it is a brand-new application or has always been a successful one in the past, SedoInfo will contain only a string value “none”. In this case, the underlying agents will send the setup connection tree information to the analyzer. Otherwise, SedoInfo will contain some caller-callee pairs. Caller in a caller-callee pair is the ID of an agent appearing in the setup connection tree, where callee is the ID of one of the underlying agent or resource attaching to the caller agent. If an underlying agent finds that it matches the caller itself from SedoInfo, it will select the callee for the job in a higher priority, i.e. regardless of the rules or policies applied usually as long as it is healthy.

When an agent connects to its underlying agents with the XOR pattern, it only chooses either one of them and sends it the job request. Nevertheless, unselected agents may hide extra connection tree information beneath. In order to send the complete setup tree information to the analyzer, the agent sends a ‘fake’ job request message to all the unselected agents. SedoInfo in a ‘fake’ job request contains a string value “tree” so that the called agent will just send connection tree information to the analyzer and forward the ‘fake’ job requests to all its underlying agents. Hence the setup connection tree received by the analyzer is a complete representation.

Provided that SedoInfo is “none” which has no diagnostic information, either the first available resource in the local resource list or a preferred one indicated by the job order from the agent head will be chosen. The broker will look for another available resource if the resource is unhealthy. Nevertheless, if SedoInfo contains diagnostic information, an agent will not choose other available resources from the resource list even if the agent/resource specified by SedoInfo is

found unhealthy. Instead, the broker will reply a message to the head that indicates a failed resource selection. The watcher hence simply chooses the next set of resource from the table the next time a similar application is to be setup.

BadList is a list of semantic faulty agents or resources found by SEDO. An agent will always ignore these agents or resources in the highest priority unless the bad-list in the watcher of the application agent is reset by the SIB Monitor. Since each bad-list is associated with an AppID, the agent/resource avoided from BadList information in an application may still be chosen by other applications with different AppIDs.

CallList has already been introduced in section 5.3.2. It is a list of agents that have been involved during the application setup. This information is useful for simplifying the LOOP pattern that if the same agent is called again during the setup of the same application, it will find itself already on the list and will modify its ID to make itself look like another agent. This also applies to the callers of the repeated agents. An agent (the caller) may recognize that one of the agents in CallList is the one which it is going to select. Hence it modifies the ID of the called agent the same way as the called agent will do. Consider the example shown in Figure 5-9. The CallList information received by A1 at the first time was [A0]. Then A1 added itself to the list and the new list had been passed along with the job request to A3. A3 got the CallList information [A0,A1] and found that A1 -- the agent it was going to call was already in the list so that it modified A1 into A1##01 and used this new ID in the setup connection tree information. It also put itself on the list and hence the CallList information received by A1 at the 2nd time became [A0, A1, A3]. A1 realized that it was called before so it used A1##01 too as its new ID for both the

new CallList and the setup connection tree information. Thus the new CallList became [A0, A1, A3, A1##01]. If A1 is called again later, the new CallList will become [A0, A1, A3, A1##01, A1##002,], and so on.

The modified broker in the application agent will also create a Tester object for application performance tests as discussed in section 5.3.7.

5.4 Simulation -- Applying SEDO to the Fault Scenario

Figure 5-20 and 5-21 are the message sequence charts (MSCs) that show the messages exchanged between agents in the fault scenario discussed earlier in section 4.2. In this simulation, SEDO has been applied to solve the problem.

Assume that a client wants to send modem data to Washington. Since MCI (A2) offered the most economic price for long-distance service to U.S. based on the rules defined in the script of agent S, it was chosen for the application. However, in the fault scenario, MCI used compressed trunk and led to the semantic error on modem data transfer. Figure 5-20 is a message sequence chart demonstrating how the agents behaved during the setup and operation of this newly created application in the simulation.

Figure S-21: The setup and operation of the 2nd modem data transfer application

Figure 5-21 demonstrates another client who asked for the modem data transfer service in the simulation. The destination was again Washington. The watcher in agent D recognized that it was an application having the same nature as the first one and hence retrieved information from the STB. The STB contained the table generated by the analyzer since the analyzer realized the occurrence of the semantic error in the first application and had finished the analysis process. The table was [[A2], [A3], [A1]]. The watcher sent the job request including the SedoInfo information: [A3:S] to agent S which found A3 (Sprint) from SedoInfo and chose it instead of A2 (MCI). Sprint did not use compressed trunk and thus the application was a successful one. The watcher sent the notification of success to the analyzer which in turn updated the badlist in watcher, sent detailed information to the SIB Monitor and cleared the record of this application in the STB. The system administrator can then select one of the AppIDs from the AppIDs option window (Figure 5-17). There is the letter 'N' just before an AppID (not shown on the Figure) that indicates a newly created SIB record which has not been examined by anyone. By clicking the AppID, 'N' will be changed to the minus symbol '-' and all corresponding diagnostic information will be shown on fields 'Details', 'Setup Tree', 'Rsc Table', 'Failed Tests' and Suspect List' accordingly. In this simulation, the content of each field is shown below:

