Proud: An integrated reverse engineering system for software maintenance

Huang, Hai, Ph.D.

University of Hawaii, 1994

PROUD - AN INTEGRATED REVERSE ENGINEERING SYSTEM FOR SOFTWARE MAINTENANCE

A DISSERTATION SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAII IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMMUNICATION & INFORMATION SCIENCES

AUGUST 1994

By

Hai Huang

To

My Wife and My Son

Acknowledgments

I would like to thank my advisor Isao Miyamoto for his advice and support. He indicated the general direction of this research and my many discussions with him led to many of the ideas in this dissertation. I wish to thank the other members of my dissertation committee. As the supervisor, Kazuo Sugihara provided support for the research. David Chin gave me a lot help, especially in the design of the query language for the system and in the writing of this dissertation. I also wish to thank Bill Remus and Alexander Quilici for their guidance and interest in my work.

I would like to express my special thanks to Philip Johnson for many discussions we had about research process and his comments on the first draft of this dissertation. I also wish to thank Art Lew. His comments led to many modifications of the dissertation and made it more readable.

The Software Engineering Research Lab in the Department of Information and Computer Sciences, University of Hawaii has been a very exciting place to do research in the last few years. I thank both current and former members of the Laboratory: Koji Takeda, Tatsuo Ishii, Taro Adachi, Mitsuyuki Inaba, Kazunari Yamamoto, Keiichi Nemoto, Tatsuhiro Kobayashi, Takeshi Nishimura, Xiaobo Wang, Ye Huang, Xiang-dong Ke and Douglas Wake. I would like to express my special thanks to Narasimha Polkampally and Yuk-Ping Leung for their contribution in implementing some components of Proud.

I would like to thank the sponsors of this research, which include the State of Hawaii, Fujitsu Limited and PFU Limited.

I would like to thank my parents Fu-Sheng Huang and Lan-Jun Ju for everything. My son Dennis provided wonderful (and necessary) distractions. Most of all, I would like to thank my wife, Wei Yang, for her support during these years.

Abstract

Programmers who maintain software systems face one big problem – it is difficult to get accurate and relevant information about the target system. Reverse engineering can be a solution to the problem since it obtains the information about the target system from the most reliable source – the source code of the target system.

This dissertation presents an integrated, intelligent reverse engineering system – Proud (Program Understanding System). Proud is a component for reverse engineering and program analysis in the SMA (Software Maintenance Assistant) project. Its objective is to provide accurate and relevant information of the target system for other components in SMA by extraction from and analysis of the source codes of the target system. Proud has the following unique features:

- It uses a graphical knowledge representation language that incorporates many advanced artificial intelligence features for representing different aspects and properties of software.
- It provides a graphical, non-procedural query language to access and manipulate information extracted and abstracted from the source codes of the target system.
- It adopts a flexible, adaptable approach and hence it is easily customized to fit the requirements of a particular software maintenance project.
- It uses a rule-based approach in the design and implementation of most of its extraction tools so that tools become language independent and can easily provide extra information if demanded.

The system has been tested with some source code files from Fujitsu Limited. It is also used as the front-end processor for other projects. Proud shows its capabilities in these tests and has been proven as a useful tool for software maintenance.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Most programmers face a big problem when they do maintenance on software systems – it is difficult and costly – in terms of resources and time, to obtain precise and relevant information about the systems they are maintaining. Even if the information is obtained, it may not be represented in formats with which the programmers are familiar. This problem is interesting, since knowledge and techniques from different fields of computer sciences are needed in order to solve the problem. This problem is important, because many resources are involved in software maintenance. This problem is also difficult, since many aspects of the problem are still unclear.

1.1 Software Maintenance

Software maintenance is the last stage in the entire software life cycle. It refers to all modification activities applied on a software system after it is released to users, from the first version until it is discarded. Martin and McClure [33] give the following reasons for software maintenance:

- To remove design defects and correct errors.
- To improve the design.
- To convert the programs so that it can be used with different hardware, software, system features, telecommunication facilities, and so on.
- To interface the programs to other programs.
- To make changes in files or databases.
- To make enhancements to the application.

Swanson [44] classifies maintenance activities into three categories – corrective, adaptive, and perfective. Corrective maintenance is performed to identify and correct software failures, performance failures, and implementation failures. Adaptive maintenance is performed to adapt software to changes in the data requirements or the processing environments. Perfective maintenance is performed to enhance performance, improve cost-effectiveness, improve processing efficiency, or improve maintainability.

1.2 Problems in Software Maintenance

With varying developments in computer technology, information systems are widely used in many organizations and businesses, who consider information systems as strategic weapons towards success. Huge amounts of resources, both capital and man-power, are spent in software development and maintenance. But some research surveys (Hale and Haworth, Moad) [19, 35] point out that more than 50 percent of the resources are absorbed in maintaining existing software. This limits the resources that may be available for development of new software systems that can support organizations in competition. Schneidewind [42] gives some reasons for the high cost of maintenance activities:

- There are difficulties in tracing the product or the process that created the software being used.
- Changes are not adequately documented.
- Lack of change stability.
- Ripple effect of making changes.
- Myopic view that maintenance is strictly a post-delivery activity.

Unfortunately, even though the importance of software maintenance has been recognized for a while, not many computer supported systems have been developed for software maintenance. Research in this field is relatively less than other fields of software engineering. To make the matter worse, few experienced managers and programmers are assigned to maintenance teams, due to relatively low priority of maintenance projects. It means more time and resources are needed to maintain a system properly.

For most software systems, the only reliable source of information for maintenance is the source code. This is because that the programmers who made changes to the source code might not document the changes. Original requirements and specification may not exist anymore, or may be out of date and do not correspond to the current system. If changes were not documented, no one knows how the changes were applied. The only way to know the properties and behavior of the system is to study the source code. Without the support of an automated or semi-automated system, this is labor-intensive and time-consuming work.

1.3 Reverse Engineering

The term "reverse engineering" was first used in the analysis of hardware – to decipher designs from finished products. In a landmark paper on the topic, Rekoff [39] defined reverse engineering as:

> "...the act of creating a set of specifications for a piece of hardware by someone other than the original designers, primarily based upon analyzing and dimensioning a specimen or collection of specimens."

There are several definitions for the term "reverse engineering" in the area of software engineering. One of the definitions of reverse engineering as defined by Don Yu [51] is:

> "Software activities pertaining to computer-aided extraction of specifications, design, and software components from existing software systems. Reverse-engineering implies derivation of abstract specifications from existing, 'good' software systems and usually includes transverse-engineering steps."

Chikofsky and Cross [11] give another definition of "reverse engineering" in software development and maintenance domain as:

> "...the process of analyzing a subject system to

- identify the system's components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction."

They argue that reverse engineering can be performed from any level of abstraction and at any stage of the life cycle of the software system, and it is a process of *examination*, not a process of change or replication.

They indicate that there are many subareas of reverse engineering, two of which are widely referred – redocumentation and design recovery. *Redocumentation* is the creation or revision of a semantically equivalent representation within the same relative abstraction level. *Design recovery* adds extra knowledge to the observations of the subject system in identifying meaningful higher level abstractions. Biggerstaff [4] states that:

> "Design recovery recreates design abstraction from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains ... Design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. Thus, it deals with a far wider range of information than found in conventional software-engineering representations or code."

1.4 Challenge of Software Maintenance

The program understanding process plays critical role in software maintenance. It directly affects costs and schedules of maintenance projects, and the qualities of systems to be maintained. The program understanding process also helps to estimate the status of systems so levels of maintenance on the target system can be determined.

Robson [40] points out that it has been estimated that more than 50% of maintenance time is devoted to understanding the program to be maintained. Among the time spent on program understanding, most of it is spent on studying of code, other than documentation [40]. One major reason pointed out by Sasso [41] is the low credibility of existing documentation. Automated program understanding support will speed up maintenance activities, thus reduce the cost of software maintenances which normally consume more than half of the budget in the life cycle of software.

Sasso [41] found out four major themes for design recovery in his empirical study. These themes are:

- Following the mainline flow of control through the program.
- Identifying transaction type-specific control structures.
- Using information on input and output.
- Relating code to abstract, higher-level representations.

These themes are sound for most program understanding activities. Support for activities in these themes will help a programmer more clearly and quickly understand the program.

However, the program understanding process for software maintenance is different from the program understanding process for other purposes, due to the nature of software maintenance. The levels of understanding of programs vary from project to project. Swanson [44] divides maintenance activities into three groups – corrective maintenance, adaptive maintenance, and perfective maintenance. The characteristics of each type of activities are different one from the other and therefore, support for these activities should be different.

Corrective maintenance projects focus on fixing failures discovered after the system is delivered. They are the extension of program implementation and testing. They share most of the requirements for program debugging and have the following characteristics:

- Normally problems of the system are reported by the users and the problems reported are the symptoms of errors reflected in the user interfaces.
- In most cases, the maintenance activities are unscheduled and urgent. Time is a crucial factor for this type of project.

So the first major task for a corrective maintenance project is to locate the bugs in the code that cause the symptoms. Program understanding support in this circumstance emphasizes exploring the relationships between the internal behaviors of the program and the external behaviors of the user interfaces.

Adaptive maintenance projects emphasize adapting the system to the changed environment. Two types of changes are expected in most cases: changes in data environment, such as changes in file structure or database structure, and changes in processing environment, such as changes in platform of processing or peripheral environment. It has the following characteristics:

- In most cases, it is a scheduled project. Time is not as crucial as corrective maintenance.
- In most cases, it requires no change in user interface behaviors and functionalities provided by the system.
- Levels of modification on code can vary from simple processes such as recompiling source code in the new platform to rewriting most of the system.

The first task for an adaptive maintenance project is to determine the level of modification and locating interfaces affected and relationships between components behind these interfaces and the environment. Program understanding support in this circumstance focuses on exploring relationships between components of the target system and their environments for changing the processing environment, and interfaces between processing components and data structures for changing the data environment.

Perfective maintenance projects are to enhance the existing system so that it can provide new services or perform new functions. A survey estimated that 65% of the maintenance phase is taken up with perfective maintenance. It has the following characteristics:

- The time constraint varies depending on the urgency of the new services.
- New services are normally related to existing functions.
- The process of developing a new function is similar to developing a new system.

Program understanding support for this type of project includes exploring the roles of new functions and relationships between the new functions and other components in the system.

Program understanding for software maintenance tends to be different from program understanding for other purposes in the following ways:

- Obtain information mainly from source code, which is the most reliable source of information about the program.
- Support for exploring different aspects of the target programs in different levels of abstraction, from basic control flow to high level program specification.
- Support for exploring relationships among different components and aspects of the target system in different levels.
- Support for dealing with programs written in different programming languages, and representing concepts in a common format to reduce efforts required for using the system.
- Support for multiple ways of representing information to satisfy requirements of the users' view and filtering non-relevant information to reduce the information overload on users.

1.5 Current Approaches and Their Weaknesses

Recently, many methods and approaches for reverse engineering have been proposed (See chapter 2). In a survey, Miyamoto [36] finds that many commercial CASE tools in the market claim to support reverse engineering. Many of them only support reverse engineering on data. Some of them support reverse engineering on processes are limited to a particular languages or require special comments in the source code. The techniques and approaches adopted in previous research works on reverse engineering can be grouped into two categories: AI-based and non-AI techniques.

Most reverse engineering methods based on non-AI techniques provide mostly cross reference information. The C Information Abstractions system [9, 10], for example, provides cross reference information among the macro definitions, user defined data types, functions, files, and global variables. This information is important for software maintenance, but is not sufficient. Maintainers tend to request much more information than cross reference information.

On the other hand, AI-based approaches have the capability to provide higher level abstracted information associated with functionalities of the target program. The plan-based approach, which is used in PAT [20, 21], is one of the popular methods for abstraction of program functionalities from source code. Due to the limitation of the state-of-the-art in artificial intelligence, the capability of these approaches is limited by knowledge representation formats for plans, search space exploration, and other problems. They are still not powerful enough to deal with real software systems.

The current approaches for reverse engineering and program understanding share some common weaknesses towards software maintenance:

- They deal with only one or a few aspects of the program at only a few levels of details. They are incapable of providing multiple views of the program.
- Most tools are stand-alone and due to the lack of the common representation format, it is very hard to integrate several tools into one environment to support maintenance activities.
- They lack flexibility to represent information. Most tools represent information only in a few formats and it is not easy to change the way in which the information is represented. This imposes some restrictions on maintainers for their understanding of information obtained by the tools.
- They lack flexibility to obtain different information from source code. Many tools obtain information about one or a few aspects of the target system. It is not easy to obtain other information. To obtain other information requires either major modification of the tools or development of new tools. This restricts the usage of reverse engineering tools in software maintenance.

1.6 Proud – A Solution to the Problem

A new system is needed to support programmers maintaining industrial strength software systems, which may contain millions of lines of code. A system called "Proud" (Program Understanding System) provides a partial solution to the problem. It is an integrated, intelligent reverse engineering system. It provides information about the target system in a graphical format, via a non-procedural query interface. The information regarding the target system is represented in RMA *(Requirement Modification and Analysis)* models [43]. The system also allow programmers to define the types of information they want to obtain and to choose the proper representation formats to some degree. With a non-procedural, graphical query interface, the system requires minimal training to use it. The system also provides a framework for tool integration and a high level of openness for adding new tools by using a plan-based facility management approach. The facility management component does dependency checking to select the proper tool for generation of the required information.

Compared to other dependency checking facilities, such as *make* in Unix, the dependency checking in Proud has a few advantages. Proud integrates the retrieval process with the information generation process and the unit for checking is a graph, not a file. This gives Proud an advantage over other dependency checking facilities, since users do not have to know which file contains the information they want. The only thing they need to provide to Proud is a pattern that is used as the retrieval criteria. Proud only generates the information that is required for retrieval if it is not already in the database of the system.

Another advantage of dependency checking in Proud is it uses more information to do the checking. Unlike *make*, which uses only the file name and time stamp for dependency checking, Proud uses not only file name and time stamp, but also other information such as the programming language in which the source code is written, the table type, the model type, etc. This gives Proud more control over which tool to use in order to generate the right information. For instance, two source code files written in different programming languages are processed by two different parsers, even though they may have same name.

The following example shows how Proud can help a programmer to save time and effort when working on a maintenance project. The scenario is: maintenance of an airline reservation program is assigned to a programmer who is not familiar with the program. In order to do the maintenance, the programmer needs to become familiar with the program. One effective tool for studying programs is the data flow diagrams for the program.

To create data flow diagrams from source code manually, the programmer must spend a lot of time to study the source code, understand the source code one section at a time, and create a piece of the diagram at a time. It requires a lot of effort and the diagrams created are error prone.

Even if the programmer uses reverse engineering tools to assist him, he still has to do a lot of work. For example, if he uses the reverse engineering tools provided in Proud to create a data flow diagram, he must determine which source code file he should study first, generate the syntax tree from the source code file by using the proper parser, then generate five cross reference tables one by one, using *Table Generator*, and finally generate the data flow diagrams (called CPM in Proud) by using *Model Builder*. After the graphs are generated, the programmer has to go through the graphs one by one to find the right graph that he needs.

Proud frees the programmer from all this trouble. The only thing he has to do is to create a query in a graphical format, such as the query shown in Figure 1.1. This query requests Proud to return a data-flow diagram that contains a task named "RESVPG" since the programmer knows the name of the program is "RESVPG" and he wants to see the overall data flow of the program.

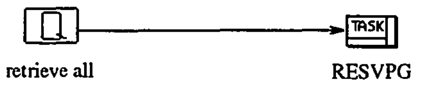


Figure 1.1: Query for Retrieval of the Top Level CPM Graph

Upon receiving this query, Proud determines what information is required and what is missing, generates the required information if necessary, and returns the graph that matches the requirements of the query to the programmer. The answer for this query is shown in Figure 1.2.

After the programmer studies the data flow diagram he just retrieved, he decides to study more detailed data flow diagrams. Since he does not know the names of the

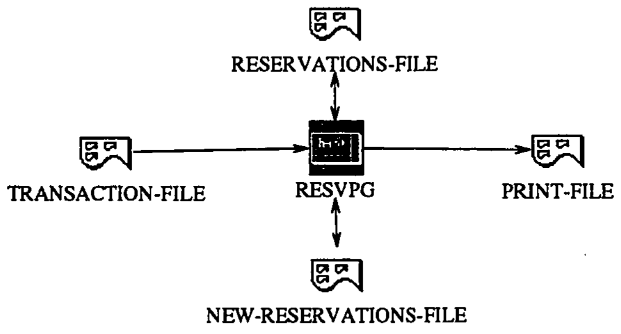


Figure 1.2: Result of the Retrieval of the Top Level CPM Graph Query

second level modules, he tries to retrieve data flow diagrams related to tasks whose names start with "RESVPG". He issues another query shown in Figure 1.3 to Proud. Proud does not invoke the generation process since Proud learns the information requested by the programmer exists in the database of the system and retrieves the graphs matching the programmer's retrieval pattern, shown in Figure 1.2 and 1.4.

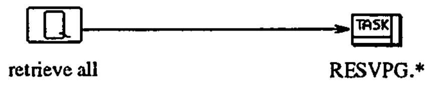


Figure 1.3: Query for Retrieval of Another CPM Graph

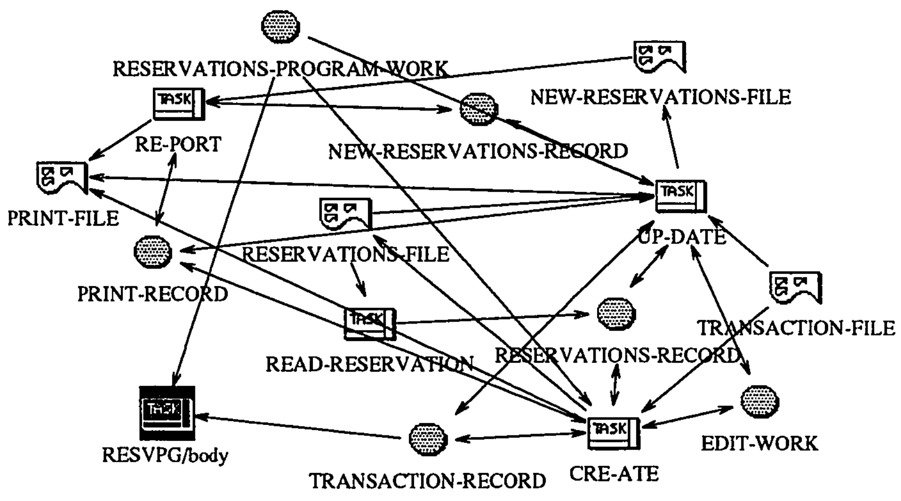


Figure 1.4: Result of the Retrieval of Another CPM Graph Query

By examining the data flow graphs, the programmer gains some knowledge about the structure of the program. But in order to maintain the program, he needs to know more about the program in detail. He decides to examine a portion of the program that invokes the "CRE-ATE" module. He issues a query to retrieve the control flow diagram (called IOPM2 in Proud). The query is shown in Figure 1.5. Proud returns the answer to that query after generation of the required information, by using different generation tools than the one used for generation of data flow diagram. The result is shown in Figure 1.6.

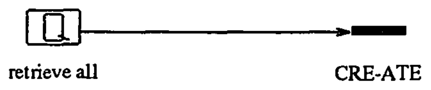


Figure 1.5: Query for Retrieval of IOPM2 Graphs

The organization of this thesis is the following:

- Chapter 2 discusses some research on the topics related to this research.
- Chapter 3 presents the overall structure of Proud and some issues involved in the research of Proud.
- Chapter 4 to Chapter 6 present the design of components of Proud.
- Chapter 7 illustrates the internal process of Proud by showing the processing of the example queries used in this chapter.
- Chapter 8 concludes the dissertation.
- Appendix A provides the brief description of formalisms of RMA models generated by Proud.
- Appendix B lists the data generated by reverse engineering tools for a sample program.
- Appendix C provides the grammars of generation rules.
- Appendix D provides the information required for handling programs written in ANSI COBOL.
- Appendix E provides the information related to linkages among the RMA models.

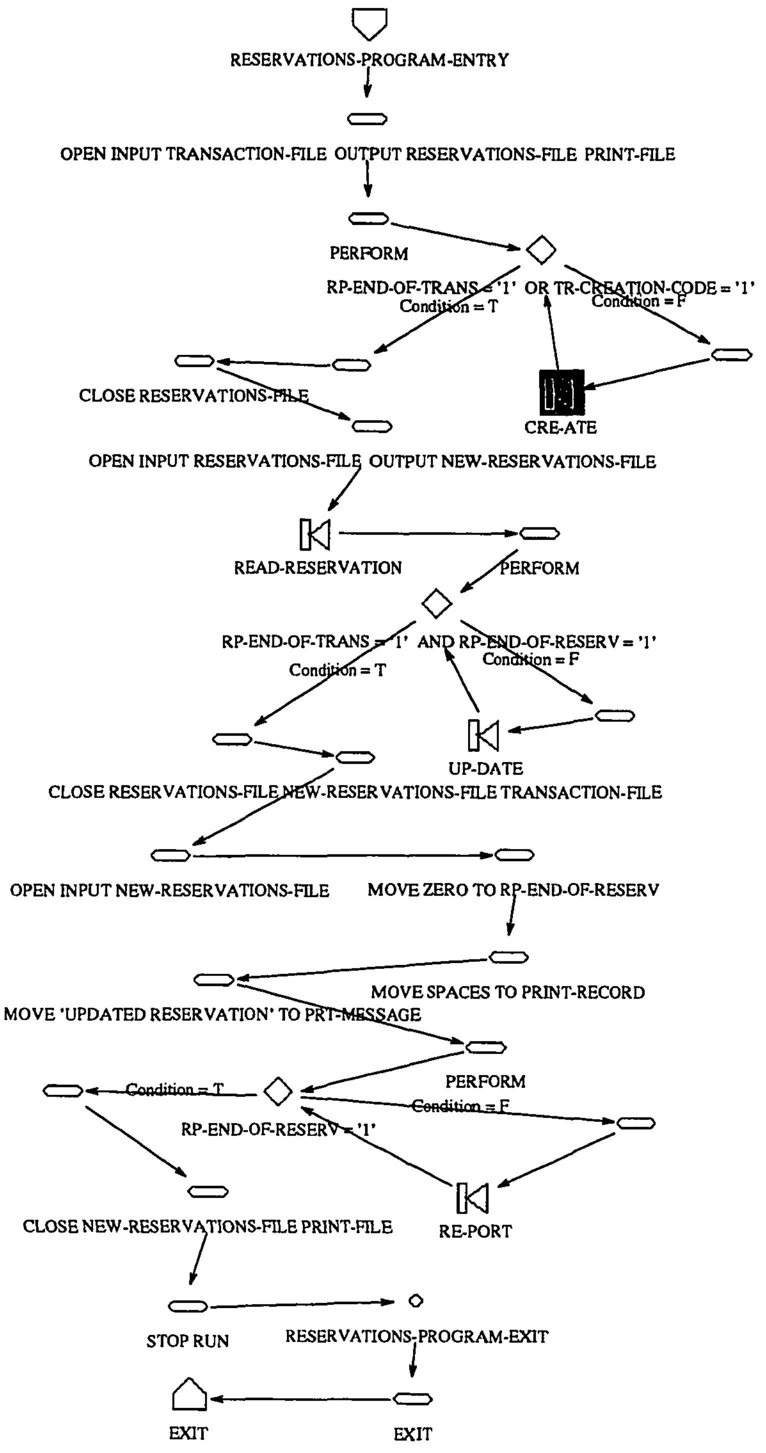

RESERVATIONS-PROGRAM-ENTRY
OPEN INPUT TRANSACTION-FILE OUTPUT RESERVATIONS-FILE PRINT-FILE
PERFORM
RP-END-OF-TRANS = '1' OR TR-CREATION-CODE = '1'
Condition = T
Condition = F
CLOSE RESERVATIONS-FILE
CRE-ATE
OPEN INPUT RESERVATIONS-FILE OUTPUT NEW-RESERVATIONS-FILE
READ-RESERVATION
PERFORM
RP-END-OF-TRANS = '1' AND RP-END-OF-RESERV = '1'
Condition = T
Condition = F
UP-DATE
CLOSE RESERVATIONS-FILE NEW-RESERVATIONS-FILE TRANSACTION-FILE
OPEN INPUT NEW-RESERVATIONS-FILE
MOVE ZERO TO RP-END-OF-RESERV
MOVE SPACES TO PRINT-RECORD
MOVE 'UPDATED RESERVATION' TO PRT-MESSAGE
PERFORM
Condition = T
Condition = F
RP-END-OF-RESERV = '1'
RE-PORT
CLOSE NEW-RESERVATIONS-FILE PRINT-FILE
STOP RUN
RESERVATIONS-PROGRAM-EXIT
EXIT
EXIT

Figure 1.6: Result of the Retrieval of IOPM2 Graph Query

Chapter 2

Research in Software Maintenance and Reverse Engineering

Research on Program Understanding and Reverse Engineering can be roughly divided into three groups – engineering methodologies, reverse engineering, and program understanding based on artificial intelligence techniques. Research interests and focuses are slightly different from one group to another. Engineering methodologies focus on methodologies that guide the software maintenance process. Reverse engineering focuses on extracting information from source code and organizing the information in a proper way. Program understanding tries to find the properties and behaviors of the target system from source code. The following is a review of previous research in each group.

2.1 Engineering Methodologies

Using proper methodologies for program maintenance is critical for reducing the time of the maintenance task, as well as ensuring the quality of the task. But there is no formal methodology widely accepted for software maintenance. Only a few methods have been proposed in recent conferences for software maintenance.

One method for documenting and maintaining an undocumented program was proposed by Fay and Holmes[17]. The method consists of five steps:

1. Learn the structure and organization of the program. Start on the structure of the program by going through the code and finding where each module is called by other modules. Use the techniques preferred by the maintainers to represent the structure of the program. Each time through the code, write down any questions that arise.

2. Determine what the program is currently doing. Go through the code carefully (module by module), trying to grasp everything. Use flowcharts or other techniques to represent the information obtained from the code. Answer the questions that were raised at step 1. Go through the code again and reconstruct flowcharts or other representations. Put comments into the code according to the current understanding of the program.

3. Begin documentation. Develop a module prologue for each module. Develop a data dictionary for the program or update the existing one. Check with a manager to see if the modifications being made should be placed under configuration management. Make a copy of original source codes.

4. Make the required update. Be sure what is needed. Repeat previous test procedures to ensure the same results as documented in the test reports. Determine where the updates should be implemented within the code and implement the updates. Test the updates in every possible way.

5. Finish the documentation. Make sure the code itself is well documented. Start pulling together all those notes that have been kept and form a Program Description Document. Form a Version Description Document that contains all the information pertaining to this version of the program. Create or update the User's Manual. Create or modify a test document.

This method is a good approach for maintainers to understand and maintain an undocumented program. It focuses on updating the documents about the program and comments in the source codes. This effort will pay off by reducing the effort needed for future maintenance. Some automated tools can help to speed up the process under this method. For example, tools to automatically generate structure chart, flowcharts, etc.

This method also requires the maintainers to have a lot of knowledge. The knowledge includes general knowledge about the programming language used in the program, the system environment, and special knowledge about the domain of the program. Even with an automated support environment, this method is still very time and resource consuming.

Another method applies structured program analysis for software maintenance [47]. It consists of seven steps, each one designed to extract more information from the source code than the previous step. Its main purposes are to map out the structure and functions of various program parts and to provide a mapping of various program features to the program listing for quickly locating features. The steps in their logical order are:

1. Executive procedure analysis. The executive procedure sets up the immediate environment for the program. This is crucial for maintenance knowledge, especially for the files assigned to the program. The guideline for this step are: Be general; explain commands by functions only; show set-ups for program executions; and put data files or procedures into the set-ups by their purpose or contents.

2. Program analysis by subprograms and file-elements. The result of this step is a basic program structure chart. The guidelines for building it are: Start with the main routine; repeat calls throughout the chart; all subprograms' calls should be included; ignore any decision branches; and no text descriptions should appear in this step.

3. Program analysis by subprograms and file-elements with the module's purpose and I/O. This step expands the information on the structure chart produced in step 2. A short statement of the purpose of each module is listed next to its name and file-element. Include any I/O actions the module performs.

4. Individual module analysis by subprograms and file-elements, module's purpose and I/O, and block comments. This step is to complete the information collected in the prior steps. This step handles one subprogram at a time and only includes the calls made directly by the module itself.

5. Data flow diagrams. Data flow diagrams are an excellent form for mapping the flow of data through the program.

6. Console message and print format lists with file-elements. This step is to list all the messages output at console or printer. Guidelines for building these lists are: There should be one list for each output destination. To locate the message in the program, list the file-element where the message is found.

7. Line-by-line code analysis. At this point, maintainers have a functional chart of the executive procedure, a structure chart of the program with various levels of documentation, data flow diagrams, and message format lists. Any code analysis at this point is not isolated.

This method mainly focuses on analyzing the structure of the program. The strong point of this method is that it considers not only the source codes of the program, but also the environment of the program. Like the Fay and Holmes method, this method requires the maintainers to have plenty of knowledge. Also, this method requires some additional information about the program to be analyzed, such as the environment in which the program is executed, the structure of the files or other external resource used by the program, etc.

2.2 Reverse Engineering

The objective of reverse engineering is to recover the specifications and designs from source codes. With state-of-the-art technology, not all knowledge, decisions, and experience implemented in codes can be automatically extracted from the codes. However, some of the design information, such as the logical structure of the program, the data flow of the system, and the data dependency information, can be extracted and abstracted from source codes. There have been several methods proposed. One of them is to explore the relationship between different components by using techniques like relational databases. One example of this method is the *C Information Abstractions* System [9, 10].

C Information Abstractions System is a tool for system analysis. This system consists of five components: *Make*, *C Abstractor*, *Program Database*, *Information Viewer*, and *Software Investigator*. The *C Abstractor* uses a conceptual model to convert the textual description of a set of C source modules (files) into a set of objects and relationships. *Make* checks the time-stamps of each file and invokes the *C Abstractor* on a file only when that file has been modified since the creation of the last program database. The information of each source file is recorded in an incremental database. The incremental databases are then linked together to construct a large *Program Database*, which can be processed by other tools. The objects collected in *Program Database* are: *file*, *macro*, *global variable*, *data type*, and *function*. Each type of objects has a set of attributes to represent the information about each object. Objects in the database are linked by two types of relations: *includes* and *refers to*. The information stored in *Program Database* can be accessed by *Information Viewer*, which provides relational views to programmers and allows them to browse the program text in a nonlinear way. *Software Investigator* is a collection of tools to uncover interesting aspects of program structures. The tools include graphical views, subsystem extraction, program layering, dead code elimination, and binding analysis. Those tools use the information provided by *Information Viewer*.

This is a powerful system for maintainers to analyze C programs. By using relational database techniques, information in the program database can be retrieved easily and flexibly. Many interesting analyses about the target program can be carried out by using the information in the program database. The weakness of this system is that it only provides static information about the target program. The dynamic aspects of the program, such as data flow, control flow, and user interface behaviors, are not provided by the system.

Another method in reverse engineering is to extract information on data and data structures. Colbrook and Smythe [13] proposed an approach for program maintenance that is fallen in this area. There are two major points in their approach: structured data form and program reading. Structured data form is to use a set of well-defined data structures to replace other data structures. Program reading is to replace a program section with logical comments based upon the functionality of the program block. The approach first identifies critical sections of the program, and then uses a restructuring tool to transform the critical section of the program into a structured control flow program. The proper program flow-graph is generated for the critical section, and the flow-graph is divided into primitive programs. The primitive programs are replaced by logical comments in a bottom-up manner. After the logic comments for a whole section are derived, patterns in the logic comments are examined and known structure data forms are used to replace existing data structures. Finally, the critical section is structured both in data and control flow.

It is not mentioned in this paper whether the logical comments are derived from source codes automatically, or manually. The structured data form is not widely accepted in the software community. The approach concentrates on low level program transformations. This limits the usage of their approach. The major weakness of the approach is that it does not address one important aspect of the program - data flow.

Another approach in reverse engineering is to develop new methodologies for extracting design information from source code of a target system. An example of this method is to construct data flow diagrams from programs written in Pascal proposed by Benedusi, Cimitile, and Carlini [3]. The diagrams are constructed, based on a hierarchy of calling relationships. The method has two phases. In phase 1, the structure chart of the program, which represents the calling relationships between modules, is examined and then modules in the chart are classified into two categories: terminal modules and transform modules. Each transform module is allocated to the corresponding level of the data flow diagram.

Phase 2 is to complete the data flow diagram through identification of repositories in each level and their connection to transform modules. A bottom-up strategy is used to complete the data flow diagram. It is divided into three sub-phases. Phase 2a is to replace all formal parameters with actual parameters. Phase 2b is to produce the actual list of data items that certainly contribute to the formation of repositories. Phase 2c is to construct the repositories.

This method provides a way to reconstruct high-level documents - data flow diagrams from source codes. The processes used in the method are simple, yet powerful enough to handle programs written in layered programming languages. But the method is not suitable for COBOL program, since the two major processes for producing an actual list of data items contributed to data flow are not effective in COBOL.

There is other research in the reverse engineering area. *Ccall* is a system for analyzing C programs[30]. It generates a calling tree and calling matrix of the program from source codes. Also, it generates some statistics that may be useful in determining the complexity of the program. This is a simple system and does not extract much information from source codes. Cimitile and De Carlini present a set of algorithms for program graph production [12]. They propose an algebraic representation of program modules. Three types of graphs are defined – the program flow-graph that represents the control flow of the program, the program nesting tree that represents the structure of the program, and the exemplar-path tree that represents the path tree of the program. Paul and Prakash describe a source code retrieval method using high-level patterns [38]. A pattern language is developed for describing patterns in the programs and the descriptive power of this language exceeds previously existing pattern matching description languages.

2.3 Program Understanding

Understanding of a program is regarded as construction of the descriptions of the program that indicate what the program does and how the program does it. Artificial intelligence is involved in research on automatic construction of such descriptions from source codes.

Lukey developed a theory of program understanding and debugging in the late 70s[32]. The theory was embodied in a system, *PUDSY*, which was able to understand and debug some simple Pascal programs. The method proposed by the theory has 4 key features:

- the segmentation of a program,
- the description of the flow of information,
- the recognition of debugging clues, and
- the description of the values of variables.

The input program is first segmented into small chunks. This is the unit for analysis. Then *PUDSY* uses its knowledge base to find out what each chunk does. Chunks that have not been recognized are symbolically examined. The flow of information between chunks is determined and the assertions describing the entire program are generated. A record of how the assertions are derived from the assertions of individual chunks is kept for debugging phase.

In the debugging phase, the derived assertions are compared with specifications of the program. Mismatches between them indicate possible bugs. The system traces back from assertions of the entire program to the assertions of individual chunks. A discrepancy between expected and actual assertions indicates a bug and the discrepancy is used to propose a way to fix it.

One of the problems of this system is the limited processing power. The system cannot deal with recursion and goto statements. Moreover, the system cannot use knowledge about a "real-world" problem that a program intends to solve. It is very difficult, if not impossible, to process "real" programs by using this system.

Talus was developed for automatic program debugging for programs written in Lisp. *Talus* separates knowledge representation into three different levels:

- Tasks are basic programming assignments given to students.
- Algorithms are alternative ways of solving tasks.
- Functions are sub-elements of algorithms.

Talus uses the following four steps to analyze input programs:

Program simplification: It simplifies programs by (1) transforming them from an extended Lisp dialect to a simpler core dialect, and (2) reducing all conditionals to If-Normal form.

Algorithm recognition: The key to algorithm recognition in *Talus* is its use of E-frames – data structures that contain slots representing various program abstractions.

Bug detection: By using symbolic evaluation, *Talus* determines whether an equivalence exists between input and reference functions.

Bug recognition: Techniques used for correcting bugs are based on theorem proving and other heuristic methods.

Talus's strengths are: It allows reference functions to be stored as program code, can detect localized mismatches automatically, can handle functions containing functions, and can recognize equivalent forms of the same expression.

The limitations of *Talus* are: It has a limitation on the programming language that it can handle. It can handle only programs written in a subset of Lisp called extended dialect. It assumes that the task is already known. So it verifies the correctness of the program instead of understanding an arbitrary program. It also assumes that sub-function definitions are prescribed. For *Talus* to properly recognize the input program, the input program must allocate its sub-functions exactly as *Talus* does. It is oriented to recursive programs that update data structures. It has problems with large programs and imperative programming style. It provides limited data structure definitions.

Proust was designed to do on-line analysis and understanding of Pascal programs written by novice programmers [27, 28]. The input to the system is a program to be analyzed and a non-algorithmic description of the program requirements. The description contains a list of goals that must be satisfied and a list of objects that must be manipulated by the program. *Proust* has a knowledge base containing a plan to satisfy the goals and to manipulate the objects. The system uses a problem-driven approach to understand the program. It picks a goal from the description and finds out all the plan that may satisfy the goal and maps the individual plan with the actions in source codes. If some goal could not be satisfied, it means that there are some bugs in the program.

Four types of knowledge in *Proust* are program descriptions, goals, plans, and object classes. Program descriptions are a codification of the requirements for the problem to be solved. Goals are identified in the program description, and indicate the specific attributes the program must have, although such attributes can be implemented in various ways. Plans are templates for program fragments that are stereotypical for the operation to be performed. Object classes refer to the way in which *Proust* represents data structures.

Goal decomposition is important to understand even a small program. A goal decomposition consists of:

- a description of hierarchical organization of subtasks in a problem,
- indications of relationships and interactions among the subtasks, and
- a mapping from the subtask requirements to the plans used to implement them.

Since goal decomposition may not be unique, many possible decompositions and programs have to be searched. Since one program may be associated with several goal decompositions, ambiguous interpretations should also be solved. *Proust* evolves goal decomposition and plans analysis of the program simultaneously in order to reduce search space. The plan analysis is performed by using a database of correct and buggy plans, transformation rules, and bug-misconception rules to construct and evaluate interpretations for the program under consideration. The evaluation process is prediction-driven. *Proust* operates in a continuous cycle, as follows:

Select a goal from the agenda.

Determine if that goal should be subsumed or unified with other goals already on the agenda.

Expand the goal by identifying implied goals and elaborating the properties of any objects that the goal manipulates.

Retrieve implementation methods for goal. Implementation methods may correspond to object implementations or to plans.

Attempt to match retrieved plans with the input program. Plan matching may stop because of match errors (which may indicate the presence of bugs), or because the plan contains a subgoal. If an error is indicated, *Proust* analyzes the error for the presence of bugs. If a subgoal is encountered, *Proust* adds the subgoals to the agenda and starts a new cycle. If no plan fits the program, *Proust* begins a new cycle with the goal for the current cycle removed from the agenda.

Choose one of the plans that matches the current goal if more than one plan exists. If the current goal is a subgoal of some previous goal, *Proust* reactivates the previous goal. Otherwise, by using an updated goal agenda, *Proust* continues on to the next cycle.

The strengths of *Proust* are primarily derived from its knowledge representation. They are:

- Top-down template matching – *Proust* uses an opportunistic approach to program recognition. The depth-first technique minimizes the search space required for successful program identification. *Proust* can synthesize a solution that is not part of a stored database.

- Partial recognition – It recognizes large parts of a program and uses its current plan database to identify localized sections it can not recognize. Its recognition process identifies the program as being associated with a given problem solution, even though it cannot completely identify all parts of the input program.

Proust has three basic limitations. It requires a problem description, transformation rules, and templates:

- Problem description – *Proust* relies on the fact that the problem to be solved has been specified. *Proust*'s search strategy does not easily extend to the domain of identifying problem descriptions with unknown programs.

- Transformation rules – It has no notion of a calculus for programs. *Proust* generates each variation of a construct, using a heuristic that must be included in the knowledge base.

- Templates – It limits *Proust* in two way. First, templates place constraints relative to the recognition of equivalence classes. Second, templates are tied to a particular programming language, so it is impossible to port the program recognition implementation to other languages.

PAT (Program Analysis Tool) is a system to analyze and debug programs[20, 21]. There are two concepts involved in the system: program plan and program event. Program plans are abstract representations of the algorithmic structure of programs. Events are language independent objects that are derived from source codes and recognized from existing events.

The system consists of seven process components and three knowledge bases. The three knowledge bases are used to store different knowledge for program understanding and knowledge about the program being processed. The knowledge bases are:

- *Plan Base* contains program plans that are developed by human experts and parsed by Plan Parser. These program plans are used by *Understander* to derive high level events of the program.

- *Event Base* stores both the lowest level events obtained from source codes and higher level events derived by the knowledge inference engine.

- *Justification-based Truth Maintenance System* (JTMS) records the results and justifications of the inference process performed by *Understander.* The final state of the *JTMS* reflects the system's eventual understanding of the given program.

The process components are:

- *Plan Parser* converts the plans developed by humans into an internal format and stores them in *Plan Base.*
- *Program Parser* obtains the lowest level program events from source codes.
- *Understander* uses the program plans stored in *Plan Base* as inference rules and program events stored in *Event Base* as the facts to derive new high-level events.
- *Explanation Generator* provides explanations on the events derived by *Understander*, using the information in *JTMS.*
- *Paraphraser* translates the explanation generated by *Explanation Generator* into natural language descriptions.
- *Debugger* examines the final set of recognized events and uses information contained in program plans to identify possible misimplementations.
- *Editor* provides an editing capability to modify the buggy codes.

Events in the *Event Base* are organized in a hierarchy. At the lowest level are events representing language concepts. At a higher level are events corresponding to common programming patterns and strategies. A program plan has four components: *path expression* that specifies lexical and control sequence requirements; *test condition* that specifies constraints of the plan; *text* that provides documentation information; and *miss* that stores near-miss debugging information. The understanding power of the system comes from the pattern-directed inference engine that uses the Plan Base. Plan rules are triggered by events defined in the plan's path. The rule body is an assertion that declares a new event when the trigger patterns and test conditions are satisfied.

The strength of this system is that it does not require system specification for a target system, and it can handle programs with missing, extra, or buggy parts, and can avoid the combinatorial barriers in analysis of large programs.

The drawback of the system is the difficulty to write a program plan that can cover most program patterns for an algorithm or other program concepts. It cannot handle the situation where the knowledge of a program concept is not captured in the Plan Base. Thus it needs a large Plan Base to handle real situations in software maintenance.

The idea of a plan is further extended into a program concept [16, 29]. A program concept is defined by two parts. The first part defines the meaning of the concept and its attributes. The second part defines the components the concept should contain and the relationships and constraints among the components. All concepts in the knowledge base are organized hierarchically. The descriptions of the components are programming language independent so these concepts can be used for understanding programs written in different programming languages.

Calliss, Khalil, Munro, and Ward proposed a knowledge-based maintenance tool[7]. The key idea in their proposal is a way to organize the knowledge base. The knowledge contains two different kinds of knowledge:

- Maintenance Knowledge about how maintenance programmers do their work. This knowledge will provide heuristic knowledge to dictate weighting patterns for search through the expert system.

- Program Plans. There are two categories of knowledge: General Program Plans that show activities commonly occurring in program, and Program Class Plans that are common to a particular type of program. Some heuristic rules accompany the Program Class Plans to guide the use of the plans.

Separating common program plans and specific program plans can help to recognize a new program pattern without heavily increasing the space and time needed for the search process.

Das presented a knowledge-based technique for understanding programs in terms of their plans[14]. The technique is used to enhance quality control of software. It derives plans of code and compares them with the logical structures of the code in program design language (PDL). The basic approach to generate the plans of the code is similar to the approach mentioned in Harandi and Ning's approach.

This approach needs some information on design and implementation of the software. However, such information is rarely available. Since the PDL describes the logical structure of the software, the plans derived from code are not very abstract. The information used in the technique is well organized in an object-orient fashion. All these reduce difficulties to achieve the goal of inspecting consistency between the code and the PDL. However, this approach is not suitable for most other maintenance activities.

Program Recognizer [50] recognizes occurrences of stereotyped computational fragments in computer programs. Using the plan calculus, it can analyze a program, convert the program plan to a flow graph projection, parse the flow graph projection with a grammar derived from a library of cliches, and check the constraints on the matched flow graph (constrains also derived from the library of cliches).

A plan for a program is a graph, with nodes representing operators and edges representing control flow and data-flow. The flow analyzer converts program source code into a plan, which occurs in two stages: macro-expansion, followed by control flow and data-flow analysis. The cliche library provides a taxonomy of standard computational fragments and data structures represented as plans. Flow graphs resemble plans but have some significant differences. To parse a program, the system attempts to match a flow graph representing a cliche with a subgraph occurring in the program definition. After matching based on the flow graph, the system checks constraints added to the input graph against those that apply to the cliche being matched. If the constraints agree, a successful parse has occurred and the process continues. If not, the cliche is removed from the current item list.

The Program Recognizer's strengths are:

- Abstraction capabilities – The use of control flow analysis means that arbitrary ordering decisions in the input program do not affect recognition. The data-flow analysis removes the sensitivity to the order of statement or function calls not affecting program data-flow.

- Recognition capabilities – it does not require that all program components be recognized. It identifies components that match cliches and ignores the rest.

It does not require all reference functions to be present in the input code. But it does not recognize a high level cliche unless all required lower level cliches are present.

The Program Recognizer's limitations are:

- Ease of extension – It requires that grammar rules be manually added whenever it adds a new cliche to the library.
- Coverage limitations – It does not provide coverage for side-effecting operations, full recursion, arbitrary data abstraction, or functional arguments.

Chapter 3

Proud Subsystem

The *SMA (Software Maintenance Assistant)* project was initiated in Fall 1990 at the Software Engineering Research Laboratory, Department of Information and Computer Sciences, University of Hawaii, together with the Software Engineering Laboratory, Department of Computer Science and Engineering, University of California at San Diego. It aims at developing an integrated environment for supporting various maintenance activities. SMA consists of several subsystems such as Program Understanding (Proud), Requirements Modification and Analysis (RMA), Maintenance Consultant (MC), Object Base (OB), etc. Figure 3.1 shows the overview of the SMA system.

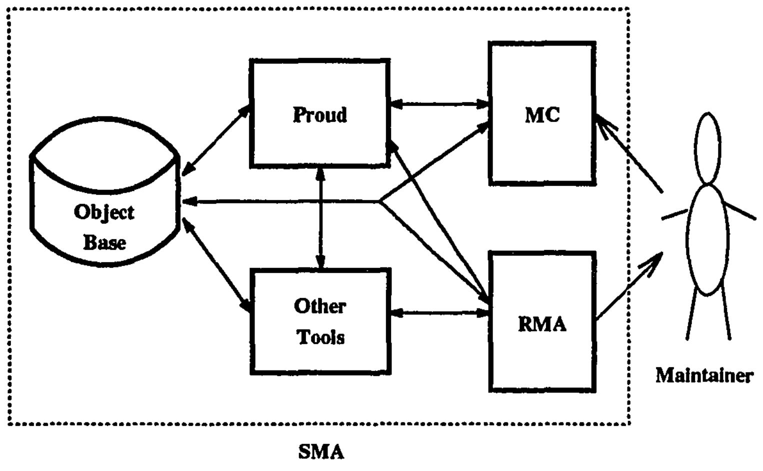


Figure 3.1: Overview of SMA System

As the reverse engineering part of SMA system, the Proud subsystem has the responsibilities to extract information from source code of the target system and analyze the extracted information to provide the proper information to maintainers' requests about the target software system. The objectives of Proud are:

1. to have the capability to extract and analyze a target software system from its source code,
2. to have the capability to attach displaying instructions for displaying tools to emphasize the information in the answer that is relevant to the client's query,
3. to provide a unique interface for clients to access information about the target system, and
4. to shield clients from changes in PROUD tools.

To fulfill the goals of the Proud subsystem, some new ideas and approaches are adopted in the system. These new ideas and approaches try to overcome some of the weaknesses of existing approaches and methodologies.

3.1 New Ideas and Approaches

The following new ideas and approaches are used in Proud subsystem: model representation, plan-based facility manager, rule-based reverse engineering tools, and integrated reverse engineering and analysis environment. This section will briefly discuss all of the new ideas and approaches. Some ideas and approaches are discussed in detail in the subsequent chapters.

3.1.1 Graphical Representation Language

Proud provides a graph query language for accessing Proud. This query language provides users the ability to find particular properties of the target system. This query language is based on an extended ERA representation – the MERA (META Entity Relation Attribute) language[45]. The same MERA language is also used to represent most of the information extracted from source code, the results of the analysis processes, and the knowledge used in Proud.

MERA – A Flexible Graphical Software Description Meta-Language

MERA, Meta Entity Relation Attribute, is a graphical language designed for representation of all types of software specifications and other properties of software. A unique feature of MERA is that it can be used to define many different graphical languages by creating a *meta-graph* in MERA that describes the formalism (lexicon and grammar) of the specific graphical language.

The MERA language also includes many features that are commonly found in artificial intelligence knowledge representation languages. MERA applies semantic networks, sets, ontological markers for intensionality, and user-adaptation to visual representation of both software engineering information as well as artificial intelligence concepts and artifacts.

Following the entity-relationship data model [8], the basic building blocks of MERA are *entities*, *relations*, and *attributes*. Entities are represented as icons and relations as lines. Attribute are name-value pairs and are associated with *objects*, which are defined to be all entities and relations. Attributes have type constraints, range constraints on numeric types, a constraint on when values are required, a set constraint that specifies whether an attribute can consist of a set of values, and an optional default value. Relations have a source and a sink that are objects. Objects have names and constraints on when a name is required. Relations also have constraints on the types of their sources and sinks, connectivity, reflexivity, and multiple relations of the same type on the same pair of source and sink objects.

MERA includes many features commonly found in artificial intelligence knowledge representation languages such as the KL-ONE [5] and KODIAK [49] family of semantic network based knowledge representation languages. MERA has a type hierarchy organized in a lattice with subsumption of subtypes by supertypes. The subsumption hierarchy has strict inheritance of all constraints of supertypes by their subsumed subtypes. Another feature of MERA that is found in AI knowledge representation schemes is the explicit encoding of intensionality. A special attribute of all objects, *number*, is used to encode the intensionality in MERA.

In MERA, *definitional* information such as attribute values, object names, and graph connections are clearly separated from *view* information such as the location of objects, the icon used to display an entity, and whether or not to display names or attributes. Separating definitional data from view data has many advantages. It allows different groups of users to specify different ways of viewing the same data. Also, programs that produce only definitional information need not be concerned with view information.

MERA provides a formal specification of the syntax used for text file storage of MERA - Common Data Format (CDF). CDF is used as a data interchange format among the many programs that use MERA representations. For more detailed information about MERA language and CDF definitions, please refer to [45]. Information extracted by Proud regarding the target system is represented in RMA *Requirement Modification and Analysis* models. Appendix A provides a brief description of these models.

MERA in Proud

An advantage of using MERA is integrity of languages used in SMA. MERA is used extensively in SMA as a representation language for describing the requirements of maintenance requests, as well as the results of analysis processes. Thus, there is less difficulty for users of SMA to use MERA for representing knowledge.

Knowledge representation in Proud has the following features:

- The separation of tools and their functions enables the use of the object-oriented concept to knowledge representation and simplification of the task for capturing the knowledge regarding tools.
- The graphical format represents knowledge in a way close to real perception of the knowledge in user's mind, makes it easier to express the knowledge, and makes maintenance of the knowledge less complicated.
- Using the same language for data and knowledge representation reduces users' effort needed to learn a system and increases users' familiarity of the language so that the knowledge can be expressed more accurately.

This approach may overcome some weaknesses of some reverse engineering tools discussed in the last chapter.

3.1.2 Knowledge-based Facility Manager and Tool Integration

Tool integration is an active research area for CASE tools. The basic model used for Integrated CASE is the model defined by the European Computer Manufacturers Association [15]. Among the different dimensions of integration for CASE, *tool integration* is the most frequently provided form of integration in CASE and other environments for software development and maintenance.

Brown and McDermid [6] defined the classification of tool integration. Five levels are defined – carrier, lexical, syntactic, semantic, and method. Carrier level integration is the lowest. One example of carrier level integration is the Unix environment, which allows tools to be integrated on a single, consistent file format. On the lexical level, tool sets share data formats and operating conventions that let them meaningfully interact. On the syntactic level, all relevant tools agree on a set of data structures, or more precisely on the rules regulating the data formation. At the semantic level, a common understanding of data structures is augmented with a common definition of the structures' semantics. The method level is the highest integration. This level requires agreement not only on the data structures and operations on them, but also on the specific development process. In other words, tools "know" their role in the process and can interact with other tools. Brown and McDermid point out that only meaningful integration for CASE occurs at least at the syntactic level.

Thomas and Nejmeh define the four relationships for tool integration – presentation, data, control, and process [46]. For data integration, there are five properties: inter-operability (how much work must be done for a tool to manipulate data produced by another), non-redundancy (how much data managed by a tool is duplicated in or can be derived from the data managed by the other), data consistency (how well do two tools cooperate to maintain the semantic constraints on the data they manipulate), data exchange (how much work must be done to make the nonpersistent data generated by one tool usable by the other), and synchronization (how well does a tool communicate changes it makes to the values of nonpersistent common data). For control integration, the properties are: provision (to what extent are a tool's services used by other tools in the environment) and use (to what extent does a tool use the services provided by other tools in the environment). For Process integration, there are three properties: process step (how well do relevant tools combine to support the performance of a process step), event (how well do relevant tools agree on the events required to support a process), and constraint (how well do relevant tools cooperate to enforce a constraint). For presentation integration, the properties are:

appearance and behavior (to what extent do two tools use similar screen appearance and interaction behavior) and interaction paradigm (to what extent do two tools use similar metaphors and mental models).

In Proud, data is represented in the MERA language. This language is used as the common representation format for most of the data used in the environment. MERA has clear definitions about syntax of data structures (definitions of CDF format for data exchange), so tool integration is achieved at least at the syntactic level. MERA also clearly defines the semantics of its META level definitions. This provides some degree of integration at the semantic level.

Data integration for the tools in Proud achieves the following features: There is little work, if not none, to do to pass data from one tool to another since all of them deal with same type of data – MERA. All data is managed by one component – OB, so there is a little redundancy. All data is represented in one language, so there is no problem for data consistency. A plan-based facility manages ensures all data are up-to-date and invokes proper tools to regenerate some of the data if they become invalid. This process ensures synchronization.

For presentation integration, all the tools in Proud have similar command formats. All the tools use the same character for the same option, and access the same environment variables. All tools use the MERA model as the mental model for data presentation, so this achieves the interaction paradigm property.

There are several problems regarding tool integration: Which tools should be used and in what order to generate the desired information? What is the required input data for a particular tool? What does a tool produce and what is the name of the product? Proud applies a knowledge-based approach to management of tools in order to solve these problems. A similar approach was employed in a compiler construction system called "Eli" [18]. This approach is also promising in order to achieve the following three goals that were given as some of the directions towards next-generation user interfaces in [37].

Noncommand-based Interaction : Users can carry out what they really want to do without describing how in detail by using commands. For example, the system, not the user, chooses a tool that is suitable for a given task.

Object Visibility and Invisibility : Many objects can be implicit and hidden, although some objects should be visible to a user. For example, the system looks for the input data of the chosen tool that is necessary to produce a specified output data. A user may not be aware of the input data.

There is a knowledge base in Proud that hold the status of data in the system as well as the knowledge of each tool under the control of Proud. The knowledge about a task performed in a tool is represented by a graph written by a specially defined formalism in MERA. Each graph indicates what task to perform, the pre-conditions and the post-conditions of the tasks, as well as the tools to carry out the task. By using this knowledge, Proud is able to launch a tool whenever it is needed.

The benefits of this approach are:

1. Releasing clients from knowing how to obtain particular information about the target system by using noncommand-based interaction. Users can carry out what they really want to do without describing how to do so in detail by using commands.

2. Converting the actual Object Base to a virtual Object Base from the viewpoint of its clients. The clients can get a particular object without knowing whether or not it is in the Object Base, because Proud will automatically invoke the appropriate tools to generate the missing object and add it to the Object Base.

3. Allowing the addition of new tools without affecting user's knowledge about the system, in most cases.

3.1.3 Rule-Based Reverse Engineering Tools

By using rule-based tools, the knowledge associated with a particular programming language or a particular model can be moved away from tools themselves and can be put into rules (knowledge). So the tools become independent of programming languages and/or models. This will extend the capabilities of the tools and bring flexibility to the tools. Much less effort is required to process software systems written in different programming language that were not included in Proud previously. To add a new programming language requires only a parser, which generates a syntax tree from the programs written this language, plus a set of table generation rules

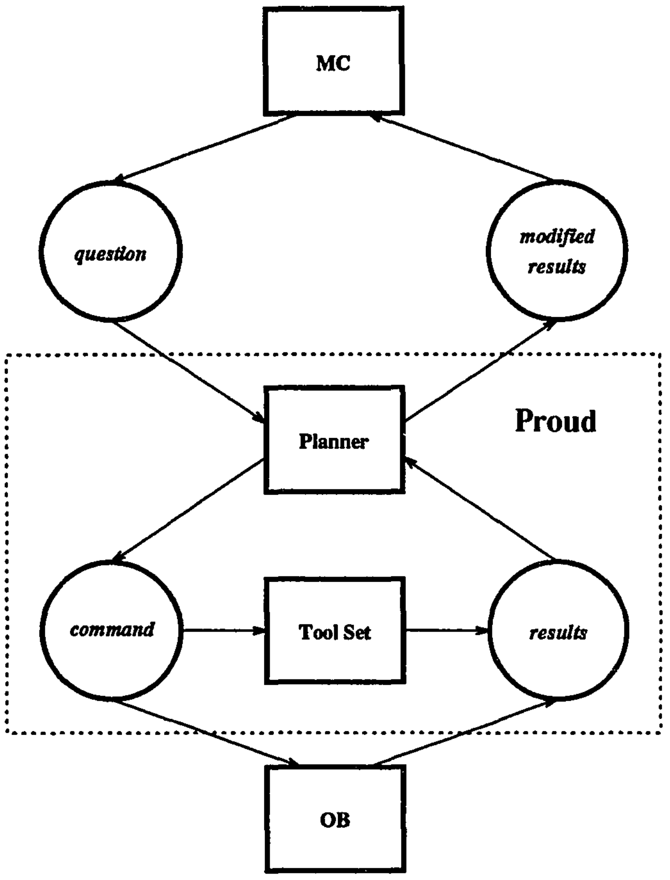


Figure 3.2: The Architecture of Proud and Its Environment

for generating tables from the syntax tree and a set of of model generation rules for generating models. Also this approach makes it possible to generating different representation for different users according to their perspective. Finally, this approach can reduce the resources needed to develop the tools and speed up the development.

3.2 System Architecture

Proud consists of two major components: Planner and Tool Set. Figure 3.2 shows the architecture of Proud and its environment.

The reverse engineering and analysis tools include:

- extracting and analyzing the organization of the target system,

- extracting and analyzing the relationships between components in the system,
- analyzing the static behaviors of the system,
- analyzing the dynamic behaviors of the system,
- tracing system behaviors from different aspects.

3.3 The Environment of Proud

As the reverse engineering and program analysis subsystem of SMA, Proud has two interfaces to the other components of SMA. One is the interface to the Object Base (OB). This interface sends queries to Object Base, either in OBGQ (Object Base Graphical Query) or OBQ (Object Base Query) for retrieval and manipulation of information in Object Base. The results expected from Object Base are graphs in CDF (Common Data Format).

The other interface is the interface to MC or other components that use Proud. The incoming information to Proud is query graphs in PUQ (ProUd Query) format. The expected information returned by Proud will be graphs in CDF that are the answers to the queries.

Since Proud is an open system, all tools in Proud can be invoked either by Planner or by a user directly. Because using the tools directly requires extensive knowledge about the tools, only users who have the knowledge should use the tools directly. For using reverse engineering tools developed in the Proud project, please refer to [22].

3.4 Definitions and Terms Used in Proud

3.4.1 Naming Schema

This section presents the naming scheme used in Proud for identifying data files. The most essential part of the naming schema is the term "core name".

Definition 1 *A* **core name** *is a string. It is created from a file name by removing any suffix defined in this section.*

For example, the core name of a file named `foo.cobol` will be `foo` since `cobol` is a suffix defined in this section.

Notations for Describing Naming Schema

Changeable items in a schema are surrounded by < and >. Other symbols in a schema represent constants and should be used as is.

Some frequently used changeable items are:

language stands for the name of a programming language.

meta stands for the name of a META graph.

table stands for the type of a table.

s-graph stands for the name of a graph used as the source in linkage generation.

d-graph stands for the name of a graph used as the destination in linkage generation.

s-meta stands for the name of the META graph of the source graph.

d-meta stands for the name of the META graph of the destination graph.

s-core stands for the name of the file containing the source graph.

d-core stands for the name of the file containing the destination graph.

Source Code File

The naming schema for source code is:

```
<core name>.<language>
```

Table 3.1 shows the suffixes for different languages. For example, a file named `foo.cobol` is assumed written in COBOL since it has a `cobol` suffix.

Table 3.1: Default Suffixes for Source Code File

Language	*Suffix*	*Alternative*
COBOL	cobol	cob
C	c	
PASCAL	pascal	pas
ADA	ada	
FORTRAN	fortran	for
LISP	lisp	
JCL	jcl	
C++	cpp	cc
PROLOG	prolog	

Syntax Tree and Tables

The naming schema for syntax tree and table files is:

```
<core name>.<table>.
```

Table 3.2 shows the suffixes currently defined.

Table 3.2: Suffixes for Syntax Tree and Table Files

Type	*suffix*
Syntax Tree	tree
Data Declaration Table	dec
Data Definition Table	def
Data Reference Table	ref
Program Structure Table	ps
Calling Structure Table	cs

For example, if a syntax tree file is generated from the source code file `sample.cobol`, then the name of the syntax tree file is `sample.tree`.

MERA Graph

The naming schema for RMA graphs generated by reverse engineering tools are named by the scheme discussed below.

The term *module* is used in the naming scheme. The definition of a module varies depending on the programming language used. In COBOL, a module will be either a section, a paragraph, or a set of sections and/or paragraphs called by a PERFORM statement. In C, a module will be a function.

The naming schema for a graph is:

```
<core name>@<module name>
```

For example, if an IOPM graph file is generated from a syntax tree file `sample.tree` that is generated from `sample.cobol`, then the name of a graph representing section `aaa` will be named `sample@aaa`.

For some types of models, an overview graph will be generated. The name of the overview graph is:

```
<core name>@OVERVIEW
```

For example, if the source code file is `sample.cobol`, the name of the overview graph will be `sample@OVERVIEW`.

For each source code file, there is one *CDF* file that contains all graphs of same type that are generated from the file. The naming schema for a graph file is:

```
<core name>.<meta>.cdf
```

For example, a file contains all IOPM2 graphs generated from `sample.cobol` will be `sample.IOPM2.cdf`.

Linkage

The naming schema for a linkage graph is:

```
{<s-graph>@<s-meta>}{<d-graph>@<d-meta>}
```

For example, if a CPM graph named `sample@aa` is linked to a DM graph named `sample@bb`, then the name of the link graph is `{sample@aa@CPM}{sample@bb@DM}`.

The naming schema for a CDF file containing a set of linkages from one type of graphs to another type of graphs is:

```
<s-core>.<s-meta>--<d-core><s-meta>.cdf
```

For example, a linkage CDF file for linkage from `sample.IOPM.cdf` to `sample.DM.cdf` will be `sample.IOPM--sample.DM.cdf`.

Other Naming Conventions Used in Proud

There are other types of files used in Proud. Most of them are the knowledge or definition information used by the tools in Proud. They are not used by ordinary users of Proud. The maintainers of Proud or SMA should know the naming conventions for these files.

Model Generation Rules The naming convention for model generation rule files is:

```
<language>.<meta>.rule
```

For example, the name of a model generation rule file for generating an IOPM2 graph from COBOL source code would be `cobol.iopm2.rule`.

Table Generation Rules The naming convention for table generation rule files is:

```
<language>.<table>.rule
```

For example, the name of a table generation rule file for generating Data Definition Table from COBOL source code will be `cobol.def.rule`.

Linkage Generation Rules The naming convention for linkage generation rule files is:

```
<s-meta>.<d-meta>.rule
```

For example, the linkage generation rules file for CPM and DM graphs is: `cpm.dm.linkage`

Tree Node Definition The naming convention for tree node definition files is:

```
<language>_node.h
```

For example, the name of the tree node definition file for the COBOL programming language is `cobol_node.h`.

3.4.2 Notations for Grammar Description

All grammar descriptions in this paper use notations defined in this section. In most cases, the notations defined in this section are the notations commonly accepted as grammar description notations.

All terminals are written in upper-case and all non-terminals are written in lower-case. Characters other than letters that are used as terminals are surrounded by a pair of single quotes.

Elements in a pair of brackets "[" and "]" are optional. Parentheses "(" and ")" are used to group several elements into a group. An asterisk "*" indicates that the element it follows repeats zero or more times. A plus "+" indicates that the element it follows repeats one or more times.

Chapter 4

PUQ - Interface of Proud

4.1 Overview

Program understanding plays a very critical role in software maintenance. Automated or semi-automated facilities dramatically reduce time and effort required for understanding a target system to be maintained by providing accurate and relevant information efficiently.

Proud contains various facilities for supporting information extraction, program analysis, and program abstraction. To use these facilities effectively and efficiently, an intelligent facility manager has been implemented. This facility manager is called PU-Planner and its objective is to coordinate facilities in Proud to provide accurate and relevant information to its clients. To achieve this objective, an interface is required to accept clients' requests and return the information back to the clients. This is PUQ (Proud Query Language).

Most information that clients request is stored in Object Base (OB). Information stored in OB is represented in an extended ERA data model - MERA (META Entity Relation Attribute). A query language that has the same or similar format as the retrieved information is more desirable because it will be easier to understand and use. Thus the MERA language is chosen as the base for PUQ, too.

Queries in PUQ are categorized into three groups:

- for retrieving information associated with target system - Proud Retrieval Query (PRQ).
- for allowing users to manipulate information they retrieved - Proud Manipulation Query (PMQ).
- for managing different version of a graph created by users - Proud Version Query (PVQ).

Since PUQ is based on MERA, all queries in PUQ have the same basic format. Due to the differences of functionalities in different groups, the detail structures and formats of queries in difference groups vary.

The rest of this chapter discusses the syntax and semantics of PUQ. Some of the concepts and ideas, such as *delivered concept* and *binary operator*, defined in [26] are used in PUQ with modification and extension. Section 4.2 discusses some definitions used in PUQ and the basic format of a query. Sections 4.3 to 4.5 discuss queries in PRQ, PMQ, and PVQ in detail.

4.2 Definitions

4.2.1 Query Structure

PUQ is defined in terms of MERA. Both incoming queries and return values are MERA graphs in CDF (Common Data Format). For the definition and description of CDF, please refer to [45].

A empty graph returned as the answer for a query indicates a failure of the requested operation. For a PRQ query, it represents no answer to the query. For a PMQ query, it represents the operation failed and information in OB is not changed. For a PVQ query, it represents either the information requested does not exist, or the operation on versions failed.

If the incoming query has syntax or semantic errors, the query itself will be returned, plus some display information to indicate the possible errors.

4.2.2 Concept of Graph

Since MERA is originally designed to represent software specifications, it lacks some properties that are necessary for a query language. Some extensions are necessary. On the other hand, MERA is too complicated to be used fully as a query language. It will cause unnecessary complications for query processing. PUQ uses a subset of the MERA language, plus some extensions focusing on query expressive power.

One restriction on the MERA language is the term *Graph.* There is no formal definition of *graph* in the MERA language. To reduce the complexity of query interpretation and analysis, a *graph* concept is formally defined in PUQ and used as the basic unit of query. The definition is the following:

Definition 2 *A* **graph** *is a collection of objects.*

A connected graph concept is used in PUQ to define the scope of a query. The definition of a connected graph is the following:

Definition 3 *A* **connected graph** *is a graph and has the following property – any object in the graph can reach any other objects in the graph by traversing the graph (the directions of relations in the graph are ignored).*

There are four types of graphs – *Query*, *Data*, *Definition*, and *Version* used in PUQ. Each query consists of one *Query* graph and optionally, several *Data*, *Definition*, or *Version* graphs. All the extensions and restrictions on the original MERA language are applied on *Query* graphs and *Definition* graphs.

Query graphs contain questions to be answered or operations to be performed. They represent the requests from the clients. Details of *Query* graphs will be discussed in Sections 4.2.3.

Data graphs are ordinary MERA graphs. They represent the scope of the query. If there is no *Data* graph, the query is applied to the information stored in OB regarding regarding the target system.

Definition graphs are used to define new concepts to be used in the future. These concepts can be used in either *Query* graphs to construct questions, or *Definition* graphs to define new concepts.

Version graphs are used for representing the version status of information in OB regarding the target system. Details of *Version* graphs will be discussed in Section 4.5.2.

4.2.3 Query Graph

A *Query* graph is defined on a specially designed formalism – **PUQM** (ProUd Query Model). This formalism contains all RMA (Requirements Modification and Analysis) and AFM (Application File Modeling) formalisms that are already defined, as well as some objects that reflect the requirements for construction of questions. Figure 4.1

shows the new objects defined in PUQM. Graphs in RMA represent properties of program and graphs in AFM represent properties of control schema, such as JCL.

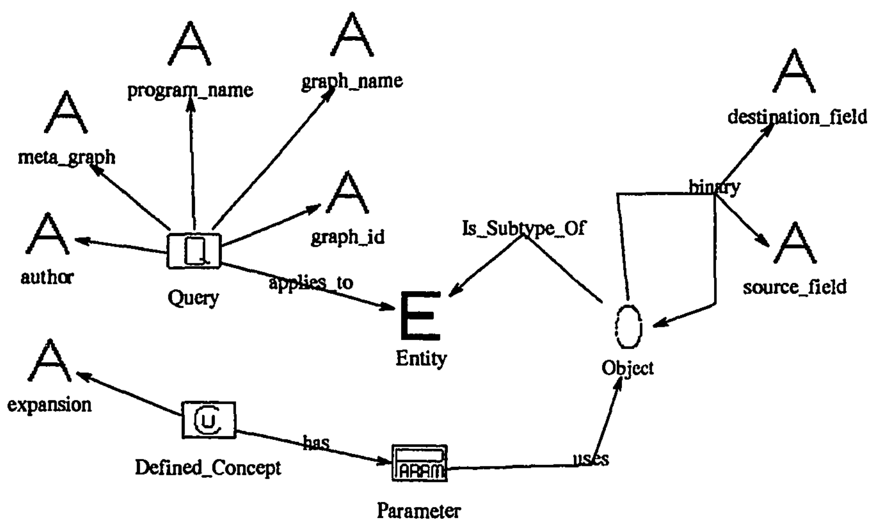


Figure 4.1: PUQM – Formalism for Proud Query Language

In the Figure, the object named **Object** represents the top level object that is defined in META (top level definition of the MERA language, all formalisms are derived from it), so it represents all objects defined in RMA as well AFM. All entities defined in PUQM are sub-classes of the class **Entity** and all relations are sub-classes of the class **Relation** in META. A *Query* graph is a connected graph and contains one and only one **Query** entity.

New Objects Defined in PUQM

The following are the definitions of objects defined in PUQM and not defined in other META graphs. These objects inherit all the properties defined in the top level of the MERA language and some of them has their own attributes.

Query entity indicates the graph is a *Query* graph. Its name is used to indicate the operation requested. It has four attributes. Except for the **Author** attribute, they are used to define the scope of the query and are optional:

author indicates the sender of the query. The value of this attribute is the user_id of a user who sends the query. If the query is sent by a process, the user_id of the owner of the process will be used as the value of the attribute.

program_name indicates the name of a source code file. The graphs generated from the source code file are the scope of the operation. The possible value is a set of regular expressions as defined in Section 4.3.2, separated by "|"s.

graph_name indicates the name of graphs. These graphs are the scope of the operation. The possible value is a set of regular expressions as defined in Section 4.3.2, separated by "|"s.

graph_id indicates the graph_id of graphs. These graphs are the scope of the operation. The possible value is a set of regular expressions as defined in Section 4.3.2, separated by "|"s.

meta_graph indicates the names of the META graphs. Graphs whose META graphs are one of these META graphs are the scope of the operation. The possible value is a valid META graph name, or several valid META graph names separated by "|".

Defined_Concept entity represents a defined concept. Its name is used to indicate the concept. It has one attribute:

expansion indicates whether or not the concept should be expanded into its definition when the results are returned. Its possible values are boolean. If its value is **true**, then the entity will be replaced by its definition in the results before they are returned.

Parameter entity represents the parameter of a concept. Its name is used as the identifier of the parameter. It is connected to an object that is used as the actual parameter. The object replaces the corresponding object in the definition of the defined concept when the defined concept is expanded by its definition. This entity has no attributes.

has relation connects a **Defined_Concept** entity with a **Parameter** entity. It means the concept has that parameter. This relation has no attributes and it is a one direction and one-to-many relation.

uses relation connects a **Parameter** entity with an object. It means that the object is used as the actual value of the parameter. This relation has no attributes and it is a one direction and many-to-many relation.

applies_to relation connects a **Query** entity to another entity. It means that the connected objects are the conditions of the query. This relation has no attributes and it is a one direction and one-to-many relation.

binary relation connects two objects that are defined in other META graphs. It represents a relationship between the values of the attributes in those two objects. This relationship must be hold in the result graphs. The relation is a one direction and one-to-one relation. The name of the relation indicates the condition that must hold and its possible values are comparison operators such as "=", ">", etc. The relation has two attributes:

source field represents the attribute in the source object whose value is used for the operation.

destination field represents the attribute in the destination object whose value is used for the operation.

Definitions for Existing Terms

Some terms in the MERA have more precise meanings in a Query Graph. These terms are sub-graph, some values of the *number* attribute, and the type of an object.

Number attribute plays a important role to form the pattern for retrieval. Some values in this attribute have special meanings in query.

0 indicates the negation condition. No object in the result graphs should match this object.

$*,+$ indicates the closure condition. There may be several objects in the results graphs that match this object and these objects should form a path. $*$ represents zero or more appearance of the matched objects and $+$ represents one or more appearance.

unspecified indicates the multiple matching condition. There may be several objects in the result graphs that match this object and all of them will be appeared in the result graphs.

Type represents the type of the object in the result graphs. If the type of this object has sub-types, then any object that is of the same type or any of the subtypes matches.

sub-graph attribute has different meanings, depending on the value of the *number* attribute of the sub-graph. If it is zero, the sub-graph represents the constraints applied to the object. Otherwise, the sub-graph is part of the pattern to be matched. If the sub-graph is used as a constraint on the object, one set of connected objects in the sub-graph forms one constraint and all sets form an **or** condition of the constraints.

4.2.4 Definition Graph

A *Definition* graph is used to define a concept that may be used later in a *Query* graph. A *Definition* graph contains no **Query** entity. It has one and only one **Defined_Concept** entity, which represents the concept to be defined. The name of the entity is used as the identifier of the concept.

A *Definition* graph contains one or more **Parameter** entities that represent the parameters of the concept. These entities connect with objects that serve as the virtual parameters. The names of the **Parameter** entities are used as the identifiers of the parameters. Figure 4.2 shows an example of a *Definition* graph. This graph defines a concept named **coupled-tasks**. The definition of this concept is that two **tasks** are coupled if both of them exchange data via some **internal data**.

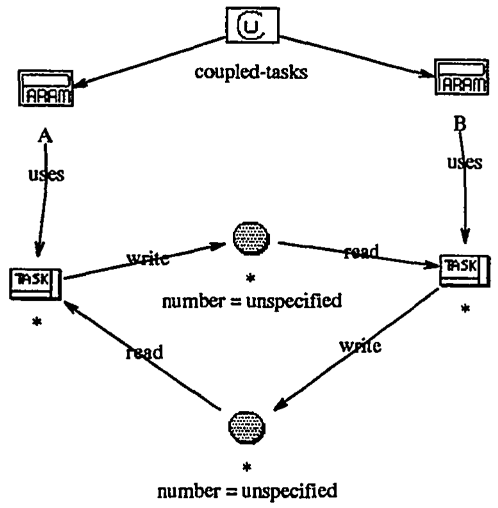


Figure 4.2: An Example of a Definition Graph in PUQ

4.3 Proud Retrieval Query – PRQ

4.3.1 Semantics

Proud Retrieval Query (PRQ) is used to obtain information about the target system. The information may already exist in OB, or may have to be generated from other sources. Some information may be provided by reverse engineering processes, and some by program analysis processes. From the client's point of view, the source of information is not important and it is not specified in the query.

The retrieval criteria are specified by three sets of information in the query: the name of the *Query* entity, the objects and their relationships in the *Query* graph, and the value of attributes of objects in the query graph. The value of an attribute can be either a constant, which serves as a condition applied to the query, or a regular expression, which serves as a variable of the query (question to be answered).

4.3.2 Regular Expression

Any attribute of an entity or a relation (including the name, type, and other field), except the **Number** attribute, in a *Query* graph can be used to form questions by using a regular expression as its value. Table 4.1 shows the summary of the regular expressions available in PUQ. A precise definition of regular expressions can be found in the Unix manual *egrep(1)*. In Table 4.1, x or y represent an ASCII character, and X or Y represent a regular expression.

4.3.3 Query Format

A query in PRQ consists of a *Query* graph and/or several *Data* graphs. The *Data* graphs specify the scope of the query. If there is no *Data* graph in the query, the scope of the query is the graphs of the entire system and may be limited by the scope attributes of the *Query* entity.

A *Query* graph must be a connected graph. If the graph becomes several disconnected sub-graphs after the *Query* entity is removed, each disconnected sub-graph serves as one set of conditions for the query, and the results for each sub-graph is combined together to form the results of the query. In other words, there is an "OR"

Table 4.1: Summary of Regular Expression

regular expression	matching pattern
x	the letter x itself
.	any one letter
$[x - y]$	range of consecutive ASCII characters
$[\hat{}X]$	any one letter except X
^	beginning of string
\$	end of string
XY	concatenation of X and Y
$X \mid Y$	union of the matches of X and Y
$X*$	Kleene closure
$X+$	closure
$\backslash x$	the letter x itself even if x is normally used to form a regular expression
(X)	group of regular expressions

relation between disconnected sub-graphs. Also, only one object in each sub-graph can be connected with the **Query** entity.

If there is more than one graph satisfying the retrieval criteria specified in the query, the most relevant one (determined by Proud, not necessarily correct) will be returned. To retrieve the other graphs, the same query can be issued to Proud again. After all qualified graphs are returned, an empty graph will be returned to indicate no other graphs satisfy the criteria.

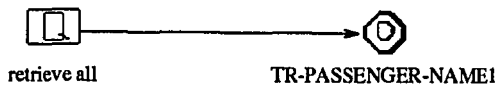


Figure 4.3: An Sample Query for Retrieve All

The operation requested is represented by the name of the **Query** entity. The formats for retrieval queries are the following:

- A *retrieve all* query requires that an entire graph that satisfies the query is retrieved. It consists of a *Query* graph and optionally, several *Data* graphs. The objects in the *Query* graph serve as the pattern for retrieval. After the result graphs are retrieved, all the objects matching the pattern are highlighted. Figure 4.3 shows a query to retrieve graphs that contain a **data_item** named "TR-

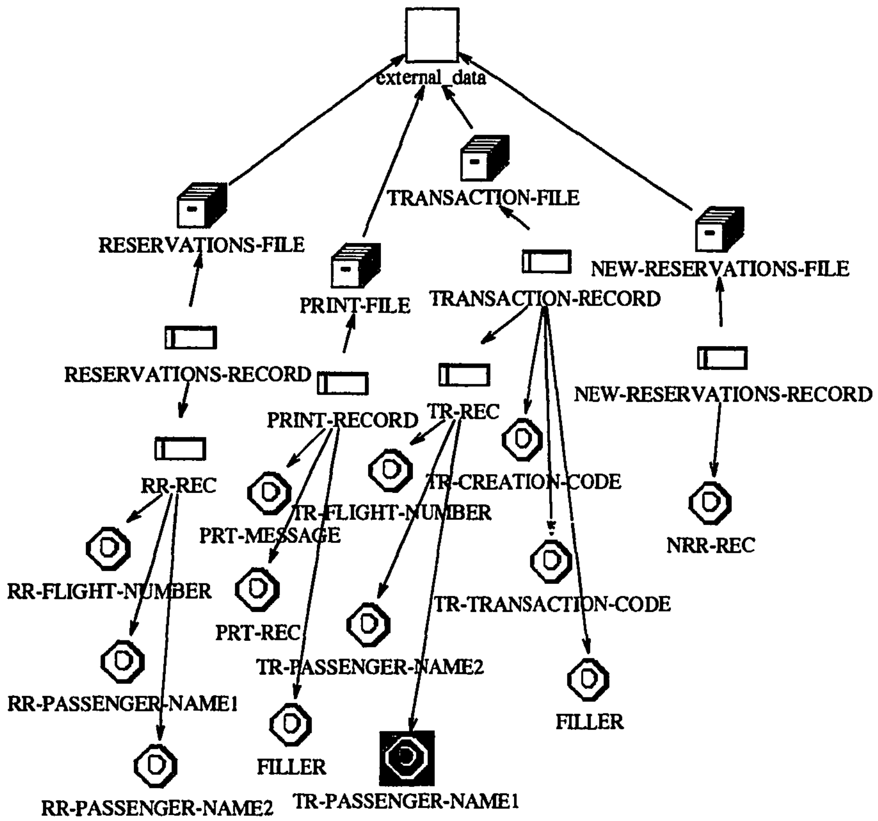


Figure 4.4: Result of the Retrieve All Query

PASSENGER-NAME1". Figure 4.4 shows the retrieval result. The matched object is highlighted.

- A *retrieve partial* query requires that only the objects mentioned in the query are retrieved. It consists of a *Query* graph and optionally, several *Data* graphs. The objects in the *Query* graph serve as the pattern for retrieval. After the result graphs are retrieved, the entire graphs are returned, but only the objects that match the pattern are displayed, other objects are hidden. If some of the objects in the pattern have a "hidden" display option, the objects in the result graphs that match these "hidden" objects are not returned. Figure 4.5 shows a query to retrieve partial graphs that contain a **data_item** named "TR-PASSENGER-NAME1". Figure 4.6 shows the retrieval result. Only the matched **data_item** entity is displayed.

- An *impact* query requires that the graphs that are affected by the objects mentioned in the query are retrieved. It consists of a *Query* graph and optionally, several *Data* graphs. The objects in the *Query* graph serve as the objects to be changed and the graphs affected by the changed objects are retrieved. The affected objects will be highlighted. Figure 4.7 shows a query to analyze the impact of a **data_item** named "TR-PASSENGER-NAME1". Figure 4.8 shows one of the graphs that may be affected by the entity. The affected entities are highlighted.

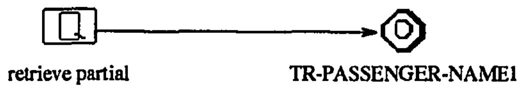


Figure 4.5: An Sample Query for Retrieve Partial

TR-PASSENGER-NAME1

Figure 4.6: Result of the Retrieve Partial Query

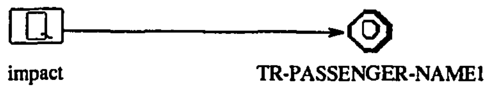


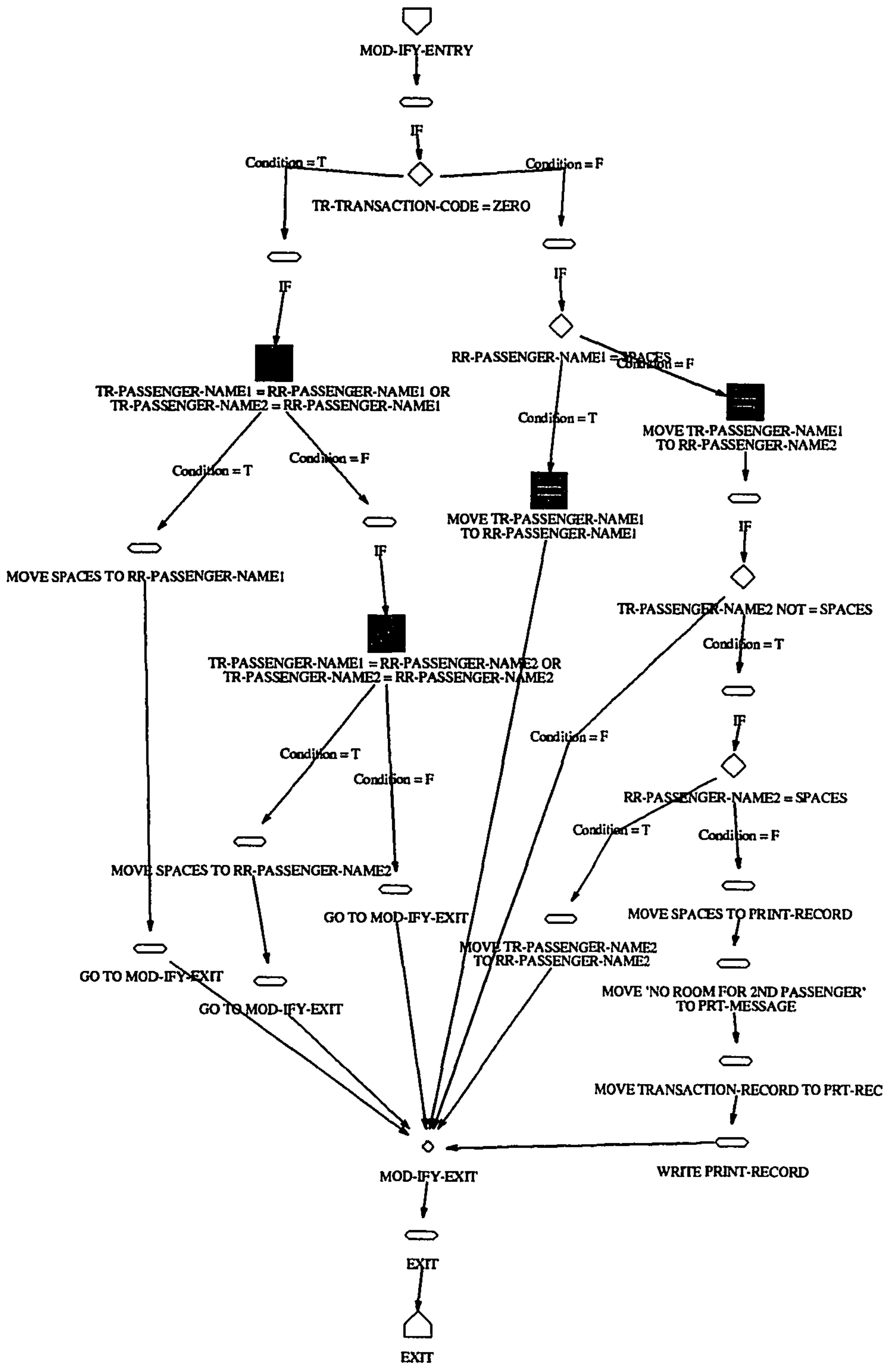
Figure 4.7: An Sample Query for Impact

4.4 Proud Manipulation Query – PMQ

4.4.1 Semantics

Proud Manipulation Query (PMQ) provides the capability for users to manipulate existing graphs in OB without worrying about detailed OBQ and housekeeping.

The manipulations that users can apply to a graph are *store* and *delete*. Both manipulations will create a new version of the graph for the target system. All the changes will be reflected in the new version. Information in the old version will not be affected. Section 4.5.1 will discuss the concept of version control in detail. *Store* allows users to store a graph into OB. *Delete* allows users to delete a graph from OB permanently. Except for the manager, a user is only allowed to delete graphs that were created by the user.



MOD-IFY-ENTRY
IF
Condition = T
Condition = F
TR-TRANSACTION-CODE = ZERO
IF
IF
RR-PASSENGER-NAME1 = SPACES
Condition = F
TR-PASSENGER-NAME1 = RR-PASSENGER-NAME1 OR TR-PASSENGER-NAME2 = RR-PASSENGER-NAME1
Condition = T
MOVE TR-PASSENGER-NAME1 TO RR-PASSENGER-NAME2
Condition = T
Condition = F
MOVE TR-PASSENGER-NAME1 TO RR-PASSENGER-NAME1
IF
MOVE SPACES TO RR-PASSENGER-NAME1
IF
TR-PASSENGER-NAME2 NOT = SPACES
Condition = T
TR-PASSENGER-NAME1 = RR-PASSENGER-NAME2 OR TR-PASSENGER-NAME2 = RR-PASSENGER-NAME2
IF
Condition = F
Condition = T
Condition = F
RR-PASSENGER-NAME2 = SPACES
Condition = T
Condition = F
MOVE SPACES TO RR-PASSENGER-NAME2
GO TO MOD-IFY-EXIT
MOVE SPACES TO PRINT-RECORD
MOVE TR-PASSENGER-NAME2 TO RR-PASSENGER-NAME2
GO TO MOD-IFY-EXIT
MOVE 'NO ROOM FOR 2ND PASSENGER' TO PRT-MESSAGE
GO TO MOD-IFY-EXIT
MOVE TRANSACTION-RECORD TO PRT-REC
MOD-IFY-EXIT
WRITE PRINT-RECORD
EXIT
EXIT

Figure 4.8: Result of the Impact Query

4.4.2 Query Format

A query in PMQ consist of a *Query* graph and, optionally, one *Data* or *Definition* graph. The *Data* or *Definition* graph is used as source for the manipulation operation. The *Query* graph contains only one entity - the **Query** entity.

The name of the **Query** entity is used to represent the requested operation. Two scope attributes of the **Query** entity may be used in PMQ. One is the **Graph_ID** and the other is the **Graph_Name**. The usage of these attributes is dependent on the requested operation. The formats for manipulation queries are the following:

- A *store* query stores a graph into OB. It contains one *Data* or *Definition* graph, which is the graph to be stored. If the **Graph_Name** attribute of the *Query* entity is not empty, then its value will be used as the name of the graph to be stored. Otherwise, the name of the graph to be stored will not be changed. If the operation is successful, a version graph (version graphs will be discussed in Section 4.5.2) will be returned and it contains one graph entity that represents the newly stored graph.
- A *delete* query deletes a graph from OB. It does not contain any *Data* or *Definition* graphs. The **Graph_ID** attribute of the *Query* entity indicates the graph in OB to be deleted. If the operation is successful, a version graph will be returned and it contains one graph entity that represents the newly deleted graph.

4.5 Proud Version Query

4.5.1 Concept of Version Control

The software maintenance process tends to adopt an incremental approach for modification. The version control in PUQ is used to support this approach. PUQ provides an access method to obtain and manipulate information regarding versions. The location of the version control agent in *SMA* is not in Proud and PVQ is just provided as a uniform interface for uses.

There are two types of version control defined in Proud. One is global version control and the other is personal version control. Both types of version are defined in the information about the entire target system. The difference between two types of version is the visibility. The global version is visible to all users of Proud. On the other hand, the personal version is visible only to a particular user.

The information about versions are maintained in OB and organized as a tree structure. Each node in the version tree represents one version. The root of the version tree is the version that contains two types of graphs. One is the graphs generated by reverse engineering tools. The other is the graphs generated by analysis tools from the first type of graphs. Any modification of graphs in a particular version will create a new version, which will be the child of the old version. Proud maintains one global version tree for all users and one personal version tree for each user. Each version contains the following information about the version:

- When it was created.
- Who created it.
- Who is using this version now.
- Names of graphs that belong to this version.

The operation of version control is described as follows:

- When a user issues a PMQ, a new personal version is created for that user. If a personal version tree for the user exists, the new version is attached to the tree. Otherwise, a new personal version tree will be created for the user. The modification is only visible to the user and no one else will see it.
- When the user finishes modification and wants to publish it to all other users, the user should explicitly instruct Proud to do so. In this case, the personal version specified by the user will become a new global version. The personal version tree of the user will be removed.
- A set of special operations is designed for project managers to manage the global version tree. These operations include create a new global version from existing global versions, remove a particular global version, or discard all versions except the root version.

Before a user issues a PRQ, Proud has to know that version the user will like to work on. That version is called the working version for the user. There are two ways to define the working version for the user. One way is to use the version that the use worked on most recently. Or the user tells Proud via a PVQ query that version she/he desires.

4.5.2 Version Graph

Version graph is defined by the PUVM (ProUd Version Model) formalism and is used to represent the current states of information managed by Proud. It has two types of entities and two types of relations. Figure 4.9 shows the graphical representation of the formalism.

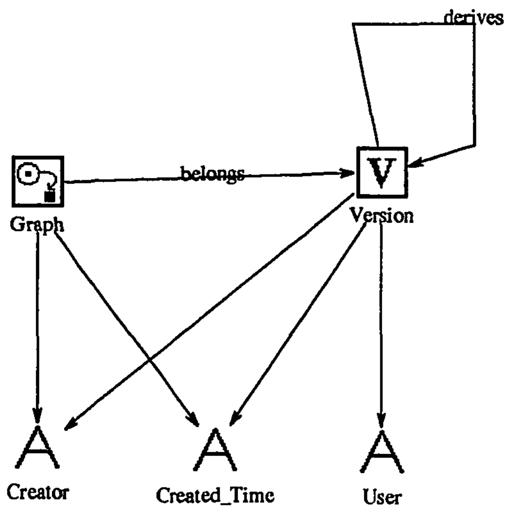


Figure 4.9: PUVM – Formalism for Proud Version Control

The types of entities in PUVM are:

Version entity represents a particular version of information in OB. The name of the entity is the version number of the version the entity represents. The name is required and should be unique. The entity has three attributes and all of them are required:

Create_time When the version was created.

Creator Who created the version.

User Who are using the version now.

Graph entity represents a particular graph in OB. The name of the entity is the graph_id of the graph the entity represents. The entity has two attributes and both of them are required:

Create_time When the version was created.

Creator Who created the version.

The types of relations in PUVM are:

derives relation means one version is derived from the other. The source of the relation is a **version** entity representing the original version and the sink of the relation is a **version** entity representing the derived version. The name of the relation has no meaning and is optional. The relation has no attributes.

belongs relation means a graph belongs to a version. The source of the relation is a **version** entity representing the version and the sink of the relation is a **graph** entity representing the graph. The name of the relation has no meaning and is optional. The relation has no attributes.

4.5.3 Semantics

There are two sets of PVQ. One set of PVQ can be used by all users:

- view versions (global or personal),
- view graphs in version (global or personal),
- check the differences between two versions,
- move to a particular version,
- release personal versions,
- delete one personal version, or
- discard all personal versions.

The other set of PVQ can only be used by project managers. These include:

- create a new version from existing versions,
- delete one global version, or
- discard all global versions.

View version query retrieves either the global or personal version tree. Retrieved information includes the version tree and information for each version.

View graphs query retrieves graphs belonging to a version or several versions. Each graph is represented as an graph entity in a version graph. If the user only wants to view graphs in one version, then all graphs belonging to that version will be returned. If user wants to view graphs in several versions, all graphs in the versions indicated by the query will be returned.

Version differences query checks the differences of graphs between two versions, and returns only graphs that are different to the user.

Set working version query sets the version as the working version for the user.

Delete version query removes the specified version, reorganizes the version tree so that versions derived from the deleted version will be linked to the version from that the deleted version was derived. Then the most recent version of the personal versions is made the working version.

Delete personal version query removes all personal versions for the user, and sets the working version for the user to the global version from that the personal versions were derived.

Release personal versions query creates a new global version from the personal version that is indicated in the query and deletes all personal versions that belongs to the user. The working version of the user will be set to the newly created global version.

Create version query allows managers to reorganize the information in versions to form a new version. This query returns a version graph that contains graphs belonging to the version.

Delete global version query works almost the same as delete one personal version query except that all users whose working version is this version will have no working version. Any PRQ or PMQ query from these affected users will be answered by a null graph with a message in the result attribute to indicate that the version does not exist. These users will have to define another working version.

Discard all versions query removes all global versions except the root version. Any PRQ or PMQ query from the user, whose working version is not the root version, will be answered with a null graph with a message in the result attribute to indicate that the version does not exist.

4.5.4 Query Format

A query in PVQ contains only a *Query* graph. The *Query* graph contains one *Query* entity and optionally other objects defined in the *Version* graph to describe the constraints for the operation. No other types of objects are allowed.

The result will be graphs in PUVM. If the operation requested fails, a null graph will be returned.

The name of the *Query* entity indicates the operation for version control. The formats of version queries are the following:

- A *view personal versions* query contains only a **Query** entity.
- A *view global versions* query contains only a **Query** entity.
- A *view graphs* query contains a **Query** entity and several **Version** entities. The **Version** entities represent the versions that contains the graphs that the user want to see.
- A *version differences* query contains a **Query** entity and two **Version** entities that represent the two versions to be compared.
- A *set working version* query contains a **Query** entity and one **Version** entity that represents the desired working version.
- A *delete version* query contains a **Query** entity and one **Version** entity that represents the version to deleted.
- A *delete personal versions* query contains a **Query** entity.
- A *release personal version* query contains a **Query** entity and a **Version** entity representing the personal version to be released as a global version.
- A *create version* query contains a **Query** entity and **Version** entities and **Graph** entities representing the graphs that should be included in the new version.
- A *delete global version* query contains a **Query** entity and a **Version** entity representing the version to deleted.
- A *delete all versions* query contains a **Query** entity.

Chapter 5

Planner

5.1 Design Concepts

Planner is the central control component of the Proud subsystem. It controls access to and creation of information associated with the target system to be maintained. The information is stored in OB, which is the main storage of the entire SMA system, and/or generated by reverse engineering tools in Proud. *Planner* accepts queries from the user or other SMA components, which are considered clients of *Planner*, and returns required information back to them. Besides these, *Planner* also keeps track of information about existing graphs and relationship between them. Figure 5.1 shows the relationship between *Planner* and its environment.

Queries from clients are analyzed by *Planner* and either translated into OB queries to retrieve information in OB, or converted into plans, which are executed by *Planner* to invoke tools to generate the desired information. Information is integrated into a more suitable form according to the queries before it is returned.

The objectives of *Planner* are:

- Ease of use. This is achieved by providing a uniform graphical query language to access information created or managed by Proud. This language provides a non-procedural interface to the users of Proud.
- Efficiency. This is achieved by an plan-based facilities management approach. *Planner* has knowledge of all tools in Proud and the relationships between tools and data. By using this knowledge, Proud is able to obtain the user-requested information in the most efficient manner.

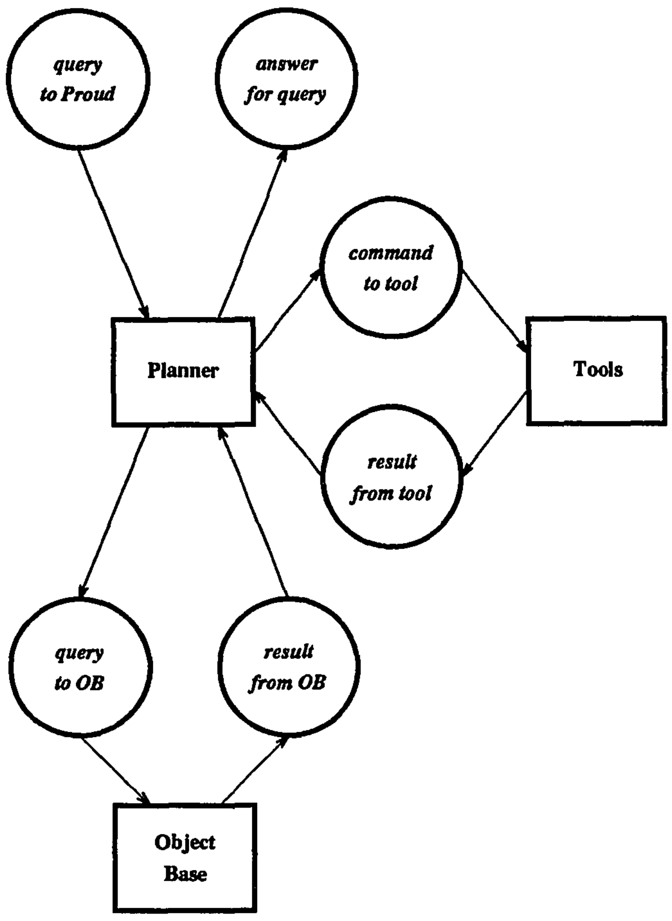


Figure 5.1: The Environment of *Planner*

5.2 Knowledge Representation

In Proud, knowledge regarding functionality of a tool and knowledge regarding other properties of the tool are represented separately. Since one tool may perform several functions and a function may be carried out by two or more tools, the separation makes it easier to represent the knowledge. Otherwise, the knowledge representation for powerful tools may become very complicated and cause problems on creation, modification and maintenance of the knowledge. In addition, the separation makes the planning process using the knowledge simpler, since it deals with simpler knowledge.