Details: "Root : ModemCallAgent_1 (Real AgentID of agent D in simulation)
Script : 001
Type : long_distance
AreaCode : Washington"

Setup Tree: A set of facts that represent the simplified tree information (not shown here)

Rsc Table: [[MCI1], [Sprint1], [AT_T1]] (Real AgentIDs for A2, A3 and A1 respectively)

Failed Tests: [TransferRate>14.3]

Suspect List: [MCI1]

Provided that Sprint also introduces a semantic problem to the application, the analyzer will update the suspect list in the SIB and the new suspect list will become [MCI1, Sprint1]. In addition, the letter 'U' will be shown before the AppID. The administrator can then notice that the existing diagnostic information of that AppID has been *updated*. The information of other fields will not be changed until the record has been cleared and the SIB Monitor can accept new information again.

When some similar applications are to be set up in the future, agent S will no longer choose MCI since the information in Badlist contains [A2]. Instead it will select A3 (Sprint). New setup connection tree information will also be sent to the analyzer and the new table will become [[A3], [A1]]. The system administrator may select the AppID representing the application on the SIB Monitor and then press the 'clear' button to reset the application. A reset message will be sent to the watcher to clear the badlist.

Although MCI will be rejected by the long-distance modem data applications in this case, other applications such as the long-distance voice call applications will still accept MCI since the AppIDs of the voice applications are different from those of the modem ones.

5.5 SEDO and 'Trust' among Agents

The algorithm of SEDO discussed so far is a deterministic approach. It avoids the faulty agents or resources once they are determined to be the sources of semantic problems. In some multi-agent systems, the understanding relationship between two agents is more complicated in that it con-

tains 'trust' to determine whether the callee is the most trustworthy or reliable one to employ. The formal definition of 'trust' can be found in [M94]. Basically, 'trust' in a caller-callee pair in a job can be expressed as a quantitative value or magnitude that measures the trust the callee places in the caller for the job. For instance, in the application domain described in section 4.1, the trust of agents A1, A2 and A3 in agent S is initially the same, say 1.0 for the modern data transfer application. All these resources should have the same priority to get the job assuming that the head of agent S does not enforce its body to choose a preferred resource. However, after A2 has been found to be the faulty resource, SEDO will send the BadList information to the underlying agents as before in the setup of a similar application, but the agent (S) that employs the faulty resource (A2) will have less trust on that resource instead of remove it from the local resource list. The difference is that if the other two agents (A1 and A3) are off-line or they have lower trust values later, the faulty resource (A2) can be re-selected. Hence the trust can be treated as the probability of being successful based on the past history (the statistics of success).

The components of a multi-agent system are far more complicated and unstable than one imagines in practice. The overall performance of an application may vary a lot depending on how busy are the components involved, how old are the hardware components (need maintenance?) and many other factors. A resource may have caused a semantic problem in only one special occasion before but it works perfectly in other instances. Thus SEDO in the trust model brings the multi-agent system a pseudo-probabilistic approach to solve the semantic problem 'softly' (i.e. not a deterministic approach). This approach may be good for the dynamic behavior of a multi-agent system. The drawback is that the reaction may not be fast enough to replace or eliminate the faulty resources in some life-critical situations. The best way should be to combine both

approaches dynamically according to the needs of an application.

Chapter 6

Conclusion

In this thesis, a new fault localization algorithm, SEDO, has been developed to solve the semantic problem based on the simple fault model of semantic errors presented in section 5.2. A long distance service has been simulated in a multi-agent environment to demonstrate the occurrence of a semantic error. Then SEDO has been applied to the fault scenarios described in sections 4.2 and 5.4 in the simulation and successfully solved the problem. The simulation has not only focused on the operation of SEDO but also how the agents in a multi-agent system communicate internally as well as externally with other agents, and how this has affected the design and implementation of SEDO.

SEDO can work dynamically depending on the nature of an application. After locating the faulty resource, SEDO can either avoid it completely, or decrease the trust placed in it to lower the intention of selecting it by other agents. Hence SEDO can work in a deterministic or pseudo-probabilistic fashion. Since a semantic error will affect only a specific type of application, a faulty

resource or agent detected by SEDO in a specific type of application can also be used by other applications. In the long distance example, MCI will not be selected by modem applications but can still be employed by voice applications. In this way SEDO shows its ability to focus on the problem without affecting the operation of the rest of the system.