5.2.1 META Language for Knowledge Representation

The knowledge used in Proud can be classified into three groups – data, tool, and task. *Data* represents the hierarchical relationship between different types of data. *Tool* represents the hierarchical relationship between different tools used in Proud. *Task* represents the tasks performed by tools, the required input data, and the products produced by the tasks.

Data

Hierarchical class definitions are used for representing information used by the tools in Proud. All the information in Proud are organized into a tree of classes, whose root is a class called **Data**. The classification is according to the type of the information, as well as the relationship between the information and tools that use it. Each piece of information is represented by an instance in one of the classes. Each instance has at least two attributes: the *type* and the *name* of the information. These attributes are defined as the attributes of the class **Data** and inherited by all its subclasses under it. The *type* attribute of a piece of information is denoted by the name of the class that it belongs to. A piece of information can be represented by a superclass of its own class. A piece of information may have additional attributes to express its unique properties. For instance, a piece of information associated with a programming language has an attribute to describe which programming language it is associated with. These additional attributes are defined as the attributes of a class that is the smallest class containing all necessary information in the hierarchy. Figure 5.2 shows the class hierarchy definition of information in the current Proud configuration.

The class hierarchy definition remains stable, unless a new class is created and needs to be added into the class hierarchy definition. On the other hand, the instances representing pieces of specific information are frequently added and removed during the process. A collection of the instances represents the status of information in Proud. When a piece of new information is created, an instance representing this piece of information is added into a class representing the type of the information. When a piece of information is deleted from Proud, the instance representing this piece of information is removed.

There is a type of relation between instances called **depend on**. This relation indicates that a piece of information x depends on the other y. If y is removed or

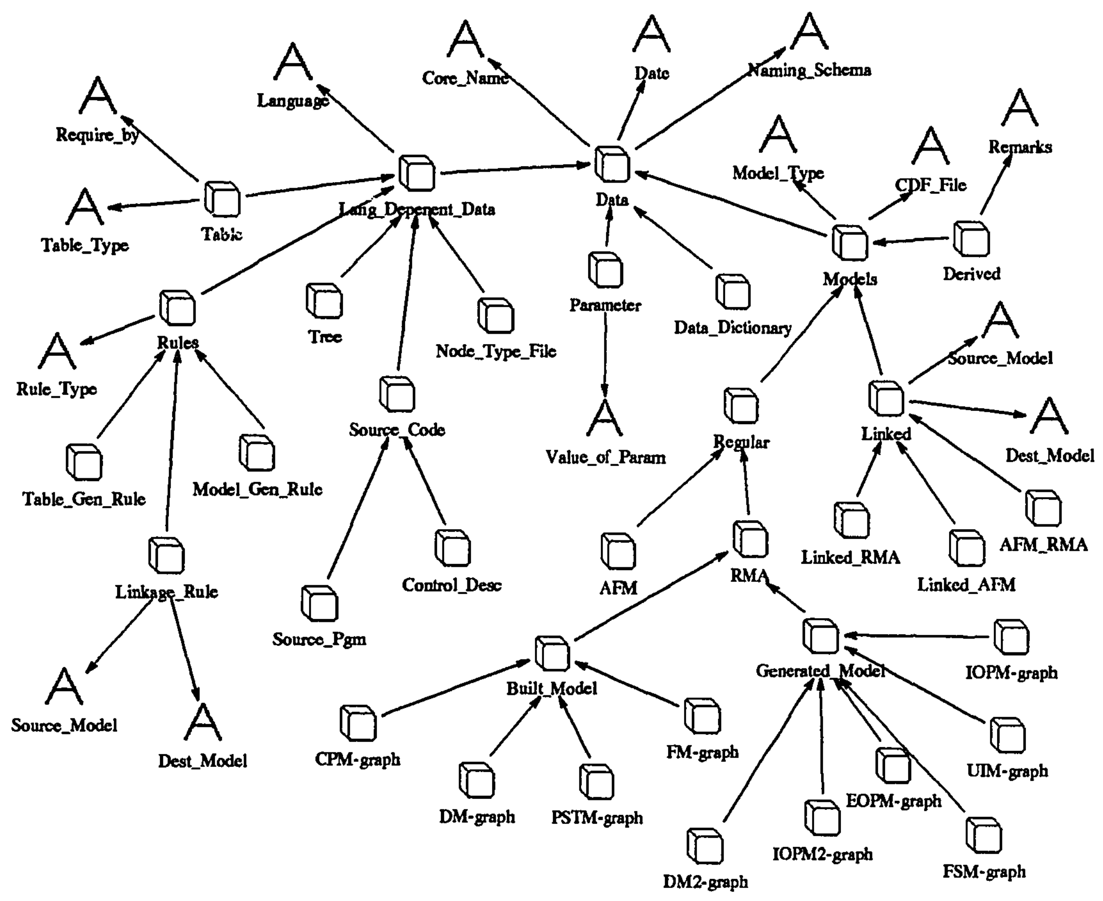


Figure 5.2: Hierarchical Class Definition of Data

updated, x becomes invalid and should no longer be used, since the current content of x may be out-of-date. For example, a syntax tree is generated from a source code file. A **depend on** relation is created from the instance for the syntax tree to the instance for the source code file. If the source code file is modified, the syntax tree no longer represents the syntax of the new source code. This type of relations is used for the planning process, as well as housekeeping.

Tool

Similar to data, tools are organized in a class hierarchy. Tools are classified by their functions performed. The largest class for tools is called **Tool** and has three attributes: *name*, *command format*, and *product*. These attributes are inherited by all instances representing tools. The *name* indicates the name of the tool. The *command format* indicates how to invoke the tool in a Unix environment. It includes the Unix commands of the tool, options, and how to obtain values for options. The *product* indicates the names of the products produced by the tool, as well as the types of the products. Figure 5.3 shows the class hierarchy definition of tools for the current Proud configuration.

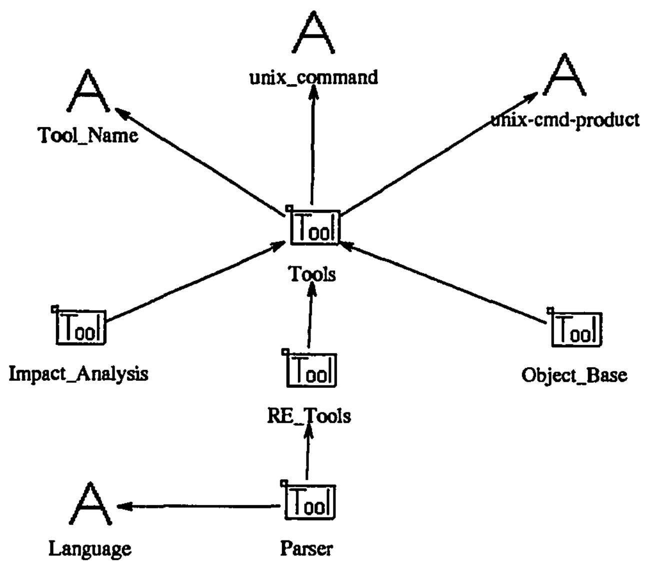


Figure 5.3: Hierarchical Class Definition of Tools

The process for adding a new tool varies depending on the functions performed by the new tool. If the functions performed by the new tool have already been defined in the class hierarchy, only one instance representing the tool is added into the class representing the functions performed by the tool. If the functions have not been defined in the class hierarchy, then a new class must be defined and a new instance is created in the new class.

Task

Knowledge regarding each function performed by one or more tools is represented by **Task** graphs. Each **Task** graph represents a primitive function performed by one tool. A **Task** graph is said to be *generic* if it is not associated with a set of particular input and/or output data. Each **Task** graph consist of three types of entities and up to four types of relations. Figure 5.4 shows the formalism of **Task** graph.

The types of entities in a **Task** graph are:

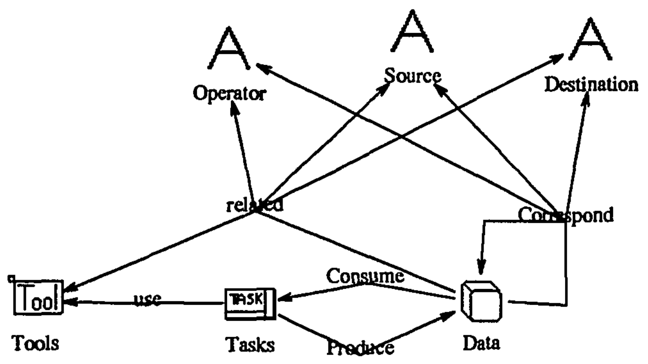


Figure 5.4: Formalism of **Task** graph

Tasks entity represents the function to be performed. There is only one **Tasks** entity in one **Task** graph, since the **Task** graph represents one function to be performed.

Data entities represent the input data and output data. They represent classes in the class hierarchy definition for information.

Tools entities represent the tools that can perform functions. They can be either entities representing particular tools or classes in the tool class hierarchy that represents a set of tools that can perform similar functions on different types of data.

The types of relations in a **Task** graph are:

Consume relation connects from a **Data** entity to a **Tasks** entity and indicates that the data is used as input for the function.

Produce relation connects from a **Tasks** entity to a **Data** entity and indicates that the data is produced as output of the function.

use relation connects from a **Tasks** entity to a **Tools** entity and indicates that the tool is used to perform the function.

Correspond relation connects between two **Data** entities and represents a constraint between a pair of attributes in the two **Data** entities.

related relation connects from a **Data** entity to a **Tool** entity and represents a constraint between a pair of attributes in the two entities.

The **Correspond** and the **related** relations play very important roles for expressing knowledge about a function explicitly. A **Correspond** or **related** relation has three attributes: the *name* serving as the identifier for the relation, the *operation* indicating the operation to be performed, and *names* of attributes in both objects used in the operation. It performs two functions, depending on the values of indicated attributes. If both attributes for the operation have values, this relation acts like a condition check. In this case, a constraint of the Correspond relation is satisfied if the result of the operation indicated by the relation is true. Otherwise, the constraint is not satisfied, which implies that either the function represented by the **Task** graph cannot be performed or one of the entities connected by the relation needs to be removed. If one of the attributes does not have a value, then the relation acts like an assignment. It assigns a value to that attribute so that the result of the operation is true. If both attribute do not have values, then it indicates an error in the **Task** graph. The next section will give some examples.

5.2.2 Generic Plans

A Generic plan graph represents the knowledge of a tool in Proud. Basically each plan contains a **Tasks** object. The **Tasks** object denotes the action to be performed. Currently there are seven generic plans:

- Parsing task,
- Table_Generating task,
- Model_Generating task,
- CPM_Building task,
- DM_Building task,
- FM_Building task, and
- PSTM_Building task.

Figure 5.5 shows the generic plan for the *Parsing* task. The parsing task requires a source code file as input data and produces a syntax tree file. The task is executed by a parser. Several parsers may be associated with the task, but only one of them will be used at one time. Which parser to be used is determined by the language that the source code is written in. This is shown by the **related** relations between the **Parser** and **Tree** entities. For example, the source code is written in COBOL, the only **related** relation whose condition is satisfied is the one that connects **COBOL_PARSER** and **Source_Pgm**. In this case, the *COBOL Parser* will be used to execute the task. The **Correspond** relations state that the **Core_Name** and **Language** of the source code file and syntax tree file should be same. Since the tree file will created, then the name of the tree file will has the same Core_Name as the source code file.

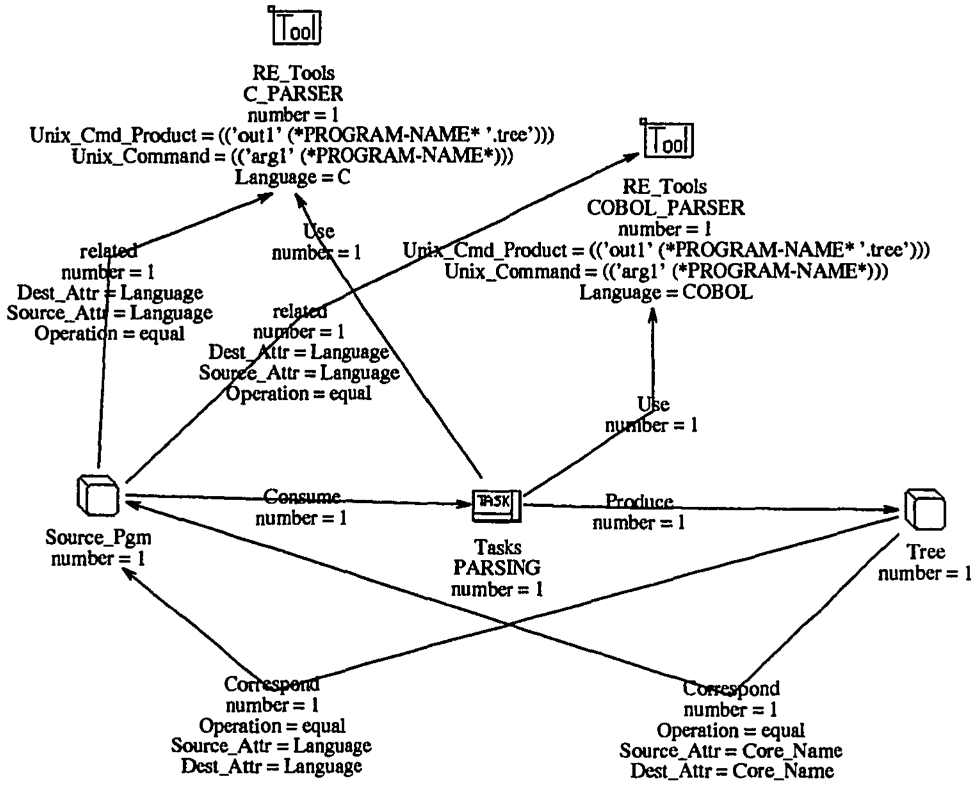


Figure 5.5: Generic Plan for the Parsing Task

Figure 5.6 shows the generic plan for the *Table_Generating* task. It requires three inputs – a syntax tree, a table generation rule file, and a node type definition file. It produces a table. The task is executed by a dedicated tool – *Table Generator*. The generated table will have the same Core_Name as the tree file, indicated by a **Correspond** relation between them. A table generation rule file contains knowledge of generating a particular type of table from a particular language. A node type file contains mappings of node types from integer representations that are used in syntax tree to mnemonic representations that are used in generation rules for a particular language. The node definition file and the table generation rule file to be used should deal with the same language as the syntax tree, indicated by two **Correspond** relations between the **Table_Gen_Rule**, **Node_Type_File**, and **Tree** entities. The table generation rule file to be used should contain the knowledge for generating the particular type of table as required, indicated by a **Correspond** relation between the **Table_Gen_Rule** and **Table** entities.

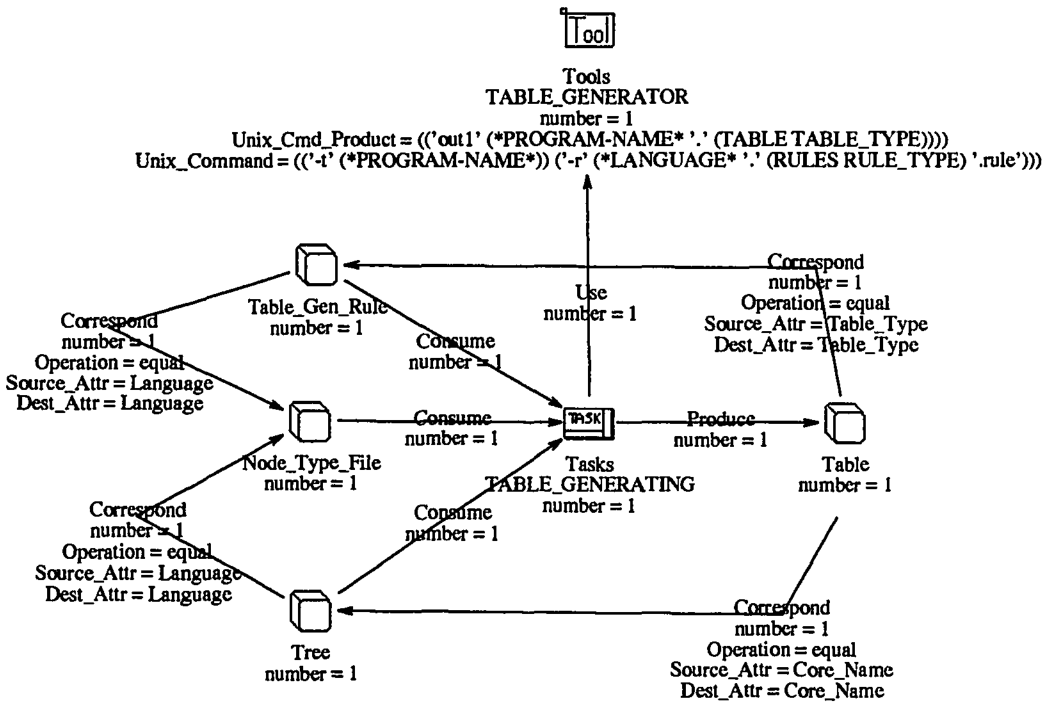


Figure 5.6: Generic Plan for the Table_Generating Task

Figure 5.7 shows the generic plan for the *Model_Generating* task. It requires three inputs – a syntax tree, a model generation rule file, and a node type definition file. It produces a file containing a particular type of graphs. The task is executed by a dedicated tool – *Model Generator.* The generated graph will have the same Core_Name as the tree file, indicated by a **Correspond** relation between them. A model generation rule file contains knowledge to generate a particular type of graph from a particular language. A node type file contains mappings of node types from integer representations that are used in syntax tree to mnemonic representations that are used in generation rules for a particular language. The node definition file and the model generation rule file to be used should deal with the same language as the syntax tree, indicated by two **Correspond** relations between the **Model_Gen_Rule**, **Node_Type_File**, and **Tree** entities. The model generation rule file to be used should contains the knowledge of generating the particular type of graphs as required, indicated by a **Correspond** relation between the **Model_Gen_Rule** and **Model** entities.

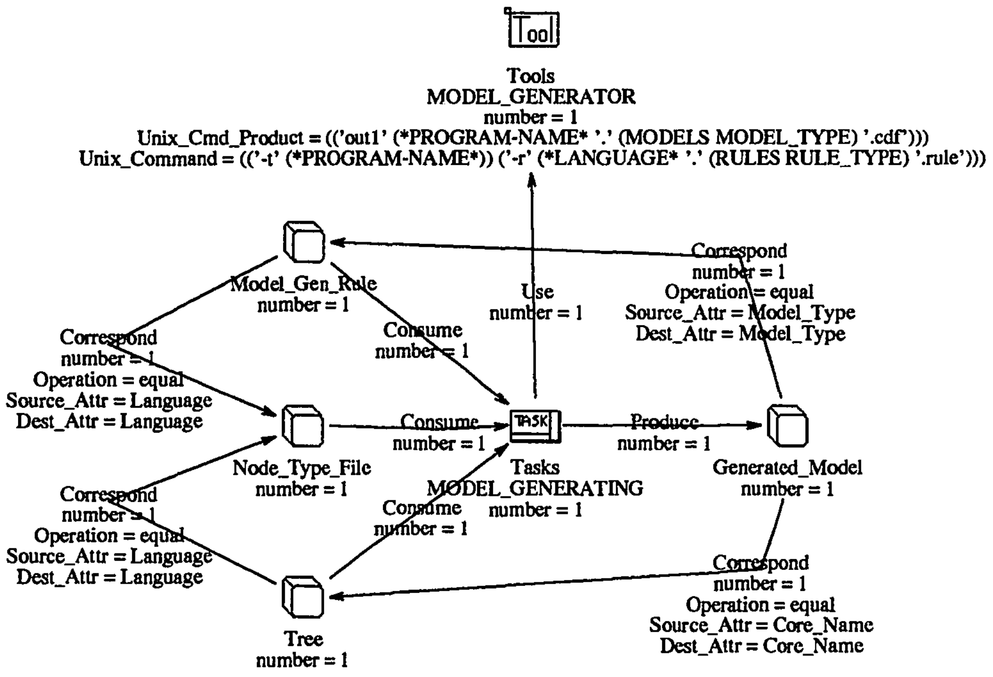


Figure 5.7: Generic Plan for the Model_Generating Task

Figure 5.8 shows the generic plan for the *CPM_Building* task. It requires six inputs – a data definition table, a data declaration table, a data reference table, a calling structure table, a program structure table, and a parameter indicating the type of graphs to be generated – CPM. It produces a file containing CPM graphs. The task is executed by a dedicated tool – *Model Builder*. All the tables should have the same Core_Name, indicated by four **Correspond** relations between entities representing the tables. The generated graph will have the same Core_Name as the table files, indicated by a **Correspond** relation between the **CPM_graph** and a **Table** entity.

Figure 5.9 shows the generic plan for the *DM_Building* task. It requires two inputs – a data declaration table and a parameter indicates type of graphs to be generated – DM. It produces a file contains DM graphs. The task is executed by a dedicated tool – *Model Builder*. The generated graph will have the same Core_Name as the table files, indicated by a **Correspond** relation between the **DM_graph** entity and the **Table** entity.

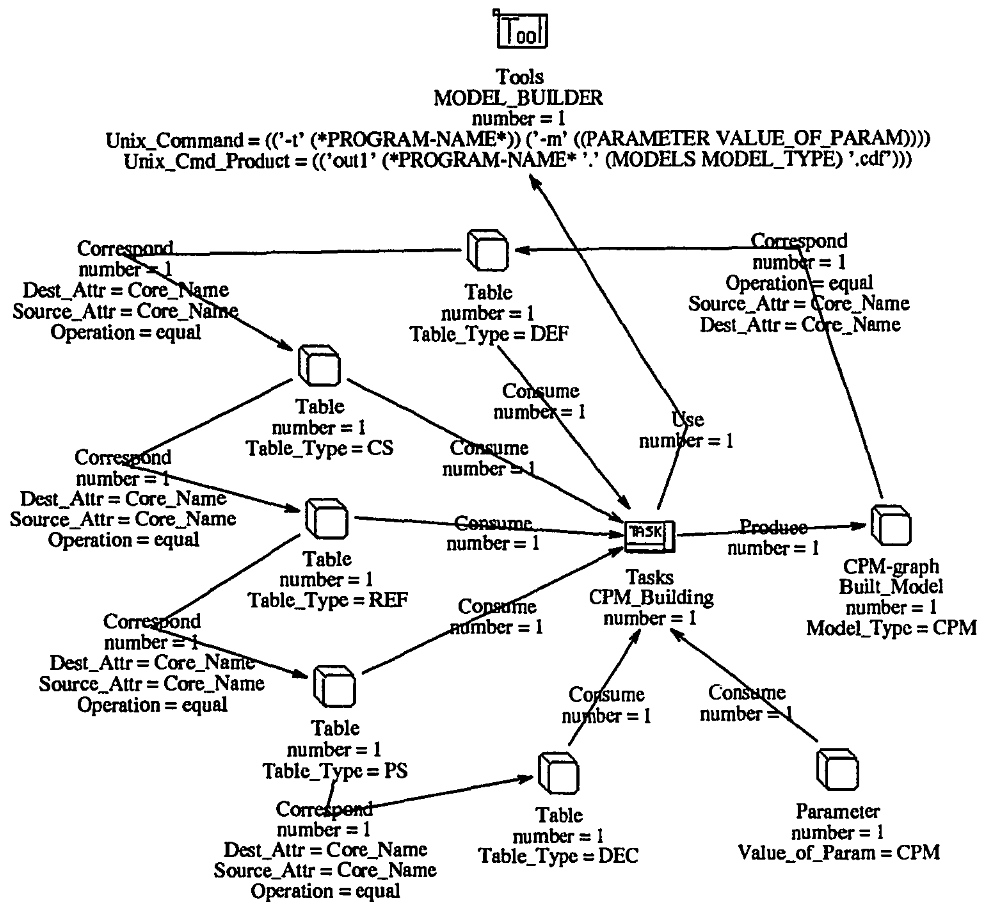


Figure 5.8: Generic Plan for the CPM Building Task

Figure 5.10 shows the generic plan for the *FM_Building* task. It requires six inputs – a data definition table, a data declaration table, a data reference table, a calling structure table, a program structure table, and a parameter indicating the type of graphs to be generated – FM. It produces a file containing FM graphs. The task is executed by a dedicated tool – *Model Builder.* All the tables should have the same Core_Name, indicated by four **Correspond** relations between entities representing the tables. The generated graph will have the same Core_Name as the table files, indicated by a **Correspond** relation between the **FM_graph** entity and a **Table** entity.

Figure 5.11 shows the generic plan for the *PSTM_Building* task. It requires three inputs – a calling structure table, a program structure table, and a parameter indicating the type of graphs to be generated – PSTM. It produces a file containing

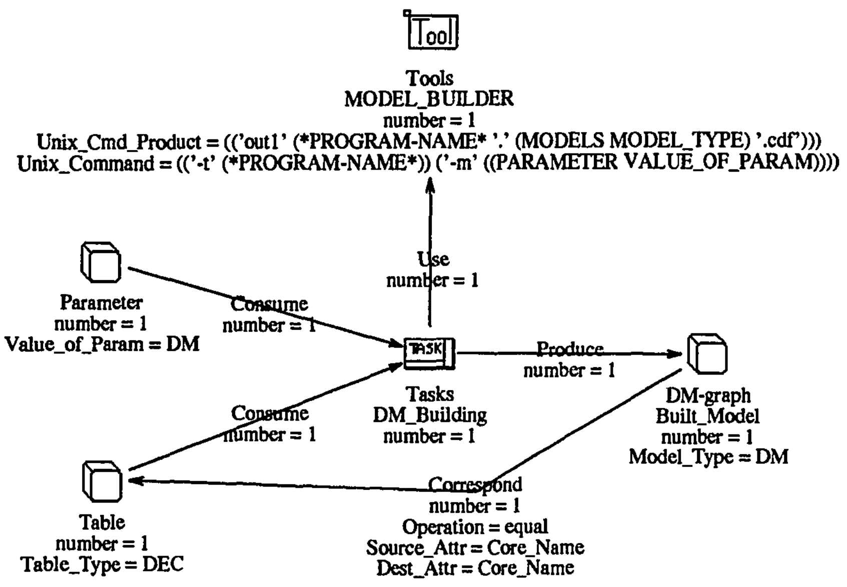


Figure 5.9: Generic Plan for the DM Building Task

PSTM graphs. The task is executed by a dedicated tool – *Model Builder*. All the tables should have the same Core_Name, indicated by a **Correspond** relation between entities representing the tables. The generated graph will have the same Core_Name as the table files, indicated by a **Correspond** relation between the **PSTM_graph** entity and a **Table** entity.

5.3 Process Flow

Basically, there are five steps to process a query that is issued by a client. The steps are: *Query Analysis*, *Planning*, *Model Generation and Retrieval*, *Model Inspection and Analysis*, and *Model Manipulation*.

The *Query Analysis* step analyzes the incoming query and detects the requirements of the query. This step is performed by the *Query Interpreter* component.

The *Planning* step checks the requirements of the query and the status of OB, then controls the processing of the rest of the steps. This step is performed by the *Planning* components.

The *Model Generation and Retrieval* step obtains the graphs that may satisfy the requirements of the query. These graphs are generated by reverse engineering tools

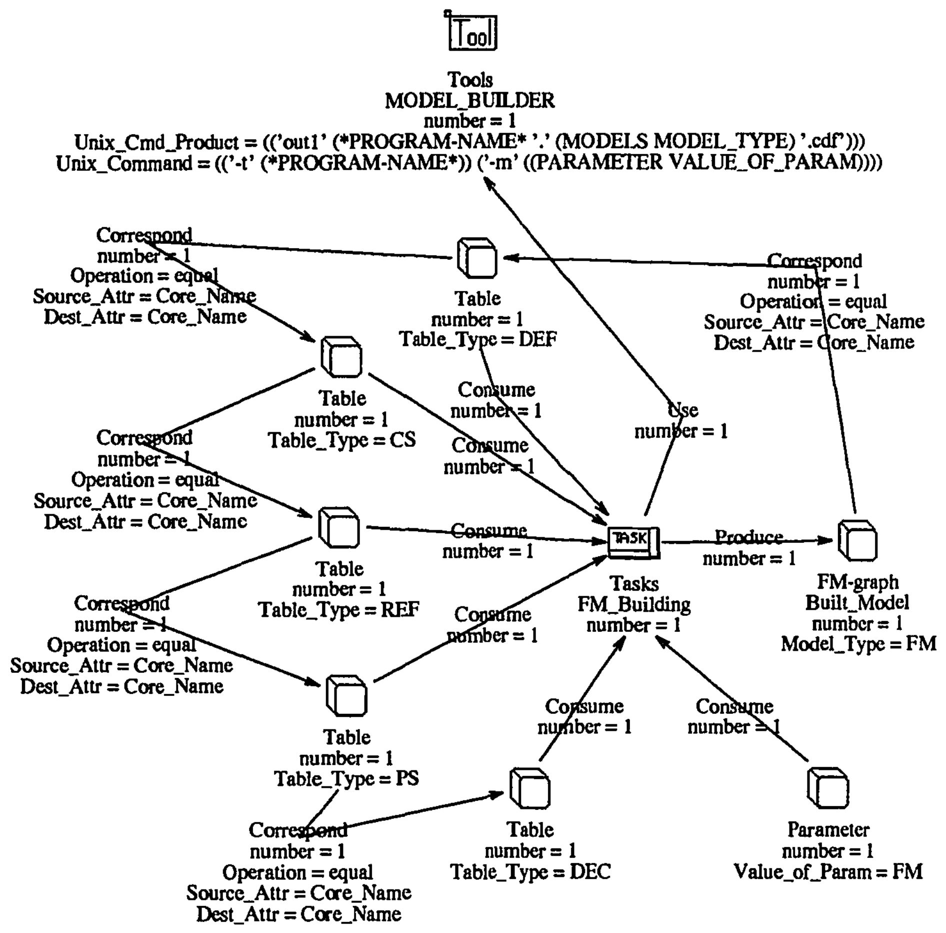


Figure 5.10: Generic Plan for the FM Building Task

if they do not already exist in the Object Base, or are retrieved from OB. This step is performed mainly by the *Information Base* component, the *OB Access Controller* component, the *Tool Controller* components, and the reverse engineering tools in Proud.

The *Model Inspection and Analysis* step checks the obtained graphs to see if they match the requirement or not. This step is performed by functions and tools in Proud.

The *Model Manipulation* step reformats the graphs that are the answer to the query before the graphs are returned to client. This step is performed by the *Result Integrating* component.

Among these processes, only the first two, *Query Analysis* and *Planning*, are

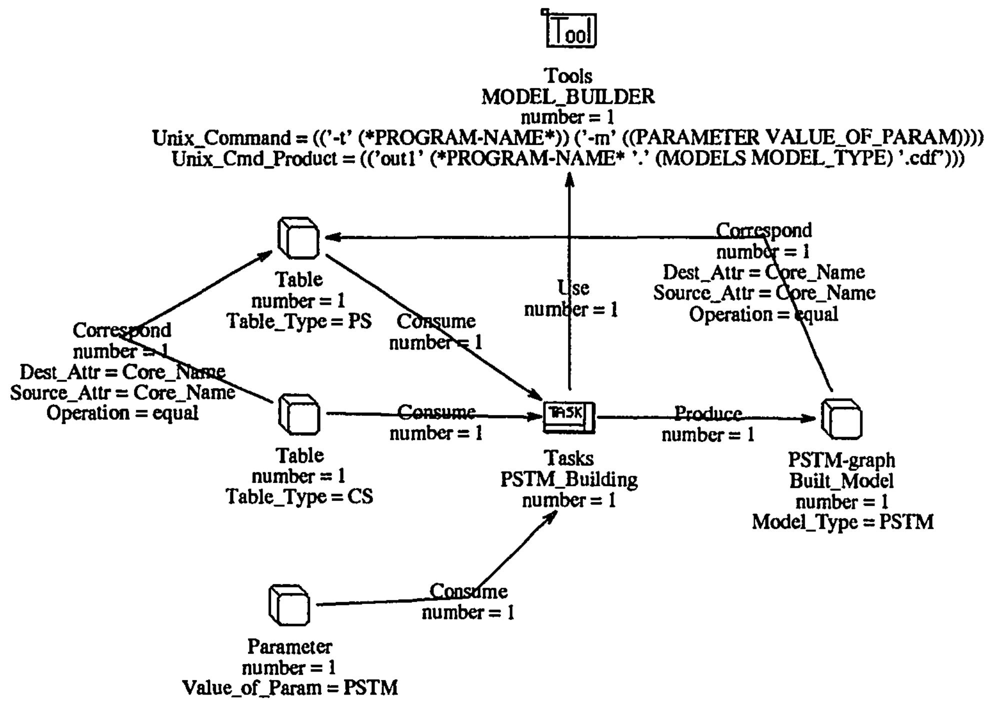


Figure 5.11: Generic Plan for the PSTM Building Task

performed for each incoming query. The other processes are performed depending on the requirements of the incoming query and the status of information in OB. Figure 5.12 shows the relationship of these processes.

5.3.1 Query Analysis

There are three sub-steps in the *Query Analysis* process – *check the correctness*, *check the requirements*, and *find graphs.*

Check the Correctness

The first step is to check the correctness of the incoming query. The check criteria are:

- Each incoming query should have one and only one query graph.
- In the query graph, there is one and only one **Query** entity.
- There is no relation whose destination is the **Query** entity.

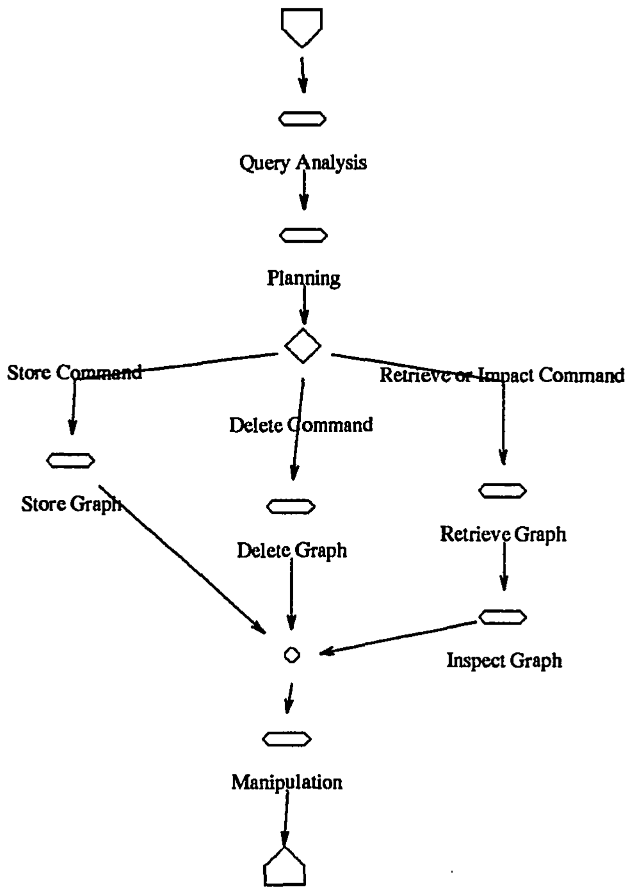


Figure 5.12: Relationship between Processes

- The query graph should be a connected graph (ignoring the direction of relations).
- The objects in the query graph should be defined in PUQ.
- If there is more than one relation in the query graph whose source is the **Query** entity, the graph should consists of several disconnected sub graphs and each of them is connected to the **Query** entity by one and only one relation.

Check the Requirements

The following requirements are checked in each incoming query:

- The operation required by the query. It is indicated by the name of the the **Query** entity.
- The META graph of the information required by the query. It is indicated either explicitly by the *meta_graph* attribute of the **Query** entity or implicitly by the types of the other objects in the query graph.
- The program in which the information required by the query can be obtained. It is indicated explicitly by the *program_name* attribute of the **Query** entity.
- The graphs in which the information required by the query may be obtained. It is indicated either explicitly by the *graph_name* attribute or implicitly by the names of the entities in the query.
- The union of multiple sets of requirements in the query. These requirements are formed by the **applies_to** relations from the **Query** entity to each of the set. These sets of requirements form an "or" condition for the query.
- The negation of the existence of entities or relations. Negation is indicated by a "0" value in the *number* attribute of the object.
- The path to be matched. This is indicated by a "*" or "+" value in the *number* attribute of the object.
- The constraint on an object. The constraint is indicated by a subgraph attached to the object and objects in the subgraph must be sub-types of the object in the class hierarchy.
- The binary relation between objects in the query. This is indicated by a **binary** relation in the query and the *operation* attribute of the relation indicates the type of constraint between the value of some attributes in the two objects (specified by the *source* and the *destination* attributes of the **binary** relation).
- The return criteria of the query. This is indicated by the view information of the objects in the query. If an object is "hidden," that means the objects in the result graph that correspond to that object should be removed.

These requirements are checked and the results are placed into a set of lists that will be used by the *Planning* process.

Find Graphs

The entities in the query graph are checked with a set of tables to find potential graphs that may be the answer to the query. Three tables are checked – data declaration table, program structure table, and calling structure table. These tables are generated from source code files by reverse engineering tools.

This process returns a list of graphs. Each graph consists of three pieces of information – the name of the graph, the name of the META graph of the graph, and the name of the program from which the graph is generated.

If there is no information about the META graph, there is a set of rules to determine the META graph: If the name of the entity matches one entry in the data declaration table, the default META graph name is "DM2". If the name of the entity matches one entry in the program structure table, the default META graph name is "PSTM". If the name of the entity matches one entry in the calling structure table, the default META graph name is "FM".

5.3.2 Planning

The *Planning* process takes the outputs of the Query Analysis process and guides the rest of the processes to obtain the required information. Among the information it gets from the Query Analysis process, the most important is the operation required by the query.

The *Planning* process uses two sets of knowledge – knowledge about the routines and the tools in Proud and knowledge of how to process each operation. Knowledge about routines and tools include the operations they perform, the input data required for them, and the outputs they produce. The knowledge is captured in Mera graphs. One graph represents knowledge of one function. The knowledge about how to process each operation is captured in scripts. One script represents the knowledge of one operation. The knowledge captured in a script describes the steps needed to process the operation and the information required for each step. Figure 5.13 shows the general view of the script of *Model Generation and Retrieval* process.

The first step in the *Planning* process is to check the scripts to find the right script for the current operation. Then, a plan of generating required information is created for each step described in the script. The mean-ends approach is used for plan generation.

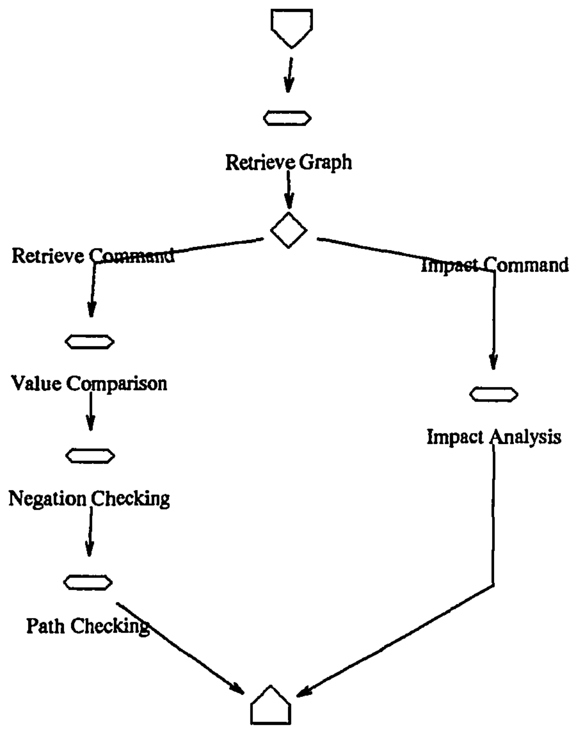


Figure 5.13: Model Generation and Retrieval Process

The next step is executing the plan. The *Planning* process monitors the execution. If one tool or routine fails, the *Planning* process tries to recreate a plan, or report the error.

5.3.3 Model Generation and Retrieval

The *Model Generation and Retrieval* process retrieves required graphs from OB if they are in OB. Otherwise, it generates the graphs from source code by using reverse engineering tools. The retrieval criteria are a list of graph id numbers that represent the graphs, or a set of patterns to be matched.

Check for Existence

The first step in this process is to check if the required graphs are in OB or not. This is done by the *Information Base.* When a graph is generated, it is registered in the

Information Base. So the *Information Base* contains the status of the graphs in the OB.

Model Generation

If the graphs are not in OB, the next step is to determine how to generate them. This is also done by the *Information Base.* By using knowledge about individual reverse engineering tools and the relationships between tools and data, the *Information Base* generates a plan that can guide the generation of the graphs.

Then the plan is carried out by the *Tool Controller* to generate the required graphs. After the graphs are generated, they are stored in OB and registered in the *Information Base.*

Model Retrieval

If the graphs are in OB, these graphs are retrieved. The retrieval process is different depending on the retrieval criteria.

Retrieval by Pattern Matching If the retrieval criteria are a set of patterns, the first step is to check the objects in the pattern to see if there is an object whose type is a superclass of some other types. If such an object is found, the pattern is duplicated into several copies, the object in each copy is replaced with an object whose type is one of the subtypes of the original object. For example, Figure 5.14 is the original retrieval pattern. The type of "EDIT" entity is **Transition**. **Transition** is a superclass of **Fork**, **Wait**, **Invoke**, and **Process**. The pattern is duplicated into 4 retrieval patterns, shown in Figure 5.15 to Figure 5.18.

The second step is to retrieve the graphs by using the OBGQ (Object Base Graphical Query) facility. Each pattern is sent to OB one by one, and all the returned results are collected.

Retrieval by Graph ID Numbers The OBQ (Object Base Query) facility is used for the retrieval process. First, the graph definition of the graph is retrieved, then the objects in the graph are retrieved. Finally, the view information about the graph is retrieved.

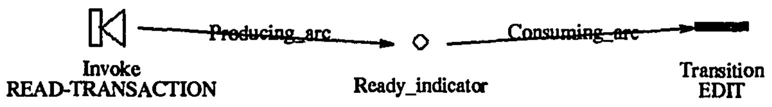


Figure 5.14: Original Retrieval Pattern

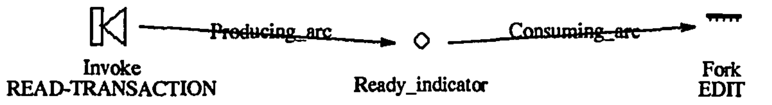


Figure 5.15: Retrieval Pattern with **Fork**

5.3.4 Model Inspection and Analysis

This process checks the graphs obtained from Object Base or generated by reverse engineering tools and filters out the irrelevant information according to the query.

There are two steps in this process. The first one is model inspection. Since the pattern match capability of OB is not powerful enough to process all kinds of patterns found in incoming queries, the graphs obtained from OB are the superset of the answer to the query. This step is to filter out the extra information. Before the inspection can be performed, the mappings between the pattern and the data graphs are established.

There are three types of inspection, in order of process:

Value Comparison checks the values of some attribute of objects connected by a **binary** relation. If the relation does not hold between two values, the inspection fails.

Negation looks for of objects that should not exist. The objects are indicated in the pattern by a "0" value in their *number* attributes. If such an object is found in a data graph, the graph is filtered out.

Path examines the paths in the data graph that matches the paths in the query graph. OB can process paths with simple constraints, so sometimes the re-

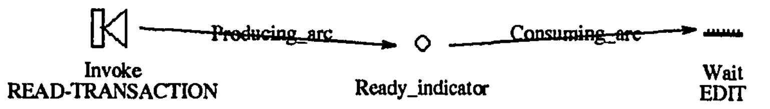


Figure 5.16: Retrieval Pattern with **Wait**

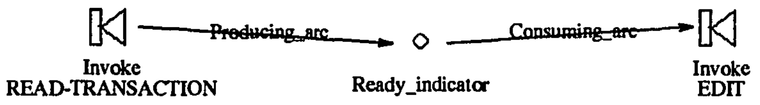


Figure 5.17: Retrieval Pattern with **Invoke**

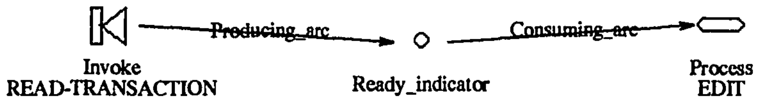


Figure 5.18: Retrieval Pattern with **Process**

trieval criteria must be relaxed for retrieving from OB. So extra paths may be retrieved from OB.

The second step in this process is model analysis. This process is done by analysis tools in Proud. The ability of analysis is dependent on the analysis tools. Two tools are proposed for Proud – impact analysis and similarity analysis.

5.3.5 Model Manipulation

The *Model Manipulation* process reformats and organizes the graphs so they can meet the retrieval criteria of the query. There are two types of manipulations in this process: set view information and remove unwanted objects. Figure 5.19 illustrates the process graphically.

Set View Information

If the operation of the query is *retrieve all* and no return criteria is given in the query, the objects in the graph that match the patterns for retrieving are indicates by highlighting them. If the operation is *retrieve partial*, the unmatched objects in the query are hidden. If the operation is *impact*, no set view information is altered since the impact analysis tool already sets the view information.

Remove Unwanted Objects

If the operation of the query is *retrieve partial* and some objects in the query graph are marked as hidden objects, the objects in the data graph that match the objects marked hidden in the query are removed from the graph.

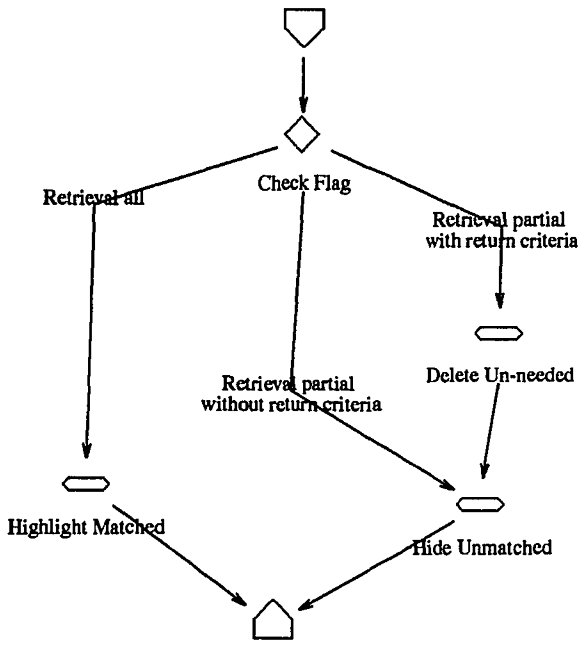

Check Flag
Retrieval all
Retrieval partial with return criteria
Retrieval partial without return criteria
Delete Un-needed
Highlight Matched
Hide Unmatched

Figure 5.19: Model Manipulation Process

5.4 Architecture

Planner consists of six components: *Query Interpreter*, *Planning*, *OB Access Controller*, *Tool Controller*, *Result Integrating*, and *Information Base.* Figure 5.20 shows the architecture of *Planner.*

Query Interpreter accepts incoming queries and indicates which data graphs may contain answers for the queries. *Planning* analyzes the queries and generates plans for answering the queries. *Planning* inquires *Information Base* about the status of the data graphs. OB returns TRUE if the graphs are in OB, or a plan containing a set of tasks for generating these graphs. The tasks are executed by *Tool Controller*, which converts the actions into proper Unix commands. The generated graphs are stored in OB and registered in *Information Base.* Then *Planning* converts the incoming queues into retrieval patterns and passes these patterns to *OB Access Controller. OB Access Controller* translates the retrieval patterns into OB queries and receives the answers from OB. Finally, the retrieved graphs are integrated by *Result Integrating* and returned as the answers to the incoming queries.