6.1 Limitation

SEDO assumes that the performance or quality of services can always be verified. The system or the applications themselves have to define a set of methods to test the quality of outcome. Then SEDO will start the fault localization mechanism accordingly. Also, SEDO cannot detect more than one faulty resources simultaneously in each turn of operation. If there are two resources that will cause the semantic problem, SEDO will eliminate the first one it detects and update the information in the first turn of operation. Then later it will repeat the same procedures when the second faulty resource is employed and semantic error is encountered. The operation of SEDO is dependent upon the dependence graph formed during the application setup. If the dependence graph is just a chain of agents connected together and there is an faulty agent in this chain, SEDO will not be able to determine which agent has caused the semantic problem. In this case, the system administrator or designer has to examine where and what the problem is.

So far the scale of the multi-agent applications demonstrated in this thesis are small. All of them involve less than 6 levels of connection tree hierarchy and the total number of agents is less than 10. In this scale SEDO will not introduce serious performance issue. However, in large and complicated applications that involve hundreds of agents, the overhead caused by SEDO may

affect the system performance.

6.2 Future works

Future work may include the study of applying SEDO on different multi-agent systems and improve SEDO so that it can recognize more patterns of agents actions. For example, more complex application may introduce the patterns of XOR-Join or AND-Join, or some ORDER patterns which may affect the setup connection tree in the future. In addition, a counter may be applied on the LOOP pattern so that SEDO can keep track on the number of physical cyclic connection in order to determine whether it is a logical infinite loop or merely a course of distinct tasks. Overhead of SEDO can also be simplified for better performance purpose.

New development of SEDO may include an advanced artificial intelligence system that has the capability to analyze the real cause of semantic problems based on the information provided by current SIB in the future. SEDO can also adopt mobile agent technology in other multi-agent systems so that mobile analyzers can be deployed and collect diagnostic information actively throughout the hierarchy of agents. Also, the analysis process (the most time consuming process) can be operate anywhere so that system resources can be used more efficiently.

Most multi-agent systems are currently under development. The system architecture, agent structure, agent communication protocols, etc. have not yet been standardized (or will never be). As a result, network management in a multi-agent system will always be an open issue. It is very difficult or currently infeasible to derive a general algorithm to diagnose the same problem in all

existing multi-agent system because of different needs, purposes and focuses of different system.

Networking management in multi-agent systems will be very challenging since the flexibility, adaptability and dynamic nature of agents have to be considered.

References

- [AP96] J. F. Arcand and S. J. Pelletier, *Cognition based multi-agent architecture*. In Intelligent Agents Volume II - Proceedings of the Second International Workshop on Agent Theories, Architectures, and Languages, M. Wooldridge, J. P. Muller & M. Tambe (Eds.), Lecture Notes in Artificial Intelligence, Springer-Verlag, 1996
- [B96] M. Banville, *Sonia: an Adaptation of Linda for Coordination of Activities in Organizations*, Conference Coordination '96, April 1996.
- [B97] M. Barbuceanu, *How To Make Your Agents Fulfil Their Obligations*, Proceedings of PAAM-98, London, England, 1998. <http://www.ie.utoronto.ca/EIL/ABS-page/paam98.ps>.
- [D96] P. Desjardins, *Mitel-CATA (The Micmac Software Testbed)*, Technical Report from CITI, 1996.
- [DK93] N. W. Dawes and P. Kalab, *Competitive software agents providing communication services*, Technical Report SCE-93-20, Carleton University, 1993.
- [DL97] M. d'Inverno and M. Luck, *Development and Application of a Formal Agent Framework*, In Proceedings of the First IEEE International Conference on Formal Engineering Methods, Hiroshima, Japan, 1997.
- [F94] P. G. S. Florissi, *QuAL: Quality Assurance Language*, Technical Report CUCS-007-94, 1994.
- [G89] R. P. Grimaldi, *Disjunctive and Conjunctive Normal Forms*, Discrete and Combinatorial Mathematics - An Applied Introduction, 2nd Edition, Addison Wesley, USA, pp 595, 1989.
- [G95a] T. Gray, *Report of the CATA Evolution Project*, A MITEL Restricted Document, 1995.
- [G95b] T. Gray, *Diagnosis Within CATA*, A MITEL Restricted Document, January 1997.
- [GK94] M. R. Genesereth and S. P. Ketchpel, *Software Agents*, Communications of the ACM, Vol.37, No.7, July 1994.
- [H95] B. Hayes-Roth, *An architecture for adaptive intelligent systems*, Artificial Intelligence, 1995.
- [JW95] N. R. Jennings and M. J. Wooldridge, *Intelligent Agents: Theory and Practice The Knowledge Engineering Review*, 1995. <http://www.elec.qmw.ac.uk/dai/pubs/KER95/>.

- [JW98] N. R. Jennings and M. J. Wooldridge, *Agent Technology Foundations, Applications, and Markets*, UNICOM, U.S.A., March 1998.
- [LGD97] M. Luck, N. Griffiths, M. Inverno, *From Agent Theory to Agent Construction: A Case Study*. In *Intelligent Agents Volume III - Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, J. P. Muller, M. Wooldridge & N. R. Jennings (Eds.), *Lecture Notes in Artificial Intelligence*, 1193, 46-93, Springer-Verlag, 1997
- [M97] Z. Meglicki, *Introduction to Linda*, February 1997. <http://www.qpsf.edu.au/workshop/linda/linda.html>
- [M94] S. P. Marsh, *Optimism and Pessimism in Trust*, Technical Report CSM-117, August 1994.
- [MF94] M. Fisher, *Representing and executing agent-based systems*. *Intelligent Agents: Theories, Architectures, and Languages*. *Lecture Notes in AI - Volume 890*, Amsterdam, Netherlands, 1994.
- [R88] D. W. Rolston, *Scripts*, *Artificial Intelligence and Expert Systems Development*, McGraw Hill, USA, pp 54-56, 1988.
- [F98] E. J. Friedman-Hill, *Jess, The Java Expert System Shell*, Version 4.0, <http://herzberg.ca.sandia.gov/jess>, 1998.
- [R98] Gary Riley, *CLIPS, A Tool for Building Expert Systems*, <http://www.ghg.net/clips/CLIPS.html>, 1998.
- [RG95] A. S. Rao and M. P. Georgeff, *BDI Agents: From Theory to Practice*, Technical Note 56, The Australian Artificial Intelligence Institute, April 1995.
- [S80] R. Smith, *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver*, *Interaction Languages, Structures and Protocols*, May 1980.
- [S98] A. Sloman, *What's an AI Toolkit For*. In B. Logan and J. Baxter (Eds.) *Proceedings AAAI-98. Workshop on Software Tools for Developing Agents*, Madison, July 1998.
- [WGD95a] M. Weiss, T. Gray and A. Diaz, *An Agent-based Multimedia Service Environment*, *International Conference on Tools with AI*, Herndon VA, Summer 1995.
- [WGD95b] M. Weiss, T. Gray and A. Diaz, *A Middleware for Developing Distributed Multimedia Applications*. In *Proceedings of the ISCA International Conference on Parallel and Disrupted Computing*, Orlando, Summer 1995.
- [WGD95c] M. Weiss, T. Gray and A. Diaz, *A Service Environment for Distributed Multimedia Applications*, *Workshop of Distributed Information Networks, IJCAI 95*, Montreal.

Appendix

A -- Script ‘analysis.clp’

B-- Function ‘transform’

C-- Function ‘genTable’

Appendix A – the source code of analysis.clp

```
~~~~~
; analysis.clp
;-----
; The script used by the analyzer to analyze the setup connection tree model of an
; application. It will first reduce the setup connection tree, then generate an
; expression and transform it. Finally, it will generate a table from the
; transformed expression. The table will contains sets of resources for future
; resource selection.
;-----
; By Daniel Choi
;~~~~~

(defglobal
  ?*list* = "(" ; stores the expression of reduced tree
  ?*changed* = false ; var to ensure all facts received are asserted
                    ; before starting the reduction process
)

(deffacts tree
  (agent-list) ; list to store all agents in the tree
  (fixed-rsc-list) ; list to store non-optional resources that will
                  ; be chosen anyway by the application
  (initial-selected-resources) ; list to store initial set of resource
                              ; selected
)

;=====
; The followings are the rules that establishes the setup connection tree model
; from the facts received from all the underlying agents.
;=====

;; Put the selected resources, i.e. the operation connection tree (at the 1st
;; time) into the expert system

(defrule set-initial-resources
  (declare (salience 100)) ; assigns high priority
  ?s <- (selected-resources ?x) ; resources selected
  ?i <- (initial-selected-resources $?r)
=>
  (retract ?s)
  (retract ?i)
  (assert (initial-selected-resources $?r ?x))
  ; puts the selected rsc into list
)

;-----

;; Rule to help seting up the agent-list

(defrule assert-agent
  (declare (salience 100)) ; assigns high priority
```

```

        (node Agent ?a)                                ; an agent exists
=>
        (assert (Agent ?a))                            ; a temp fact
    )
;-----

; // Records all agents into the agent-list

(defrule set-agent-list
    (declare (salience 100))                          ; assigns high priority
    ?l <- (agent-list $?x)                             ; agent-list exists
    ?agent <- (Agent ?a)                               ; temp fact exists
=>
    (retract ?l)
    (retract ?agent)
    (assert (agent-list ?x ?a))                        ; put the agent into agent-list
)
;-----