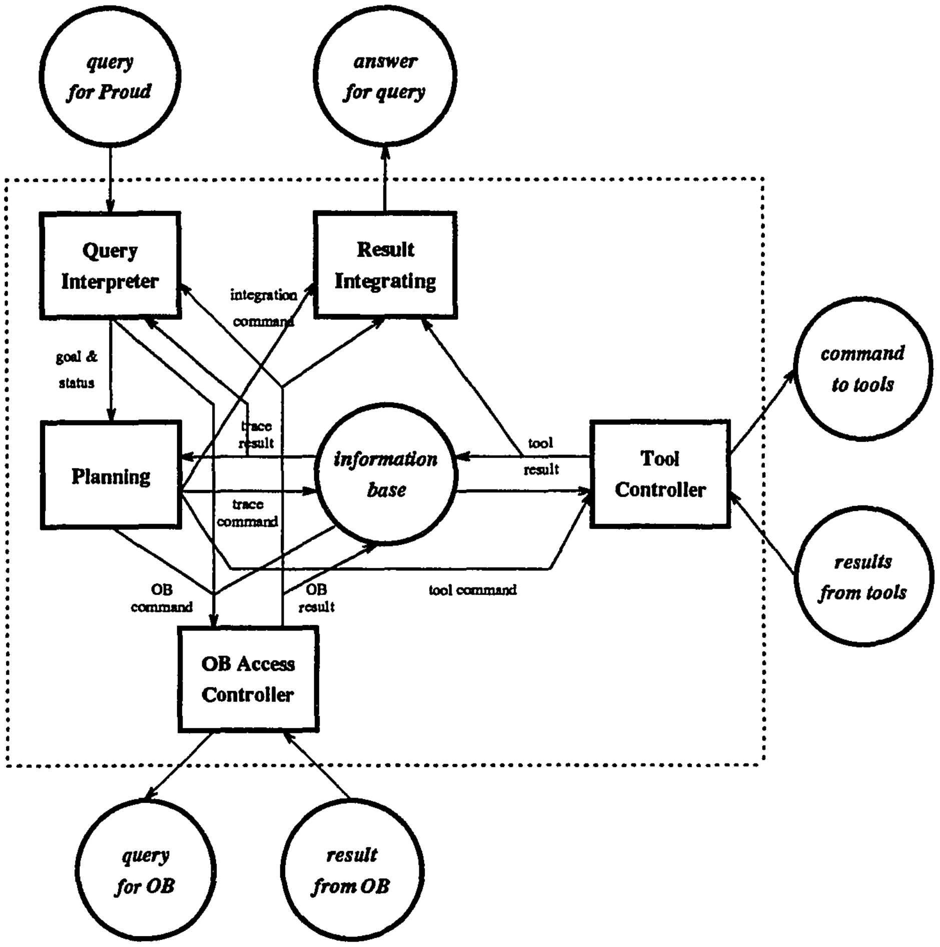


Figure 5.20: Architecture of *Planner*

5.4.1 Query Interpreter

Query Interpreter serves as the main interface between *Planner* and its users. Its main function is to check the incoming query for syntactical errors, and do some pre-processing to reduce the work of *Planning.* Figure 5.21 shows the structure of *Query Interpreter.*

Receptor

Receptor accepts queries from clients and converts them into internal structures. The queries are in two formats: CDF format or MERA CLOS format. If an incoming

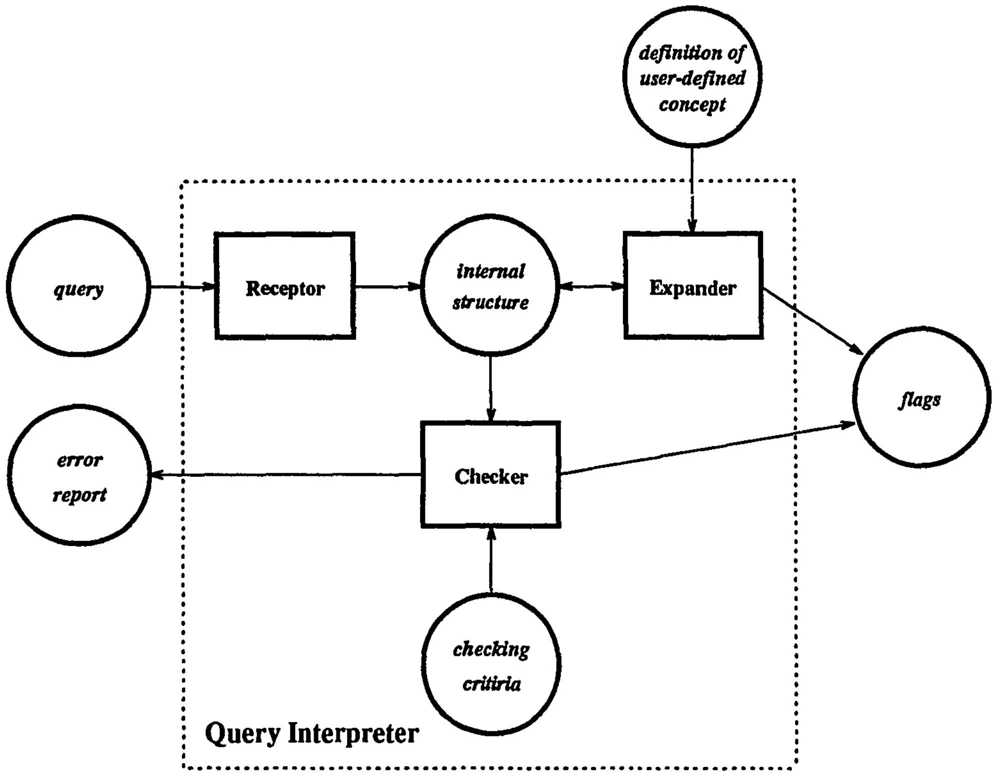


Figure 5.21: Structure of *Query Interpreter*

query is in MERA CLOS format, *Receptor* dose nothing, except store the query in internal MERA CLOS structures. If the query is in CDF format, *Receptor* will convert it into MERA CLOS format and store it.

Expander

To reduce the effort needed to create complicated queries by the user, Proud provides defined-concepts in its query language (PUQ). The user can define definitions of some concepts by using PUQ and store them in OB. Then, the defined-concepts can be used in queries. These concepts are replaced with their definitions in *Expander.*

There are three tasks for *Expander*:

- Retrieve the definitions of the concepts from OB.
- Replace the definitions into the original query and set the proper flags.

- Integrate the definitions with the original query by replacing the virtual parameters of the definitions with actual parameters.

Figure 5.22 shows a query that uses a defined-concept – **coupled-tasks**. The definition of the concept is shown in Figure 4.2. The expanded query is shown in Figure 5.23. In this graph, the **task** pointed by the **parameter** *A* in the definition graph is replaces by the **task** pointed by the **parameter** *A* in the query graph. The same replacement is also applied to the **parameter** *B*.

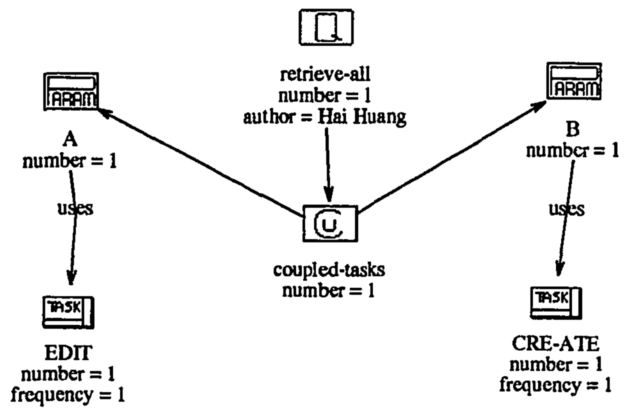


Figure 5.22: An Example of Query Graph Using Defined-concept

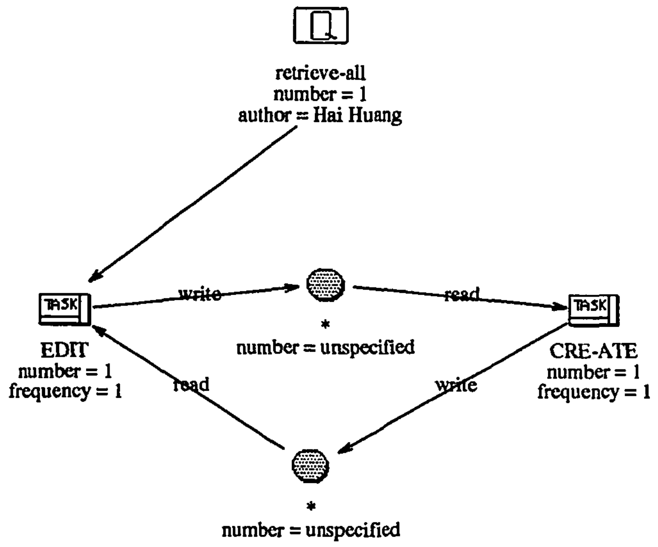


Figure 5.23: The Expanded Query Graph

Checker

Checker is the main component in *Query Interpreter.* It performs all three sub-steps in the *Query Analysis* process. That is:

- Check the correctness of the query.
- Check the requirements of the query.
- Find potential graphs that may contain the results for the query.

5.4.2 Planning

The main function of *Planning* is to generate a plan, which contains a series of actions. The execution of the plan will obtain the information requested in the incoming query. *Planning* uses *mean-ends analysis* [2] for plan generation. The requirements of the incoming query serve as the goals for planning. *Planning* generates a plan that will produce the required information from the existing information. The *Planning* process is also guided by the proper script.

The generated plan is represented as a sequence of tasks. Each task represents a command that may invoke a tool in Proud or a query to OB. The execution of the command is monitored by *Planning* to detect any failure. If a failure occurs, a re-planning process is invoked to find an alternative plan, if it is possible.

5.4.3 OB Access Controller

OB Access Controller is the interface between *Planner* and OB. It consists of two parts. Both of them are called *ob_controller.* One is written in C (called the C version in this dissertation) and the other is written in CLOS (called the LISP version in this dissertation). The C version does the actual work to communicate with OB, and the LISP version bridges *Planner*, which is written in CLOS, and the C version.

Inputs to the C version are a command that instructs the *ob_controller* what to do and a MERA graph in CDF format. Depending on the action to be taken, the graph represents the query graph, deletion graph, or storage graph.

The format to invoke the C version *ob_controller* is:

```
ob_controller [-h host] [-b database] [-I path] -c command [-o output] [input]
```

The definitions of the options are:

host is the host server on which the OB is running. If this option is omitted, the default will be the value of the environment variable **OB_HOST**, or the local host if the environment variable is not presented.

database is the name of the database to be used. If this option is omitted, the id of the current user will be used by default.

path is the search paths for files included in the current file by the `#include` clause in the current file. If this option is omitted, only the current directory will be searched for included files.

command is the action to be performed by the *ob_controller*. It is required. There are six valid actions:

- **store-all** is to store all graphs, including those in the included files, into OB.
- **store-partial** is to store only the graphs in the input file. The graphs in the included files will not be stored.
- **delete-all** is to delete all graphs, including those in the included files, from OB.
- **delete-partial** is to delete graphs in the input file. The graphs in the included files will not be deleted.
- **retrieve-all** is to retrieve the graphs that match the query without modification.
- **retrieve-partial** is to retrieve the graphs that match the query, and add display information to indicate which parts of the graphs actually match the query.

output represents the file name to store the output of *ob_controller*. If this options is omitted, the result of the execution will be sent to the standard output device.

input is the input file name. If this is omitted, *ob_controller* takes its input from the standard input device.

For *store-all* and *store-partial*, the input graphs are the graphs to be stored in OB. If the graphs exist in OB, the previous graphs will be deleted and the new graphs will be stored.

For *delete-all* and *delete-partial*, the input graphs indicate the graphs to be deleted from OB. Only the D-GRAPH or D-DIAGRAM clauses in the input file have meaning. All other clauses are ignored. The return value is the graph names that are deleted from OB.

For *retrieve-all* and *retrieve-partial*, the input graphs represent the retrieval conditions. The search scope is limited by the META-graph-IDs of the input graphs. The return value are the graphs that match the retrieval conditions.

The LISP version converts the query graph from MERA CLOS into CDF format, writes it into a temporary file, and invokes the C version of *ob_controller*. Then it collects the output of the C version of *ob_controller* and passes it to the other components in *Planner*.

5.4.4 Tool Controller

Tool Controller is the interface between *Planner* and other tools that run in the Unix environment and it performs the model generation process. It invokes tools by using proper Unix commands and stores temporary results generated by the tools for the next command.

There is a temporary directory in the Unix file system to keep all the source code files of the target system, as well as the temporary results generated by tools. All the input and output data for tools that are under control of Proud should be store in that directory.

Tool Controller uses *Information Base* to obtains the proper Unix command in order to invoke a tool, as well as information about the product of the tool. This information includes the file name of the product and the type of the product in the data class hierarchy of *Information Base*. If the product is directed to standard output, a temporary file will be created to hold the output, and it will be used as standard input for the next command.

Tool Controller also monitors the performance of the tools. When one execution of a tool finishes successfully, *Tool Controller* registers the products into *Information Base* to update the status. If the execution fails for some reason, *Tool Controller* stops execution of the remaining commands, and notifies the *Planning* component about the failure.

Tool Controller is implemented in CLOS and uses some special features (foreign function feature) of Allegro Common LISP. These features may not be portable to other common LISP environments.

5.4.5 Result Integrating

Result Integrating performs the *Model Manipulation* process. It reorganizes and reformats the results before they are returned. Since the results are MERA graphs, all functions provided by *Result Integrating* deal with MERA graphs.

Merge Merge several graphs into one graph. There are two ways to merge graphs. One is to create a new graph that contains all objects from the graphs to be merged. The other way is to created a new graph that contains several entities, each of which represents a graph to be merged. The first way, the merged graph serves as one piece of information that answers incoming query. The second way, the merged graph indicates several pieces of information that are related to answering the incoming query.

Format Add display data to the graph to be returned to indicate the important parts of the graph. *Planner* always return an entire graph, not a portion of a graph. In some cases, only part of the graph is relevant to the query. So the irrelevant information in the graph should be hidden. In other cases, some parts of the graph are more relevant to the query than the other parts, these parts should be highlighted.

5.4.6 Information Base

Information Base serves as a knowledge base for *Planner. Information Base* is implemented in MERA CLOS, which is an extension of standard CLOS.

Features

Two major features of *Information Base* are:

- to maintain objects in *Information Base* that represent the status of information regarding the target system in OB and other places.
- to generate information generation plans by using tool knowledge in *Information Base.*

A graph in *Information Base* is defined for each class in the data hierarchy of *Information Base.* Every graph has a unique name. The naming schema is: `graph.<class>-base`. For example, the name of the "IOPM" graph is "graph.iopm-base". A model graph, such as an IOPM model graph, a table, such as a data declaration table, a syntax tree file, or a source code file is represented as an object in *Information Base* and stored in the graph that represents its class. Objects that belong to the same class are captured in one graph in *Information Base.* The following functions are used to maintain the objects in *Information Base.*

Registering Objects Every object has to check in with *Information Base* to register its existence. There are two ways to check in an object, check in with existence checking or without existence checking. For check in without existence checking, the user must take great care, otherwise it might result in duplicate objects in the same graph.

Check Existence To identify an object in *Information Base*, use the scheme : ¡graph name¿.¡object name¿ to identify the object. The *graph name* here is the class name that the data belongs to and the *object name* is actually the name of the data graph. For example, to check whether a IOPM graph named "EDIT" is registered in *Information Base* or not, the identification used for checking is IOPM.EDIT. Check existence will search for the object in the corresponding graph and notify the user of the results.

Delete Objects After identifying the existence of the object, call standard CLOS functions to remove the object from the graph.

Information Base is able to generate plans for RMA or AFM model generation. There are three major functions to perform this task.

Plan Proposal This function produces a proper plan to generate the desired information. It uses the basic AI-planning approach to achieve this task: It checks the existence of the required information. If it does not exist, it tries to find a task that may generate it, and checks the required inputs of the tasks, until a set of tools are found that can produce the required information from the existing information, or discovers that there are not tasks available to produce such information. The detailed algorithm will be presented in the following sections.

Unix Command Generation This function is used to generate the actual Unix command for a task that can be run under the Unix operating system. After a request to Plan Proposal, a list of task objects will be returned. Each task object represents a command to a particular tool. In order to invoke the tool properly, a Unix command must be formed according to the tool, since each tool has its own command format. This function uses the special *Unix_Command* slot in the tools class and relationships between tools and objects to form the actual Unix command that is used to invoke the tool. The language used to describe the information in the *Unix_Command* slot is presented in the following section.

Product Information Generating This function takes an approach similar to Unix command generation to get the product name for a task. It produces the products name by using the *Unix_Cmd_Product* attribute in tools class. This attribute specifies how to form the file name of the product of the tool. The same language used in *Unix_Command* slot is used here to form the product name.

Planning Algorithm

Planning in *Information Base* is to create a plan for basic model generation. The reasons to do this planning in *Information Base* are:

1. the information required to generate this type of plan is available in *Information Base.* If planning is done in another component, a high volume of communication may be generated and this affects the performance of the system, and
2. the planning algorithm for this type of plan is simple. The basic problem solving algorithm is powerful enough to create the desired plan; no advanced techniques in planning are needed.

So, this approach will reduce the complexity and increase the performance of the entire system.

Dependency analysis is a main issue in plan proposal. In order to generate some objects, it is necessary to use some reverse engineering tools and the tools may require some other objects, which are called the consumed objects. So the target object depends on the consumed objects. If any of the consumed objects does not exist, a generation plan is needed for that consumed object. The dependency checker will work on this consumed object and so on. The fundamental algorithm for plan proposal follows.

First, read the target object name and type, check if the target object already exists or not. If the object exists in the system, inform requester that the object already exists and there is no need to create a generation plan. Otherwise, look for tasks that produce this object. Traverse the source objects of all consume relation and see whether these objects exist or not. Every source object will in turn become a target object read recursively by this algorithm. This algorithm repeats until all related objects are inspected. The task found for producing the target object will be placed in a result list. Finally the result list containing task objects will be returned. Order in the list is crucial, and they have to be executed in sequence.

The following is the pseudo code of the planning algorithm.

```
plan-proposal(object-name, object-type)
{
     if (check-existence(object-name,object-type))
          inform user object already exists, no need for a generation plan;
     else
          return (plan-gen (object-name,object-type,null));
}

plan-gen(object-name,object-type,plan-list)
{
     not-found = true;
     while (not-found)
     {
          inspect not examined generic plans;
          if ((class of sink object of produce relation)==object-type)
          {
               task = task-object of the generic plan;
               not-found = false;
          }
     }
     push task to plan-list;
     consume-list = list of all source objects of consume relations;
     for (every item in the consume-list)
     {
          Check-name= object-name of the item;
          check-type = object-type of the item;
          if (check-existence(check-name,check-type) == false)
              plan-gen(check-name,check-type,plan-list);
     }
     return(plan-list);
}
```

Command Expression Language

Command Expression Language is used to specify the format of the Unix command of a tool. The grammar the language is:

```
command-expression : ( argument+ )
argument           : ( OPTION ( option-value* ) )
                   | ( "argl" option-value )
                   | ( STRING option-value )
                   | ( "outl" option-value )
option-value       : ( slot+ )
                   | VARIABLE-NAME
                   | STRING
slot               : ( CLASS SLOT-NAME )
```

OPTION is the option for a command. It uses the Unix convention that each option should start with "-". Each tool may have its own options list.

"argl" is the argument passed in for command generation.

"outl" is the output format.

VARIABLE-NAME represents a variable in Lisp.

STRING represents a string that is delimited by single quotes.

CLASS represents an object in the plan whose class matches CLASS.

SLOT_NAME represents a slot in a class.

For example, the Tools entity representing *Model Generator* has the following attributes:

name `model_generator`

Unix_Command

```
    ('-t' (*PROGRAM-NAME*)) ('-r' (*LANGUAGE* '.' (RULES RULE_TYPE)
    '.rule'))
```

Unix_Cmd_Product `(('out1' (*PROGRAM-NAME* '.' (MODELS MODEL_TYPE) '.cdf')))`

The variables `*PROGRAM-NAME*` and `*LANGUAGE*` are used to generate the values of the argument `-t` and value of the argument `-r`. The value of the slot named *RULE_TYPE* in the object *Model_Gen_Rule* whose class is *RULES* is also used to generate the value of the argument `-r`. If the value of the `*PROGRAM-NAME*` is "sample ", the value of the `*LANGUAGE*` is "cobol", the value of the *MODEL_TYPE* is "IOPM ", and the value of the *RULE_TYPE* is "iopm", the generated command is:

```
model_generator -t sample -r cobol.iopm.rule
```

The name of the generated product is:

```
sample.IOPM.cdf
```

Chapter 6

Reverse Engineering Tool Set

6.1 Overview

Proud provides maintainers with a set of reverse engineering tools and enables them to capture the static and dynamic behavior of a target system. The objective of the design for the reverse engineering tool set is to provide a flexible, adaptable, yet powerful tool set to generate the desired information from the source code. To achieve this objective, a rule-based approach is adopted. A set of generation rules used by the tools allows the user to extract different information from source code written in different programming languages and represents the extracted information in a format suitable to their requirements [25]. The configuration of the tools is shown in Figure 6.1.

There are five tools for automatic model generation in *Proud*: *Parser*, *Table Generator*, *Model Generator*, *Model Builder*, and *Linkage Generator*. Among the these tools, three of them are rule-based tools: *Model Generator*, *Table Generator*, and *Linkage Generator*. *Parser* is the only programming language dependent tool. For every programming language to be processed, a corresponding parser is needed. *Model Generator* and *Model Builder* are the main tools for model generation. Currently, *Model Generator* produces IOPM, EOPM, FSM, DM2, and UIM; and *Model Builder* produces CPM, FM, DM, and PSTM. *Table Generator* creates various tables from a syntax tree of a program and used by *Model Builder* and *Linkage Generator*. *Linkage Generator* creates various relationships between different diagrams in the same model or different models.

6.2 Reverse Engineering Process

Each program contains several different kinds of information. Roughly, the information can be classified into three groups – problem domain information that represents

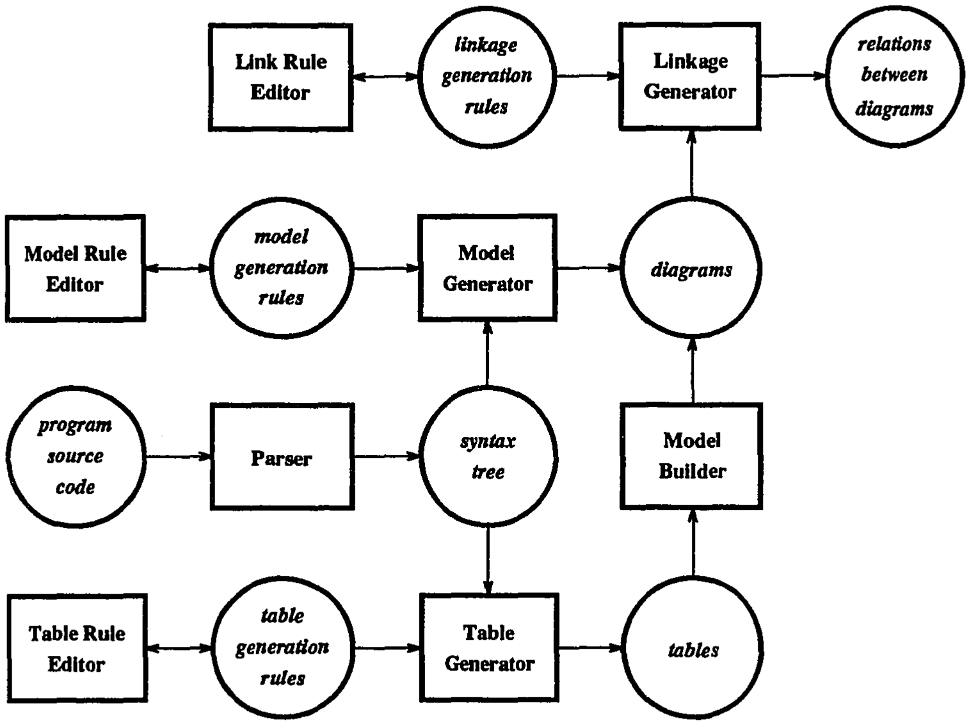


Figure 6.1: Configuration of Reverse Engineering Tool Set

the information about the problems to be handled by the system, design information that represents the design decisions made on the problem domain information, and programming language information about the design decisions.

In forward engineering, the problem domain information is obtained via software requirement analysis and represented as system requirement specification *Pspec*. Then this information is transferred into design information via preliminary and detailed design process. The design information is represented as different design documents *Ddoc*. Finally, the design information is translated into programming language information via the implementation process. The programming language information is represented as the source code of the program *Scode*. These processes can be defined as follows:

$$Ddoc_i = Dmethod_i(Pspec, Ddecision_i) \tag{6.1}$$

$$Scode_j = coding_j(Ddoc_1 \ldots Ddoc_k) \tag{6.2}$$

In formula 6.1, $Ddoc_i$ represents one design document, *Dmethod* is the design method, *Pspec* is the problem domain information, and $Ddecision_i$ is the design decision taken in the process. In formula 6.2, $Scode$ represents the source code of the program. $Ddoc_1 \ldots Ddoc_k$ represent the design documents, and $coding_j$ represents the coding in programming language L_j.

In reverse engineering, the design information and problem domain information are extracted and abstracted from source code. In the Proud, only design information is extracted from source code. The following discussion focuses on processes for design information extraction.

The process of extracting a kind of design information, say $Ddoc_i$, from source code $Scode_j$ written in programming language L_j can be described as follows:

$$Ddoc_i = Dextract_i(Scode_j, L_j) \tag{6.3}$$

Or viewed as a set operation:

$$Ddoc_i = (Scode_j - Lconcept_j) \wedge Dconcept_i \tag{6.4}$$

In formula 6.3, $Dextract$ is the extraction process. In formula 6.4 $Lconcept_j$ represents the language related information with the programming language L_j and $Dconcept_i$ represents the concepts related to design method $Dmethod_i$.

Because of the tightly coupled design information and programming language information, it is very difficult to define $Dextract_i$, which can handle multiple programming languages.

One approach is to find an extracting method $Dextract_{ij}$ that extracts $Ddoc_i$ from $Scode_j$ written in language L_j. This is a relatively easy approach. But $Dextract_{ij}$ has to be defined for one kind of design information and one programming language. For m kinds of design information and n different languages, total $m \times n$ kinds of extracting must be defined.

Another approach is to split the extracting process into two steps. The program is first converted into another source code written in a "universal language". The converted program should have the same functionalities and behavior as the original one. Then the design information is extracted from the converted source code. The process is shown as follows:

$$Scode_u = convert_j(Scode_j) \tag{6.5}$$

$$Ddoc_i = Dextract_u(Scode_u) \tag{6.6}$$

The $convert_x$ in formula 6.5 is the converting process that translates source code written in language x into source code written in the "universal language". There are two problems associated with this approach. One is related with the "universal language" L_u. It is very difficult to define such a language since different programming languages have their own special features, and sometimes some features in one language conflict with some features in other languages. Another problem is related to defining the converting algorithm even if L_u is defined.

To avoid these problems, one solution is to shift some of the work in step 1 into step2. By doing this, the requirement of a universal language can be relaxed to some degree and a common representation form and the corresponding converting algorithm can be defined. Proud uses this approach in its extracting process.

At first, the information about the syntax of a programming language L_j is removed from the source code and the rest of the program is kept in an abstract syntax tree. The process is shown as:

$$Tree_j = parsing_j(Scode_j) \tag{6.7}$$

In formula 6.7, $Tree_x$ is a syntax tree generated from source code and $parsing_y$ is the parsing process that generates the syntax tree from the source code written in language y. After removing the syntactic information from the source code, the extraction processes are very similar for different programming languages. This enables removing knowledge of the programming language from the extraction process and makes the process able to extract $Ddoc_i$ from source code written in different languages, as shown in formula 6.8:

$$Ddoc_i = Dextract_u(Tree_j, Lknowledge_j) \tag{6.8}$$

In formula 6.8, $Lknowledge_x$ is the syntactic knowledge of the language x.

Further study shows that the extraction processes for some design information are similar, in some contexts. This suggests another idea to further simplify the extraction processes by moving the knowledge of extracting a particular design information from the extraction process and keeping it separate, as shown in formula 6.9

$$Ddoc_i = Dextract(Tree_j, Eknowledge_i, Lknowledge_j) \tag{6.9}$$

In formula 6.9, $Eknowledge_x$ is the knowledge of extracting design document $Ddoc_x$.

A rule-based approach is developed based on this idea and used in Proud. A set of rules captures the knowledge of a programming language and the knowledge of extracting a kind of design information. The process can be viewed as follows:

$$Ddoc_i = Dextract(Tree_j, Mrule_{ij}) \tag{6.10}$$

$Mrule_{xy}$ in formula 6.10 captures the knowledge of extracting $Ddocx$ from $Tree_y$.

Due to limits in the expressive power of the rule description language used in the Proud (Section 6.3 discusses this in detail), not all kinds of design information can be extracted by using this method. Another method is defined to extract some of the design information that cannot be extracted by the rule-based approach.

In the second method, a set of tables is extracted from a syntax tree. The tables contain no information about programming language, so they are programming language independent. The rule-based approach is used in the table extraction process. The process can be viewed as follows:

$$Table_i = t - extract(Tree_j, Trule_{ij}) \tag{6.11}$$

In formula 6.11, $Trule_{ij}$ represents the knowledge needed to extract $Table_i$ from source code in programming language L_j. Then the design information is extracted from these tables, see formula 6.12

$$Ddoc_k = Dextract_k(Table_1 \ldots Table_m) \tag{6.12}$$

The following formulas represent the extraction process in terms of set operations:

$$Tree_j = Scode_j - Lsyntax_j \tag{6.13}$$

$$Ddoc_i = (Tree_j - Lsemantics_j) \wedge Eknowledge_i \tag{6.14}$$

$$Table_l = (Tree_j - Lsemantics_j) \wedge Tknowledge_l \tag{6.15}$$

$$Ddoc_m = (\bigvee_{i=1}^{n} Table_i) \wedge Eknowledge_m \tag{6.16}$$

In these formulas, $Lsyntax_j$ and $Lsemantics_j$ represent information regarding the syntax and semantics of a programming language L_j, respectively. $Eknowledge_i$ represents the information regarding the concepts of design method $Dmethod_i$. $Tknowledge_l$ represents the information for extracting $Table_l$.

The following discussion focuses on one aspect of a program to show this approach. A *MOVE* statement assigns a value to a variable. The value can be obtained from either a constant or another variable. So a *MOVE* statement creates a data flow. According to the grammar of ANSI COBOL 85, the following definitions are given for assigning a value to a data item or referring to a data item.

$$\forall s \forall d(assign_{move}(s,d)) \Longleftrightarrow (statement(move,s) \wedge variable(d) \wedge after(TO,s,d,1)) \qquad (6.17)$$

$$\forall s \forall d(refer_{move}(s,d)) \Longleftrightarrow (statement(move,s) \wedge (variable(d) \vee constant(d)) \wedge after(MOVE,s,d,1)) \qquad (6.18)$$

In the above definitions, $assign_{move}(x,y)$ means that the *MOVE* statement x assigns a value to data item y, $refer_{move}(x,y)$ means that the *MOVE* statement x uses the value of data item y, $statement(x,y)$ means that y is a x type statement, $variable(x)$ means that x is a variable, $constant(x)$ represents that x is a constant, and the $after(i,j,k,l)$ function means that k is the lth identifier after i in statement j.

These definitions are language dependent since they use the grammar of the *move* statement, the position of the identifier in the statement, etc. The parsing process removes some syntactic information by transforming the source code into a syntax tree. The following definitions define the source and destination in the syntax tree for a MOVE statement:

$$\forall s \forall d(source_{move}(s,d)) \Longleftrightarrow (node(move,s) \wedge \exists t(child(s,t) \wedge node(source,t) \wedge child(t,d)) \wedge (node(variable,d) \vee node(constant,d))) \qquad (6.19)$$

$$\forall s \forall d(destination_{move}(s,d) \Longleftrightarrow (node(move,s) \wedge \exists t(child(s,t) \wedge node(to,t) \wedge child(t,d)) \wedge node(variable,d)) \qquad (6.20)$$

In the above definitions, $source_{move}(x,y)$ means that y is the source of MOVE node x, $destination(x,y)$ means that y is the destination of MOVE node x, $node(x,y)$ means that y is a type x node, $child(x,y)$ means that y is a child node of x.

The transformation from an assign in a MOVE statement to a source in a syntax tree is defined as follows:

$$source_{move}(s,d) \equiv \exists x \exists y(assign_{move}(x,y) \wedge represent(s,x) \wedge represent(d,y)) \qquad (6.21)$$

In the above definition, *represent*(a, b) means that a is the node in syntax tree that represents b and b is an element in the source code .

The language information is removed entirely from the data definition table and data reference table. The definition of *refer* in data definition table is defined as follows:

$$\forall s \forall d(refer(s, d)) \Longleftrightarrow \exists t \exists x(table(t, reference) \wedge entry(t, x) \wedge field(s, x, statement) \wedge field(d, x, dest - data)) \tag{6.22}$$

In the above definition, $refer(x, y) means that statement x refers to data item y$,table(x, y) means that y is a x type table, $entry(x, y) means that y is an entry in table x$,field(x, y, z) means that in the field z of the entry y has the value x.

The transformation from a $source_{move}$ to *refer* in a table is defined as follows:

$$refer(s, d) \equiv \exists x \exists y(source_{move}(x, y) \wedge same(lineno, s, x) \wedge same(name, d, y)) \tag{6.23}$$

The function $same(a, b, c)$ means nodes b and c have the same value for attribute a. Now the construction of data flow can be defined:

$$dataflow_{refer}(m, d) \equiv module(m) \wedge data\_item(d) \wedge \exists s(s \in m \wedge \exists x \exists y(refer(x, y) \wedge same(lineno, x, s) \wedge same(name, d, y))) \tag{6.24}$$

$module(x)$ means that x is a module in a program and $data\_item(x)$ means that x is a data item define in the program.

6.3 Generation Rules

Generation rules play a very important role in achieving the design objectives of the tool set – flexible, adaptable, and powerful. There are three different types of generation rules – *model generation rules*, *table generation rules*, and *linkage generation rules.* Both *model generation rules* and *table generation rules* are used to extract information from the syntax tree and they are very similar in structure and other definitions. On the other hand, *linkage generation rules* are quite different from the other two types of rules. The discussion in this section mainly applies to the first two types of rules, but the basic concepts also apply to *linkage generation rules.*

6.3.1 Design Objective

The basic objective of the rule-based approach is to separate the knowledge of a certain task from the implementation of the application and capture them into a set of rules. By using different knowledge, the application can be behave differently without any modification of the application itself. So the application tends to be more flexible and adaptable.

There are three factors affecting the design of a rule description language – the expressive power, the readability, and the correctness checking ability. It is really difficulty to maximize all three. A balance point must be found to maximize the overall value of the language. In Proud, the balance point for the generation rule description language is the programming language concepts. Each rule is powerful enough to specify the actions to handle one, but only one language concept. Any sub-concepts contained in the concept should be processed by other rules. The rules are organized in a hierarchy so the rules dealing with lower level language concepts, such as "identification", are invoked by the rules dealing with the higher level concepts.

One reason to select this balance point is for correctness checking. There are few formal verification and validation methods available to do the correctness checking so far. The only effective method is the *walk through* method: manually read the rules and to see if they do the right thing or not. By selecting the balance point on language concept, each rule describes the process for exactly one programming language concept. This will increase the readability and the correctness checking ability. But the expressive power is reduced and some very complicated graphs cannot be generated by the rule-based approach.

6.3.2 Structure of Generation Rules

Basically, a rule consists of two major components: conditions to be satisfied and tasks to be performed after the conditions are satisfied. The former ones are called **triggers** and the later ones are called **actions** in this document.

One rule may contains more than one trigger, and each trigger may have more than one condition to be satisfied. The actions in a rule will be executed if and only if one of the triggers in the rule is triggered. One trigger is triggered if and only if all conditions in the trigger are satisfied. This is called firing a rule.

Each action in a rule does a specific task. There are three types of actions: the first one does the main tasks designed for the tool; the second one controls the order of other actions to be executed; and the last one does auxiliary work to help the first or second type of actions.

6.3.3 Definitions Used in Generation Rules

There are three definitions used in generation rules – *syntax tree node*, *path description*, and *expression.* These definitions are important to understand to write the generation rules. These definitions are associated with the structure of a syntax tree and all of them are used to construct the triggers and some types of actions.

Syntax Tree Node

Each node in a syntax tree contains the following information called the *node content*:

NODE-ID is the identifier of the node. Each node in a syntax tree has a unique identification number. Every node contains this information.

NODE-TYPE indicates the language concept represented by the node, such as iteration, branch, etc. Nodes that have the same type will have similar types of children in a similar order. Every node contains this information.

NODE-NAME represents the language concept the node represented in more detail. Since one language concept may have several variants, the variant is indicated here. Every node contains this information.

NODE-VALUE represents any value that the node may have. For example, the value of an integer node represents the value of the integer. Some nodes do not contain this information.

MODE-ELEMENT represents the portion of a program represented by the subtree whose root is the current node. This field is used only in the node representing a statement or smaller portion.

NODE-KEY represents additional information associated with the portion of a program represented by the node. There are four node-key fields, NODE-KEY-1 to NODE-KEY-4. Only some nodes use these fields.

NODE-LINE-NO indicates the line number of the portion of the program represented by the node. Every node contains this information.

To access this information, a *path description* to the node and a field identifier must be specified. The definition of a *path description* is defined below.

Path Description

Generation rules deal with the syntax tree. A rule works on one node in the syntax tree. In order to process another node or to access information in other nodes, a mechanism is needed to specify which node. This is called the *path description* in the generation rules.

A *path description* consists of a series of *path descriptors*. A *path descriptor* indicates a node in a position relative to the node indicated by the *path descriptors* preceding of the current *path descriptor*, or the node currently being processed if there is no preceding *path descriptor*. The *path descriptors* are defined as follows:

child indicates a child node. There are two ways to specify a child. One is by order and the other is by node type. To specify a child by order, a number is used to used to indicate the desired child. To specify a child by node type, a list of types is used to indicate the desired child. For example, a *path descriptor* **1** indicates the first child. A *path descriptor* **"ADD_ST" OR "SUBTRACT_ST"** indicates a child whose type is either "ADD_ST" or "SUBTRACT_ST" and no preceding nodes have type "ADD_ST" or "SUBTRACT_ST".

parent indicates the parent node of the current node.

elder indicates a sibling node on the left of the current node. It can have a type list or keyword "ANY" as a constraint. If the constraint is "ANY" or missing, it indicates the node just on the left of the current node. Otherwise, it indicates the rightmost node to the left of the current node and whose type matches one of the types in the type list. For example, a node has five children. The third one is being processed now. a *path descriptor* **ELDER("ADD_ST")** indicates the first child if it is a "ADD_ST" node and the second one is not.

younger indicates a sibling node on the right of the current node. It can have a type list or keyword "ANY" as a constraint. If the constraint is "ANY" or missing, it indicates the node just on the right of the current node. Otherwise, it indicates the leftmost node to the right of the current node and whose type matches one of the types in the type list.

ancestor indicates a node on the path from the root node to the current node. It can have a type list or keyword "ANY" as a constraint. If the constraint is "ANY" or missing, it indicates the parent node of the current node. Otherwise, it indicates the node closest to the current node whose type matches one of the types in the type list.

offspring indicates a node in the subtree whose root node is the current node. It can have a type list or keyword "ANY" as a constrain. If the constraint is "ANY" or missing, it indicates the first child node of the current node. Otherwise, it indicates the first node in the subtree whose type matches one of the types in the type list, in depth first search order.

root indicates the root of the syntax tree. This *path descriptor* can only be used as the first *path descriptor* in a path. It converts the path reference point from relative to absolute.

Each *path descriptor* can have a condition associated with it. The condition puts a constraint on the path search. The condition consists of two parts, separated by an "=". The first part consists of a *path description* and a *node content.* The reference point of the *path description* is the node referred to by the current *path descriptor.* The second part is an expression. The reference point for nodes in the expression is the node being processed. These two parts must be equal in order to satisfy the condition. If the condition is not satisfied and the *path descriptor* is *elder*, *younger*, *ancestor*, or *offspring*, searching for a node matching the current *path descriptor* will continue to the next node. The following is an example of a *path description* and *node content.* It is a part of the table generation rules for generating data definition tables from COBOL source code.

```
PATH(/
     OFFSPRING("FILE_DES_FILE_NODE")
       [PATH("FILE_NAME_NODE"/1):NODE-VALUE =
        PATH("DUM_SOURCE_NODE"/1):NODE-VALUE]
     /"FILE_DATA_NODE"
     /1)
     :NODE-VALUE)
```

This example is to access a value of a node. It contains two parts in the top level – a *path description* and a *node content.* The *node content* is the NODE-VALUE. The *path description* contains four *path descriptors*:

1. *root*, represented by `/`
2. *offspring*, which contains a condition, represented by

```
OFFSPRING("FILE_DES_FILE_NODE")
  [PATH("FILE_NAME_NODE"/1):NODE-VALUE =
   PATH("DUM_SOURCE_NODE"/1):NODE-VALUE]
```

   The condition contains two parts. `PATH("FILE_NAME_NODE"/1):NODE-VALUE` is the first part and it also contains two parts – a *path description* and a *node content.* The reference point for the *path description* is the node specified by the *path description* `\OFFSPRING("FILE_DES_FILE_NODE")`. The second part of the condition is `PATH("DUM_SOURCE_NODE"/1):NODE-VALUE`. It also contains two parts – a *path description* and a *node content.* But the reference point for this *path description* is the node to be processed.

3. *child*, specified by type of and represented by the string `"FILE_DATA_NODE"`.
4. *child*, specified by order and represented by `1`.

Figure 6.2 shows the relationship between nodes in the example. The node being processed currently is node **B**. The field to be accessed is the NODE-VALUE field of node **G**, which is the first child of node **D**. Node **D** is a child of node **A** and is a "FILE_DATA_NODE" node. Node **A** is a "FILE_DES_FILE_NODE" node and is not referred from node **B** since the path starts from the root (indicated by the leading "/"). Node **A** has to satisfy the following condition: the value of the NODE-VALUE attribute of node **F** has to be equal to the value of the NODE-VALUE attribute of node **H**. Node **F** is the first child of node **C**. Node **C** is a child of node **A** and is a "FILE_NAME_NODE" node. Node **H** is the first child of node **E**. Node **E** is the child of node **B** and is a "DUM_SOURCE_NODE" node.

Expression

An expression can be a string, a variable, or an arithmetic expression. An element in an arithmetic expression can be a constant, a variable, or a node content. If an element is a variable or a node content, the value of the element must be a number or a string that can be converted into a number, such as "123".

Each rule-based tool has a certain number of variables available for storing temporary values. A variable is identified by an integer, which is called *variable identifier.*

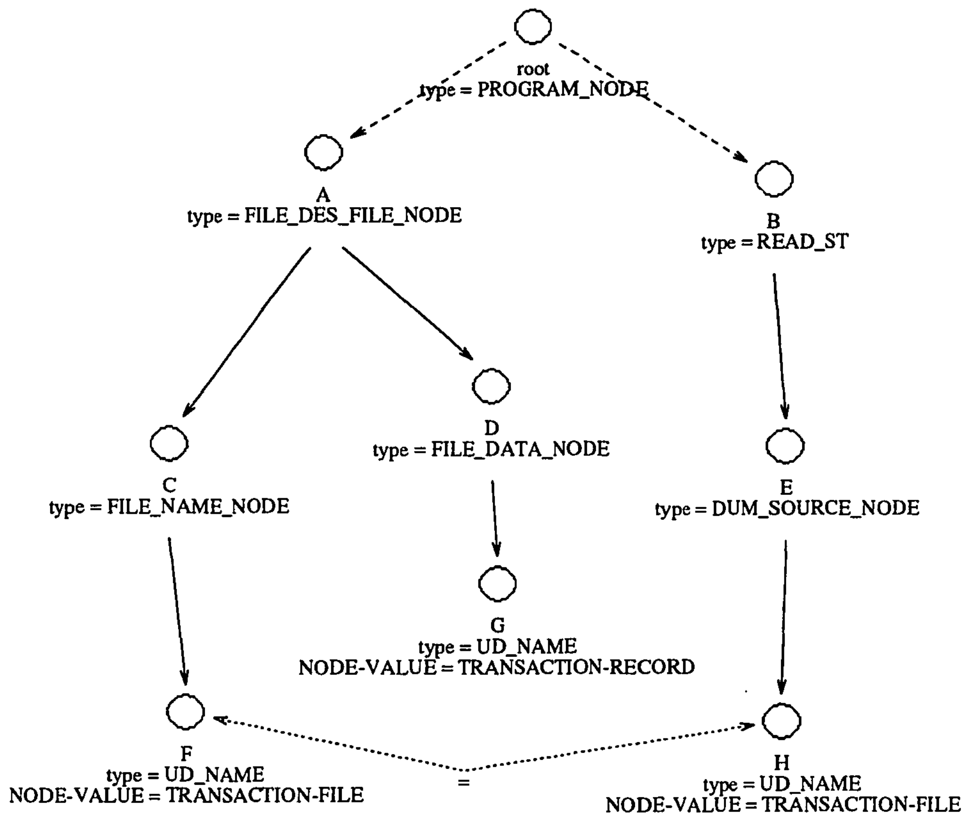


Figure 6.2: An Example of Path

A variable is accessible anywhere in a set of rules, so it is a "global" variable. The types of values that can be stored in a variable vary depending on the tool and will be discussed in the following chapters.

6.3.4 Trigger

As mentioned above, a trigger is a firing condition for a rule. A rule may have more than one trigger. In such a case, the actions in the rule are executed when any of the triggers is triggered. Logically, all triggers in a rule form an **or** relation.

Since each rule deals with one programming language concept, naturally the node type becomes the primary condition of a trigger. A trigger may have additional conditions. Only when all conditions in a trigger are satisfied, then the trigger is triggered. All conditions in a trigger are joined by the keyword **AND**. There are three types of additional conditions available to be used in triggers:

comparison condition consists of three parts: a comparison operator and two expressions to be compared.

node condition checks for the existence of a node. The node is indicated by a *path description.*

flag condition is a string. This condition is true only if its value matches the value of a flag passed by a *process* action that is processing a node.

6.3.5 Action

There are three types of actions defined in generation rules. One is the primary actions that does the main tasks designed for a tool. The detailed definitions and semantics of this type of action vary according to the application. This type of action will be discussed in the sections associated with the particular applications.

The second type of action is the flow control actions, which control the execution of other actions. There are five actions defined in generation rules for this purpose - *process*, *group*, *branch*, *break*, and *exit.* Even though the syntactic format of these actions may differ slightly in different generation rules, the semantics are the same.

The last type of action is the auxiliary actions, which assist other actions. There are two actions in this class - *variable* and *macro* action.

Process Action

A *process* action invokes another rule to process a node. The node to be processed is specified by a *path description.* When a suitable rule is found for that node, the actions in that rule will be executed. After the execution of the actions in that rule, the action following the *process* action will be executed.

A flag can be attached to the *process* action as a constraint. Only a rule that has the same flag as an additional condition will be fired. This provides an extra control for firing a rule. The following example shows how this works. It is a part of the table generation rules for generating data definition table for COBOL source code.

```
RULE(CONDITION("ADD_ST")
   ...
```

```
    PROCESS(PATH(1) FLAG("SOURCE")))
RULE(CONDITION("LITERAL" AND FLAG("SOURCE"))
    ...)
RULE(CONDITION("LITERAL" AND FLAG("SINK"))
    ...)
```

The *process* action in the first rule will invoke the second if the first child of the current node is a "LITERAL" node, but will never invoke the third rule since the flag in the trigger of the third rule does not match the flag in the *process* action.

Group Action

A *group* action collects a set of actions as a unit and applies it repeatedly to a set of nodes that share the same parent node. The nodes to be processed are indicated by two *path descriptions* that represent the first node and the last node to be processed. The *path description* to the last node can be omitted. In this case, the execution will be stopped when there is no node to be processed.

The effects caused by the actions in a *group* action are limited to one loop. When a new loop starts, all states that may affect the execution of the actions are reset to the condition before the *group* action is executed. The following example shows how this works. It is a part of model generation rules for generating IOPM graphs from COBOL source code.

```
RULE(CONDITION("DIVISION_NODE" RETURN(1 2))
   GRAPH(NAME(PATH(1/1):NODE-VALUE) "IOPM" 1 2)
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2)) ) )
RULE(CONDITION("SECTION_NODE" RETURN(1))
   ENTITY(1 "process" NAME(PATH(0):NODE-VALUE)
      ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
   GRAPH(NAME(PATH(0):NODE-VALUE) "IOPM" 4 5)
   ENTITY(2 "entry" NAME("") RESERVED
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   ENTITY(3 "exit" NAME("EXIT") RESERVED
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(4 5))
      CONNECT(13 "dummy" NAME("") 5 4) )
   RELATION(14 "dummy" NAME("") 2 4)
   RELATION(15 "dummy" NAME("") 5 3) )
```

in the above rules, *graph* action creates a graph, *entity* action creates an entity, *relation* and *connect* actions create relations between two objects, and *attribute* action sets an attribute in an object. These actions are defined in Section 6.6.2.

The first rule creates a graph **A**, then processes the children of the current node by using a *group* and a *process* actions in the rule. The *process* action in the first rule invokes the second rule for each child. When the second rule is fired, the current graph is **A** and the first *entity* action in the second rule creates an entity in **A**. When the *graph* action in the second rule executes, the current graph is set to the graph created by the graph action, say **B**, and objects created by the following actions are in graph **B**. When the execution of the actions in second rule is finished, the *group* action in the first rule starts to process the second child and resets the current graph to **A**. So the entities created by the first *entity* action in the second rule are alway in graph **A**.

Branch Action

A *branch* action executes one of the two set actions it contains, according the conditions specified in the action and node to be processed.

Break and Exit Action

The *break* and the *exit* action is used in actions grouped by a *group* action. They terminate the execution of the loop. The difference is that a *break* action terminates the current loop and starts to execute the next loop. In contrast, an *exit* terminates not only the current loop, but also the *group* action and starts to execute the action following the *group* action.

Variable Action

A *variable* action sets the value of a variable. This value then can be used by actions in other rules later.

Macro Action

A *macro* action is defined to reduce the work the programmer needs to write a set of generation rules. To use a *macro* action, the macro must be defined first. The *macro* action expands the definition of the macro in the current position and does proper substitution of identifications used in the definition of the macro. An example of using *macro* and *if* actions is shown as follows. It is a part of the model generation rules for generating IOPM2 graphs from COBOL source code.

```
MACRO("test" ARGUMENT(1 2 3 4)
      IF ((TYPE OLD-OBJ(1) = "Ready_indicator")
          CONNECT(3 "Producing_arc" NAME("") 2 1)
      ELSE
          CONNECT(4 "Exec_next" NAME("") 2 1)))
RULE(CONDITION("SECTION_NODE" RETURN(1 1))
   ...
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(4 5))
      MACRO("test" ARGUMENT(4 5 9 6)))
   ...)
```

The first part defines a macro named "test". The *macro* action is used in a rule for processing "SECTION_NODE" node. When the rule is read in by a tool, the macro in the rule will be replaced by its definition and the object IDs in the definition are replaced by the object IDs in the macro action. The following shows the rule after expansion and identifier replacement.

```
RULE(CONDITION("SECTION_NODE" RETURN(1 1))
   ...
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(4 5))
      IF ((TYPE OLD-OBJ(4) = "Ready_indicator")
          CONNECT(9 "Producing_arc" NAME("") 5 4)
      ELSE
          CONNECT(6 "Exec_next" NAME("") 5 4)))
   ...)
```

6.3.6 Data Structures for Generation Rules

There are three basic data structures for rules – one for the trigger, one for the node and the *path description*, and one for the action.

Data structure for trigger . This data structure represents one trigger.

node-type The type of node that this rule will process.

condition A set of conditions that has to be satisfied in order to trigger the rule.