; // Rule to help adding caller-callee pair into the expert system

(defrule prepare-add-pair1
    (declare (salience 100))
    (node Agent ?x)                                    ; x : agent
    (node ?t&:(or (eq ?t Rsc) (eq ?t Agent)) ?z)      ; z : rsc or agent
    (connect ?x ?y $?j1)                               ; y : anything, x connects to y
    (connect ?y ?z $?j2)                               ; y connects to z
    (appID ?id)                                         ; appID initialized
=>
    (if (eq ?*changed* false)                         ; if reduction not yet started
        then
        (assert (pair ?x ?z))                          ; caller: x, callee: z
    )
)
;-----

; // Rule to help adding caller-callee pair into the expert system

(defrule prepare-add-pair2
    (declare (salience 100))
    (node Agent ?x)                                    ; x : agent
    (node ?t&:(or (eq ?t Rsc) (eq ?t Agent)) ?y)      ; y : rsc or agent
    (connect ?x ?y $?j)                               ; x connects to y
    (appID ?id)                                         ; appID initialized
=>
    (if (eq ?*changed* false)                         ; if reduction not yet started
        then
        (assert (pair ?x ?y))                          ; caller: x, callee: y
    )
)

```

; The followings are the rules that reduce the setup connection tree model

;/---Rule to eliminate hard components (except root) from the tree

```
(defrule elimination-of-hard-components
  ?node1 <- (node ?t1 ?n1 $?js1)      ; Any component
  ?node2 <- (node ?t2 ?n2)             ; Only agent or resource
  ?node3 <- (node ?t3 ?n3 $?js3)      ; Any component
  ?c1-2 <- (connect ?n1 ?n2 ?j1)      ; Connection between node1 & node2
  ?c2-3 <- (connect ?n2 ?n3 $?j2)     ; Connection between node2 & node3
  ?list <- (agent-list $?agent-list)   ; A list of agents in tree

=>
  (retract ?node2)
  (retract ?c1-2)
  (retract ?c2-3)
  (bind ?*changed* true)
  (assert (connect ?n1 ?n3 ?j2))
  (if (or (eq ?t1 AND) (eq ?t1 OR))   ; A Soft-component (i.e. AND/OR)
      then
        (retract ?node1)
        (bind ?new (delete$ ?js1 (member$ ?j1 ?js1) (member$ ?j1 ?js1)))
        (bind ?new (insert$ ?new (+ (length$ ?new) 1) ?j2))
        (assert (node ?t1 ?n1 ?new ))
      )
  (if (eq ?t2 Agent)                  ; if node2 is an agent
      then
        (retract ?list)
        (assert (agent-list (delete$ ?agent-list (member$ ?n2 ?agent-list)
                                                    (member$ ?n2 ?agent-list))))
        ; clear node2 from the agent-list
      )
  )
)
```

;/---Rule to combine soft components of the same relational type (i.e. AND or OR)

```
(defrule combination-of-soft-components
  ?node1 <- (node ?t1 ?n1 $?js1)      ; Any component
  ?node2 <- (node ?t2 ?n2 $?js2)      ; Any component
  ?c1-2 <- (connect ?n1 ?n2 ?j1)      ; Connection between node1 & node2

=>
  (bind ?*changed* true)
  (if (and (eq ?t1 ?t2) (or (eq ?t1 AND) (eq ?t1 OR)))
      then ;if node1 and node2 are both OR or AND
        (retract ?node1)
        (retract ?node2)
        (retract ?c1-2) ; combines the nodes
        (bind ?new (delete$ ?js1 (member$ ?j1 ?js1) (member$ ?j1 ?js1)))
        (bind ?new (insert$ ?new (+ (length$ ?new) 1) ?js2))
        (assert (node ?t1 ?n1 ?new ))
        (assert (ready-move-from-to ?n2 ?n1))
      )
  )
)
```

```

;-----
;---Rule to move all connections from one node to another node

(defrule move-node-connections
  (ready-move-from-to ?n1 ?n2)
  ?c1-2 <- (connect ?n1 ?any ?j)          ; Connection bet node1 & any node
=>
  (bind ?*changed* true)
  (retract ?c1-2)                          ; delete the connection
  (assert (connect ?n2 ?any ?j))           ; connects node2 to that node
)

;-----

;---Rule to eliminate AND components that have only one link to underlying
; component

(defrule elimination-of-meaningless-AND
  ?node1 <- (node ?t1 ?n1 $?js1)           ; node1: any component
  ?node2 <- (node AND ?n2 $?js2)           ; node2: AND component
  ?node3 <- (node ?t3 ?n3 $?js3)           ; node3: any component
  ?c1-2 <- (connect ?n1 ?n2 $?j1)          ; Connection between node1 & node2
  ?c2-3 <- (connect ?n2 ?n3 $?j2)          ; Connection between node3 & node4
=>
  (bind ?*changed* true)                   ; if reduction process started
  (if (= (length$ ?js2) 1)                 ; if only one link under AND node
    then
      (retract ?node2)                     ; delete the AND node
      (retract ?c1-2)
      (retract ?c2-3)
      (assert (connect ?n1 ?n3 ?j2))
      (if (or (eq ?t1 AND) (eq ?t1 OR))    ; if node1 is AND/OR node
        then                                ; move the link from node2 to node1
          (retract ?node1)
          (if (< (length$ ?j1) 0)
            then
              (bind ?new (delete$ ?js1
                                (member$ ?j1 ?js1) (member$ ?j1 ?js1)))
            else
              (bind ?new ?js1)
          )
          (bind ?new (insert$ ?new (+ (length$ ?new) 1) ?j2))
          (assert (node ?t1 ?n1 ?new ))
        )
      )
  )
)

```

://--Rule to eliminate faked AND components (have no link to underlying component)

```
(defrule elimination-of-faked-AND
  ?node1 <- (node ?t1 ?n1 $?js1)          ; node1: any component
  ?node2 <- (node AND ?n2 $?js2&:(eq (length$ ?js2) 0)) ; node2: faked AND
  ?c1-2 <- (connect ?n1 ?n2 $?j1)         ; Connection between node1 & node2
=>
  (retract ?node2)                        ; delete node2
  (retract ?c1-2)
  (bind ?*changed* true)                  ; if reduction process started
  (if (< (length$ ?j1) 0)                  ; size of $?js1 < 0, i.e. Soft-component
    then                                   ; update links of node1
      (retract ?node1)
      (bind ?new (delete$ ?js1 (member$ ?j1 ?js1) (member$ ?j1 ?js1)))
      (assert (node ?t1 ?n1 ?new)))
  )
)
```

://---Rule to determine whether the reduction of tree has finished

```
(defrule check-reduction-finish
  (declare (salience -100))               ; low priority
  (agent-list ?a0)                        ; all agent except the application agent
                                          ; have all been eliminated (tree reduced)
                                          ; any node under the application agent
=>
  (connect ?a0 ?next link)
  (assert (ptr ?next))                    ; point to the node under the app. agent
  (assert (reduction finished))
  (assert (hit-list))                     ; list to store node already visited
  (assert (go-down))                     ; ready to search downward and put nodes
                                          ; into an expression
)
```

://---Rule to search downward in the tree and update the expression with node

:// found

```
(defrule down-search
  ?dir <- (go-down)                       ; search downward
  ?p <- (ptr ?ptr)                         ; current position
  ?h <- (hit-list $e)                     ; nodes already visited
  ?node1 <- (node ?t1 ?ptr $?js1)         ; node1: current node
  ?choice <- (index ?ptr ?i)              ; under which branch of a OR node
  ?c1-2 <- (connect ?ptr
    ?n2&:(eq (member$ ?n2 ?e) FALSE)
    ?j1&:(eq ?j1 (nth$ ?i ?js1)))
                                          ; if there is a node under node1
  ?node2 <- (node ?t2 ?n2 $?js2)
=>
  (if (= ?i 1)
    then
      (if (eq ?t1 AND)                    ; if node1 is an AND node
        then
```

```

        (bind ?*list* (str-cat ?*list* "^"))
        else
        (bind ?*list* (str-cat ?*list* "|"))
    )
)
(if (eq ?t2 Rsc)                                ; if node2 is a resource (end of branch)
    then
    (retract ?h)
    (bind ?*list* (str-cat ?*list* "(" ?n2 "))")
    (assert (hit-list $?e ?n2))
    (assert (go-up))                            ; return back to upper level
    else
    (assert (go-down))                          ; if node2 is NOT a resource
    (retract ?p)                                ; continue to search downward
    (assert (ptr ?n2))
    (bind ?*list* (str-cat ?*list* "("))
)
)

```

;;---Rule to search upward in the tree and update the expression with node found

```

(defrule up-search
    ?dir <- (go-up)                                ; search upward
    ?node0 <- (node ?t0 ?n0 $?js0)                 ; node0: any node
    ?p <- (ptr ?ptr)                               ; current position
    ?c0-1 <- (connect ?n0 ?ptr $?j0)               ; node0 is above node1
    ?node1 <- (node ?t1 ?ptr $?js1)               ; node1: current node
    ?choice1 <- (index ?ptr ?i1)                  ; under which branch of a OR node
=>
    (retract ?dir)
    (if (>= ?i1 (length$ ?js1))                   ; if all branches have been visited
        then
        (bind ?*list* (str-cat ?*list* ""))
        (retract ?p)
        (assert (ptr ?n0))                        ; change current position to upper node
        (assert (go-up))                          ; continue to search upward
        else
        (retract ?choice1)                        ; still some branches beneath not visited
        (assert (index ?ptr (+ ?i1 1)))
        (assert (go-down))                        ; search downward in branches not visited
    )
)

```

;;---Rule to check if the expression has been made for the reduced tree