Data structure for node and path description . This data structure represents one *node descriptor.*

path-type indicates the type of the *path descriptor.* It can be *child order*, *child type*, *parent*, *root*, etc.

check-type indicates the node types to be checked.

condition indicates a set of conditions that must be satisfied for this *path descriptor.*

element indicates the field in the node to be accessed.

Data structure for actions . This data structure represents one action. Each particular type of action has its own data structure to represent the parameters used by the action.

type The type of the action.

parameter A set of parameters for the action.

6.4 Parser

A piece of source code is first converted into a syntax tree by a parser. The output of the parser is the basis for all other reverse engineering tools. Since parsers are programming language dependent, one parser is required for one programming language. There are several advantages to using the syntax tree in reverse engineering process.

- Semantic information in the syntax tree is much better organized than that in the source code. This reduces the effort needed for other tools to process the information.
- Similar language components in different programming languages can have very similar syntax tree structures. For example, a syntax tree structure for an *IF* statement in COBOL is almost the same as that for an *if* statement in C and Pascal. This results in the possibility of making the other reverse engineering tools be programming language independent.

A parser consists of two major parts: lexical analyzer and syntactic analyzer. Currently, the Unix utilities *flex* and *bison* are used to generate the lexical analyzer and the syntactic analyzer, respectively. An ANSI 85 COBOL parser has been implemented by using these utilities on Sun SPARC workstations.

The lexical analyzer scans the source code file and sends to the syntactic analyzer a token that represents a keyword or other lexical unit recognized by the lexical analyzer. The syntactic analyzer analyzes the tokens to see whether they satisfy the grammar rules or not, and generates a partial syntax tree that represents the source code satisfying the grammar rule. After a source code file is analyzed, the syntax tree is written to a file.

There is a node definition file used by the parser. This file consists of mappings between pairs of integers and strings. Each pair represents same node type. This file is programming language dependent and one such a file is required for each programming language. This file provides the best of two worlds for processing nodes – integers are efficiently processed by tools and strings are easy for human to use. This file also adds a layer of correctness checking – if a string used in a set of generation rules as a node type and the string is not found in the mappings, it indicates an error.

6.5 Table Generator

To bridge the gap between the requirement of language independent tools and language dependent information source – syntax tree, a special tool – *Table Generator* is implemented. This tool adopts a rule-based approach. The tool uses a set of rules to generate a particular type of table from the syntax tree representing programs written in a particular programming language. Figure 6.3 shows the relationship between *Table Generator* and its environment.

6.5.1 Architecture

There are two inputs for *Table Generator* – the syntax tree of the program and *Table Generation Rules*, which guide the operations of *Table Generator*. The output of *Table Generator* is the generated table.

Table Generator consists of four components: *Syntax Tree Reading*, *Rule Searching*, *Action Executing*, and *Table Formatting*. *Syntax Tree Reading* reads the syntax

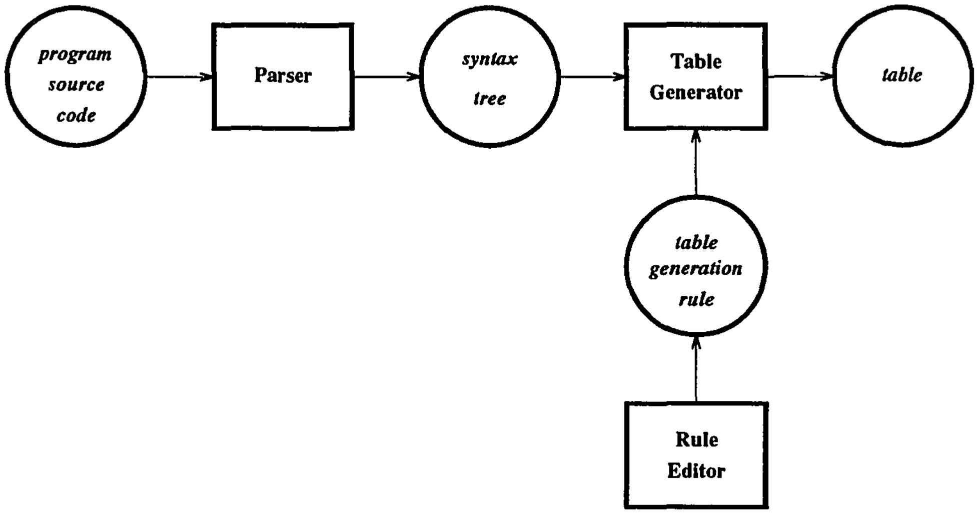


Figure 6.3: The Environment of *Table Generator*

tree of the source code and builds the internal representation of the tree. *Rule Searching* reads the required *table generation rules* and searches for proper rules to fire according to the node being processed and other conditions. *Action Executing* executes the five actions defined in the rules – *entry*, *put-field*, *process*, *group*, and *put-variable*. *Table Formatting* organizes the table and prints it out. Figure 6.4 shows the structure of *Table Generator*.

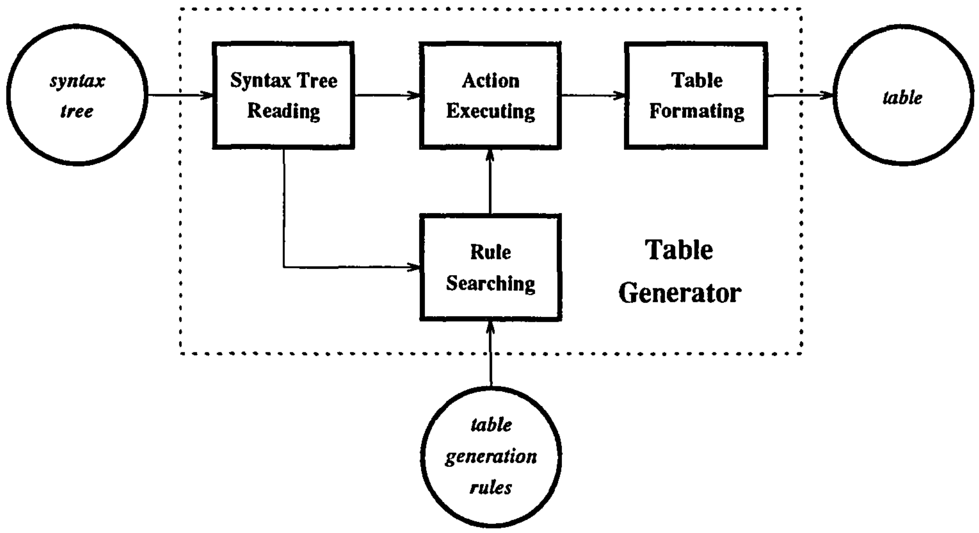


Figure 6.4: Architecture of *Table Generator*

The format of the generated table is as follows:

- The first line indicates the name of the table and number of fields in each entry of the table.
- The second line is the definitions of the fields. Each field consists of two parts – the field name and type of values to be put into the field. They are separated by a colon. The the definitions of fields are separated by a comma.
- The entries of the table start from the third line.
- One entry takes one line. A comma is used to separate fields.
- If one field has more than one value, a space is used to separate different values.
- Each string value is surrounded by a pair of double quotes.

A portion of a calling structure table for an airline reservation program is shown below. The second line describes the format of the subsequent lines.

```
TABLE(CALLING_STRUCTURE,4)
LINE_NO:INTEGER,CALLING_BLOCK:STRING,CALLED_BLOCK:STRING,LAST_BLOCK:STRING
87,"RESERVATIONS-PROGRAM-ENTRY","CRE-ATE",,
91,"RESERVATIONS-PROGRAM-ENTRY","READ-RESERVATION",,
92,"RESERVATIONS-PROGRAM-ENTRY","UP-DATE",,
100,"RESERVATIONS-PROGRAM-ENTRY","RE-PORT",,
115,"CRE-ATE-ENTRY","READ-TRANSACTION",,
138,"READ-TRANSACTION-ENTRY","EDIT",,
194,"UP-DATE-ENTRY","FINISH-RESERVATION",,
197,"UP-DATE-ENTRY","FINISH-TRANSACTION",,
```

6.5.2 Table Generation Rules

A *table generation rules* file captures the knowledge for generating a particular type of table from the syntax tree of a program written in a particular language. Some definitions defined in Section 6.3 are used and the two primary actions for table generating – entry and put-field are defined in *table generation rules.*

A *table generation rules* file consists of two parts – table definition and generation rules. The table to be generated is defined by *table-definition* clauses. The name of the table is defined by *table-name.* The number of fields in the table is defined by *number-of-field.* Each field is defined by a *field-definition* clause, which has three parameters – field-name, field-type, and multiple-value. The number of the *field-definition* clauses must match the value of *number-of-field.* The *field-name* defines the name of the field. The *field-type* specifies which type of data will be put into the field – integer or string. If the *multiple-value* flag is present, the field will accept multiple values.

There are two types of primary actions defined in table generation rules – *entry* and *put-field.* Two control actions are used – *process* and *group.* Also one type of auxiliary action is used – *put-variable.*

A *entry* action creates an entry in the table. It takes one argument – *entry-id.* The scope of the *entry-id* is the actions following this action in the same rule.

A *put-field* action puts values into the fields of an entry. It takes two arguments – *entry-id* and a list of *field contents.* The *entry-id* is optional and if it is missing, the values will be put into the fields of the most recently created entry. Each *field content* consists of three parts. The first is the name of the field that the value is put in. The second is the value to be put. The last one is the concatenate option, which is a string. If the string is present, the value to be put into this field is concatenated with the last value of the field, connected by the concatenation string. Otherwise, the value is added into the field as a new value. A *field content* can be any one of the following:

- A *global id* is a counter. It is incremented each time an entry is created. The current value of the *global id* is used as the identifier of the entry to be created.
- A *group id* is a counter. It is cleaned up each time a group action starts and incremented each time an entry is created in the group action.
- A *table id* is the id of an entry. It has two parameters – *check-field* and *check-content.* It returns the id of an entry that has a value in the field specified by *check-field* if the entry matches the value of *check-content.*
- A *table content* is the values of a field in an entry. It has three parameters – *check-field*, *check-content*, and *value-field.* It works similarly to *table id* but returns the value of the field specified by *value-field* instead of the id of the entry.
- A *node value* is the value of a field specified by the *node content* in a node specified by the *path description.* If the *path description* contains no *path descriptor*s, the current node will be used.

- A *variable* is the value of a variable. It has one parameter – *variable-id* to identify the variable to be used. A value must be assigned to the variable before it can be referred.
- A *constant* can be either a string or an integer.

The put-field action in the example below shows the usage of some field contents. It is a part of table generation rules for generating data declaration table from COBOL source code.

```
PUT-FIELD(("ID"           GLOBAL-ID)
          ("NAME"         PATH("FILE_NAME_NODE"/1):NODE-VALUE)
          ("TYPE"         NODE-VALUE)
          ("PARENT"       0)
          ("POSITION"     GROUP-ID)
          ("LENGTH"       0)
          ("SCOPE"        VARIABLE(0))
          ("LINE_NO"      NODE-LINE-NO))
```

The ID field of the entry is filled by the global id. The NAME field is filled by the NODE-VALUE of the first child of a "FILE_NAME_NODE" node. The "FILE_NAME_NODE" is a child of the current node. The PARENT and the LENGTH field are filled by an integer constant "0". The POSITION field is filled by the *group-id*. The SCOPE field is filled by the value of variable 0. The LINE_NO field is filled by the value of the NODE-LINE-NO field of the current node.

6.5.3 Algorithm

This section describes the basic data structures and algorithm used in *Table Generator*. The algorithm is written in C-like pseudo-code.

Data Structure

There are three sets of data structures – data structures for syntax tree, data structures for rules, and data structures for tables.

Data Structures for Syntax Tree The data structure defined in Section 6.3.3 for syntax trees is used.

Data Structures for Rules The data structures defined in Section 6.3.6 for generation rules are used.

Data Structures for Tables The data structures for tables are linked lists. There are two major data structures – entry and field.

Data Structure for entry:

id identification of the entry.

field fields in the entry.

Data Structure for field is:

type indicating the field contains integer or string.

number indicating the number of values in the field.

head pointing to the head of a linked list that contains the values of the field.

tail pointing to the tail of a linked list that contains the values of the field.

Algorithm

Input The root of the syntax tree to be processed,
table generation rules.

Output Generated tables

```
table_generator(root)
{
    if ((root != NULL) && (find_rule(root->type))) {
        for (each action in the action list of the rule) {
            switch(action->type) {
            case ENTRY :
                entry = create_entry(entry_action->id);
                global_id++;
                group_id++;
                break;
            case PUT_FIELD :
                value = get_value(put_action->value, root);
                field = find_field(put_action->field);
                set_value(field, value);
                break;
            case PROCESS :
                node = get_node(process_action->node, root);
                table_generator(node);
                break;
            case PUT_VARIABLE :
                variable(variable_action->variable_id) =
                    get_value(variable_action->value, root);
                break;
            case GROUP :
                start = get_node(group_action->start, root);
                stop = get_node(group_action->stop, root);
                group_id = 0;
                for (each node between start and stop) {
                    for (each sub-action in the sub-action list of
                         the group action) {
                        switch(sub-action->type) {
                        case ENTRY :
                            same as entry process above;
                        case PUT_FIELD :
                            same as put_field process above
                        case PROCESS :
                            same as process process above
                        case PUT_VARIABLE :
                            same as put_variable process above;
                        }
                    }
                break;
            }
        }
    }
}
```

6.5.4 An Example for the Table Generating Process

To illustrate the process of table generating, an example is shown here.

```
    MOVE TRANSACTION-RECORD TO PRT-REC
```

The following is the generated syntax tree in original text format.

```
(429 "Move_st" 2026 1 0 "MOVE TRANSACTION-RECORD TO PRT-REC" 142 "" "" "" ""
      435 424)
  (427 "Sourcd_d" 3503 1 0 "TRANSACTION-RECORD" 142 "" "" "" "" 429 0 )
    (425 "ud_name" 5012 3 "TRANSACTION-RECORD" "TRANSACTION-RECORD" 142
         "" "" "" "" 427 0)
  (428 "TO" 3005 1 0 "TO PRT-REC" 142 "" "" "" "" 429 427)
    (426 "ud_name" 5012 3 "PRT-REC" "PRT-REC" 142 "" "" "" "" 428 0)
```

Figure 6.5 shows the tree in graphical form. To simplify the description, the name of the node is replaced by the NODE-ID that uniquely identify each node.

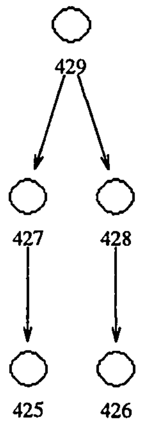


Figure 6.5: Syntax Tree for the Example of the Table Generating Process

The table to be generated is a data definition table for the program. The portion of the table generation rules used in this example is shown below:

```
TABLE("DATA_DEFINITION" 5
      FIELD("LINE_NO"   INTEGER-TYPE)
      FIELD("DEST_NAME" STRING-TYPE)
      FIELD("NODE_ID"   INTEGER-TYPE)
      FIELD("SRC_TYPE"  STRING-TYPE)
      FIELD("SRC_NAME"  STRING-TYPE))
RULE(CONDITION("MOVE_ST")
     GROUP(PATH("TO_NODE"/1)
           ENTRY(1)
           PUT-FIELD(("LINE_NO"      PATH(../..):NODE-LINE-NO)
                     ("NODE_ID"      PATH(../..):NODE-ID))
           PROCESS(PATH(0)  FLAG("DEST"))
           PROCESS(PATH(../../"DUM_SOURCE_NODE"/1) FLAG("SOURCE"))))
```

```
RULE(CONDITION("UD_NAME_NODE", AND FLAG("DEST"))
     PUT-FIELD(("DEST_NAME"     NODE-VALUE)))
RULE(CONDITION("UD_NAME_NODE" AND FLAG("SOURCE"))
     PUT-FIELD(("SRC_NAME"      NODE-VALUE)
               ("SRC_TYPE"      "VARIABLE")))
```

The data definition table to be generated has 5 fields – LINE_NO representing the line number of the statement to be processed, DEST_NAME representing a data item to which the value is assigned, NODE_ID representing the id of the node in the syntax tree representing the statement to be processed, SRC_TYPE indicating the value to be assigned is a constant or from a variable, and SRC_NAME representing the name of a data item from which the value is obtained or the value itself if it is a constant.

Assume that the node to be processed now is node 429. It is a "MOVE_ST" node so the "MOVE_ST" rule, which is the first rule, is fired. The only action in the rule, a *group* action, is executed. The group action applies the actions contained in it repeatly to all children of the "TO" node. In this example, there is only one node – node 426. It become the current node for this loop.

The first action in the *group* action is an *entry* action. It creates a new entry and inserts the entry into the table. This entry becomes the current entry and all *put-field* actions put values into the fields of this entry until a new entry is created. The second action is a *put-field* action, which puts values into two fields of the newly created entry. One is the LINE_NO field and the value to be assigned is the value of the NODE-LINE-NO field of the parent of the parent of the current node, which is node 429. The value of the LINE_NO field in node 429 is 142. The other field to be assigned is the NODE_ID fields and the value to be assigned is the value of the NODE-ID field of the parent of the parent of the current node, which is also node 429, and the value is 429.

The third action is a *process* action with a "DEST" flag. The node to be processed is the current node. The current node is a "UD_NAME" node. There are two "UD_NAME_NODE" rules. Since the flag in the second rule matches the flag passed by the *process* action, it is fired. The third rule is not fired for this *process* action since the flag in the rule does not match the flag passed by the *process* action.

The action in the second rule is a *put-field* action and it puts a value into the DEST_NAME field of the current entry. The value to be put is the NODE-VALUE of the current node, which is node 426. The value is "PRT-REC".

The fourth action in the first rule is another *process* action and the node to be processed is the first child of a "DUM_SOURCE_NODE" node, which is a child of the parent of the parent of the current node. The parent of the parent of the current node 426 is node 429. The "DUM_SOURCE_NODE" child node of node 429 is node 427 and the first child of node 427 is node 425. So the node to be processed is node 425 and it is a UD_NAME_NODE node. The *process* action also carries a flag "SOURCE". This *process* action fires the third rule since its conditions are satisfied and makes node 425 the current node.

The *put-field* action in the third rule puts values into two fields of the current entry. The first value is the NODE-VALUE of the node to be process. It is "TRANSACTION-RECORD" and it is put into the SRC_NAME field of the current entry. The second value to be put is a string constant "VARIABLE" and the field to accept this value is the SRC_TYPE.

Now, the current entry contains following values:

```
142,"PRT-REC",429,"VARIABLE","TRANSACTION-RECORD",
```

6.6 Model Generator

Model Generator takes a syntax tree of the source code as input and produces a set of particular type of graphs as output. The the most important design concept for this application is to move the knowledge related to generation of a particular type of graph from a program written in a particular programming language away from the tool. so that the tool can be used as a multi-purpose model generation tool.

Model Generator is a rule-based application [24]. Figure 6.6 shows the relationship between *Model Generator* and its environment.

6.6.1 System Structure

The *Model Generator* has four components: *Syntax Tree Reading*, *Rule Searching*, *Action Executing*, and *Graph Layout*. Figure 6.7 shows the architecture of the *Model Generator*.

The functionalities of *Syntax Tree Reading* and *Rule Searching* are similar to their counterpart in *Table Generator*. *Action Executing* executes actions defined in *model generation rules* to create graphs. *Graph Layout* relocates the entities so that the

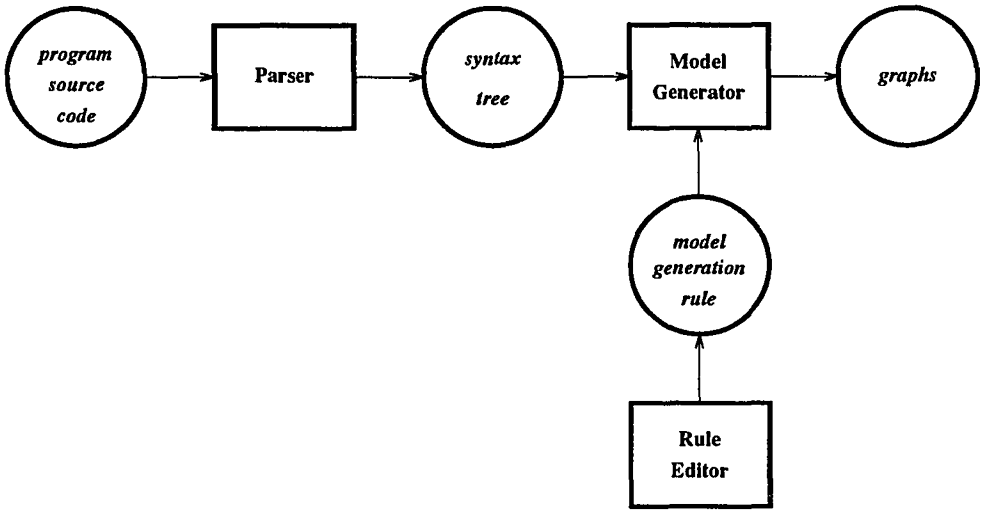


Figure 6.6: The Environment of *Model Generator*

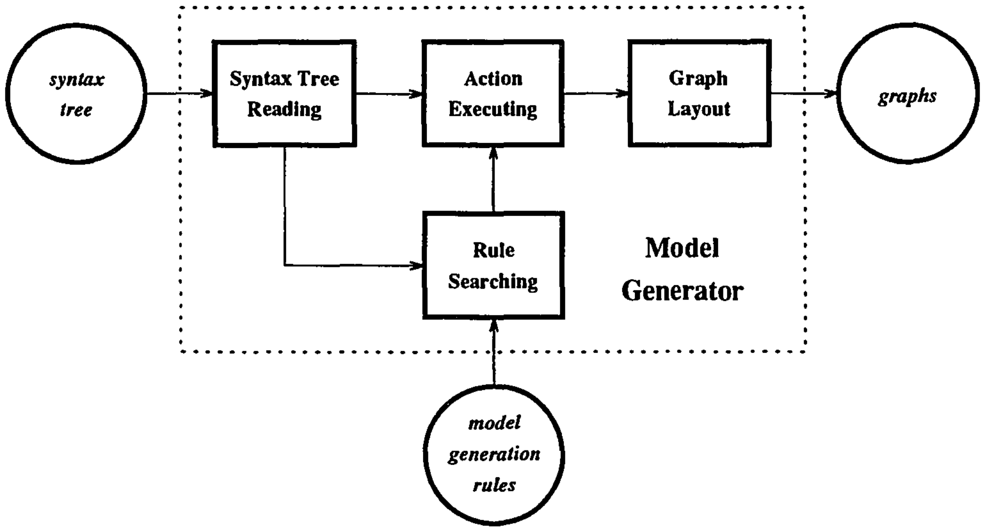


Figure 6.7: Architecture of *Model Generator*

graphs can be viewed better. The generated graphs are stored in a file and written in CDF (*Common Data File*) format.

6.6.2 Model Generation Rules

Some Definitions Used in Model Generation Rules

Naming Schema A mechanism is defined for composing a string from various sources, which is called the *naming schema* in *model generation rules.* The *naming schema* is to concatenate the elements in a *name clause* into one string. An element can be a string, the value of a variable, a loop counter, or a piece of information in a node. All numbers are converted into strings before the concatenation.

Dummy Object Another concept used in *model generation rules* is the *dummy* object. All *dummy* objects are regular objects, except the type of the objects is "dummy" and they are removed after nodes in the syntax tree are processed and before the created graphs are output.

There are two reasons to create dummy objects. One is to connect objects. There is sometimes no rule for a particular type of node. A *process* action may try to process a particular type of node but there may not be any such nodes found in the path or there may be no rule for the node to be processed. In these cases, a *dummy* entity is created so the objects created before and after this *process* action can be connected via this *dummy* entity. The second reason is to make writing generation rules easier. Creating *dummy* objects and removing them later enables one rule to be used in a greater variety of situations and makes the rules more compact and readable. It is easier to verify rules if every rule produces a fragment that has similar types of objects as the connect points for connecting with other fragments.

All *dummy* entities are removed before any *dummy* relations are removed. To remove a *dummy* entity, a new relation is created for each pair of incoming and outgoing relations of that entity. For example, in Figure 6.8, the *dummy* entity has two incoming relations – **A** and **B** and three outgoing relations – **C**, **D**, and **E**. After removing this *dummy* entity, six new relations are created: **AC**, **AD**, **YE**, **BC**, **BD**, and **BE**.

To remove a *dummy* relation, the two objects connected by the *dummy* relation are merged into a new objects. For example, in Figure 6.8, the source object of the *dummy* relation has two incoming relations A and B and one outgoing relation D, besides the *dummy* relation. The sink object of the *dummy* relation has one incoming relation C besides the *dummy* relation and one outgoing relation E. After removing this *dummy* relation, the source object and the sink object of this *dummy* relation

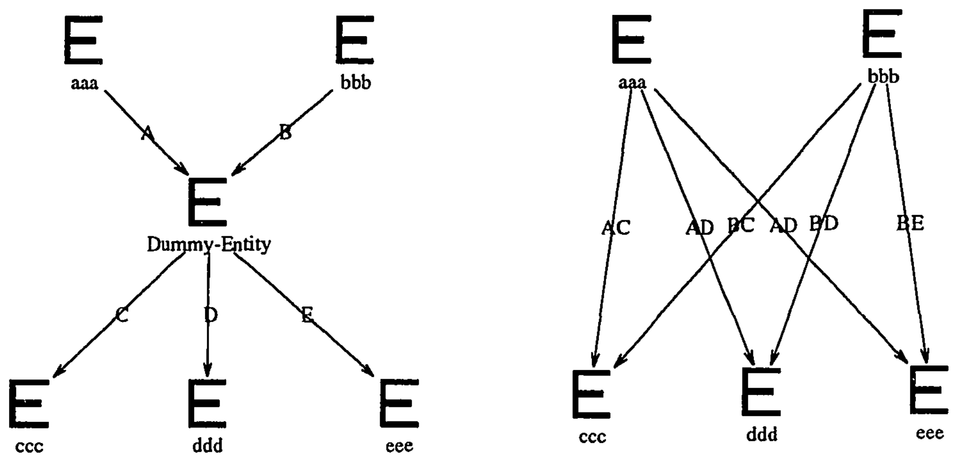


Figure 6.8: An Example of Removing a Dummy Entity

are merged into one object that has three incoming relation A, B, and C and two outgoing relation D and E.

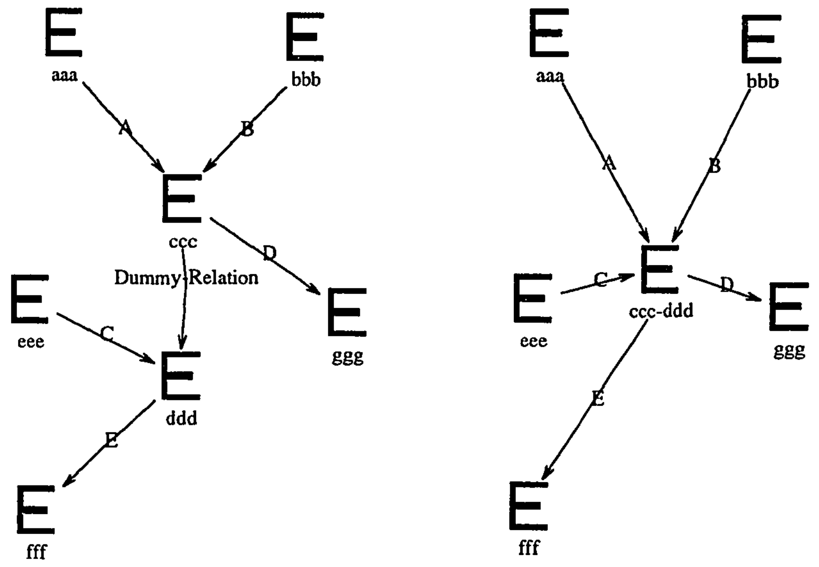


Figure 6.9: An Example of Removing a Dummy Relation

The object newly created by the removing process inherits the properties of either of the objects which it represents. The general rule is that the new object inherits the properties of the sink object unless the source object has a RESERVED flag. There is an exception: If the desired object does not have a name, the name of the other object is used. Table 6.1 shows the relationships between the new object and the old objects. In the table, a mark $\times$ indicates the existence of the condition is ignored. A mark $\surd$ indicates the existence of the condition. *src* indicates that a value from the source object is inherited and *dst* indicates that a value from the destination object is inherited.

Table 6.1: Relationship between the New Object and the Old Objects

Objects to be Merged				New Object	
Source Object		Destination Object			
Reserved	Has Name	Reserved	Has Name	Name	Other Properties
		$\times$	$\times$	dst	dst
	$\surd$	$\times$		src	dst
	$\surd$	$\times$	$\surd$	dst	dst
$\surd$	$\times$	$\times$		src	src
$\surd$		$\times$	$\surd$	dst	src
$\surd$	$\surd$	$\times$	$\surd$	src	src

Actions in Model Generation Rules

A *model generation rules* file captures the knowledge for generating a particular type of graph from the syntax tree of a program written in a particular language. Some definitions defined in Section 6.3 are used and six primary actions are defined for model generation rules.

A *model generation rules* file consists of three parts – a set of relation type definitions, a set of macro definitions, and a set of generation rules. The first two parts are optional.

A *relation type definition* defines a type of relation that connects two particular types of objects. This definition has the highest priority and overwrites the type of relation defined previously by other rules. The purpose of introducing the *relation type definition* concept is to overcome the difficulty of keeping the relations in the generated models correct according to the constraints defined in the META graph for that type of graph and to reduce the complicity of rules. For META graphs that define a lot of relations between entities, the number of actions required to add a relation will be very large since a series of tests must be conducted to find the correct relation to be created. The *relation type definition* takes care the correctness of the relation type after all graphs are created, so the relations created in the rules can be arbitrary types and no testing is required.

The purpose of introducing macros is to reduce the work for writing generation rules. The definition of a macro is given before it can be used in a generation rule by a *macro* action. Each *macro* action will be expanded into its definition when it is read in.

There are six types of primary actions defined in model generation rules – *graph*, *entity*, *item*, *relation*, *connect*, and *attribute*. Five control actions are used – *process*, *group*, *if*, *break*, and *exit*. Two auxiliary actions are used – *macro* and *variable*.

A *graph* action creates a graph. It takes four arguments – the name of the graph to be generated, the name of the META graph for the graph, the entry entity of the graph, and the exit entities of the graph. The scope of the graph action includes all the subsequent actions contained in the rule and actions in rules that are fired by some of the subsequent actions, unless another graph action is encountered. One exception is the graph action as a sub-action of a group action. In this case, the scope of the graph action is limited to one loop. When the next loop starts, the graph that was the current graph when the group action starts are restored as the current graph again.

An *entity* action creates an entity in a graph. It takes six arguments – the id of the entity, the type of the entity, the name of the entity, an optional reserved flag, an optional unique flag, and the attributes of the entity. The reserved flag of the entity indicates whether the properties of the entity is reserved and will be kept if it is merged with another entity. The unique flag indicates whether there is only one entity of the specified type having that name in the graph. The action should check the entity list of the graph to see if the entity exists or not. If the entity exists, this entity is used as the current entity. Otherwise, a new entity will be created.

The *item* action is almost the same as the *entity* action and takes same arguments, except it can only be used as a sub-action in a *group* action. It executes only from the second loop of the *group* action.

The *relation* action establishes a relation between two entities. It takes seven arguments – the id of the relation, the type of the relation, the name of the relation, the source object, the sink object, an optional reserved flag, and the attributes of the relation. The source object and the sink object for a relation can be in different graphs. For an object that is not created by the actions in the current rule, a graph name and an object name are used to locate the object. The reserved flag indicates whether the properties of the relation is reserved and will be kept if it is merged with another relation.

A *connect* action acts almost the same as the *relation* action, except it only can be used as a sub-action in a *group* action and has one more argument – an optional reverse flag. It establishes a relation between two objects that are created in two subsequent loop of a group action. The source object is created by last loop of the *group* action and the sink object is created by this loop of the *group* action, or vice verse if the reverse flag is present.

An *attribute* action puts an attribute into an object. It takes three arguments – the id of the object that takes the attribute, the name of the attribute, and the value of the attribute.

The *process* action in *model generation rules* has one more parameter than the *process* action discussed in Section 6.3.5 – a set of return entities. These are the entities generated by the actions in a rule fired by the *process* action and used as the connecting points. The return entities correspond to the return entities in the trigger condition of the rule that processes the node. If there are more return entities in the trigger condition than in the action, extra entities will be discarded. If there are less return entities in the trigger condition than in the action, the rest of the return entities in the action will be set to *dummy* entities.

6.6.3 Algorithm

This section describes the basic data structures and algorithm used in *Model Generator*. The algorithm is written in C-like pseudo-code.

Data Structure

There are three sets of data structures – data structures for the syntax tree, data structures for rules, and data structures for graphs.

Data Structures for the Syntax Tree The data structure defined in Section 6.3.3 for the syntax tree is used.

Data Structure for Rules The data structures defined in Section 6.3.6 for generation rules are used.

Data Structure for Model The data structures for a model are linked lists. There are three major data structures - graph, object, and attribute.

The data structure for graphs is:

name is the name of the graph.

formalism is the META graph for the graph.

object is a list of objects in the graph.

The data structure for objects is:

name is the name of the object.

type is the type of the object.

class indicates the object is an entity or a relation.

sub-list A list of relations that start from the object.

obj-list A list of relations that end at the object.

subject The source of the object if it is a relation.

object The sink of the object if it is a relation.

attribute A set of attributes associated with the object.

The data structure for attributes is:

type is the type of the attribute.

value is the value of the attribute.

Algorithm

Input The root of syntax tree to be processed,
list of rules.

Output Generated Model

```
model_generator(root)
{
    if ((root != NULL) && (rule->find-rule(root->type))) {
        for (each action in the list of actions of the rule) {
            switch(action->type) {
            case GRAPH :
                name = get_name(graph_action->name, root);
                graph = create_graph(name, graph_action->formalism);
                break;
            case ENTITY :
            case ITEM :
                name = get_name(entity_action->name, root);
                entity = create_entity(name, entity_action->type);
                set_flag(entity, entity_action->flag);
                set_attribute(entity, entity_action->attribute, root);
                break;
            case RELATION :
                name = get_name(relation_action->name, root);
                subject = get_object(relation_action->subject);
                object = get_object(relation_action->object);
                relation = create_relation(name, relation_action->type,
                                           subject, object);
                set_attribute(entity, relation_action->attribute, root);
                break;
            case ATTRIBUTE :
                object = get_object(attribute->object);
                value = get_name(attribute->value, root);
                set_attribute(object, attribute->type, value);
                break;
            case VARIABLE :
                value = get_name(variable->value, root);
                set_variable(variable->id, value);
                break;
            case PROCESS :
                node = get_node(process_action->node, root);
                model_generator(node);
                set_return(node, process_action->returns);
                break;
            case IF :
                if (evaluate(if->condition, root) != false)
                    execute(if->true_action);
                else
        execute(if->false_action);
    break;
case EXIT :
    return;
    break;
case GROUP :
    start = get_node(group_action->start, root);
    stop = get_node(group_action->stop, root);
    clear_buffer();
    graph_save = graph;
    loop_counter = 0;
    for (each node between start and stop) {
        loop_counter++;
        for (each sub-action in the sub-action list of group action) {
            switch(sub-action->type) {
            case GRAPH :
                same as graph process above
                break;
            case ENTITY :
                same as entity process above
                break;
            case ITEM :
                if (loop_counter > 1)
                    same as entity process above
                break;
            case RELATION :
                same as relation process above
                break;
            case CONNECT :
                name = get_name(connect_action->name, node);
                subject = get_old_object(connect_action->subject);
                object = get_new_entity(connect_action->object);
                relation = create_relation(name, connect_action->type,
                                           subject, object);
                set_attribute(relation, relation_action->attribute,
                              node);
            case PROCESS :
                same as process process above
                break;
            case IF :
                same as if process above
                break;
            case BREAK :
```

```
                                break;
                            }
                        }
                        graph = graph_save;
                    }
                    break;
                }
            }
        }
    }
```

6.6.4 An Example for the Model Generating Process

This section illustrates the model generating process by showing how to generate a segment of a IOPM2 graph from an IF statement written in ANSI COBOL. The sample code is shown below.

```
IF RR-PASSENGER-NAME2 = SPACES
  MOVE TR-PASSENGER-NAME2 TO RR-PASSENGER-NAME2
ELSE
  MOVE SPACES TO PRINT-RECORD
  MOVE 'NO ROOM FOR 2ND PASSENGER' TO PRT-MESSAGE
  MOVE TRANSACTION-RECORD TO PRT-REC
  WRITE PRINT-RECORD
```

The following is the generated syntax tree in the original text format.

```
(880 "If_st" 2021 1 0 "IF RR-PASSENGER-NAME2 = SPACES THEN ELSE" 374 "" ""
     "" "" 883 0)
  (854 "Rel_op" 3504 3 "=" "RR-PASSENGER-NAME2 = SPACES " 374 "" "" "" ""
       880 0)
    (852 "ud_name" 5012 3 "RR-PASSENGER-NAME2" "RR-PASSENGER-NAME2" 374
         "" "" "" "" 854 0)
      (853 "SPACE" 5001 3  " " "SPACES" 374 "" "" "" "" 854 852 )
  (879 "THEN" 3000 1 0 "THEN" 375 "" "" "" "" 880 854)
    (859 "Move_st" 2026 1 0 "MOVE TR-PASSENGER-NAME2 TO RR-PASSENGER-NAME2"
         375  ""  ""  ""  "" 879 0)
      (857 "Sourcd_d" 3503 1 0 "TR-PASSENGER-NAME2" 375 "" "" "" "" 859 0)
        (855 "ud_name" 5012 3 "TR-PASSENGER-NAME2" "TR-PASSENGER-NAME2" 375
             "" "" "" "" 857 0)
      (858 "TO" 3005 1 0 "TO RR-PASSENGER-NAME2" 375 "" "" "" "" 859 857)
        (856 "ud_name" 5012 3 "RR-PASSENGER-NAME2" "RR-PASSENGER-NAME2" 375
            "" "" "" "" 858 0)
  (878 "ELSE" 3001 1 0 "ELSE" 377 "" "" "" "" 880 879)
    (864 "Move_st" 2026 1 0 "MOVE SPACES TO PRINT-RECORD" 377 "" "" "" "" 878 0)
      (862 "Sourcd_d" 3503 1 0 "SPACES" 377 "" "" "" "" 864 0)
        (860 "SPACE" 5001 3 " " "SPACES" 377 "" "" "" "" 862 0)
      (863 "TO" 3005 1 0 "TO PRINT-RECORD" 377 "" "" "" "" 864 862)
        (861 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 377 "" "" "" ""
             863 0)
      (869 "Move_st" 2026 1 0 "MOVE 'NO ROOM FOR 2ND PASSENGER' TO PRT-MESSAGE"
           378 "" "" "" "" 878 864)
        (867 "Sourcd_d" 3503 1 0 "'NO ROOM FOR 2ND PASSENGER'" 378 "" "" "" ""
             869 0)
          (865 "LITERAL" 5001 3 "'NO ROOM FOR 2ND PASSENGER'"
               "'NO ROOM FOR 2ND PASSENGER'" 378 "" "" "" "" 867 0)
        (868 "TO" 3005 1 0 "TO PRT-MESSAGE" 378  "" "" "" "" 869 867)
          (866 "ud_name" 5012 3 "PRT-MESSAGE" "PRT-MESSAGE" 378 "" "" "" ""
               868 0)
      (874 "Move_st" 2026 1 0 "MOVE TRANSACTION-RECORD TO PRT-REC" 379 "" ""
           "" "" 878 869)
        (872 "Sourcd_d" 3503 1 0 "TRANSACTION-RECORD" 379 "" "" "" "" 874 0 )
          (870 "ud_name" 5012 3 "TRANSACTION-RECORD" "TRANSACTION-RECORD" 379
                "" "" "" "" 872 0)
        (873 "TO" 3005 1 0 "TO PRT-REC" 379 "" "" "" "" 874 872)
          (871 "ud_name" 5012 3 "PRT-REC" "PRT-REC" 379 "" "" "" "" 873 0)
      (877 "Write_st" 2053 1 0 "WRITE PRINT-RECORD" 380 "" "" "" "" 878 874)
        (876 "Source_rec" 3503 1 0 "PRINT-RECORD" 380 "" "" "" "" 877 0)
          (875 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 380 "" "" "" ""
               876 0 )
```

Figure 6.10 shows the tree in graphical form. To simplify the description, the name of the node is replaced by the NODE-ID, which uniquely identifies each node.

The model generation rules used in this example are shown below:

```
RULE(CONDITION("IF_ST" RETURN(1 6))
   ENTITY(1 "Process" NAME("IF"))
   ENTITY(2 "Branch" NAME(PATH(1):NODE-ELEMENT) RESERVED
            ATTRIBUTE("node_id" NAME(NODE-ID))
            ATTRIBUTE("Branch_type" NAME("IF")))
   ENTITY(3 "Process" NAME(""))
   ENTITY(4 "Process" NAME(""))
   ENTITY(5 "Ready_indicator" NAME("")
            ATTRIBUTE("Initial_no_tokens" NAME("0")))
```

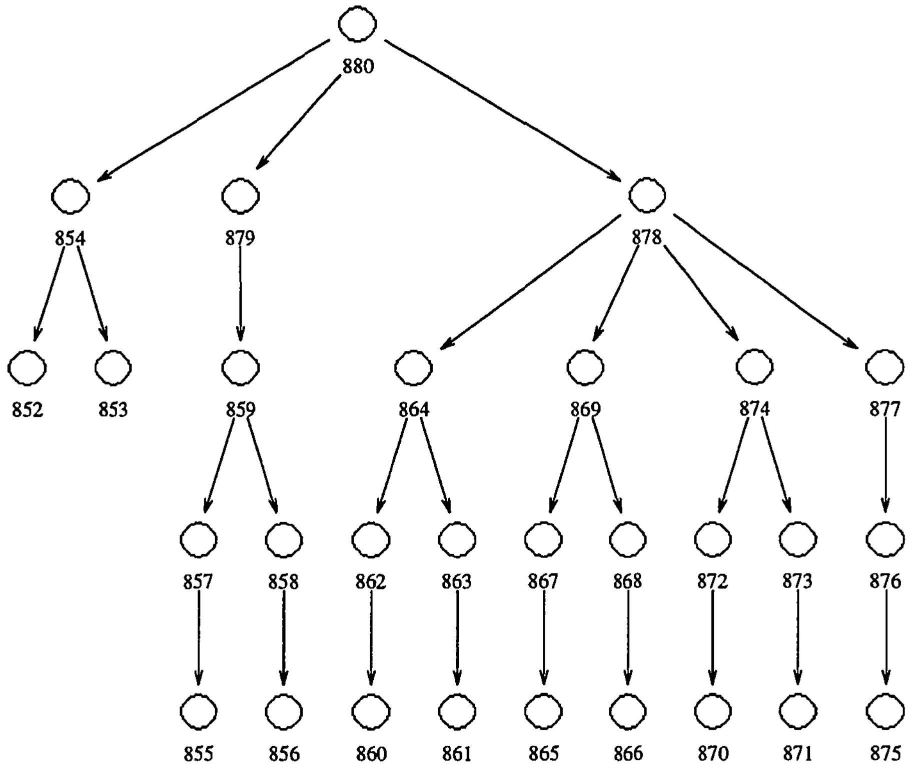


Figure 6.10: Syntax Tree for the Example of Model Generating Process

```
   ENTITY(6 "dummy" NAME(""))
   RELATION(11 "Producing_arc" NAME("") 1 2 RESERVED)
   RELATION(12 "Consuming_arc" NAME("") 2 3 RESERVED
      ATTRIBUTE("Condition" NAME("T")))
   RELATION(13 "Consuming_arc" NAME("") 2 4 RESERVED
      ATTRIBUTE("Condition" NAME("F")))
   PROCESS(PATH("THEN_NODE") RETURN(7 8))
   PROCESS(PATH("ELSE_NODE") RETURN(9 10))
   RELATION(14 "Exec_next" NAME("") 3 7)
   RELATION(15 "Exec_next" NAME("") 4 9)
   RELATION(16 "Producing_arc" NAME("") 8 5 RESERVED)
   RELATION(17 "Producing_arc" NAME("") 10 5 RESERVED)
   RELATION(18 "Consuming_arc" NAME("") 5 6 RESERVED))
RULE(CONDITION("THEN_NODE" RETURN(1 2))
     CONDITION("ELSE_NODE" RETURN(1 2))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2))
      CONNECT(11 "Exec_next" NAME("") 2 1 )))
RULE(CONDITION("MOVE_ST" RETURN(1 1))
   ENTITY(1 "Process" NAME(NODE-ELEMENT)
      ATTRIBUTE("node_id" NAME(NODE-ID))))
RULE(CONDITION("WRITE_ST" RETURN(1 9))
   ENTITY(1 "Process" NAME(NODE-ELEMENT)
      ATTRIBUTE("node_id" NAME(NODE-ID)))
   PROCESS(PATH("INVALID_KEY_NODE") RETURN(2 3))
   PROCESS(PATH("NOT_INVALID_KEY_NODE") RETURN(4 5))
   RELATION(49 "Exec_next" NAME("") 1 2)
   RELATION(110 "Exec_next" NAME("") 3 4)
   PROCESS(PATH("END_OF_PAGE_NODE") RETURN(6 7))
   PROCESS(PATH("NOT_END_OF_PAGE_NODE") RETURN(8 9))
   RELATION(55 "Exec_next" NAME("") 5 6)
   RELATION(56 "Exec_next" NAME("") 7 8) )
RULE(CONDITION("INVALID_KEY_NODE" RETURN(2 6))
     CONDITION("NOT_INVALID_KEY_NODE" RETURN(2 6))
     CONDITION("END_OF_PAGE_NODE" RETURN(2 6))
     CONDITION("NOT_END_OF_PAGE_NODE" RETURN(2 6))
   ENTITY(2 "Branch" NAME(NODE-NAME) RESERVED
      ATTRIBUTE("node_id" NAME(NODE-ID))
      ATTRIBUTE("Branch_type" NAME(NODE-NAME)))
   ENTITY(3 "Process" NAME(""))
   ENTITY(4 "Process" NAME(""))
   ENTITY(5 "Ready_indicator" NAME("")
      ATTRIBUTE("Initial_no_tokens" NAME("0")))
   ENTITY(6 "dummy" NAME(""))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(7 8))
      CONNECT(11 "Exec_next" NAME("") 8 7))
   RELATION(13 "Consuming_arc" NAME("") 2 3 RESERVED
      ATTRIBUTE("Condition" NAME("T")))
   RELATION(14 "Consuming_arc" NAME("") 2 4 RESERVED
      ATTRIBUTE("Condition" NAME("F")))
   RELATION(15 "Exec_next" NAME("") 3 7)
   RELATION(16 "Producing_arc" NAME("") 8 5)
   RELATION(17 "Producing_arc" NAME("") 4 5)
   RELATION(18 "Consuming_arc" NAME("") 5 6))
```

Assume that the current node is node 880 and it is an "IF_ST" node so the "IF_ST" rule is fired. The first six *entity* actions in the rule create six entities – A **Process** entity with name "IF", labeled 1, a **Branch** entity labeled 2 whose name is obtained from the NODE-ELEMENT field of node 854, which is the first child of the current node, two **Process** entities labeled 3 and 4 with no names, a **Read_indicate** entity labeled 5, and a dummy entity labeled 6. The dummy entity will be remove later.

The first *relation* action establishes a **Producing_arc** relation from entity 1 to entity 2. The second *relation* action establishes a **Consuming_arc** from entity 2 to entity 3 with a **Condition** attribute of value "T". The third *relation* action establishes a **Consuming_arc** from entity 2 to entity 4 with a **Condition** attribute of value "F".

The first *process* action sets the "THEN_NODE" node, node 879, to be the current node and fires "THEN_NODE" rule. The *group* action in the "THEN_NODE" rule processes the statements in the then part of the IF statement one by one. There is only one statement in the then part. The only loop of the *group* action sets node 859 as the current node since it is the only child of node 879 and fires "MOVE_ST" rule since node 879 is a "MOVE_ST" node.

The action in the "MOVE_ST" rule creates a **Process** entity and returns it as the entry and the exit point of the segment of graph created by the rule to the *process* in the "THEN_NODE" rule. Since there is no more statement to be processed, the "THEN_NODE" rule returns this entity as the entry and exist point of the segment of the graph created by the rule. This entity is labeled as entity 7 and 8 in the "IF_ST" rule.

Then the second *process* action in the "IF_ST" rule makes node 878 as the current node and fires "THEN_NODE" rule which is the same rule for "THEN_NODE". The first loop of the *group* action in the rule processes node 864. This is a "MOVE_ST" node and the rule for "MOVE_ST" node return a **Process** entity as the segment of the graph created by the rule. The *connect* action dose not executed since this is the first loop of the *group* action. The *group* action sets node 869 as the current node and processes it. Node 869 is a "MOVE_ST" node and the rule for "MOST_ST" node returns another **Process** entity. The *connect* action executes this time and establishes an **Exec_next** relation from the first **Process** entity to the second **Process** entity. Then the *group* action sets the node 874 as the current node and fires a rule for that node. The rule returns a **Process** entity and it is connected to the second **Process** entity by an **Exec_next** relation. The last node to be processed by the *group* action is node 877 and it is a "WRITE_ST" node. The first action in the "WRITE_ST" rule creates a **Process** entity. Then the *process* actions in the rule tries to find a node to process. Since no nodes is found for process, each *process* actions returns a dummy entity as the segment created by the rule. These dummy entities are connected by the *relation* actions to the **Process** entity and the **Process** entity and the last dummy entity are returned the segment of graph created by the "WRITE_ST" rule. The **Process** entity return by the "WRITE_ST" rule is connected to the **Process** created by the rule which processed node 847. Then the **Process** entity created by rule which processed node 864 and the last dummy entity created by the rule which processed node 877 are returned as the segment of graph for then part to the "IF_ST" rule and they are labeled as entity 9 and 10 in "IF_ST" rule.

The last five *relation* actions established relations from entity 3 to 7 and 4 to 9 with type **Exec_next**, 8 to 5 and 10 to 5 with type **Producing_arc**, and 5 to 6 with type **Consuming_arc**. The final graph is shown in Figure 6.11. Since the dummy type of entities is defined in META graph of IOPM2 and will be deleted in the late process, **Ready_indicate** entities are used in the graph instead and labeled "dummy".

6.7 Model Builder

Due to limitations of the rule-based approach, not all types of graphs can be generated by *Model Generator. Model Builder* is designed to deal with this problem. *Model Builder* is mainly used to generate CPM graphs and FM graphs which are difficult to generate by the rule-based approach alone. It also generates other DM graphs and PSTM graphs. It is a programming languages independent tool. The relationship between *Model Builder* and its environment is shown in Figure 6.12.

6.7.1 Design Concepts

The design of the tool is centered on CPM generation. The algorithm of CPM generation is derived from the methodology proposed by Benedusi et. al. [3] with major modifications to fit the special programming language free requirement.

One of the biggest problems associated with programming languages such as COBOL is the inconsistency between the program structure and unit of invocation. Unlike other procedural programming languages, the unit of program structure and the unit of invoking in COBOL can be different. For instance, a PERFORM can

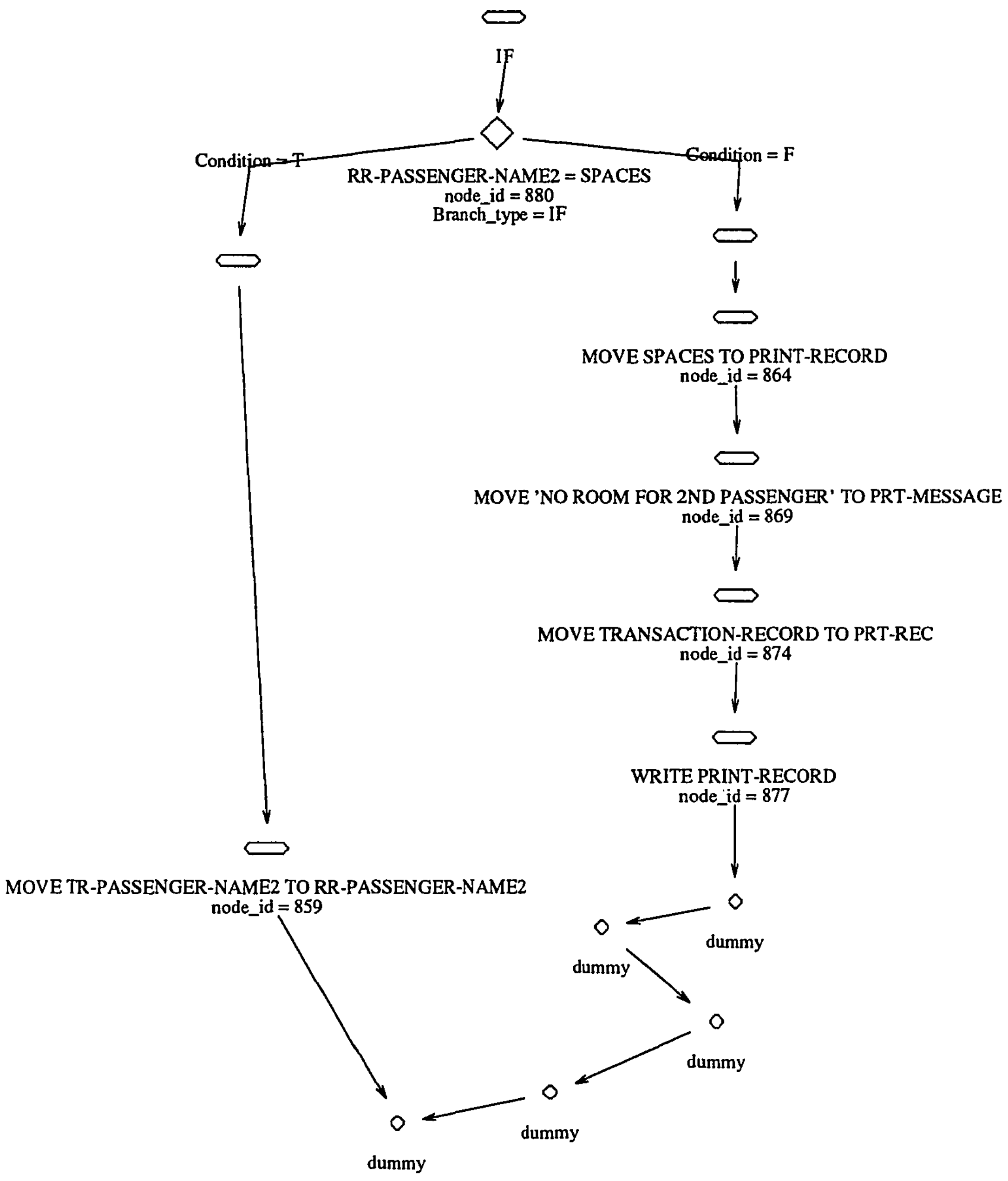

IF
Condition = T
RR-PASSENGER-NAME2 = SPACES
node_id = 880
Branch_type = IF
Condition = F
MOVE SPACES TO PRINT-RECORD
node_id = 864
MOVE 'NO ROOM FOR 2ND PASSENGER' TO PRT-MESSAGE
node_id = 869
MOVE TRANSACTION-RECORD TO PRT-REC
node_id = 874
WRITE PRINT-RECORD
node_id = 877
MOVE TR-PASSENGER-NAME2 TO RR-PASSENGER-NAME2
node_id = 859
dummy
dummy
dummy
dummy
dummy

Figure 6.11: Generated Graph for the Example of Model Generating Process

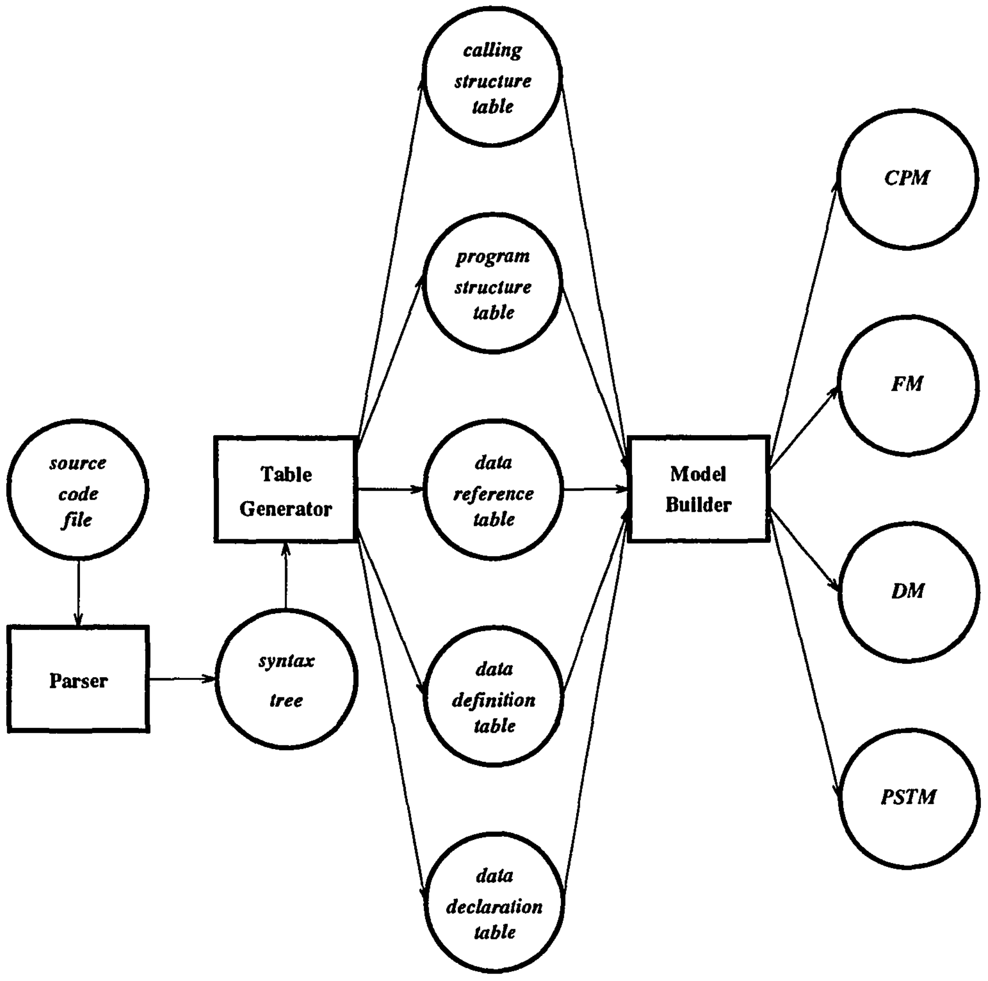


Figure 6.12: Environment of *Model Builder*

invoke one section, one paragraph, several sections, or several paragraphs, and the invoked paragraphs can across the boundary of sections. This makes it difficult to define modules. In Proud, a module is defined as a set of code that is invoked by one or more PERFORM statements. For example, the there are four modules defined in the following code:

```
SECTION A.
A1.
      ...
    PERFORM B THROUGH C1.
      ...
    PERFORM B2 THROUGH C2.
A2.
```

```
    ...
    EXIT.
SECTION B.
B1.
    ...
    PERFORM C.
    ...
B2.
    ...
    EXIT.
SECTION C.
C1.
    ...
C2.
    ...
C3.
    ...
    EXIT.
```

Section A is the first module. The first PERFORM statement in paragraph A1 defines the second module including section B and paragraph C1. The second PERFORM statement in A1 defines the third module including paragraphs B2, C1, and C2. Finally, the PERFORM statement in paragraph B1 defines the fourth module including section C. The relationship between code blocks and modules in the this example is shown in Figure 6.13.

To meet the requirement of programming language independence, this information is captured in a calling structure table. Each entry in the table has two fields to indicate the beginning and the ending code segments. For highly structured programming language such as C or Pascal, only the first field is needed since the unit of invocation is same as the unit of structure.

Another weak point in the original method is the accuracy of the generated data flow diagrams. If large number of global variables are used in the target program, some extra data flow may be generated. The problem is that the original method does not distinguish the order of assignments and references on the same variable. Even if a reference is ahead of an assignment, a flow from the assignment task to the reference task is still generated. To overcome this problem, a modification based on data value analysis was proposed [23]. The basic idea is to find out the value cover space of each assignment and use that information to eliminate the faulty information. This modified method is not implemented in the current implementation, due to time constraints.

Model Builder takes up to five tables as inputs, depending on the type of graphs

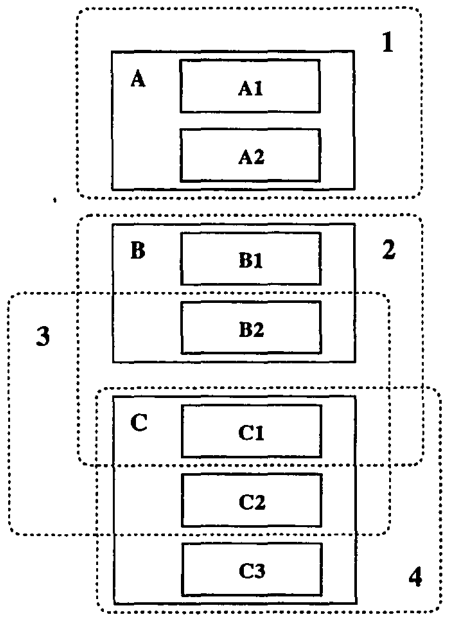


Figure 6.13: Relationship between Code Blocks and Modules

to be generated. All of the tables are generated from the syntax tree by *Table Generator*. Table 6.2 shows the relationship between the tables and the types of graphs to be generated.

Table 6.2: Relationship between Tables and Types of Graphs

Table	*CPM*	*DM*	*FM*	*PSTM*
data declaration table (dec)	√	√	√	
data definition table (def)	√		√	
data reference table (ref)	√		√	
program structure table (ps)	√		√	√
calling structure table (cs)	√		√	√

6.7.2 Architecture

Model Builder has several components. The major components are *Data Structure Builder*, *Structure Chart Builder*, and *Read/Write Lists Builder*. *Data Structure Builder* creates the data structure model that represents data structures used in a target program. *Structure Chart Builder* creates the structure chart that represents structures of the program and its modules. *Read/Write Lists Builder* adds a list of read/write dependencies between modules into the structure chart. Each type of graphs is then generated by an *ad hoc* builder. Figure 6.14 shows the configuration of *Model Builder*.

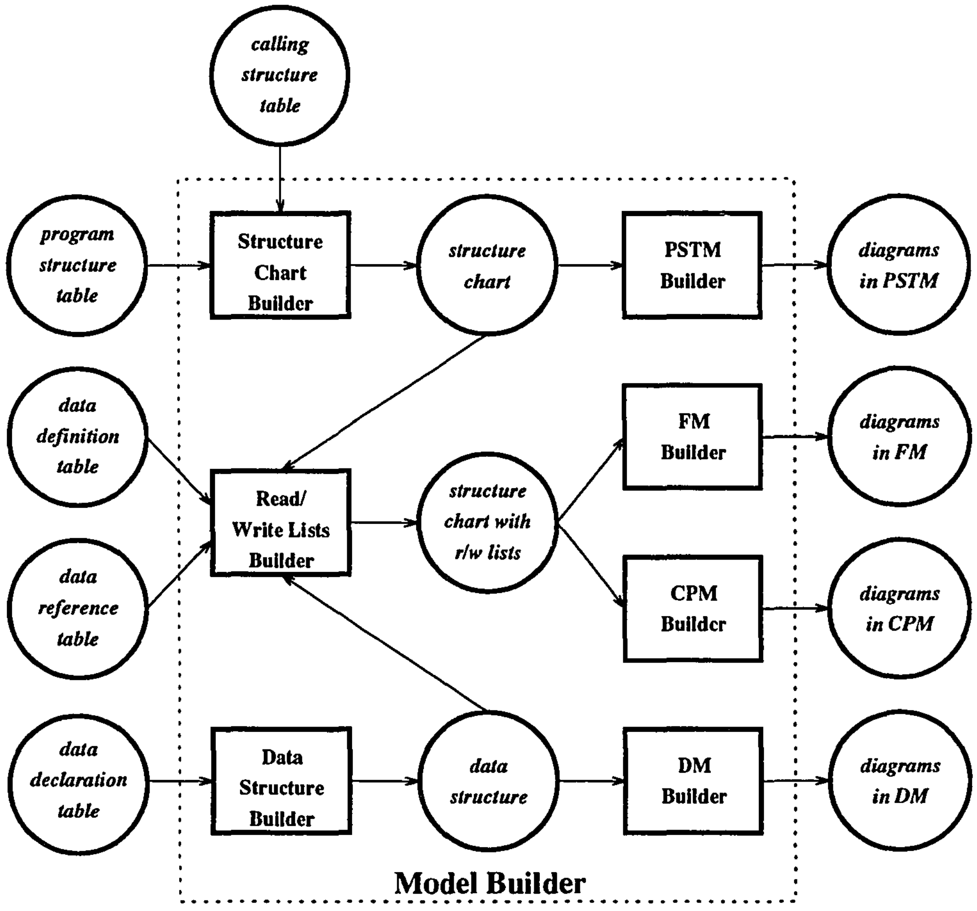


Figure 6.14: Structure of *Model Builder*

6.7.3 Data Structure Builder

Data Structure Builder creates a hierarchical representation of the data items declared in the target system. Its input is a *data declaration table*. The table contains the following information regarding a data item:

ID An integer that uniquely identifies the data item.

NAME A string that represents the name of the data item.

TYPE An integer that indicates whether the data item is an integer, real, string, record, or file.

PARENT An integer that is the ID of another data item that contains this data item.

POSITION An integer that indicates relative position of the data item in the structure if it is a member of another data item.

LENGTH An integer that indicates the length of the data item.

FORMAT A string that represents the format of the data item.

ALIAS A string that represents the name of a data item that shares the same storage space with this data item.

VALUE A string that represents the initial value of the data item.

SCOPE A string that represents the scope of the data item. It can be either "EXTERNAL", "INTERNAL", or "PARAMETER".

LINE_NO An integer that indicates the line number where the declaration of the data item is specified.

Data Structure Builder consists of two components: input component and building component. The input component reads a *data declaration table* and creates a linked list that contains the information obtained from the table. The building component uses the PARENT and POSITION information of each element to build the data structure. The format of the data structure is a forest. The first level data items represent files, internal data, and parameters, if any of them exist in the program. Redefined items are linked together by special pointers.

6.7.4 Structure Chart Builder

Structure Chart Builder consists of five components: *program structure table input*, *calling structure table input*, *program structure building*, *module identifying*, and *structure chart building*. Figure 6.15 shows structure of *Structure Chart Builder*.

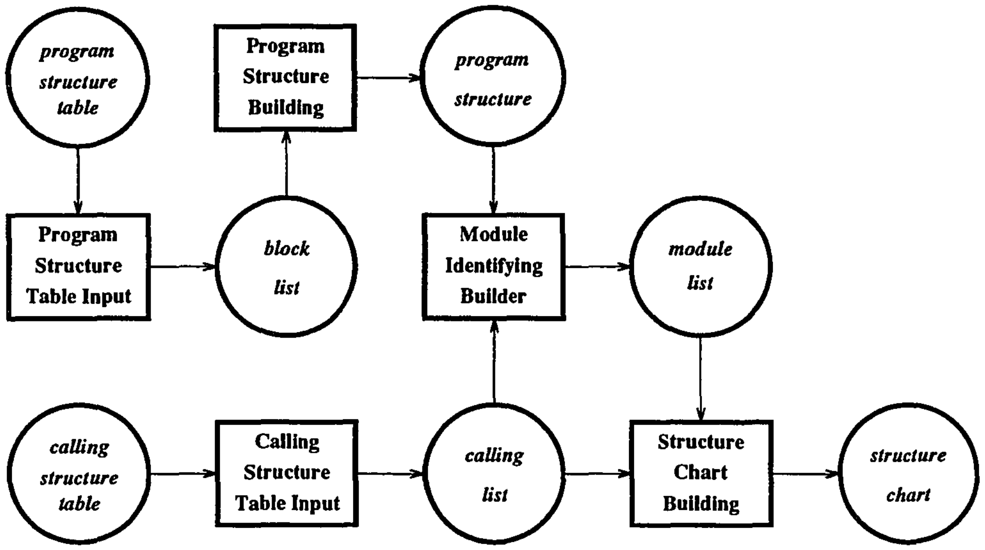


Figure 6.15: Structure of *Structure Chart Builder*

Program Structure Table Input Component

This component reads a *program structure table* and creates a linked list to hold the input information. Each entry of the table represents a code block in the program. The table contains the following information regarding a code block:

Program Structure Table

ID An integer that uniquely identifies the code block.

LINE_NO An integer that indicates the line number of the beginning of the block.

NAME A string that represents the name of the block.

POSITION An integer that indicates the relative position of the block in a code block containing this code block.

PARENT An integer that represents the id of a code block containing this code block.

NODE_ID An integer that represent the id number of a node that represents this code block in the syntax tree.

Calling Structure Table Input Component

This component reads a *calling structure table* and creates a linked list to hold the obtained information. Each entry in the table represents an invocation statement that appears in the program. The table contains the following information:

Calling Structure Table

LINE_NO An integer that indicates the location of an invocation statement.

CALLING_BLOCK A string that represents the name of a code block containing the invocation statement.

CALLED_BLOCK A string that represents the name of a code block. This block can be the only block to be invoked, or the first block to be invoked by the invocation statement.

LAST_BLOCK A string that represents the name of a code block. If field is not empty, it represents the last code block being invoked. CALLED_BLOCK field and this field indicate the range of the invocation. If this field is empty, only the code block indicated by CALLED_BLOCK is invoked.

Program Structure Building Component

This component builds the program structure by using the PARENT and POSITION information obtained from a *program structure table.*

Module Identifying Component

The task of this component is to identify modules in the program. The following part describes the algorithm used for identifying modules:

1. Create an empty module list.
2. Choose the first invocation statement that has not been processed from the *calling structure table.*
3. Find all code blocks invoked by the invocation statement.
4. Check the module list to see if any existing module is in the same range. If a module exists, skip to step 6.

5. Add a new module to the module list. The new range of the new module is the code blocks invoked by the invocation statement.
6. If any invocation statement has not been processed, go back to step 2.

Each module contains a list to indicate the range of the module, or in other words, the code blocks belonging to the module.

Structure Chart Building Component

This component builds a structure chart from the module list and the *calling structure table.* It adds two lists to each module. One is the list of modules that are called by this module, and the other is the list of modules that calls this module. The following describes the algorithm used for building the structure chart:

1. Choose the first invocation statement that has not been processed from the calling structure table.
2. Find the module that contains this statement.
3. Find the module that is invoked by the statement.
4. Add the called module to the calling list of the calling module.
5. Add the calling module to the called list of the called module.
6. Repeat step 1 through step 5 until all the statements in the *calling structure table* are processed.

6.7.5 Input/Output Lists Builder

Input/Output Lists Builder consists of three components: *list input*, *initial list attachment*, and *list refinement.* The objective of *Input/Output Lists Builder* is to identify data items referred to or defined by a module or modules invoked by the module.

List Input Component

This component reads two cross reference tables and stores obtained data in two lists. The two cross reference tables are a *data reference table* and a *data definition table.* The information contained in two tables are shown as below:

The Data Reference Table

LINE_NO An integer that indicates where in the source code this reference is made.

DATA_NAME A string that represents the name of the referred data item.

NODE_ID An integer that represents the id of a node in the syntax tree. The node represents a statement that makes the referencing action.

The Data Definition Table

LINE_NO An integer that indicates where in the source code this definition is made.

DEST_NAME A string that represents the name of the data item to that a new value is assigned.

NODE_ID An integer that represents the id of a node. The node represents a statement which makes this defining action.

SOURCE_TYPE A string that represents the type of the source data item. It can be either "CONSTANT", "FILE", or "VARIABLE".

SOURCE_NAME A string that represents the name of the source data item.

Initial List Attachment Component

This component attaches referring and defining information to a module. Data items are divided into two groups. One is internal data, which refers to data items internal to the program. The other is external data, which refers to files or other input/output items. The process adds four more lists to each module. The first one is an internal read list that indicates the internal data items referred to by the module. The second one is an external read list that indicates the external data items referred to by the module. The third one is an internal write list that indicates the internal data items defined by the module. The last one is an external write list that indicates the external data items defined by the module. These lists are called basic r/w lists. The algorithm used for this process is described as follow:

R/W List Attach Algorithm

1. Choose one data item from the reference table.

2. Find the module containing this reference.
3. If the data item belongs to internal data group, add it to the internal read list of the module. Otherwise, add it to the external read list of the module.
4. Repeat steps 1 through step 3 until all data items in the reference table are processed.
5. Repeat step 1 through 4 for the definition table, except add data items to the internal or external write list.

List Refinement Component

This component refines the read/write lists so that they can represent the referring and defining of data items in each module more precisely. There are two steps of refinement. The first step is to remove redundant information from the basic lists by using the following rules. The second step is to create read/write lists for a subtree rooted at a module.

Redundant Elimination Rules

1. Only one identical data item is in a list.
2. No data item should be in a list if it is a member of a data item that is in the list.
3. If all members of a data item are in a list, the members should be replaced by the data item.

The creating read/write lists for a subtree rooted at a module process adds four more lists to the module, called subtree lists that represent the internal/external, read/write lists of the subtree. The process adopts a bottom up approach. There are two cases in this process. If the module does not call any other module, then the subtree lists are same as module's basic lists. Otherwise, the called modules' subtree lists are merged with the calling module's basic module lists to form the module's subtree lists. After the lists are formed, redundant data items are removed from the lists by using the same process as in elimination of redundant information from a module's basic lists.

```
WORKING-STORAGE SECTION.
01  RESERVATIONS-PROGRAM-WORK.
```

```
        05  RP-END-OF-TRANS          PIC X VALUE ZERO.
        05  RP-END-OF-RESERV         PIC X VALUE ZERO.
        05  RP-INVALID-RECORD        PIC X VALUE ZERO.
```

For the source code listed above, the internal read list for module "UP-DATE" is shown below:

```
UP-DATE -- Original
  Body R/W
    Internal read: RP-END-OF-RESERV RP-END-OF-TRANS RP-END-OF-TRANS
                   RP-END-OF-RESERV RP-END-OF-TRANS RP-END-OF-RESERV
                   RP-END-OF-RESERV RP-END-OF-TRANS
```

The read list after eliminating redundant data item is as following:

```
UP-DATE -- After merge
  Body R/W
    Internal read: RP-END-OF-RESERV RP-END-OF-TRANS
```

6.7.6 CPM Builder

CPM Builder generates CPM graphs from the structure chart that contains read/write lists. One CPM graph is created for each module that calls other modules. The following algorithm shows how to create one CPM graph for one module.

CPM Building Algorithm

1. Create an internal list by merging the subtree internal read/write lists and removing redundancy.
2. Create an external list by merging the subtree external read/write list and removing redundancy.
3. Create a **task** entity for the module and a **task** entity for each module called by this module.
4. Create an **internal data** entity for each data item in the internal list. If only constant values are assigned to the data item, a "HIDDEN" view option is set for the entity since it does not contribute to data transfer from input data to output data.

5. Create an **external data** entity for each data item in the external list.
6. For each data item in the module's basic internal read list, find the entity representing the item or its ancestor. Link the entity representing the data item and the entity representing the module with an **input** relation.
7. For each data item in the module's basic internal write list, find the entity representing the item or its ancestor. Link the entity representing the module and the entity representing the data item with an **output** relation.
8. For each data item in the module's basic external read list, find the entity representing the item or its ancestor. Link the entity representing the data item and the entity representing the module with a **receive** relation.
9. For each data item in the module's basic external write list, find the entity representing the item or its ancestor. Link the entity representing the module and the entity representing the data item with a **generate** relation.
10. Repeat the above four steps for every module called by this module.

For example, module CRE-ATE invokes module read-transaction. The read/write lists for each module are as follows:

```
CRE-ATE -- After merge
  Body R/W
    Internal read: TR-REC RESERVATIONS-RECORD TR-CREATION-CODE
                   RP-INVALID-RECORD RP-END-OF-TRANS
    Internal write: RESERVATIONS-RECORD
    External write: RESERVATIONS-FILE
  Module R/W
    Internal read: PRINT-RECORD TRANSACTION-RECORD RP-INVALID-RECORD
                   EDIT-WORK RP-END-OF-TRANS RESERVATIONS-RECORD
    Internal write: PRINT-RECORD RP-END-OF-TRANS TRANSACTION-RECORD
                    RP-INVALID-RECORD EDIT-WORK RESERVATIONS-RECORD
    External read: TRANSACTION-FILE
    External write: PRINT-FILE RESERVATIONS-FILE
READ-TRANSACTION -- After merge
  Body R/W
    Internal read: PRINT-RECORD TRANSACTION-RECORD RP-INVALID-RECORD
    Internal write: PRINT-RECORD RP-INVALID-RECORD RP-END-OF-TRANS
                    TRANSACTION-RECORD
    External read: TRANSACTION-FILE
    External write: PRINT-FILE
  Module R/W
    Internal read: EDIT-WORK RP-INVALID-RECORD TRANSACTION-RECORD
```

```
                 PRINT-RECORD
Internal write: EDIT-WORK RP-INVALID-RECORD TRANSACTION-RECORD
                 RP-END-OF-TRANS PRINT-RECORD
External read: TRANSACTION-FILE
External write: PRINT-FILE
```

The corresponding CPM graph is shown in Figure 6.16.

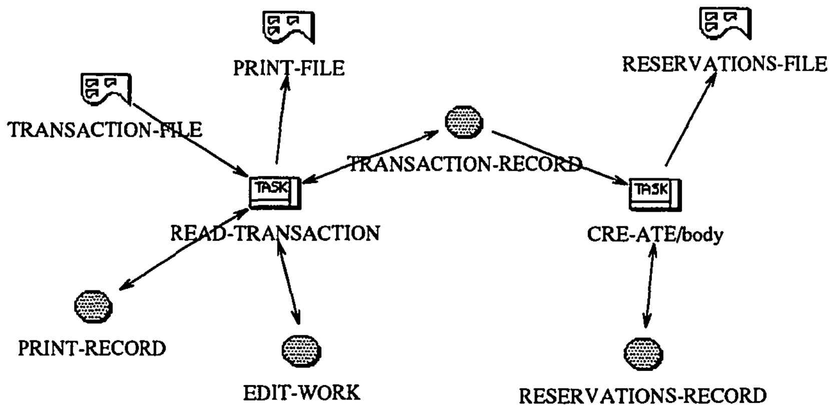


Figure 6.16: Example of CPM graph

6.7.7 FM Builder

FM Builder generates FM graphs from the structure chart that contains read/write lists. One FM graph is created for each module. The following algorithm shows how to create one FM graph for one module.

FM Building Algorithm

1. Create a read list by merging the module's subtree internal read list and external read list. If one internal data item represents the record of an external item, the internal item is removed from the list and attached to the external item.
2. Create a write list by merging the module's subtree internal write list and external write list. If one internal data item represents the record of an external item, the internal item is removed from the list and attached to the external item.
3. Create a **function** entity for the module.

4. Create a **sub_function** entity for each module called by the module.
5. Create an **input_data_set** entity for each data item in the read list. If a data item has attached items, a **sub_input_data** entity is created and linked to the **input_data_set** entity with a **consists_of** relation.
6. Create an **output_data** entity for each data item in the write list. If a data item has attached items, a **sub_output_data** entity is created and linked to the **output_data_set** entity with a **consists_of** relation.
7. Link the **function** entity and each **sub_function** entity with an **is_decomposed_into** relation.
8. Link **input_data_set** entities and the **function** entity with **is_input_to** relations.
9. Link the **function** entity and **output_data_set** entities with **outputs** relations.

The FM graph for the module "CRE-ATE" in section 6.7.6 is shown in Figure 6.17.

6.7.8 DM Builder

DM Builder generates DM graphs from the data structure table. It creates several graphs. One graph represents the overview of the data structure. Additional graphs are generated to represent the data structure of each file. One graph represents the internal data items. Another graph represents the data items used as parameters. The algorithms used for generating different graphs are described as follows:

Overview Graph Building Algorithm

1. Create a **data_bank** entity.
2. Create a **file** entity for each file data item in the data structure.
3. Create an **internal_data** entity if any internal data item is defined.
4. Create an **internal_data** entity if any data item is defined as parameter.
5. Link the **data_bank** entity and **file** entities with **has_file** relations.
6. Link
   the **data_bank** entity and **internal_data** entities with **has_internal_data** relations.

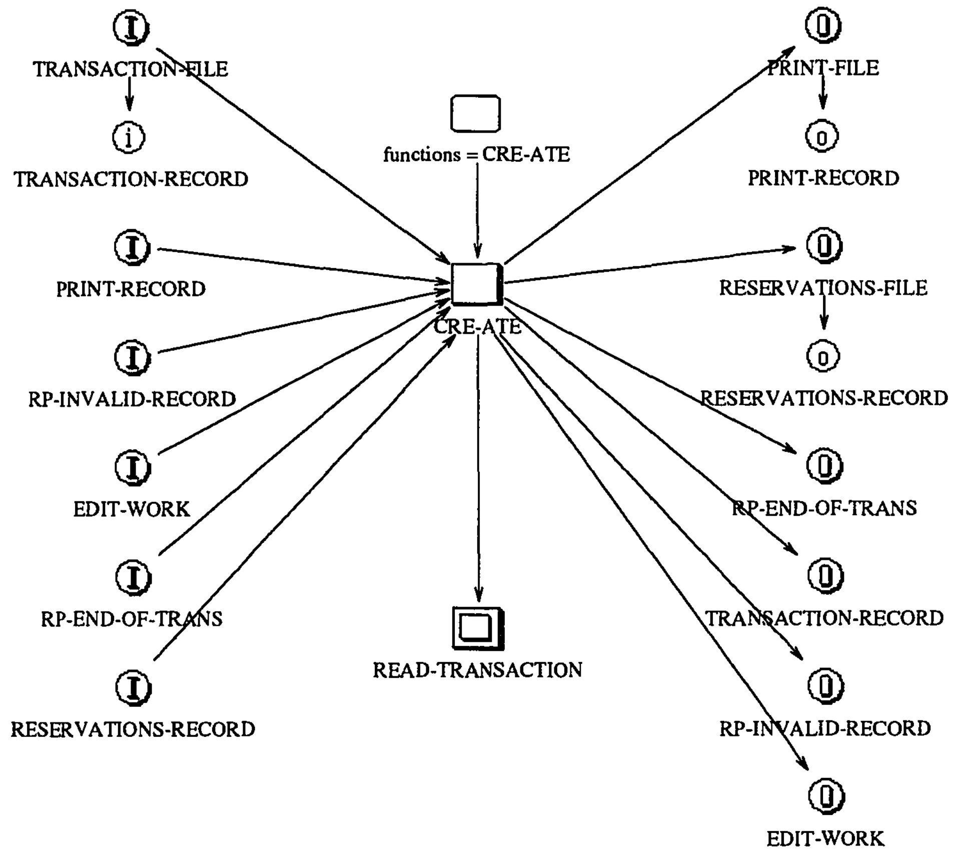


Figure 6.17: Example of FM graph

File Graph Building Algorithm

1. Create a **file** entity.
2. Create a **record** entity for each compounded data item contained in the file.
3. Create a **data_item** entity for each primary data item contained in the file.
4. Link the entities with **contains** relations according to their ownership.

Internal Data and Parameter Graph Building Algorithm

1. Create an **internal_data** entity.
2. Create a **record** entity for each compounded data item defined internally or used as parameter.

3. Create a **data_item** entity for each primary data item defined internally or used as parameter.

4. Link the entities with **contains** relations according to their ownership.

The DM graph for the following code is shown in Figure 6.18.

```
FILE SECTION.
FD  TRANSACTION-FILE
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS TRANSACTION-RECORD.
01  TRANSACTION-RECORD.
  05  TR-REC.
    10  TR-FLIGHT-NUMBER         PIC 9(6).
    10  TR-PASSENGER-NAME1       PIC X(30).
    10  TR-PASSENGER-NAME2       PIC X(30).
  05  TR-CREATION-CODE           PIC X.
  05  TR-TRANSACTION-CODE        PIC X.
  05  FILLER                     PIC X(12).
```

6.7.9 PSTM Builder

PSTM Builder generates PSTM graphs from program structure, module structure and structure chart. Only one graph is generated for one source code file. The following algorithm shows how to create the PSTM graph.

PSTM Building Algorithm

1. For each code block, create a **block** entity.

2. For each code block create a **contains** relations from the **block** entity representing the block to other **block** entities representing the blocks contained by the block.

3. For each module, create a **module** entity.

4. For each module create **invokes** relations from the **module** entity representing the module to other **module** entities representing the modules invoked by the module.

5. For each module, create **includes** relations from the **module** entity representing the module to **block** entities representing the code blocks included in the module.

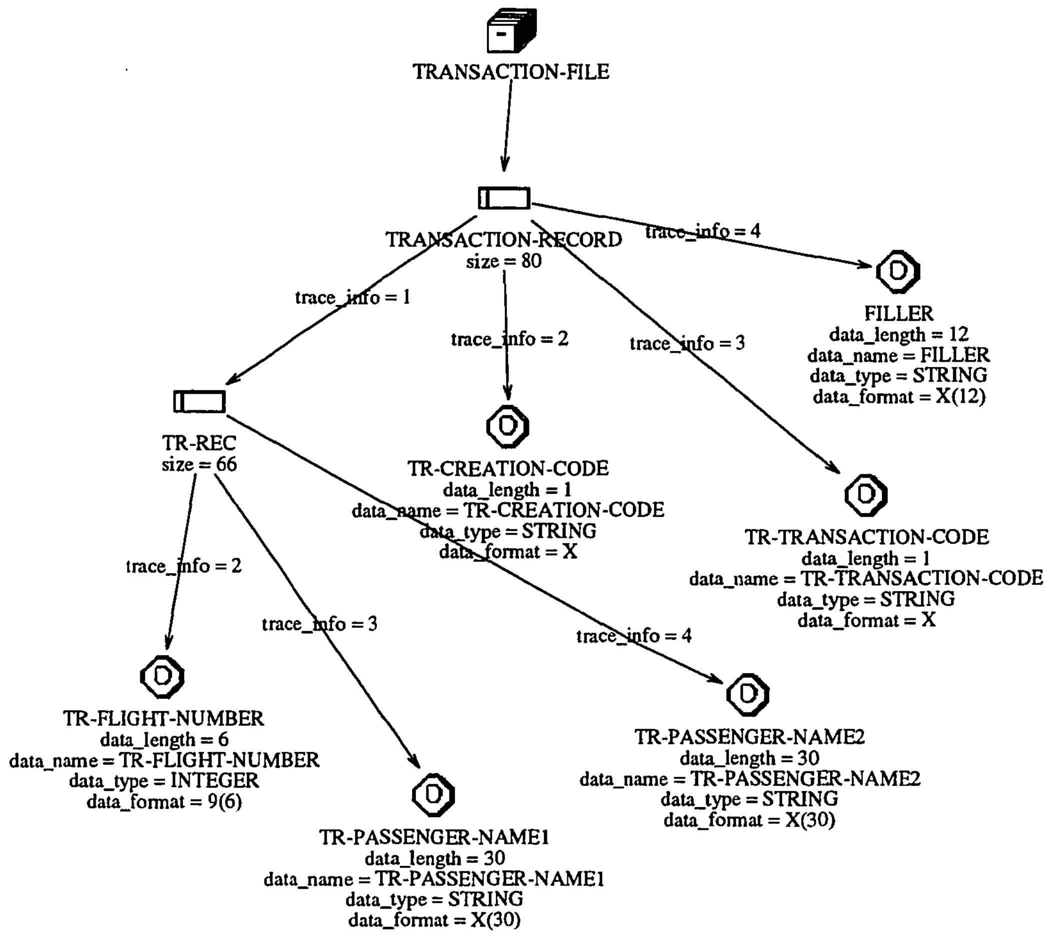


Figure 6.18: Example of DM graph

The PSTM graph for `sample.cobol` (The source code is listed in Appendix B.1) is shown in Figure 6.19.

6.8 Linkage Generator

Linkage Generator is a rule-based tool. Its major function is to create relations between two graphs. The two graphs can be in different MERA formalisms. It uses *Linkage Generation Rules* and cross reference to locate the proper objects in the graphs and add relations to connect them.

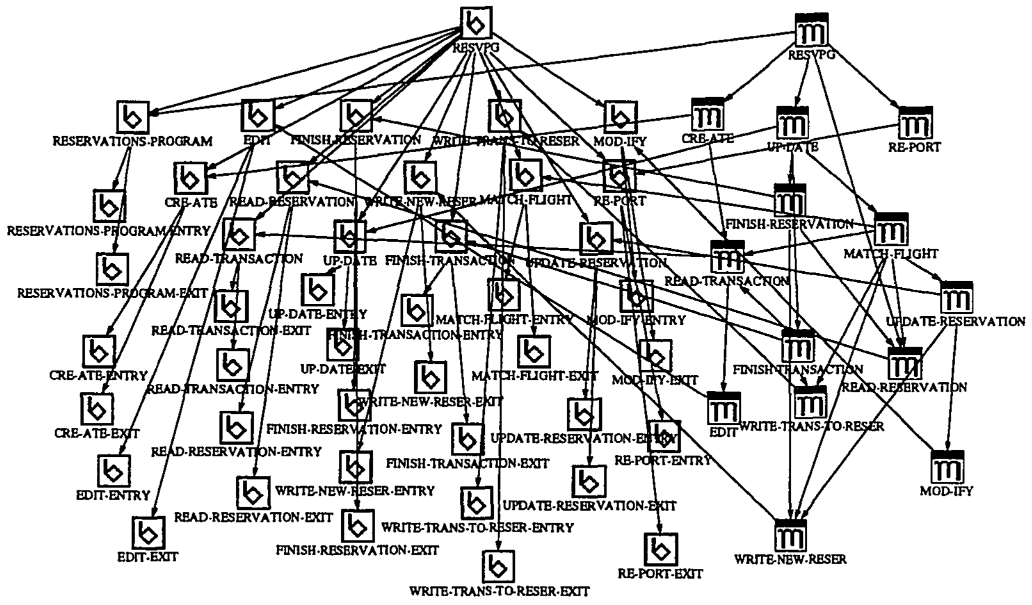


Figure 6.19: Example of PSTM graph

6.8.1 Architecture

Linkage Generator consists of six components: *Table Reading*, *Model Reading*, *Rule Reading*, *Rule-Based Generating*, and *Model Writing.* Figure 6.20 shows the architecture of *Linkage Generator.*

The *Table Reading* component reads in cross reference tables. The *Model Reading* component reads in source and destination *RMA* models. The *Rule Reading* component reads in *linkage generation rules.* The *Model Writing* component writes the generated linkages out. The *Rule-Based Generating* component is used to generate linkages according to the conditions set in each rule. The generating process in these components is controlled by *linkage generating rules.*

6.8.2 Linkage Generation Rules

Linkage generation rules capture the knowledge for generating linkage relations between two particular types of graphs. Different from model generation rules and table generation rules, the *linkage generation rules* have no actions. They are collections of conditions for connecting two objects in two models.

A *linkage generation rules* file for establishing linkages between two particular

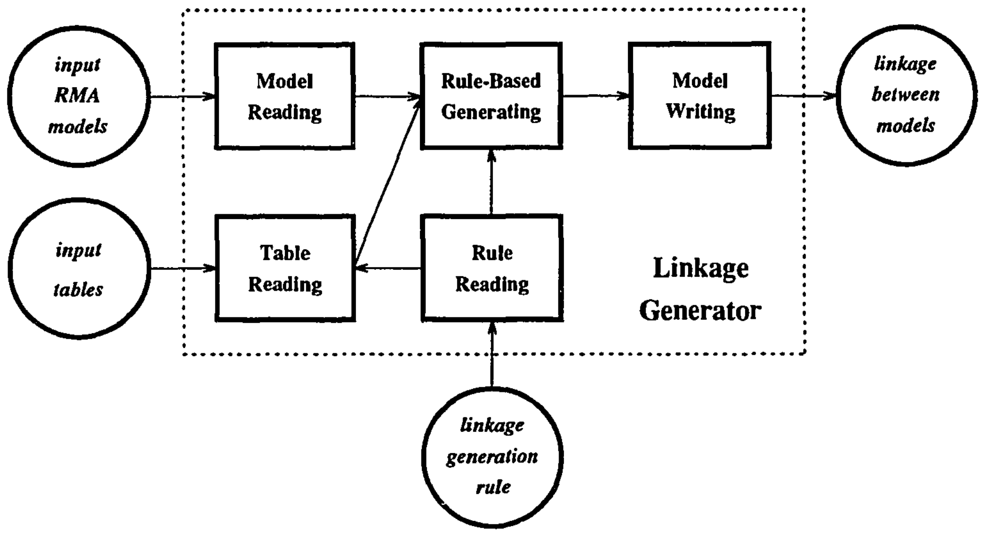


Figure 6.20: Architecture of *Linkage Generator*

types of graphs consists of two components – table declaration and generation rules. The table declaration component is optional. It indicates additional tables required for linkage generation. Each entry in the table declaration has a field indicating the type of table required, and an optional field to indicate the name of the file that contains the table if the name does not follow the name convention defined in Proud.

At the beginning of each rule the desired types of source and destination *RMA* graphs are described. Linkage between these types of graphs will be established.

Each rule consists of several link-definitions to restrict the links to be created. Objects for linkages can be objects or entire graphs. The definition consists of the following parts:

- Type of linkage that should be established.
- The source and destination objects. It can take any of the following values:

  GRAPH indicates that the object is a graph.

  ITEM indicates that the object is an entity.

  RELATION indicates that the object is a relation.

  ANY indicates that the object is either an entity or a relation.

  STRING LIST indicates that the object is either an entity or a relation whose type matches one string in the string list.

- A set of conditions connected by logical operators serves as the constraint for establishing the linkage. A linkage will be established if the conditions are satisfied. Each condition consists of three parts – left expression, right expression, and comparison operator. An expression is a arithmetic expression. The element in an expression can be:
  - A constant – integer or string.
  - A value from an object in a graph. Keyword "SRC" indicates the source graph and "DST" indicates the destination graph. The value can be the name of the graph, indicated by keyword "GRAPH", the name of the current object, indicated by keyword "NAME", the type of the current object, indicated by keyword "TYPE", or a value of an attribute of the current object, indicated by a string representing the name of the attribute.
  - A value from a field in an entry of a table. The table is indicated by its name. If the keyword "SAME" is presented, the entry used last time will be used again. The field is indicated by its name.
- A set of attributes to be attached to the generated links. Each attribute is defined by two arguments – the type of the attribute and its value. The attributes are optional.

6.8.3 An Example for Linkage Generating Process

The following shows the Linkage Generation Rules for IOPM2 and DM2.

```
TABLE(DEF)
TABLE(REF)

LINK(("IOPM2", "DM2")
    ("refer", "Process" OR "Branch", ENTITY,
     REF:NODE_ID = SRC:NODE_ID & SAME:DATA_NAME = DST:NAME)
    ("define", "Process" OR "Branch", ENTITY,
     DEF:NODE_ID = SRC:NODE_ID & SAME:DEST_NAME = DST:NAME))
```

The first two lines in the rules indicate that two tables are required – a *data definition table* and a *data reference table*. A **refer** linkage is established from a **Process** or a **Branch** in an IOPM2 graph to an entity in a DM2 graph, if there is an entry in the *data reference table* such that the value of the NODE_ID field in the entry equals the value of NODE_ID attribute of the entity in the IOPM2 graph and the value of the DATA_NAME field in the same entry equals the name of the entity in the DM2. A **define** linkage is established from a **Process** or a **Branch** in an IOPM2 graph to an entity in a DM2 graph, if there is an entry in the *data definition table* such that the value of the NODE_ID field in the entry equals the value of NODE_ID attribute of the entity in the IOPM2 graph and the value of the DEST_NAME field in the same entry equals the name of the entity in the DM2.

For the following portion of the source code of an airline reservation program:

```
...
WORKING-STORAGE SECTION.
01  RESERVATIONS-PROGRAM-WORK.
  05  RP-END-OF-TRANS          PIC X VALUE ZERO.
  05  RP-END-OF-RESERV         PIC X VALUE ZERO.
  05  RP-INVALID-RECORD        PIC X VALUE ZERO.
01  EDIT-WORK.
  05  EW-THIS-FLIGHT-NUM.
    10  EW-THIS-CREATION       PIC X.
    10  EW-THIS-FLIGHT         PIC 9(6).
  05  EW-THIS-FN REDEFINES EW-THIS-FLIGHT-NUM.
    10  EW-THIS-FLIGHT-NUMBER  PIC 9(7).
  05  EW-FLIGHT-NUMBER         VALUE ZERO.
    10  EW-CREATION            PIC X.
    10  EW-FLIGHT              PIC 9(6).
  05  EW-FN REDEFINES EW-FLIGHT-NUMBER.
    10  EW-LAST-FLIGHT-NUMBER       PIC 9(7).
...
IF EW-THIS-FLIGHT-NUMBER NOT > EW-LAST-FLIGHT-NUMBER
  MOVE '1' TO RP-INVALID-RECORD
ELSE
  MOVE EW-THIS-FLIGHT-NUMBER TO EW-LAST-FLIGHT-NUMBER.
...
```

the corresponding *data reference table* and *data definition table* are shown below:

```
TABLE(DATA_DEFINITION,5)
LINE_NO:INTEGER,DEST_NAME:STRING,NODE_ID:INTEGER,SRC_TYPE:STRING,
   SRC_NAME:STRING
162,"RP-INVALID-RECORD",468,"CONSTANT","'1'",
164,"EW-LAST-FLIGHT-NUMBER",473,"VARIABLE","EW-THIS-FLIGHT-NUMBER",


TABLE(DATA_REFERENCE,3)
LINE_NO:INTEGER,DATA_NAME:STRING,NODE_ID:INTEGER
161,"EW-THIS-FLIGHT-NUMBER",476,
161,"EW-LAST-FLIGHT-NUMBER",476,
162,"'1'",468,
164,"EW-THIS-FLIGHT-NUMBER",473,
```

Figure 6.21 and Figure 6.22 show the corresponding IOPM2 graph and DM2 graph. Figure 6.23 shows the linkage between these two graphs.