```

(defrule check-finish-search
    ?dir <- (go-up)
    ?p <- (ptr ?ptr)
    ?i <- (initial-selected-resources $?r)
    ?t <- (tree-info ?tree)
    ?f <- (appID ?id)

```

```

      (root ?ptr)                                ; all node visited and current position is
                                                  ; back to the root (i.e. application agent)
=>
      (retract ?i)
      (retract ?p)
      (retract ?t)
      (retract ?dir)
      (retract ?f)
      (print "Analyst) Reduced list: " ?*list*)    ; display the expression of reduced tree
      (bind ?newlist (transform-list ?*list*))      ; transform the expression
      (print "Analyst) Transformed list: " ?newlist) ; display the transformed expression
      (bind ?table (generate-table-from ?newlist ?r ?id ?tree)) ; generate table
      (print "Analyst) Rsc Table:" ?table)          ; display the table generate
      (assert (ready add-pair ?id))                 ; ready to add caller-callee pairs
      (assert (ready clear-all))
    )

;-----

;/--Rule to add caller-callee pairs

(defrule add-pair
  (declare (salience 100))
  ?p <- (pair ?x ?y)
  ?r <- (ready add-pair ?id)
=>
  (retract ?p)
  (add-pair-to-map ?id ?x ?y)                    ; add caller-callee pairs into map
)

;-----

;/--Rule to clear all facts in the expert system

(defrule clear-all
  (declare (salience -100))
  (ready clear-all)
=>
  (retract *)
)

```


Appendix B – the source code of function transform()

```

/////////////////////////////////////////////////////////////////
// Function transform
// -----
// Pre   : a list representing the expression of a reduced tree is input
// Post  : a list representing the expression of a transformed tree in the
//         disjunctive normal form (d.n.f.) is produced
/////////////////////////////////////////////////////////////////

String transform(String list) {
    String newList = list;           // the transformed list
    String left = null, right = null; // left & right sublists
    Vector leftLeaves = null;        // leaves in left sublist
    Vector rightLeaves = null;       // leaves in right sublist
    String l1, l2;
    int leftStart, leftEnd,          // pos. of left sublist in list
        rightStart, rightEnd;       // pos. of right sublist in list
    int count = 0;
    int ptr;                         // point to list

    leftStart = ptr = 2;
    rightEnd = list.length() - 2;
    do {
        if (list.charAt(ptr) == ')') {
            count--;
        }
        if (list.charAt(ptr) == '(') {
            count++;
        }
        ptr++;
    } while (count > 0);
    rightStart = ptr;                // right sublist contains branch except 1st
    leftEnd = rightStart - 1;        // left sublist is the 1st branch
    left = list.substring(leftStart, leftEnd + 1);
    right = list.substring(rightStart, rightEnd + 1);

    // =====
    // ==          if the top node is AND node          ==
    // =====
    if (list.charAt(1) == '^') {
        if (left.charAt(1) == '^') { // -- eliminate repeated sign --
            left = left.substring(2, left.length() - 1);
        }
        if (right.charAt(1) == '^') { // -- eliminate repeated sign --
            right = right.substring(2, right.length() - 1);
        }

        // check if there are more than 2 branches/nodes in the sublist
        // if so, add brackets to highlight it
        right = checkNumBranch(right, '^'); // right sublist

        newList = new String("(" + left + right + ")"); // update newList
    }
}

```

```

// -----
// if right side is a tree (top node is AND/OR) not a node
// -----
if (isRelation(right.charAt(1))) {
    right = transform(right); // parse right subtree
    // get a list of nodes from right subtree
    rightLeaves = getLeavesOf(right);
} else {

    // -- else right side is just a node --

    rightLeaves = new Vector();
    rightLeaves.addElement(right); // put the node into list
}

// -----
// if left sublist is an OR tree
// -----
if (left.charAt(1) == '|') {
    left = transform(left); // parse left subtree
    // get a list of nodes from left subtree
    leftLeaves = getLeavesOf(left);
} else {

    // -- else if left side is just a node --

    leftLeaves = new Vector();
    leftLeaves.addElement(left); // put the node into list
}

// =====
// Transformation of (^((a)(b)...)((x)(y)...)) into
// ( (^ (a)(x))(^ (a)(y))...(^ (b)(x))(^ (b)(y))...)
// =====
newList = new String("(");
for (int i=0; i<rightLeaves.size(); i++) {
    for (int j=0; j<leftLeaves.size(); j++) {
        l1 = (String)leftLeaves.elementAt(j);
        l2 = (String)rightLeaves.elementAt(i);
        newList = newList.concat("(" +
            l1.substring(l1.charAt(1)=='^'?2:0,l1.charAt(1)=='^'?
                l1.length()-1:l1.length()) +
            l2.substring(l2.charAt(1)=='^'?2:0,l2.charAt(1)=='^'?
                l2.length()-1:l2.length()) + " ");
    }
}
newList = newList.concat(")");