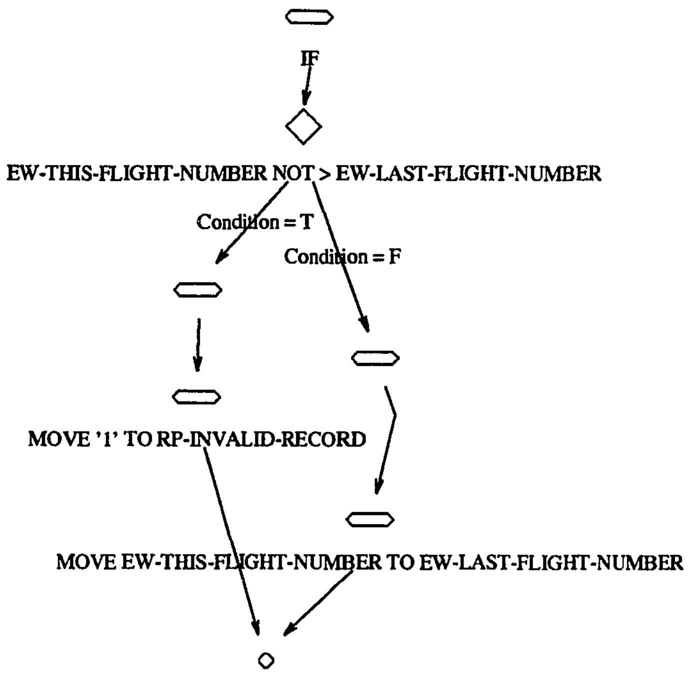

IF
EW-THIS-FLIGHT-NUMBER NOT > EW-LAST-FLIGHT-NUMBER
Condition = T
Condition = F
MOVE '1' TO RP-INVALID-RECORD
MOVE EW-THIS-FLIGHT-NUMBER TO EW-LAST-FLIGHT-NUMBER

Figure 6.21: Example IOPM2 Graph for Linkage Generating Process

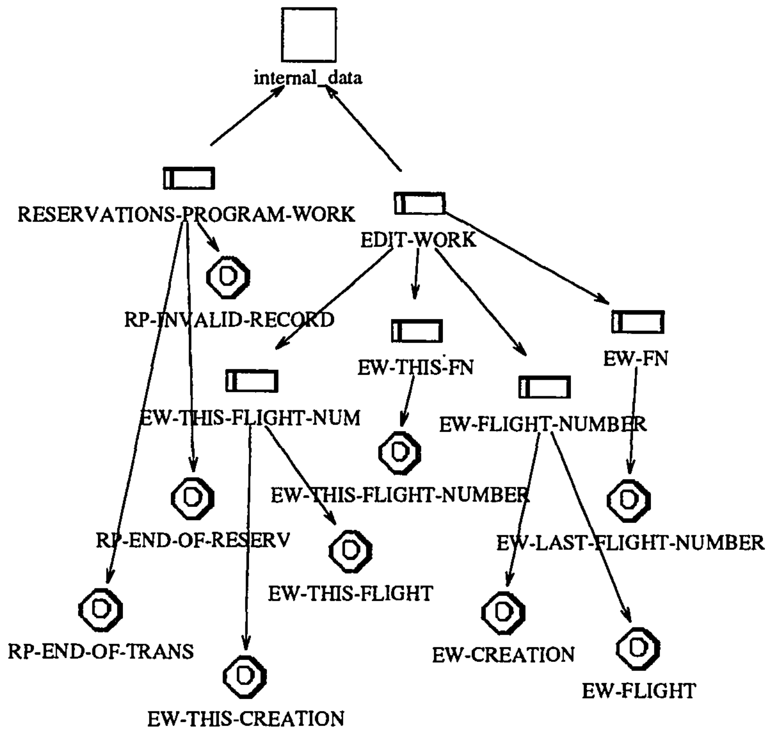


Figure 6.22: Example DM2 Graph for Linkage Generating Process

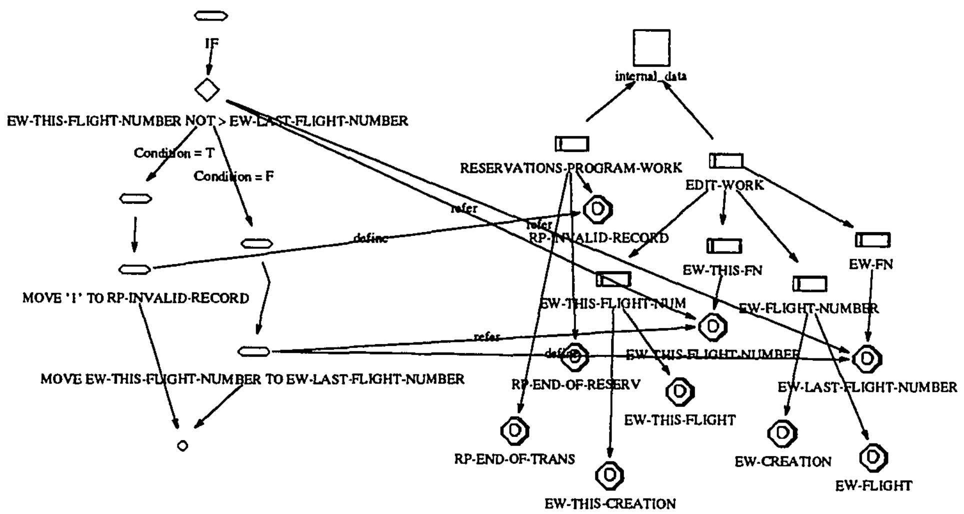


Figure 6.23: Example Linkage Graph for Linkage Generating Process

Chapter 7

Example

In order to illustrate the internal process of *Proud*, this chapter shows how Proud processes example queries shown in Chapter 1 that extract information from a target system. The scenario is stated in Chapter 1 and restated here: maintenance of an airline reservation program is assigned to a programmer who is not familiar with the program. The programmer needs to study the program and uses Proud to extract various information from the source code of the program. The source code is given in Appendix B.1.

7.1 Initialization

Planner in Proud must be initialized before Proud accepts queries from users. There are two steps in the initialization – loading knowledge related to tools and loading name reference tables. The knowledge related to tools includes table generation rules, model generation rules, generic plans for tools, etc. There are three tables used for name references – data declaration table, program structure table, and calling structure table. Since these tables do not exist initially, the *Planning* component of *Planner* initiates a process to generate these tables. *Planning* sends a request to *Information Base* to get a plan for generating the data declaration table via the following function call:

```
(plan-proposal 'sample 'dec)
```

Information Base finds that the *Table Generating* generic plan can be used to generate the required table. *Information Base* creates an instance graph of the plan and assigns values into the attributes of the instance graph that are unspecified in the generic plan. Figure 7.1 shows the instance graph.

Information Base finds that `sample.tree`, which is one of the inputs for *Table Generating* task, does not exist. It goes through the *Planning* process again and cre-

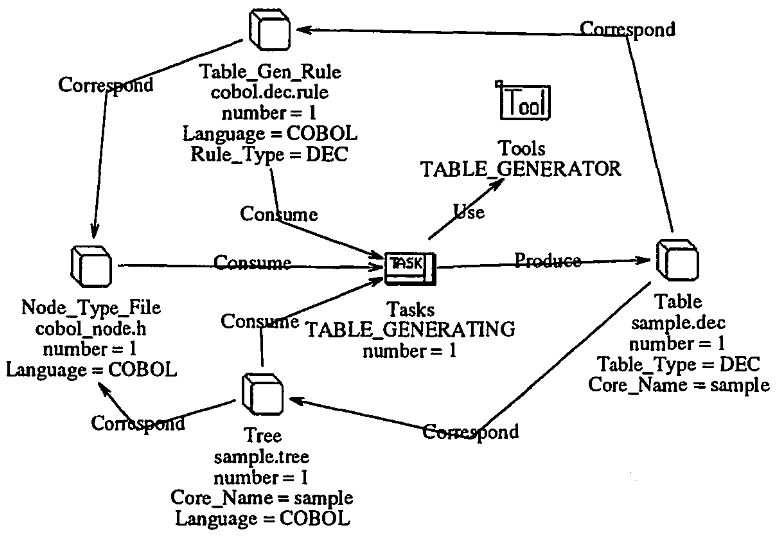


Figure 7.1: Instance Plan for Generating Data Declaration Table

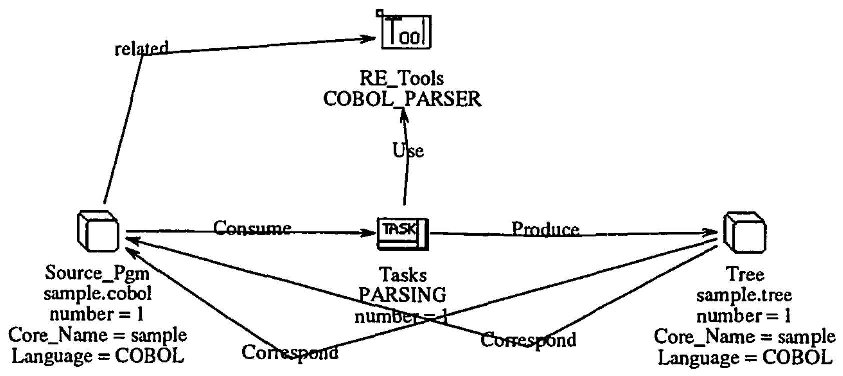


Figure 7.2: Instance Plan for Generating Syntax Tree

ates another instance graph for generation of `sample.tree` from the *Parsing* generic plan. Figure 7.2 shows the resultant instance graph. Since the source code file is written in COBOL, only the constraint of the **related** relation from Source_pgm to COBOL_PARSER is satisfied, so the other **RE_Tools** entities are removed from the instance graph.

Information Base returns these two tasks to *Planning*. *Planning* passes the *Parsing* task to *Tool Controller* first. *Tool Controller* translates the task into real Unix commands by making the following function call:

```
(unix-command-gen task-parse0 'sample)
```

The function returns the Unix command:

```
cobol_parser sample.cobol
```

This command is executed by the *COBOL Parser* and produces the syntax tree representing `sample.cobol`. The syntax tree is stored in a file named `sample.tree`. The syntax tree is shown in Appendix B.2.

After the syntax tree file `sample.tree` is generated, *Tool Controller* registers an object that represents the tree file into *Information Base* with the following function call:

```
(register-object 'sample 'tree :lang 'COBOL)
```

It informs *Information Base* that the tree file exists and does not need to be generated again unless the source code file is modified. The registration procedure is done for every piece of information generated by a tool.

Then, the second task is passed to *Tool Controller* for generating the data declaration table. The task is translated into the command:

```
table_generator -t sample -r cobol.dec.rule
```

The data declaration table is generated and stored in a file named `sample.dec`. The contents of the data declaration table are shown in Appendix B.3.1. After the table is generated, it is registered in *Information Base.*

The processes of generating the program structure table and the calling process table are similar to the process of generating the data declaration table, except only one instance graph of a task is generated for each process because the syntax tree exists. Instance graphs for these tasks are shown in Figures 7.3 and 7.4. These two tables are stored in files named `sample.ps` and `sample.cs`, respectively. The content of the program structure table is shown in Appendix B.3.4 and the content of the calling structure table is shown in Appendix B.3.5. After the three tables are generated and loaded into Proud, Proud is ready for processing incoming queries from clients (either end users or other components in the SMA system).

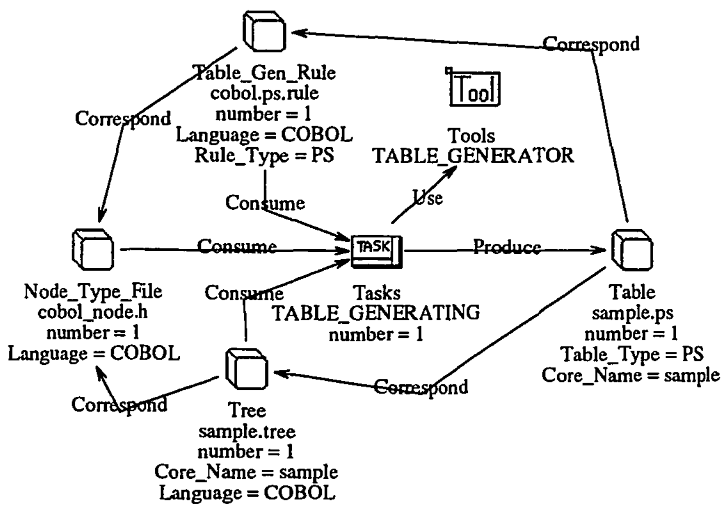


Figure 7.3: Instance Plan for Generating Program Structure Table

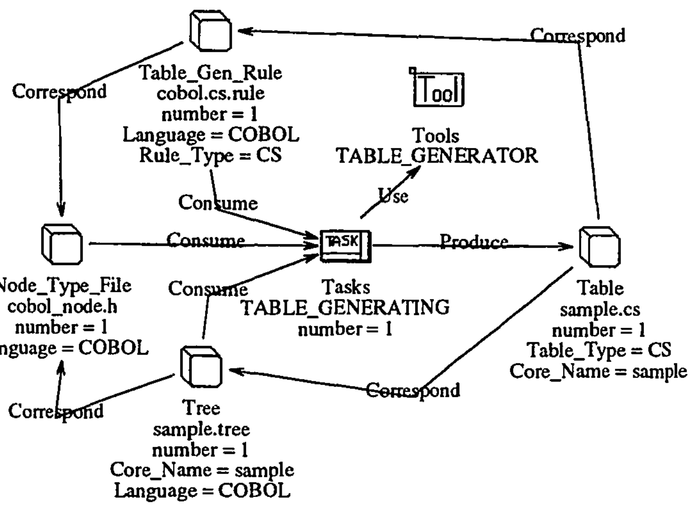


Figure 7.4: Instance Plan for Generating Calling Structure Table

7.2 Generation of CPM Graphs by Reverse Engineering Tools

The programmer knows the name of the program is "RESVPG" and he wants to see the overall data-flow of the program. He issues a query shown in Figure 1.1 to Proud. This query requests Proud to return a data-flow diagram (called CPM in Proud) that contains a task named "RESVPG".

Query Interpreter accepts this query and checks goals of the query by using the *Checker* component. *Checker* returns the following information: the operation is *retrieve all*; the META graph is *CPM* because only CPM in RMA models contains a **task** class object; the program is `sample` because one entry in a program structure table of *sample* matches the name of one entity in the query; and the graph is `sample@RESVPG`. The information is passed to the *Planning* component. *Planning* checks the existence of the graph `sample@RESVPG` by asking *Information Base*. *Information Base* returns a list of tasks to generate the graph, since it is not in OB. The tasks in the list are – generating data reference table, generating data definition table, and generating CPM model. Figures 7.5, 7.6, and 7.7 show instance graphs of these tasks, respectively. The process of creating this list is similar to that of generating the data declaration table described above. Then *Planning* passes the tasks to *Tool Controller* which translates these tasks into the following real Unix commands in order to generate the desired information:

```
table_generator -t sample -r cobol.ref.rule
table_generator -t sample -r cobol.def.rule
model_builder -t sample -m cpm
```

The data reference table, the data definition table, and the CPM graphs are shown in Appendices B.3.2, B.3.3, and B.4, respectively.

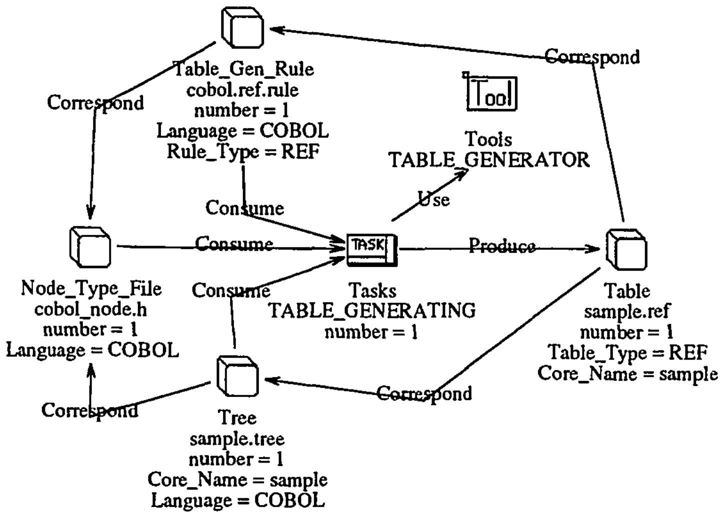


Figure 7.5: Instance Plan for Generating Data Reference Table

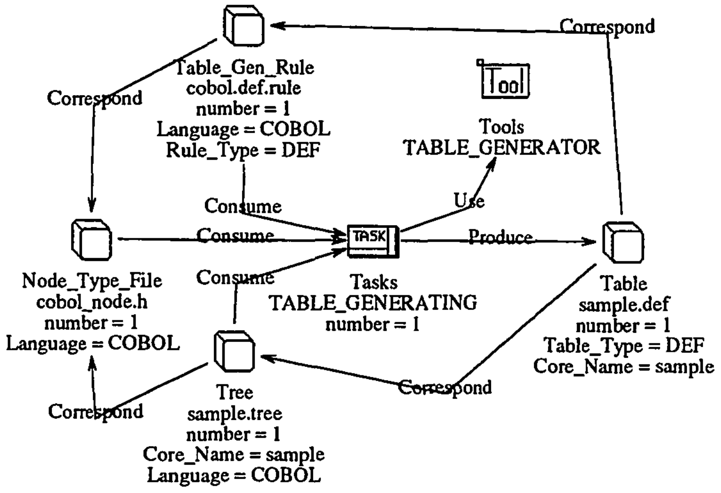


Figure 7.6: Instance Plan for Generating Data Definition Table

When the CPM graphs are generated, *Planning* issues a command to *OB Access Controller* in order to store these graphs into OB. Meanwhile, objects representing these graphs are registered in *Information Base*. *OB Access Controller* consists of two parts – the Lisp version and the C version. The C version serves as interface between Proud and OB and is invoked by the Lisp version via Unix commands. The Unix command from the Lisp version of *OB Access Controller* to the C version for this storing operation is:

```
ob_controller -c store-all sample.CPM.cdf
```

Then, *Planning* sends a retrieval command to *OB Access Controller* in order to retrieve the graphs that match retrieval criteria in the incoming query. *OB Access Controller* converts the retrieval command into OBGQ (Object Base Graph Query) and sends it to OB. The OBGQ is shown in Figure 7.8 and the Unix command from the Lisp version of *OB Access Controller* to the C version for this retrieval operation is:

```
ob_controller -c retrieve-all query.cdf
```

where `query.cdf` is the CDF file containing the OBGQ.

OB returns the matched graph to Proud. Figure 7.9 shows the retrieved graph. Since there is no other requirement specified by the incoming query, the graph is returned as the final result of the incoming query, as shown in Figure 1.2.

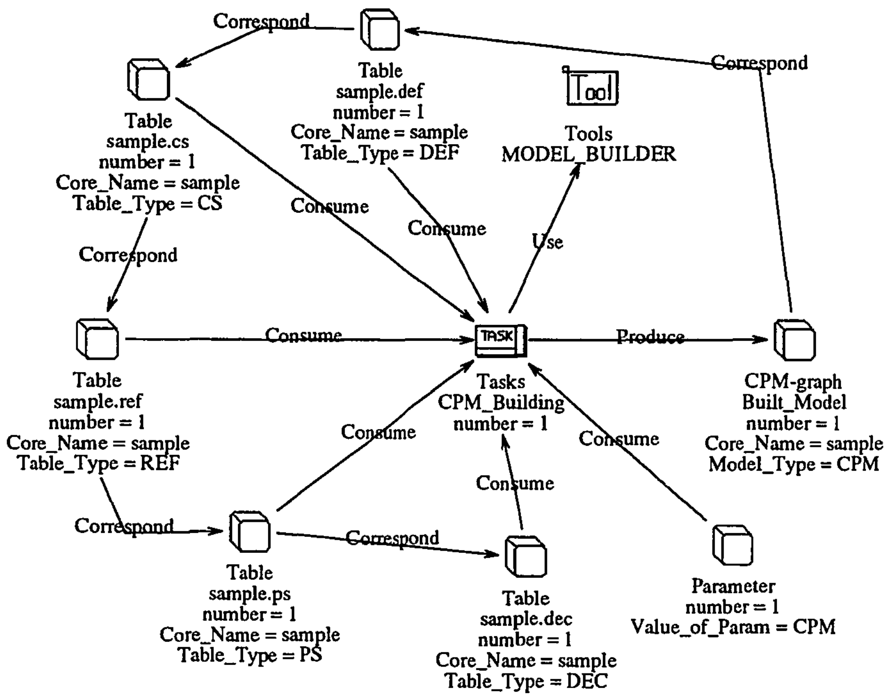


Figure 7.7: Instance Plan for Building CPM Model

Figure 7.8: OB Retrieval Pattern for the First Query

7.3 Retrieval of Existing Graphs from OB

After viewing a top-level data-flow diagram, the programmer wants to see the next level data-flow diagram. He issues another query, shown in Figure 1.3, to Proud. The string `RESVPG.*` in the query represents a regular expression for pattern matching. It matches any string starting with `RESVPG` followed by 0 or more characters.

Query Interpreter accepts the query and sends the following information to *Planning*: the operation is *retrieve all*; the META graph is *CPM*; the program is `sample`; and the graph is `sample@RESVPG`. The name of the graph sent by *Query Interpreter* is used as a reference rather than a retrieval criterion. Since all graphs of one type for a source code file is generated at once, the existence of one graph implies the existence of all graphs of this type. *Planning* checks with *Information Base* and finds that the

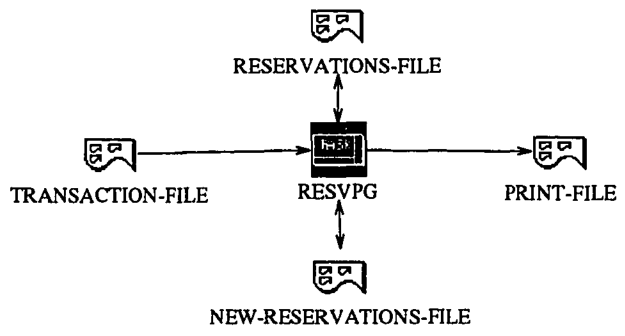


Figure 7.9: Result of OB Retrieval Operation for the First Query

TASK
RESVPG.*

Figure 7.10: OB Retrieval Pattern for the Second Query

graph indicated by *Query Interpreter* exists in OB. Thus no reverse engineering tool needs to be invoked for generating the graph. *Planning* sends a retrieval command to *OB Access Controller*. The command is converted into an OBGQ. The OBGQ is shown in Figure 7.10 and the Unix command from Lisp version of *OB Access Controller* to C version for this retrieval operation is:

```
ob_controller -c retrieve-all query.cdf
```

where `query.cdf` is the CDF file containing the OBGQ.

There are two graphs returned by OB. One is shown in Figure 7.11 and the other is shown in Figure 7.9. Both of them are returned as the answer to the query.

7.4 Generation of IOPM2 graphs by Reverse Engineering Tools

After the data flow graphs are examined, the programmer decides to examine the program in detail. He wants to study a portion of the program that invokes the "CRE-ATE' module. To do so, he issues a query to retrieve control flow diagrams (called IOPM2 in Proud) that contain the **Transition** entity named "CRE-ATE". The query is shown in Figure 1.5.

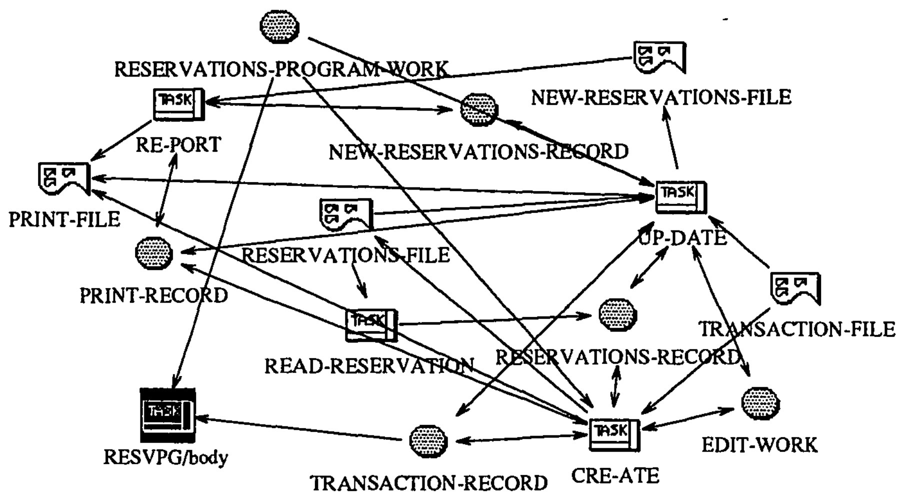


Figure 7.11: Result of OB Retrieval Operation for the Second Query

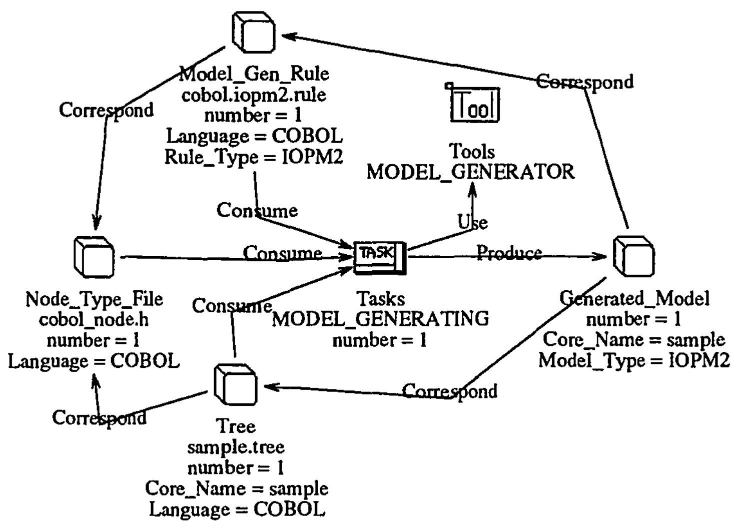


Figure 7.12: Instance Plan for Generating IOPM2 Model

Query Interpreter accepts the query and sends the following information to *Planner* – the operation is *retrieve all*; the META graph is *IOPM2*; the program is `sample`; the graph is `sample@RESVPG`. By asking *Information Base*, *Planning* finds that the graph does not exist in OB. *Information Base* provides a plan for generating the required model. There is one task in the plan – the model generating task. Figure 7.12 shows the instance graph of the task. This task uses a model generation tool to generate the IOPM2 graphs. It requires three inputs: a syntax tree named `sample.tree`, a model generation rule file named `cobol.iopm2.rule`, and a node type definition file named `cobol_node.h`. *Planning* passes the task to *Tool Controller*, which translates the task into the following real Unix command:

```
model_generator -t sample -r cobol.iopm2.rule
```

The generated IOPM2 graphs are shown in Appendix B.5. These graphs are registered in *Information Base* and stored in OB. The Unix command from the Lisp version of *OB Access Controller* to the C version for storing the generated IOPM2 graphs in OB is:

```
ob_controller -c store-all sample.IOPM2.cdf
```

Then, *Planning* sends a retrieval command to *OB Access Controller* in order to retrieve the graphs that contain the **Transition** entity named "CRE-ATE". Since the **Transition** entity is the superclass of **Invoke, Process, Fork** and **Wait**, the original retrieval pattern is duplicated by replacing the **Transition** entity with each of its subtype entity. *OB Access Controller* converts the command into five OBGQs, shown in Figure 7.13 to Figure 7.17. The Unix command from the Lisp version of *OB Access Controller* to the C version for this retrieval operation is:

```
ob_controller -c retrieve-all query.cdf
```

The command is repeated five times for each of the OBGQ.

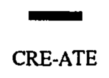


Figure 7.13: Original OB Retrieval Pattern for the Third Query

Only one graph is returned by OB. It is shown in Figure 7.18. This graph is returned as the answer for the query.

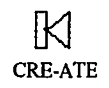


Figure 7.14: OB Retrieval Pattern of **Invoke** for the Third Query

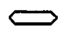


Figure 7.15: OB Retrieval Pattern of **Process** for the Third Query

Figure 7.16: OB Retrieval Pattern of **Fork** for the Third Query

Figure 7.17: OB Retrieval Pattern of **Wait** for the Third Query

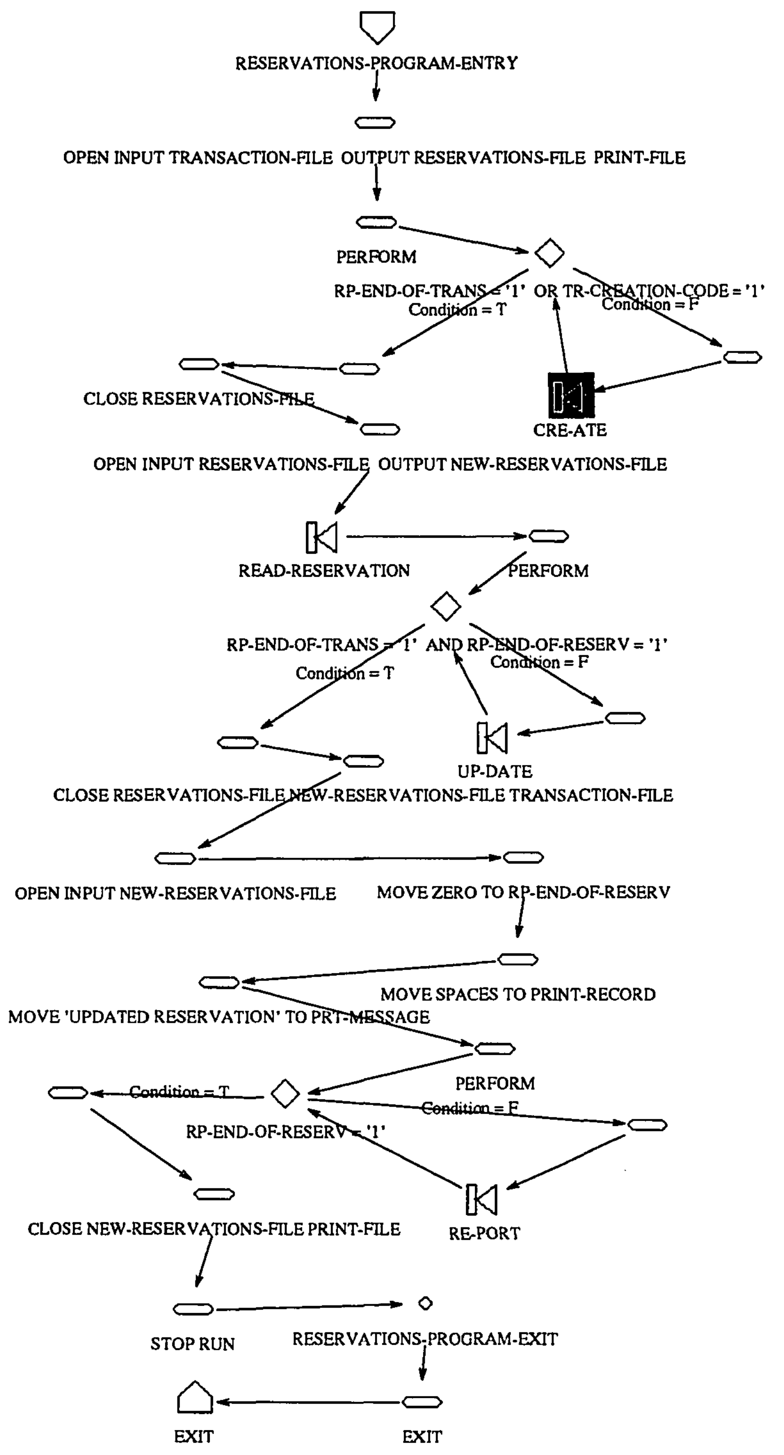

RESERVATIONS-PROGRAM-ENTRY
OPEN INPUT TRANSACTION-FILE OUTPUT RESERVATIONS-FILE PRINT-FILE
PERFORM
RP-END-OF-TRANS = '1' OR TR-CREATION-CODE = '1'
Condition = T
Condition = F
CLOSE RESERVATIONS-FILE
CRE-ATE
OPEN INPUT RESERVATIONS-FILE OUTPUT NEW-RESERVATIONS-FILE
READ-RESERVATION
PERFORM
RP-END-OF-TRANS = '1' AND RP-END-OF-RESERV = '1'
Condition = T
Condition = F
UP-DATE
CLOSE RESERVATIONS-FILE NEW-RESERVATIONS-FILE TRANSACTION-FILE
OPEN INPUT NEW-RESERVATIONS-FILE
MOVE ZERO TO RP-END-OF-RESERV
MOVE SPACES TO PRINT-RECORD
MOVE 'UPDATED RESERVATION' TO PRT-MESSAGE
PERFORM
Condition = T
Condition = F
RP-END-OF-RESERV = '1'
CLOSE NEW-RESERVATIONS-FILE PRINT-FILE
RE-PORT
STOP RUN
RESERVATIONS-PROGRAM-EXIT
EXIT
EXIT

Figure 7.18: Result of OB Retrieval Operation for the Third Query

Chapter 8
Conclusion

The Proud system is implemented on Sun Workstations. The reverse engineering tool set is implemented in the C programming language and runs on both Unix workstations and Macintosh personal computers. The PU-Planner is implemented in CLOS and runs on Unix workstations with support of Allegro Common Lisp. The system has been delivered to Fujitsu Limited, the sponsor of the SMA project, along with other components of SMA project. The reverse engineering tool set is used as front-end processor for generating of graphs in another project LDS (Logical Design Specification). The generated graphs are used by the other components in LDS to create a textual design document from source code.

8.1 Uniqueness

Compared to other systems in reverse engineering, Proud has the following advantage for supporting maintenance activities:

- It is an integrated system. The system provides a set of tools to do reverse engineering and program analysis (implemented by other members in SERL), a facility manager to managing these tools, and a graphic query language to access the information generated by the tools. This combination gives Proud more power than other systems in terms of process capability, user friendliness, and extensibility.
- It is an open system. It allows any tool to be integrated into the system, as long as the data to be processed by the tool is represented in the MERA language. Normally, the work to be done to add a new tool is to create a set of graphs to capture the knowledge regarding functionalities of the tool. Each graph represents a function to be provided by the tool, including the input data the function requests and the data the function produces.

- It can process industrial strength software systems. The system uses a create on demand policy to generate required information for users, so it does not require a large storage even if the target system is large. Most of other reverse engineering systems require all information associated with the target system to be generated in order to do the next process. This will cause a problem when industrial strength software systems are processed since most of them are large in size.
- It is easily customized to fit the needs of the programmers. The reverse engineering tool set demonstrates its flexibility in the LDS project. In that project, the graphs to be generated represent different information and use different structures than their counterparts in the RMA models. Only the generation rules associated with the graphs to be used in LDS were modified for the tool set to generate the required information. No modification was done to the reverse engineering tool set itself.
- It creates inter-graph linkages. Inter-graph linkages enable the user to explore the relationship between different aspects of the program and he/she can get a clearer view of the program. Also, inter-graph linkages can be used for other analyses of the program, such as impact analysis.

8.2 Contribution

This research has the following contributions for the field of reverse engineering and software maintenance:

- A flexible approach for generating information from source code that meets the requirements of maintainers. This will benefit the maintenance process since the maintainers can obtain information suitable for their maintenance tasks.
- A knowledge-based approach for tool integration. It provides a mechanism for representing and organizing knowledge about individual tools. By using this knowledge, a tool will be automatically invoked if it is needed. This will reduce the effort required of maintainers to obtain the required information for their tasks.
- A knowledge representation framework for multi-function tools. The knowledge representation used in Proud captures each primitive function provided by a tool separately, not all functions the tool provides. This approach reduces the

difficulty of capturing knowledge about a powerful tool. Also, this approach provides a object-oriented way for representing knowledge of tools.

- A non-command-based approach for accessing information. By providing a graphical query language, it allows users to specify the information they require instead of instructing the system to obtain the desired information step by step.

8.3 Limitations

There are some limitations in Proud. The rule-based approach and plan-based approach used in Proud contributes to most of them. But these approaches brings far more benefits, as discussed in previous chapters, compared to the limitations they cause.

The limitations of the rule-based approach are:

Expressive Power The expressive power of a rule-based application purely depends on the expressive power of the language to express the rule. If the language is too simple, the usage of the application is limited. On the other hand, if the language is too complicated, then it is very difficult to verify the correctness of the rule and efficiency of the application may be very low. So to design this kind of language, it is very important and very difficult to find a balance point between efficiency and effectiveness. Since no effective method has been found to verify the correctness of the generated rules, the design of rule-based applications in Proud favors simple languages for easy writing of rules and walk-through checking. So the expressive power of current Proud rule languages is not high and can only generate limited types of information.

Rule Verification As mentioned above, no effective method has been found to verify the correctness of rules, so the quality of rules heavily depends on the experience of the authors. For a set of complicated rules, several rounds of trial and error may be needed to verify the correctness of the rules. So it is recommended not to write complicated rules and to keep each rule as short and clean as possible.

Efficiency Even though the current languages for Proud rule-based applications are quite simple, efficiency is still a problem in some cases, especially for linkage generation. Since generalization is important in the rules, no efficient search algorithm can be used in table search. So the execution time is linearly proportional to the following items for linkage generation: the number of graphs involved, the number of objects in each graph, the number of tables involved, the number of entries in each table, and the number of rules involved.

The limitations of the plan-based approach are:

Knowledge Representation Knowledge representation is the major factor limiting the functionality of Proud. The state of art in knowledge representation is still not advanced enough to represent complicated knowledge. Only a small set of specific knowledge can be represented by the current technology.

Knowledge Base The capability of Proud depends on the knowledge captured in its knowledge base. Due to limited experience in software maintenance practice, knowledge in the current knowledge base in Proud is very limited. This restricts the functionality and performance of Proud. Currently only the knowledge regarding reverse engineering tools is captured in the knowledge base.

Query Language Query language is the interface of Proud to communicate with users. Only a few types of queries can be represented in the current query language (impact analysis, pattern retrieval). The reasons are more time and resources are needed to explore more samples.

8.4 Future Research

The direction of future research will focus on the following areas:

Knowledge about Tool Research in this area will focus on knowledge representation regarding more general tools, including constraints on input and output data, relationships among the tools, etc.

Query Language Research in this area will focus on the expressive power of the language and algorithms for graphic pattern matching.

Generation Tool Research in this area will focus on enhancement of generation rules for more expressive power, a methodology for verification of rule correctness, and the possibility of using other representation formats such as decision tables, etc.

Abstraction Research in this area will focus on generation of high level models, such as system level control flow and data flow.

Appendix A

RMA Models Generated by Proud

RMA (Requirements Modification and Analysis) models are designed for representing different aspects and properties of a software system. Every RMA model has a META graph called a formalism that defines the syntax of instance graphs. Proud generates graphs in following RMA model types:

IOPM Internal Operational Process Model,

IOPM2 new version of IOPM,

EOPM External Operational Process Model,

CPM Conceptual Process Model,

FM Functional Model,

FSM Function Structure Model,

DM Data Model,

DM2 new version of DM,

UIM User Interface Model, and

PSTM Program STructure Model.

The formalisms for these type of graphs are constructed from a TOP_LEVEL formalism. This chapter briefly describes the formalisms of IOPM2, EOPM, CPM, FM, FSM, UIM, DM, and PSTM, plus the TOP_LEVEL formalism. For more detailed information about RMA models, please refer [43].

A.1 TOP_LEVEL formalism

Every formalism automatically inherits attributes defined in the TOP_LEVEL formalism. There are three objects in TOP_LEVEL: **Entity**, **Relation**, and **Object**. **Entity** and **Relation** are subtypes of **Object**. **Entity** and **Relation** are supertypes of all Entity_type_definition's and Relation_type_definition's in all formalisms. Figure A.1 shows the definition of the TOP_LEVEL formalism.

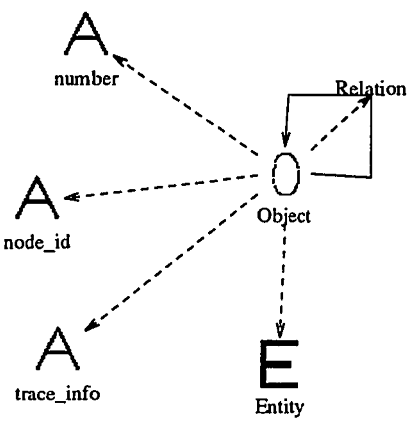


Figure A.1: TOP_LEVEL Formalism

A.2 IOPM2 Formalism

IOPM2 is used to describe the internal behavior of a system (control flow). IOPM2 is based on Petri Nets extended to describe more detailed information about the internal behavior of the system. Entities defined in IOPM2 can be divided into two groups – **Place** and **Transition**, which correspond to place and event in Petri Nets, respectively. There are four subtypes of **Place** – a **Ready_indicator** is a token holding place, a **Branch** is an intersection, an **Entry** is an entry point, and an **Exit** is an exit point. There are four subtypes of **Transition** – a **Process** represents a normal event, a **Fork** represents a starting point of parallel processing, a **Wait** represents a synchronization point of parallel processing, and an **Invoke** represents a point where another graph is invoked.

Four types of **Relation** are defined – a **Producing_arc** represents a flow from a **Transition** to a **Place**, a **Consuming_arc** represents a flow from a **Place** to a **Transition**, an **Exec_next** represents a flow from a **Transition** to a **Transition** and it is an abbreviation of a **Producing** to a **Place** and a **Consuming** from the **Place**, and a **Null_process** represents a flow from a **Place** to a **Place** and it is an abbreviation of a **Consuming** to a null **Transition** and a **Producing** from the **Transition**. Figure A.2 shows the formalism of IOPM2.

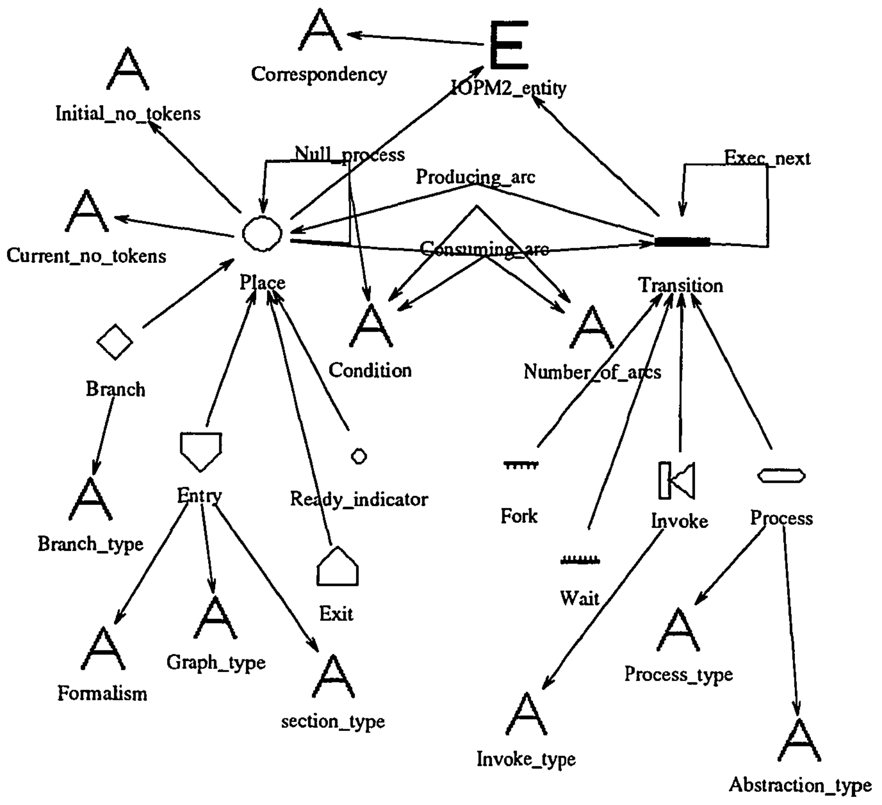


Figure A.2: IOPM2 Formalism

A.3 EOPM Formalism

EOPM is used to describe the external behaviors of a system. It is based on a transition diagram. There is only one type of **Entity** – **state** representing the state of the system and one type of **Relation** – **becomes** representing the transition conditions from one state to another. A **state** may has an attribute *factor* that represents the status of the **state**. A *becomes* has an attribute *events* that represents the condition to be satisfied. Figure A.3 shows the formalism of EOPM.

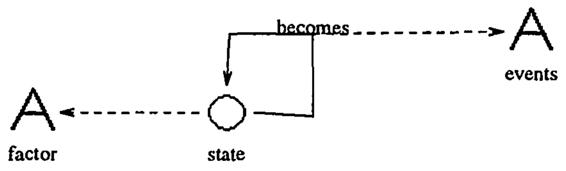


Figure A.3: EOPM Formalism

A.4 CPM Formalism

CPM is used to describe the overall system operations. It is an extended data flow model. Eight types of **Entity** are defined - a **task** represents an information processing task; a **datastore** represents temporary storage for data; an **internal_data** represents data that is internal to a computer system; an **external_data** represents data that is external to the computer system; an **external_data_element** represents data that is part of a **external_data**; a **database_system** represents a subsystem containing a database; a **section** represents an autonomous information processing unit; and **data-entity** is the supertype of **database_system**, **external_data_item**, **external_data**, **internal_data**, and **datastore**.

The types of **Relation** are - an **output** represents that a **task** outputs to an **internal_data**; an **input** represents that an **internal_data** is used as input by a **task**; a **generate** represents that a **task** generates an **external_data** or an **external_data_element**; a **receive** represents that an **external_data** or an **external_data_element** is used as input by a **task**; a **retrieve** represents that a **task** receives some information from a **database_system**; an **update** represents that a **task** updates a **database_system**; a **read** represents that a **task** reads from a **datastore**; a **write** represents that a **task** writes to a **datastore**; a **perform** represents that a **section** performs a **task**; a **manage** represents that a **section** manages a **database_system**; an **own** represents that a **section** owns an **external_data**; and a **partition** represents that a **datastore** is part of another **datastore**, an **external_data_element** is a part of an **external_data**, or an **internal_data** is a part of another **internal_data**. Figure A.4 shows the formalism of CPM.

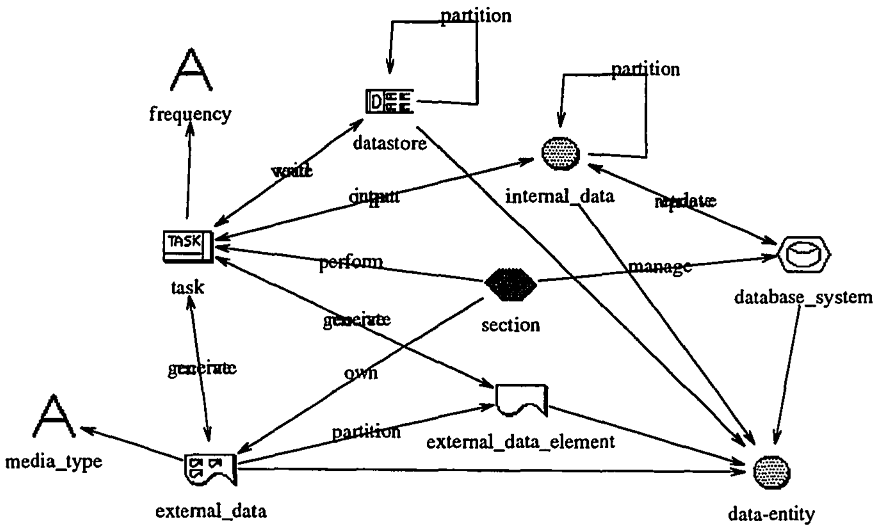


Figure A.4: CPM Formalism

A.5 FM Formalism

FM is used to describe the functional requirements of a target system. There are three pieces of information described in FM – input data, output data, and transformation from input data to output data. Eleven types of **Entity** are defined in FM and they can be divided into two groups – functional description and data description. The functional description **Entity** types are – a **function** represents a single functionality, a **sub_function** represents a subordinate function, a **description** describes the narration of the function, a **mechanism** describes a support mechanism, and an **attribute_set** represents attributes of the function.

The data description **Entity** types are – the **data_entity** represents the supertype of all other data description types, an **internal_data** represents a data item used within the function, an **input_data_set** represents a set of data items used as input data, a **sub_input_data** represents a set of data items used as input data for **sub_functions**, an **output_data_set** represents a set of data items produced by the function, and a **sub_output_data** represents a set of data items produced by **sub_functions**.

The types of **Relation** are – a **describes** represents that a **description** describes a **function**; a **specifies** represents that an **attribute_set** specifies a **function**; a **supports** represents that a **mechanism** is used to support a **function**;

an **is_input_to** represents that an **input_data_set** is used as input by a **function**, a **sub_input_data** is used as input by a **sub_function**, or an **internal_data** is used as input by a **sub_function**; an **output** represents that a **function** generates an **output_data_data** or a **sub_function** generates a **sub_output_data**; a **consists_of** represents that an **input_data_set** contains a **sub_input_data** or an **output_data_set** contains a **sub_output_data**, an **is_decomposed_into** represents that a **sub_function** is a part of a **function**, an **is_output_to** represents that a **sub_function** generates an **internal_data**. Figure A.5 shows the formalism of FM.

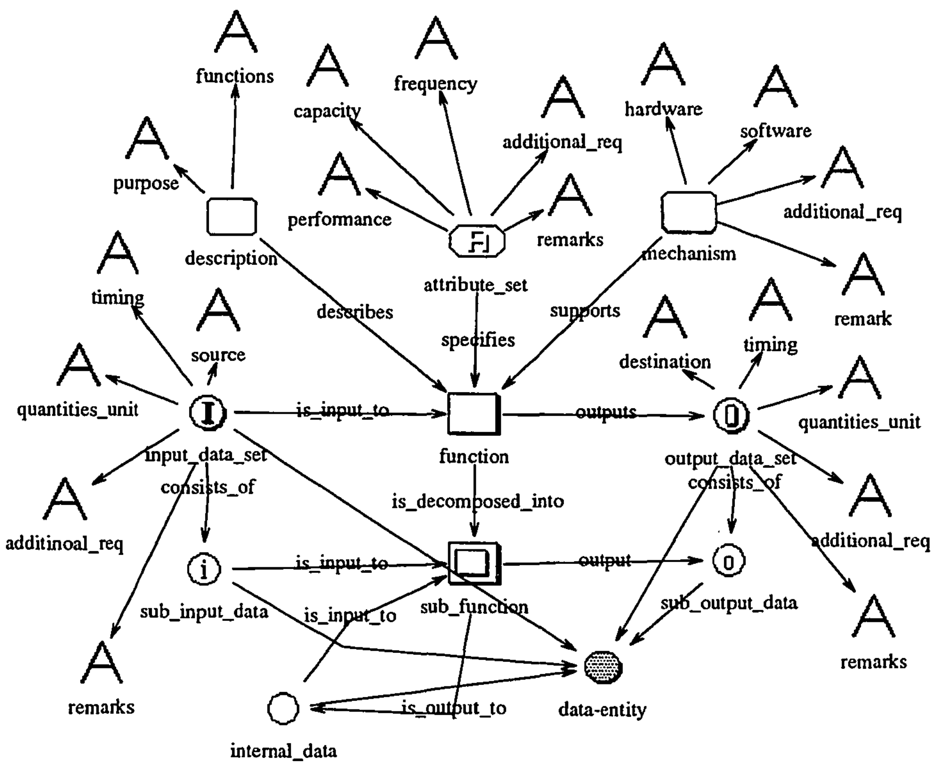


Figure A.5: FM Formalism

A.6 FSM Formalism

FSM is used to describe relationships among functions and relationships between functions and data objects. Three types of **Entity** are defined – an **actor** represents a system entity that is capable of taking actions on other entities, an **action** represents an operation an **actor** performs on an **object**, and an **object** represents a thing or collection of things.

Three types of **Relation** are defined – an **is_followed_by** represents an **action** is followed by another **action**, an **affects** represents an **action** affects an **object**, and a **takes** represents an **actor** takes an **action**. Figure A.6 shows the formalism of FSM.