// -----
// check if there is still some nodes not in between that
// has not yet reduced (it is not necessary a binary tree)
// -----

```

```

        // if there is still some OR subtree inside
        if ((newList.substring(2).indexOf('|') != -1)){
            while (newList.substring(2).indexOf('|') != -1) {
                newList = shiftList(newList);    // shift the pos. of nodes
                newList = transform(newList);    // parse the tree again
            }
        } else {

            // -- else all underlying node are leaves --

            return (newList);
        }

    } else {
        // =====
        // == if the top node is OR node ==
        // =====

        right = checkNumBranch(right, '|');

        // -----
        // if right sublist is an AND tree
        // -----
        if (right.charAt(1) == '^') {
            right = transform(right);    // parse right subtree
        }

        // -----
        // if left sublist is an AND tree
        // -----
        if (left.charAt(1) == '^') {
            left = transform(left);    // parse left subtree
        }

        // =====
        // Merge all OR nodes together
        // =====
        newList = new String("(" +
            left.substring(left.charAt(1)=='?'?2:0,
                left.charAt(1)=='?'?left.length()-1:left.length())
            + right.substring(right.charAt(1)=='?'?2:0,
                right.charAt(1)=='?'?right.length()-1:right.length())
            + ")");
    }
    return (newList);
}

```

Appendix C – the source code of function genTable()

```
/////////////////////////////////////////////////////////////////
// Function genTable
// -----
// Pre   : the AppID, the transformed list, a set of resources initially selected
//         and the setup connection tree are input
// Post  : a table is generated from the 2nd & the 3rd inputs and a new STBRecord
//         is created from the 1st & 4th inputs as well as the new table
/////////////////////////////////////////////////////////////////

String genTable(String appID, String list, Vector initRsc, String treeInfo) {
    TransformListFunction f = new TransformListFunction();    // another class
    Vector branches = f.getLeavesOf(list);                    // get all the rsc sets from list
    Vector groups = new Vector();                             // the node in a conjunction (rsc set)
    Vector table = new Vector();                              // the new table to be created
    Vector v1, v2=null;                                       // temp lists
    Vector diffInfo;                                           // stores info of difference bet 2 groups
    int tPtr=0;                                                // table pointer set to the 1st row
    int ptr=0;                                                 // temp pointer
    int i,j;                                                  // counters
    int d;                                                     // number of differences
    int diff;                                                  // reference number of differences
    int firstEqual;                                           // pos of 1st equal element from 2 groups
    boolean equal = false;                                     // boolean var to indicate eq or not

    for (i=0; i<branches.size(); i++) {
        // add nodes into rsc sets
        groups.addElement(f.getLeavesOf((String)branches.elementAt(i)));
    }

    i = 0;                                                     // first rsc set

    //=====
    // Loop to find the init set from list and put it into the 1st row
    // of table
    //=====
    while (i < groups.size()) {
        v1 = (Vector)groups.elementAt(i);

        // if the set of resources == the initial set of rsc selected
        // -----
        // note: function difference returns a vector contains the
        //       number of differences and the position of 1st equal
        //       element in the 1st list
        // -----
        if (((Integer)difference(v1,initRsc).firstElement()).intValue() == 0) {
            table.addElement(v1);                               // insert that set into table
            groups.removeElementAt(i);                          // remove the set from the list
            break;
        }
        i++;
    }
}
```

```

}

//=====
// Loop to sort the sets in order and put them into table
//=====
while (groups.size() > 0) {
    v1 = (Vector)table.elementAt(tPtr);
    j = ptr = firstEqual = 0;

    // number of differences between 1st set in list & initial set
    diff = ((Integer)difference(v1,
        (Vector)groups.firstElement()).firstElement()).intValue();

    while (j < groups.size()) {
        v2 = (Vector)groups.elementAt(j);
        diffInfo = difference(v1,v2); // info of difference bet 2 sets
        d=((Integer)diffInfo.firstElement()).intValue(); // num of diff

        // if another set has lesser differences than the selected one
        // from the current set pointed in the table, select that set
        if ((d < diff) || ((d == diff) &&
            (((Integer)diffInfo.lastElement()).intValue() > firstEqual))){
            firstEqual = d==diff?j:firstEqual;
            diff = d; // update the reference num diff
            ptr = j;
        }
        j++;
    }

    table.addElement(groups.elementAt(ptr)); // update table
    groups.removeElementAt(ptr); // remove that set from list
    tPtr++; // point to next row
}

// create new STBRecord that stores diagnostic information
as.table.put(new STBRecord(appID, treeInfo, table, 0));

return table.toString(); // return the new table generated
}

```