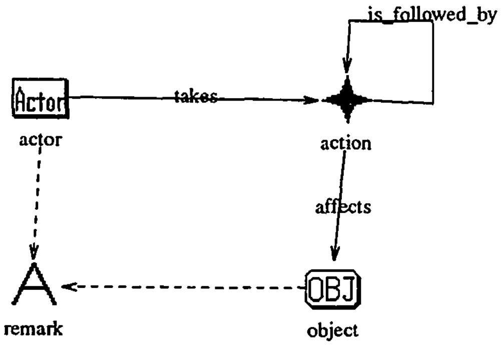


Figure A.6: FSM Formalism

A.7 DM2 Formalism

DM2 is used to describe data structures defined in a system. A **space** represents a place in the system to store a piece of data. A **data_entity** is the supertype of **file** and **data_unit**. A **file** represents a file. A **data_unit** is the supertype of **data_item** and **record**. A **data_item** represents a atomic data item. A **record** represents a collection of data items.

The types of **Relation** defined are – a **belong** represents that a **data_entity** is stored in a **space**, a **define** represents that a **data_unit** is defined in a **file**, a **redefine** indicates that a **data_unit** shares the same storage with another **data_unit**, a **contain** represents a **record** contains a **data_unit**. Figure A.7 shows the formalism of DM2.

A.8 UIM Formalism

UIM is used to describe the user interface of a system. The model consists of three sub-parts – behavior model, action list, and image. Proud generates only the be-

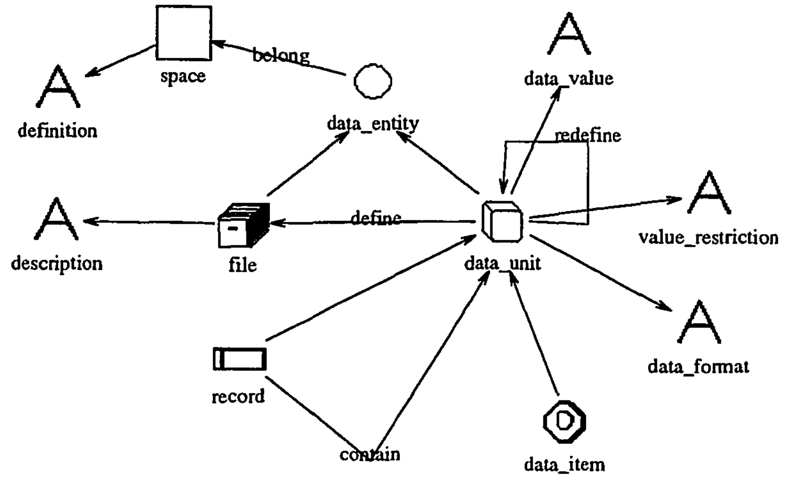


Figure A.7: DM2 Formalism

havior model. Two types of **Entity** are defined in the behavior model – **state** and **visual**. Three types of **Relation** are defined – a **transition** connects a **state** to another **state**, a **use** connects a **state** to a **visual**, and an **is_part_of** connects a **visual** to another **visual**. Figure A.8 shows the formalism of the behavior model of UIM.

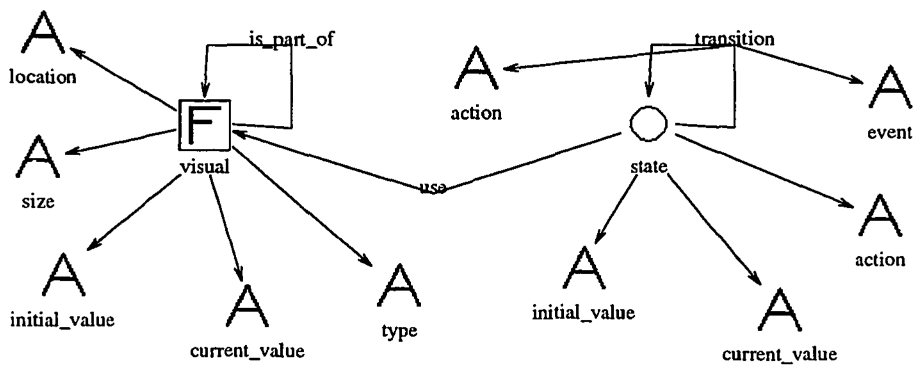


Figure A.8: UIM Formalism

A.9 PSTM Formalism

PSTM is used to describe the relationships between program code blocks and modules. Two types of **Entity** are defined in PSTM – a **block** represents a code block and a **module** represents one or more blocks invoked at once by a invoking type statement.

Three types of **Relation** are defined - a **contains** indicates a **block** contains another **block**, an **invokes** indicates a **module** invokes another **module**, and an **includes** indicates a **module** includes a **block** as part of the module. Figure A.9 shows the formalism of PSTM.

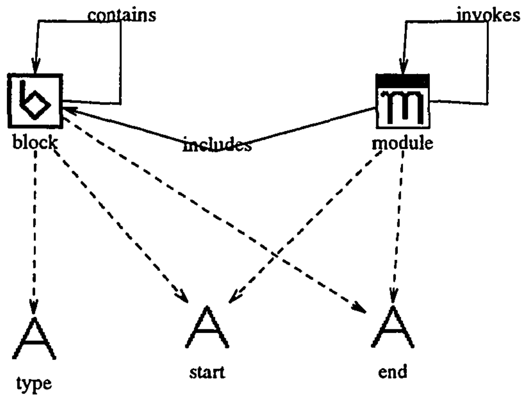


Figure A.9: PSTM Formalism

Appendix B

Generated Data for A Sample Program

This chapter lists the data generated by reverse engineering tools for a sample program. The sample program is an airline reservation system. Chapter 7 shows how the data is generated.

B.1 Source Code of the Sample Program

```
000000 IDENTIFICATION DIVISION.
       PROGRAM-ID.  RESVPG.
       AUTHOR.  MIYAMOTO.
      *
      *  THIS PROGRAM IS THE IMPLEMENTATION FOR THE RESERVATIONS
      *  PROGRAM.
      *
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER.
       OBJECT-COMPUTER.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT RESERVATIONS-FILE     ASSIGN TO MASS-STORAGE REF.
           SELECT TRANSACTION-FILE      ASSIGN TO MASS-STORAGE TR.
           SELECT PRINT-FILE            ASSIGN TO PRINTER.
           SELECT NEW-RESERVATIONS-FILE ASSIGN TO MASS-STORAGE NR.
       DATA DIVISION.
       FILE SECTION.
       FD  RESERVATIONS-FILE
           LABEL RECORDS ARE STANDARD
           DATA RECORD IS RESERVATIONS-RECORD.
       01  RESERVATIONS-RECORD.
         05  RR-REC.
           10  RR-FLIGHT-NUMBER       PIC 9(6).
           10  RR-PASSENGER-NAME1     PIC X(30).
           10  RR-PASSENGER-NAME2     PIC X(30).
```

```
FD  PRINT-FILE
    LABEL RECORDS ARE  OMITTED
    DATA RECORD IS PRINT-RECORD.
01  PRINT-RECORD.
  05  PRT-MESSAGE              PIC X(30).
  05  PRT-REC                  PIC X(68).
  05  FILLER                   PIC X(34).
FD  TRANSACTION-FILE
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS TRANSACTION-RECORD.
01  TRANSACTION-RECORD.
  05  TR-REC.
    10  TR-FLIGHT-NUMBER       PIC 9(6).
    10  TR-PASSENGER-NAME1     PIC X(30).
    10  TR-PASSENGER-NAME2     PIC X(30).
  05  TR-CREATION-CODE         PIC X.
  05  TR-TRANSACTION-CODE      PIC X.
  05  FILLER                   PIC X(12).
FD  NEW-RESERVATIONS-FILE
    LABEL RECORDS ARE STANDARD
    DATA RECORD IS NEW-RESERVATIONS-RECORD.
01  NEW-RESERVATIONS-RECORD.
  05  NRR-REC                  PIC X(66).
WORKING-STORAGE SECTION.
01  RESERVATIONS-PROGRAM-WORK.
  05  RP-END-OF-TRANS          PIC X VALUE ZERO.
  05  RP-END-OF-RESERV         PIC X VALUE ZERO.
  05  RP-INVALID-RECORD        PIC X VALUE ZERO.
01  EDIT-WORK.
  05  EW-THIS-FLIGHT-NUM.
    10  EW-THIS-CREATION       PIC X.
    10  EW-THIS-FLIGHT         PIC 9(6).
  05  EW-THIS-FN REDEFINES EW-THIS-FLIGHT-NUM.
    10  EW-THIS-FLIGHT-NUMBER  PIC 9(7).
  05  EW-FLIGHT-NUMBER         VALUE ZERO.
    10  EW-CREATION            PIC X.
    10  EW-FLIGHT              PIC 9(6).
  05  EW-FN REDEFINES EW-FLIGHT-NUMBER.
    10  EW-LAST-FLIGHT-NUMBER       PIC 9(7).
/
 PROCEDURE DIVISION.
*
*  MODULE FUNCTIONS: THIS IS THE MAINLINE MODULE OF THE
*                    RESERVATIONS PROGRAM.  IT INVOKES MODULES TO
*                    INITIALLY CREATE A RESERVATIONS FILE, TO
*                    UPDATE THIS FILE WITH NEW RESERVATIONS AND
*                    CANCELLATIONS, AND TO PRINT OUT THE CONTENTS
*                    THE NEW UPDATED RESERVATIONS FILE.
*  CONTROL VARIABLES MODIFIED: RP-END-OF-RESERV
*  MODULES INVOKED:            CRE-ATE
*                              READ-RESERVATION
*                              UP-DATE
*                              RE-PORT
*  INVOKING MODULES:           NONE
*
 RESERVATIONS-PROGRAM SECTION.
 RESERVATIONS-PROGRAM-ENTRY.
     OPEN INPUT TRANSACTION-FILE
          OUTPUT RESERVATIONS-FILE PRINT-FILE
     PERFORM CRE-ATE UNTIL RP-END-OF-TRANS='1' OR TR-CREATION-CODE = '1'.
     CLOSE RESERVATIONS-FILE
     OPEN INPUT RESERVATIONS-FILE
     OUTPUT NEW-RESERVATIONS-FILE
     PERFORM READ-RESERVATION.
     PERFORM UP-DATE UNTIL RP-END-OF-TRANS='1' AND RP-END-OF-RESERV = '1'.
     CLOSE RESERVATIONS-FILE
           NEW-RESERVATIONS-FILE
           TRANSACTION-FILE
     OPEN INPUT NEW-RESERVATIONS-FILE
     MOVE ZERO TO RP-END-OF-RESERV
     MOVE SPACES TO PRINT-RECORD
     MOVE 'UPDATED RESERVATION' TO PRT-MESSAGE
     PERFORM RE-PORT UNTIL RP-END-OF-RESERV = '1'.
     CLOSE NEW-RESERVATIONS-FILE PRINT-FILE
     STOP RUN.
 RESERVATIONS-PROGRAM-EXIT.
     EXIT.

*
*  MODULE FUNCTION: THIS MODULE INITIALLY CREATES THE RESERVATIONS
*                   FILE FORM TRANSACTION RECORDS.
*  CONTROL VARIABLES MODIFIED: NONE
*  MODULES INVOKED:            READ-TRANSACTION
*  INVOKING MODULES:           RESERVATIONS-PROGRAM
*
 CRE-ATE SECTION.
 CRE-ATE-ENTRY.
     PERFORM READ-TRANSACTION
     IF RP-END-OF-TRANS = ZERO AND RP-INVALID-RECORD = ZERO AND
        TR-CREATION-CODE NOT = '1'
       WRITE RESERVATIONS-RECORD FROM TR-REC
     ELSE
       NEXT SENTENCE.
 CRE-ATE-EXIT.
     EXIT.
*
*  MODULE FUNCTION: THIS MODULE READS A RECORD FROM THE
*                  TRANSACTION FILE. AT FILE END,
*                  RP-END-OF-TRANS IS SET TO 1.
*  CONTROL VARIABLES MODIFIED: RP-END-OF-TRANS
*  MODULES INVOKED:            EDIT
*  INVOKING MODULES:           CRE-ATE
*                              FINISH-TRANSACTION
*                              MATCH-FLIGHT
*
 READ-TRANSACTION SECTION.
 READ-TRANSACTION-ENTRY.
     READ TRANSACTION-FILE AT END MOVE '1' TO RP-END-OF-TRANS
       GO TO READ-TRANSACTION-EXIT.
     MOVE ZERO TO RP-INVALID-RECORD
     PERFORM EDIT
     IF RP-INVALID-RECORD = '1'
       MOVE SPACES TO PRINT-RECORD
       MOVE 'INLAID TRANSACTION' TO PRT-MESSAGE
       MOVE TRANSACTION-RECORD TO PRT-REC
       WRITE PRINT-RECORD
     ELSE
        NEXT SENTENCE.
 READ-TRANSACTION-EXIT.
     EXIT.
*
*  MODULE FUNCTION: THIS MODULE CHECKS FOR INPUT ERRORS IN THE
*                   TRANSACTION FILE RECORDS.
*  MODULES INVOKED:       NONE
*  INVOKING MODULES:      READ-TRANSACTION
*
 EDIT SECTION.
 EDIT-ENTRY.
     IF TR-FLIGHT-NUMBER IS NOT NUMERIC
       MOVE '1' TO RP-INVALID-RECORD
     ELSE
       MOVE TR-CREATION-CODE TO EW-THIS-CREATION
       MOVE TR-FLIGHT-NUMBER TO EW-THIS-FLIGHT
         IF EW-THIS-FLIGHT-NUMBER NOT > EW-LAST-FLIGHT-NUMBER
           MOVE '1' TO RP-INVALID-RECORD
         ELSE
           MOVE EW-THIS-FLIGHT-NUMBER TO EW-LAST-FLIGHT-NUMBER.
 EDIT-EXIT.
     EXIT.
*
*  MODULE FUNCTION: THIS MODULE READS A RESERVATIONS FILE RECORD.
*                   AT FILE END, RP-END-OF-RESERV IS SET TO 1.
*  CONTROL VARIABLES MODIFIED: RP-END-OF-RESERV
*  MODULES INVOKED:            NONE
*  INVOKING MODULES:           RESERVATIONS-PROGRAM
*                              FINISH-RESERVATION
*                             MATCH-FLIGHT
*
 READ-RESERVATION SECTION.
 READ-RESERVATION-ENTRY.
     READ RESERVATIONS-FILE AT END MOVE '1' TO RP-END-OF-RESERV.
 READ-RESERVATION-EXIT.
     EXIT.
*
*  MODULE FUNCTION: THIS MODULE UPDATES THE RESERVATIONS FILE
*                   WITH NEW TRANSACTIONS TO CREATE A NEW UPDATED
*                   RESERVATION FILE.
*  CONTROL VARIABLES MODIFIED: NONE
*  MODULES INVOKED:            FINISH-RESERVATION
*                              FINISH-TRANSACTION
*                              MATCH-FLIGHT
*  INVOKING MODULES:           RESERVATIONS-PROGRAM
*
 UP-DATE SECTION.
 UP-DATE-ENTRY.
     IF RP-END-OF-TRANS = '1' AND RP-END-OF-RESERV = ZERO
       PERFORM FINISH-RESERVATION UNTIL RP-END-OF-RESERV = '1'
     ELSE IF RP-END-OF-TRANS = ZERO AND
       RP-END-OF-RESERV = '1'
       PERFORM FINISH-TRANSACTION UNTIL RP-END-OF-TRANS = '1'
     ELSE IF RP-END-OF-TRANS = ZERO AND
       RP-END-OF-RESERV = ZERO
       PERFORM MATCH-FLIGHT
     ELSE
       NEXT SENTENCE.
 UP-DATE-EXIT.
     EXIT.
*
*  MODULE FUNCTION: WHEN THERE ARE NO MORE TRANSACTIONS TO PROCESS
*                   THIS PART OF THE UPDATE PROCESS WRITES OUT
*                   THE REMAINING RESERVATIONS ON THE NEW UPDATED
*                   RESERVATIONS FILE.
*  CONTROL VARIABLES MODIFIED: NONE
*  MODULES INVOKED:            WRITE-NEW-RESER
*                              READ-RESERVATION
*  INVOKING MODULES:           UP-DATE
*
 FINISH-RESERVATION SECTION.
 FINISH-RESERVATION-ENTRY.
     PERFORM WRITE-NEW-RESER
     PERFORM READ-RESERVATION.
 FINISH-RESERVATION-EXIT.
     EXIT.
*
*  MODULE FUNCTION: THIS MODULE WRITES A RECORD OUT ON THE NEW
*                   UPDATED RESERVATIONS FILE.
*  CONTROL VARIABLES MODIFIED: NONE
*  MODULES INVOKED:            NONE
*  INVOKING MODULES:           FINISH-RESERVATION
*                              MATCH-FLIGHT
*                              UPDATE-RESERVATION
*
 WRITE-NEW-RESER SECTION.
 WRITE-NEW-RESER-ENTRY.
     MOVE RR-REC TO NRR-REC
     WRITE NEW-RESERVATIONS-RECORD.
 WRITE-NEW-RESER-EXIT.
     EXIT.
*
*  MODULE FUNCTION: WHEN THERE ARE NO MORE RESERVATION RECORD TO
*                   UPDATE.  THIS PART OF THE UPDATE PROCESS
*                   WRITES OUT NEW RESERVATIONS RECORDS FROM THE
*                   REMAINING TRANSACTIONS.
*  CONTROL VARIABLES MODIFIED: NONE
*  MODULES INVOKED:            WRITE-TRANS-TO-RESER
*                              READ-TRANSACTION
*  INVOKING MODULES:           UP-DATE
*
 FINISH-TRANSACTION SECTION.
 FINISH-TRANSACTION-ENTRY.
     IF RP-INVALID-RECORD = ZERO
       IF TR-TRANSACTION-CODE = '1'
         PERFORM WRITE-TRANS-TO-RESER
         MOVE SPACES TO PRINT-RECORD
         MOVE 'RESERVATION ADDED' TO PRT-MESSAGE
         MOVE TRANSACTION-RECORD TO PRT-REC
         WRITE PRINT-RECORD
       ELSE
         NEXT SENTENCE
     ELSE
       NEXT SENTENCE.
     PERFORM READ-TRANSACTION.
 FINISH-TRANSACTION-EXIT.
     EXIT.
*
*  MODULE FUNCTION: THIS MODULE WRITES A RECORD OUT ON THE NEW
*                   UPDATED RESERVATIONS FILE USING A
*                   TRANSACTION RECORD.
*  CONTROL VARIABLES MODIFIED: NONE
*  INVOKING MODULES:           FINISH-TRANSACTION
*                              MATCH-FLIGHT
*  MODULES INVOKED:            NONE
*
 WRITE-TRANS-TO-RESER SECTION.
 WRITE-TRANS-TO-RESER-ENTRY.
     MOVE TR-REC TO NRR-REC
```

```
    WRITE NEW-RESERVATIONS-RECORD.
WRITE-TRANS-TO-RESER-EXIT.
    EXIT.
*
*  MODULE FUNCTION: THIS MODULE TRIES TO MATCH AN EXISTING
*                   RESERVATIONS FILE RECORD WITH A TRANSACTION
*                   FILE RECORD.
*  CONTROL VARIABLES MODIFIED: NONE
*  MODULES INVOKED:            WRITE-NEW-RESER
*                              READ-RESERVATION
*                              WRITE-TRANS-TO-RESER
*                              READ-TRANSACTION
*                              UPDATE-RESERVATION
*  INVOKING MODULES:           UP-DATE
*
MATCH-FLIGHT SECTION.
MATCH-FLIGHT-ENTRY.
    IF RP-INVALID-RECORD = '1'
      PERFORM READ-TRANSACTION
    ELSE IF RR-FLIGHT-NUMBER < TR-FLIGHT-NUMBER
      PERFORM WRITE-NEW-RESER
      PERFORM READ-RESERVATION
    ELSE IF RR-FLIGHT-NUMBER > TR-FLIGHT-NUMBER
      MOVE SPACES TO PRINT-RECORD
      MOVE 'RESERVATION RECORD ADDED' TO PRT-MESSAGE
      MOVE TRANSACTION-RECORD TO PRT-REC
      WRITE PRINT-RECORD
      PERFORM WRITE-TRANS-TO-RESER
      PERFORM READ-TRANSACTION
    ELSE IF RR-FLIGHT-NUMBER = TR-FLIGHT-NUMBER
      PERFORM UPDATE-RESERVATION
      PERFORM READ-RESERVATION
      PERFORM READ-TRANSACTION
    ELSE
      NEXT SENTENCE.
MATCH-FLIGHT-EXIT.
    EXIT.
*
*  MODULE FUNCTION: THIS MODULE UPDATES A RESERVATIONS FILE
*                   RECORD.
*  CONTROL VARIABLES MODIFIED: NONE
*  MODULES INVOKED:            MOD-IFY
*                              WRITE-NEW-RESER
*  INVOKING MODULES:           MATCH-FLIGHT
*
UPDATE-RESERVATION SECTION.
UPDATE-RESERVATION-ENTRY.
    IF TR-TRANSACTION-CODE = ZERO
    PERFORM MOD-IFY
    IF RR-PASSENGER-NAME1 = SPACES AND
```

```
        RR-PASSENGER-NAME2 = SPACES
       MOVE SPACES TO PRINT-RECORD
       MOVE 'RESERVATION RECORD DELETED' TO PRT-MESSAGE
       MOVE RESERVATIONS-RECORD TO PRT-REC
       WRITE PRINT-RECORD
     ELSE
       PERFORM WRITE-NEW-RESER
     ELSE IF RR-PASSENGER-NAME1 = SPACES OR
             RR-PASSENGER-NAME2 = SPACES
       PERFORM MOD-IFY
       MOVE SPACES TO PRINT-RECORD
       MOVE 'RESERVATION ADDED' TO PRT-MESSAGE
       MOVE TRANSACTION-RECORD TO PRT-REC
       WRITE PRINT-RECORD
     ELSE
       MOVE SPACES TO PRINT-RECORD
       MOVE 'NO ROOM ON FLIGHT' TO PRT-MESSAGE
       MOVE TRANSACTION-RECORD TO PRT-REC
       WRITE PRINT-RECORD.
 UPDATE-RESERVATION-EXIT.
        EXIT.
*
*  MODULE FUNCTION: THIS MODULE CHANGES THE PASSENGER NAMES IN
*                   THE RESERVATIONS FILE RECORD ACCORDING TO NEW
*                   INFORMATION FROM A MATCHING TRANSACTION FILE
*                   RECORD.
*  CONTROL VARIABLES MODIFIED: NONE
*  MODULES INVOKED:            NONE
*  INVOKING MODULES:           UPDATE-RESERVATION
*
 MOD-IFY SECTION.
 MOD-IFY-ENTRY.
     IF TR-TRANSACTION-CODE = ZERO
       IF TR-PASSENGER-NAME1 = RR-PASSENGER-NAME1 OR
          TR-PASSENGER-NAME2 = RR-PASSENGER-NAME1
         MOVE SPACES TO RR-PASSENGER-NAME1
         GO TO MOD-IFY-EXIT
       ELSE IF TR-PASSENGER-NAME1 = RR-PASSENGER-NAME2 OR
               TR-PASSENGER-NAME2 = RR-PASSENGER-NAME2
          MOVE SPACES TO RR-PASSENGER-NAME2
          GO TO MOD-IFY-EXIT
       ELSE
          GO TO MOD-IFY-EXIT
       ELSE
          NEXT SENTENCE.
       IF RR-PASSENGER-NAME1 = SPACES
         MOVE TR-PASSENGER-NAME1 TO RR-PASSENGER-NAME1
       ELSE
         MOVE TR-PASSENGER-NAME1 TO RR-PASSENGER-NAME2
       IF TR-PASSENGER-NAME2 NOT = SPACES
```

```
            IF RR-PASSENGER-NAME2 = SPACES
              MOVE TR-PASSENGER-NAME2 TO RR-PASSENGER-NAME2
        ELSE
          MOVE SPACES TO PRINT-RECORD
          MOVE 'NO ROOM FOR 2ND PASSENGER' TO PRT-MESSAGE
          MOVE TRANSACTION-RECORD TO PRT-REC
          WRITE PRINT-RECORD
        ELSE
          NEXT SENTENCE.
    MOD-IFY-EXIT.
        EXIT.
   *
   *  MODULE FUNCTION: THIS MODULE PRINTS OUT EACH RECORD ON THE NEW
   *                   UPDATED RESERVATIONS FILE.
   *  CONTROL VARIABLES MODIFIED: RP-END-OF-RESERV
   *  MODULES INVOKED:            NONE
   *  INVOKING MODULES:           RESERVATIONS-PROGRAM
   *
    RE-PORT SECTION.
    RE-PORT-ENTRY.
        READ NEW-RESERVATIONS-FILE
             AT END MOVE '1' TO RP-END-OF-RESERV
             GO TO RE-PORT-EXIT.
        MOVE NEW-RESERVATIONS-RECORD TO PRT-REC
        WRITE PRINT-RECORD.
    RE-PORT-EXIT.
        EXIT.
```

B.2 Syntax Tree of the Sample Program

```
(SYNTAX TREE)
(914 "Program_body" 1000 1 0 "" 1 "" "" "" "" 0 0)
 (1 "Division" 1002 3 "IDENTIFICATION" "" 1 "" "" "" "" 914 0)
  (2 "Program_name" 5018 3 "RESVPG" "" 2 "" "" "" "" 1 0)
 (3 "Division" 1002 3 "ENVIRONMENT" "" 8 "" "" "" "" 914 1)
  (4 "section" 1003 3 "CONFIGURATION" "" 9 "" "" "" "" 3 0)
   (5 "paragraph" 1004 3 "SOURCE_COMPUTER" "" 10 "" "" "" "" 4 0)
   (6 "paragraph" 1004 3 "OBJECT_COMPUTER" "" 11 "" "" "" "" 4 5)
  (7 "section" 1003 3 "INPUT OUTPUT" "" 12 "" "" "" "" 3 4)
   (8 "paragraph" 1004 3 "FILE CONTROL" "" 13 "" "" "" "" 7 0)
    (15 "file_cntl_entry" 6017 1 0 "" 14 "" "" "" "" 8 0)
     (14 "file_name" 6040 1 0 "" 14 "" "" "" "" 15 0)
      (9 "ud_name" 5012 3 "RESERVATIONS-FILE" "RESERVATIONS-FILE" 14 "" "" ""
         "" 14 0)
    (11 "f_name_list" 6016 1 0 "" 14 "" "" "" "" 15 14)
     (10 "ud_name" 5012 3 "MASS-STORAGE" "MASS-STORAGE" 14 "" "" "" "" 11 0)
    (13 "f_name_list" 6016 1 0 "" 14 "" "" "" "" 15 11)
     (12 "ud_name" 5012 3 "REF" "REF" 14 "" "" "" "" 13 0)
   (22 "file_cntl_entry" 6017 1 0 "" 15 "" "" "" "" 8 15)
    (21 "file_name" 6040 1 0 "" 15 "" "" "" "" 22 0)
     (16 "ud_name" 5012 3 "TRANSACTION-FILE" "TRANSACTION-FILE" 15 "" "" "" ""
         21 0)
    (18 "f_name_list" 6016 1 0 "" 15 "" "" "" "" 22 21)
     (17 "ud_name" 5012 3 "MASS-STORAGE" "MASS-STORAGE" 15 "" "" "" "" 18 0)
    (20 "f_name_list" 6016 1 0 "" 15 "" "" "" "" 22 18)
     (19 "ud_name" 5012 3 "TR" "TR" 15 "" "" "" "" 20 0)
   (27 "file_cntl_entry" 6017 1 0 "" 16 "" "" "" "" 8 22)
    (26 "file_name" 6040 1 0 "" 16 "" "" "" "" 27 0)
     (23 "ud_name" 5012 3 "PRINT-FILE" "PRINT-FILE" 16 "" "" "" "" 26 0)
    (25 "f_name_list" 6016 1 0 "" 16 "" "" "" "" 27 26)
     (24 "ud_name" 5012 3 "PRINTER" "PRINTER" 16 "" "" "" "" 25 0)
   (34 "file_cntl_entry" 6017 1 0 "" 17 "" "" "" "" 8 27)
    (33 "file_name" 6040 1 0 "" 17 "" "" "" "" 34 0)
     (28 "ud_name" 5012 3 "NEW-RESERVATIONS-FILE" "NEW-RESERVATIONS-FILE" 17
         "" "" "" "" 33 0)
    (30 "f_name_list" 6016 1 0 "" 17 "" "" "" "" 34 33)
     (29 "ud_name" 5012 3 "MASS-STORAGE" "MASS-STORAGE" 17 "" "" "" "" 30 0)
    (32 "f_name_list" 6016 1 0 "" 17 "" "" "" "" 34 30)
     (31 "ud_name" 5012 3 "NR" "NR" 17 "" "" "" "" 32 0)
(35 "Division" 1002 3 "DATA" "" 18 "" "" "" "" 914 3)
 (36 "section" 1003 3 "FILE" "" 19 "" "" "" "" 35 0)
  (41 "file_des_file" 6105 3 "FILE" "" 20 "FD" "" "" "" 36 0)
   (42 "file_name" 6040 1 0 "" 20 "" "" "" "" 41 0)
    (37 "ud_name" 5012 3 "RESERVATIONS-FILE" "RESERVATIONS-FILE" 20 "" "" ""
        "" 42 0)
   (38 "file_label" 6160 1 0 "" 21 "label_record_standard" "" "" "" 41 42)
   (40 "file_data" 6110 1 0 "" 22 "" "" "" "" 41 38)
    (39 "ud_name" 5012 3 "RESERVATIONS-RECORD" "RESERVATIONS-RECORD" 22 "" ""
        "" "" 40 0)
   (47 "rec_des_entry" 6131 3 "RECORD" "" 23 "" "" "" "" 41 40)
    (45 "level_number" 6048 1 0 "" 23 "" "" "" "" 47 0)
     (43 "INTEGER" 5004 1 1 "01" 23 "" "" "" "" 45 0)
    (46 "data_name" 6049 1 0 "" 23 "" "" "" "" 47 45)
     (44 "ud_name" 5012 3 "RESERVATIONS-RECORD" "RESERVATIONS-RECORD" 23 "" ""
         "" "" 46 0)
    (52 "rec_des_entry" 6131 3 "RECORD" "" 24 "" "" "" "" 47 46)
   (50 "level_number" 6048 1 0 "" 24 "" "" "" "" 52 0)
    (48 "INTEGER" 5004 1 5 "05" 24 "" "" "" "" 50 0)
   (51 "data_name" 6049 1 0 "" 24 "" "" "" "" 52 50)
    (49 "ud_name" 5012 3 "RR-REC" "RR-REC" 24 "" "" "" "" 51 0)
   (59 "rec_des_entry" 6131 3 "INTEGER" "" 25 "" "" "" "" 52 51)
    (57 "level_number" 6048 1 0 "" 25 "" "" "" "" 59 0)
     (53 "INTEGER" 5004 1 10 "10" 25 "" "" "" "" 57 0)
    (58 "data_name" 6049 1 0 "" 25 "" "" "" "" 59 57)
     (54 "ud_name" 5012 3 "RR-FLIGHT-NUMBER" "RR-FLIGHT-NUMBER" 25 "" "" ""
         "" 58 0)
    (56 "picture" 6166 1 0 "" 25 "" "" "" "" 59 58)
     (55 "Int_Type" 6163 3 "9(6)" "" 25 "6" "" "" "" 56 0)
   (66 "rec_des_entry" 6131 3 "STRING" "" 26 "" "" "" "" 52 59)
    (64 "level_number" 6048 1 0 "" 26 "" "" "" "" 66 0)
     (60 "INTEGER" 5004 1 10 "10" 26 "" "" "" "" 64 0)
    (65 "data_name" 6049 1 0 "" 26 "" "" "" "" 66 64)
     (61 "ud_name" 5012 3 "RR-PASSENGER-NAME1" "RR-PASSENGER-NAME1" 26 "" ""
         "" "" 65 0)
    (63 "picture" 6166 1 0 "" 26 "" "" "" "" 66 65)
     (62 "Text_Type" 6165 3 "X(30)" "" 26 "30" "" "" "" 63 0)
   (73 "rec_des_entry" 6131 3 "STRING" "" 27 "" "" "" "" 52 66)
    (71 "level_number" 6048 1 0 "" 27 "" "" "" "" 73 0)
     (67 "INTEGER" 5004 1 10 "10" 27 "" "" "" "" 71 0)
    (72 "data_name" 6049 1 0 "" 27 "" "" "" "" 73 71)
     (68 "ud_name" 5012 3 "RR-PASSENGER-NAME2" "RR-PASSENGER-NAME2" 27 "" ""
         "" "" 72 0)
    (70 "picture" 6166 1 0 "" 27 "" "" "" "" 73 72)
     (69 "Text_Type" 6165 3 "X(30)" "" 27 "30" "" "" "" 70 0)
(78 "file_des_file" 6105 3 "FILE" "" 28 "FD" "" "" "" 36 41)
 (79 "file_name" 6040 1 0 "" 28 "" "" "" "" 78 0)
  (74 "ud_name" 5012 3 "PRINT-FILE" "PRINT-FILE" 28 "" "" "" "" 79 0)
 (75 "file_label" 6160 1 0 "" 29 "label_record_omitted" "" "" "" 78 79)
 (77 "file_data" 6110 1 0 "" 30 "" "" "" "" 78 75)
  (76 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 30 "" "" "" "" 77 0)
 (84 "rec_des_entry" 6131 3 "RECORD" "" 31 "" "" "" "" 78 77)
  (82 "level_number" 6048 1 0 "" 31 "" "" "" "" 84 0)
   (80 "INTEGER" 5004 1 1 "01" 31 "" "" "" "" 82 0)
  (83 "data_name" 6049 1 0 "" 31 "" "" "" "" 84 82)
   (81 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 31 "" "" "" "" 83 0)
  (91 "rec_des_entry" 6131 3 "STRING" "" 32 "" "" "" "" 84 83)
   (89 "level_number" 6048 1 0 "" 32 "" "" "" "" 91 0)
    (85 "INTEGER" 5004 1 5 "05" 32 "" "" "" "" 89 0)
   (90 "data_name" 6049 1 0 "" 32 "" "" "" "" 91 89)
    (86 "ud_name" 5012 3 "PRT-MESSAGE" "PRT-MESSAGE" 32 "" "" "" "" 90 0)
   (88 "picture" 6166 1 0 "" 32 "" "" "" "" 91 90)
    (87 "Text_Type" 6165 3 "X(30)" "" 32 "30" "" "" "" 88 0)
  (98 "rec_des_entry" 6131 3 "STRING" "" 33 "" "" "" "" 84 91)
   (96 "level_number" 6048 1 0 "" 33 "" "" "" "" 98 0)
    (92 "INTEGER" 5004 1 5 "05" 33 "" "" "" "" 96 0)
   (97 "data_name" 6049 1 0 "" 33 "" "" "" "" 98 96)
    (93 "ud_name" 5012 3 "PRT-REC" "PRT-REC" 33 "" "" "" "" 97 0)
   (95 "picture" 6166 1 0 "" 33 "" "" "" "" 98 97)
    (94 "Text_Type" 6165 3 "X(68)" "" 33 "68" "" "" "" 95 0)
  (104 "rec_des_entry" 6131 3 "STRING" "" 34 "" "" "" "" 84 98)
   (102 "level_number" 6048 1 0 "" 34 "" "" "" "" 104 0)
    (99 "INTEGER" 5004 1 5 "05" 34 "" "" "" "" 102 0)
   (103 "FILLER" 5021 1 0 "" 34 "" "" "" "" 104 102)
   (101 "picture" 6166 1 0 "" 34 "" "" "" "" 104 103)
    (100 "Text_Type" 6165 3 "X(34)" "" 34 "34" "" "" "" 101 0)
(109 "file_des_file" 6105 3 "FILE" "" 35 "FD" "" "" "" 36 78)
 (110 "file_name" 6040 1 0 "" 35 "" "" "" "" 109 0)
  (105 "ud_name" 5012 3 "TRANSACTION-FILE" "TRANSACTION-FILE" 35 "" "" "" ""
       110 0)
 (106 "file_label" 6160 1 0 "" 36 "label_record_omitted" "" "" "" 109 110)
 (108 "file_data" 6110 1 0 "" 37 "" "" "" "" 109 106)
  (107 "ud_name" 5012 3 "TRANSACTION-RECORD" "TRANSACTION-RECORD" 37 "" ""
       "" "" 108 0)
 (115 "rec_des_entry" 6131 3 "RECORD" "" 38 "" "" "" "" 109 108)
  (113 "level_number" 6048 1 0 "" 38 "" "" "" "" 115 0)
   (111 "INTEGER" 5004 1 1 "01" 38 "" "" "" "" 113 0)
  (114 "data_name" 6049 1 0 "" 38 "" "" "" "" 115 113)
   (112 "ud_name" 5012 3 "TRANSACTION-RECORD" "TRANSACTION-RECORD" 38 "" ""
        "" "" 114 0)
  (120 "rec_des_entry" 6131 3 "RECORD" "" 39 "" "" "" "" 115 114)
   (118 "level_number" 6048 1 0 "" 39 "" "" "" "" 120 0)
    (116 "INTEGER" 5004 1 5 "05" 39 "" "" "" "" 118 0)
   (119 "data_name" 6049 1 0 "" 39 "" "" "" "" 120 118)
    (117 "ud_name" 5012 3 "TR-REC" "TR-REC" 39 "" "" "" "" 119 0)
   (127 "rec_des_entry" 6131 3 "INTEGER" "" 40 "" "" "" "" 120 119)
    (125 "level_number" 6048 1 0 "" 40 "" "" "" "" 127 0)
     (121 "INTEGER" 5004 1 10 "10" 40 "" "" "" "" 125 0)
    (126 "data_name" 6049 1 0 "" 40 "" "" "" "" 127 125)
     (122 "ud_name" 5012 3 "TR-FLIGHT-NUMBER" "TR-FLIGHT-NUMBER" 40 "" "" ""
          "" 126 0)
   (124 "picture" 6166 1 0 "" 40 "" "" "" "" 127 126)
    (123 "Int_Type" 6163 3 "9(6)" "" 40 "6" "" "" "" 124 0)
  (134 "rec_des_entry" 6131 3 "STRING" "" 41 "" "" "" "" 120 127)
   (132 "level_number" 6048 1 0 "" 41 "" "" "" "" 134 0)
    (128 "INTEGER" 5004 1 10 "10" 41 "" "" "" "" 132 0)
   (133 "data_name" 6049 1 0 "" 41 "" "" "" "" 134 132)
    (129 "ud_name" 5012 3 "TR-PASSENGER-NAME1" "TR-PASSENGER-NAME1" 41 ""
         "" "" "" 133 0)
   (131 "picture" 6166 1 0 "" 41 "" "" "" "" 134 133)
    (130 "Text_Type" 6165 3 "X(30)" "" 41 "30" "" "" "" 131 0)
  (141 "rec_des_entry" 6131 3 "STRING" "" 42 "" "" "" "" 120 134)
   (139 "level_number" 6048 1 0 "" 42 "" "" "" "" 141 0)
    (135 "INTEGER" 5004 1 10 "10" 42 "" "" "" "" 139 0)
   (140 "data_name" 6049 1 0 "" 42 "" "" "" "" 141 139)
    (136 "ud_name" 5012 3 "TR-PASSENGER-NAME2" "TR-PASSENGER-NAME2" 42 ""
         "" "" "" 140 0)
   (138 "picture" 6166 1 0 "" 42 "" "" "" "" 141 140)
    (137 "Text_Type" 6165 3 "X(30)" "" 42 "30" "" "" "" 138 0)
 (148 "rec_des_entry" 6131 3 "STRING" "" 43 "" "" "" "" 115 120)
  (146 "level_number" 6048 1 0 "" 43 "" "" "" "" 148 0)
   (142 "INTEGER" 5004 1 5 "05" 43 "" "" "" "" 146 0)
  (147 "data_name" 6049 1 0 "" 43 "" "" "" "" 148 146)
   (143 "ud_name" 5012 3 "TR-CREATION-CODE" "TR-CREATION-CODE" 43 "" "" ""
        "" 147 0)
  (145 "picture" 6166 1 0 "" 43 "" "" "" "" 148 147)
   (144 "Text_Type" 6165 3 "X" "" 43 "1" "" "" "" 145 0)
 (155 "rec_des_entry" 6131 3 "STRING" "" 44 "" "" "" "" 115 148)
  (153 "level_number" 6048 1 0 "" 44 "" "" "" "" 155 0)
   (149 "INTEGER" 5004 1 5 "05" 44 "" "" "" "" 153 0)
  (154 "data_name" 6049 1 0 "" 44 "" "" "" "" 155 153)
   (150 "ud_name" 5012 3 "TR-TRANSACTION-CODE" "TR-TRANSACTION-CODE" 44 ""
        "" "" "" 154 0)
  (152 "picture" 6166 1 0 "" 44 "" "" "" "" 155 154)
   (151 "Text_Type" 6165 3 "X" "" 44 "1" "" "" "" 152 0)
 (161 "rec_des_entry" 6131 3 "STRING" "" 45 "" "" "" "" 115 155)
  (159 "level_number" 6048 1 0 "" 45 "" "" "" "" 161 0)
   (156 "INTEGER" 5004 1 5 "05" 45 "" "" "" "" 159 0)
  (160 "FILLER" 5021 1 0 "" 45 "" "" "" "" 161 159)
  (158 "picture" 6166 1 0 "" 45 "" "" "" "" 161 160)
   (157 "Text_Type" 6165 3 "X(12)" "" 45 "12" "" "" "" 158 0)
(166 "file_des_file" 6105 3 "FILE" "" 46 "FD" "" "" "" 36 109)
```

```
  (167 "file_name" 6040 1 0 "" 46 "" "" "" "" 166 0)
   (162 "ud_name" 5012 3 "NEW-RESERVATIONS-FILE" "NEW-RESERVATIONS-FILE" 46
        "" "" "" "" 167 0)
  (163 "file_label" 6160 1 0 "" 47 "label_record_standard" "" "" "" 166 167)
  (165 "file_data" 6110 1 0 "" 48 "" "" "" "" 166 163)
   (164 "ud_name" 5012 3 "NEW-RESERVATIONS-RECORD" "NEW-RESERVATIONS-RECORD"
        48 "" "" "" "" 165 0)
  (172 "rec_des_entry" 6131 3 "RECORD" "" 49 "" "" "" "" 166 165)
   (170 "level_number" 6048 1 0 "" 49 "" "" "" "" 172 0)
    (168 "INTEGER" 5004 1 1 "01" 49 "" "" "" "" 170 0)
   (171 "data_name" 6049 1 0 "" 49 "" "" "" "" 172 170)
    (169 "ud_name" 5012 3 "NEW-RESERVATIONS-RECORD" "NEW-RESERVATIONS-RECORD"
         49 "" "" "" "" 171 0)
   (179 "rec_des_entry" 6131 3 "STRING" "" 50 "" "" "" "" 172 171)
    (177 "level_number" 6048 1 0 "" 50 "" "" "" "" 179 0)
     (173 "INTEGER" 5004 1 5 "05" 50 "" "" "" "" 177 0)
    (178 "data_name" 6049 1 0 "" 50 "" "" "" "" 179 177)
     (174 "ud_name" 5012 3 "NRR-REC" "NRR-REC" 50 "" "" "" "" 178 0)
    (176 "picture" 6166 1 0 "" 50 "" "" "" "" 179 178)
     (175 "Text_Type" 6165 3 "X(66)" "" 50 "66" "" "" "" 176 0)
(180 "section" 1003 3 "WORKING STORAGE" "" 51 "" "" "" "" 35 36)
 (185 "rec_des_entry" 6131 3 "RECORD" "" 52 "" "" "" "" 180 0)
  (183 "level_number" 6048 1 0 "" 52 "" "" "" "" 185 0)
   (181 "INTEGER" 5004 1 1 "01" 52 "" "" "" "" 183 0)
  (184 "data_name" 6049 1 0 "" 52 "" "" "" "" 185 183)
   (182 "ud_name" 5012 3 "RESERVATIONS-PROGRAM-WORK"
        "RESERVATIONS-PROGRAM-WORK" 52 "" "" "" "" 184 0)
  (194 "rec_des_entry" 6131 3 "STRING" "" 53 "" "" "" "" 185 184)
   (192 "level_number" 6048 1 0 "" 53 "" "" "" "" 194 0)
    (186 "INTEGER" 5004 1 5 "05" 53 "" "" "" "" 192 0)
   (193 "data_name" 6049 1 0 "" 53 "" "" "" "" 194 192)
    (187 "ud_name" 5012 3 "RP-END-OF-TRANS" "RP-END-OF-TRANS" 53 "" "" "" ""
         193 0)
   (189 "picture" 6166 1 0 "" 53 "" "" "" "" 194 193)
    (188 "Text_Type" 6165 3 "X" "" 53 "1" "" "" "" 189 0)
   (191 "data_value" 6171 1 0 "" 53 "" "" "" "" 194 189)
    (190 "ZERO" 5004 1 0 "ZERO" 53 "" "" "" "" 191 0)
  (203 "rec_des_entry" 6131 3 "STRING" "" 54 "" "" "" "" 185 194)
   (201 "level_number" 6048 1 0 "" 54 "" "" "" "" 203 0)
    (195 "INTEGER" 5004 1 5 "05" 54 "" "" "" "" 201 0)
   (202 "data_name" 6049 1 0 "" 54 "" "" "" "" 203 201)
    (196 "ud_name" 5012 3 "RP-END-OF-RESERV" "RP-END-OF-RESERV" 54 "" "" ""
        "" 202 0)
  (198 "picture" 6166 1 0 "" 54 "" "" "" "" 203 202)
   (197 "Text_Type" 6165 3 "X" "" 54 "1" "" "" "" 198 0)
  (200 "data_value" 6171 1 0 "" 54 "" "" "" "" 203 198)
   (199 "ZERO" 5004 1 0 "ZERO" 54 "" "" "" "" 200 0)
 (212 "rec_des_entry" 6131 3 "STRING" "" 55 "" "" "" "" 185 203)
  (210 "level_number" 6048 1 0 "" 55 "" "" "" "" 212 0)
   (204 "INTEGER" 5004 1 5 "05" 55 "" "" "" "" 210 0)
  (211 "data_name" 6049 1 0 "" 55 "" "" "" "" 212 210)
   (205 "ud_name" 5012 3 "RP-INVALID-RECORD" "RP-INVALID-RECORD" 55 "" "" ""
        "" 211 0)
  (207 "picture" 6166 1 0 "" 55 "" "" "" "" 212 211)
   (206 "Text_Type" 6165 3 "X" "" 55 "1" "" "" "" 207 0)
  (209 "data_value" 6171 1 0 "" 55 "" "" "" "" 212 207)
   (208 "ZERO" 5004 1 0 "ZERO" 55 "" "" "" "" 209 0)
(217 "rec_des_entry" 6131 3 "RECORD" "" 56 "" "" "" "" 180 185)
 (215 "level_number" 6048 1 0 "" 56 "" "" "" "" 217 0)
  (213 "INTEGER" 5004 1 1 "01" 56 "" "" "" "" 215 0)
 (216 "data_name" 6049 1 0 "" 56 "" "" "" "" 217 215)
  (214 "ud_name" 5012 3 "EDIT-WORK" "EDIT-WORK" 56 "" "" "" "" 216 0)
 (222 "rec_des_entry" 6131 3 "RECORD" "" 57 "" "" "" "" 217 216)
  (220 "level_number" 6048 1 0 "" 57 "" "" "" "" 222 0)
   (218 "INTEGER" 5004 1 5 "05" 57 "" "" "" "" 220 0)
  (221 "data_name" 6049 1 0 "" 57 "" "" "" "" 222 220)
   (219 "ud_name" 5012 3 "EW-THIS-FLIGHT-NUM" "EW-THIS-FLIGHT-NUM" 57 "" ""
        "" "" 221 0)
  (229 "rec_des_entry" 6131 3 "STRING" "" 58 "" "" "" "" 222 221)
   (227 "level_number" 6048 1 0 "" 58 "" "" "" "" 229 0)
    (223 "INTEGER" 5004 1 10 "10" 58 "" "" "" "" 227 0)
   (228 "data_name" 6049 1 0 "" 58 "" "" "" "" 229 227)
    (224 "ud_name" 5012 3 "EW-THIS-CREATION" "EW-THIS-CREATION" 58 "" "" ""
         "" 228 0)
   (226 "picture" 6166 1 0 "" 58 "" "" "" "" 229 228)
    (225 "Text_Type" 6165 3 "X" "" 58 "1" "" "" "" 226 0)
  (236 "rec_des_entry" 6131 3 "INTEGER" "" 59 "" "" "" "" 222 229)
   (234 "level_number" 6048 1 0 "" 59 "" "" "" "" 236 0)
    (230 "INTEGER" 5004 1 10 "10" 59 "" "" "" "" 234 0)
   (235 "data_name" 6049 1 0 "" 59 "" "" "" "" 236 234)
    (231 "ud_name" 5012 3 "EW-THIS-FLIGHT" "EW-THIS-FLIGHT" 59 "" "" "" ""
         235 0)
   (233 "picture" 6166 1 0 "" 59 "" "" "" "" 236 235)
    (232 "Int_Type" 6163 3 "9(6)" "" 59 "6" "" "" "" 233 0)
```

```
(243 "rec_des_entry" 6131 3 "RECORD" "" 60 "" "" "" "" 217 222)
 (241 "level_number" 6048 1 0 "" 60 "" "" "" "" 243 0)
  (237 "INTEGER" 5004 1 5 "05" 60 "" "" "" "" 241 0)
 (242 "data_name" 6049 1 0 "" 60 "" "" "" "" 243 241)
  (238 "ud_name" 5012 3 "EW-THIS-FN" "EW-THIS-FN" 60 "" "" "" "" 242 0)
 (240 "data_redef" 6162 1 0 "" 60 "" "" "" "" 243 242)
  (239 "ud_name" 5012 3 "EW-THIS-FLIGHT-NUM" "EW-THIS-FLIGHT-NUM" 60 "" ""
      "" "" 240 0)
 (250 "rec_des_entry" 6131 3 "INTEGER" "" 61 "" "" "" "" 243 240)
  (248 "level_number" 6048 1 0 "" 61 "" "" "" "" 250 0)
   (244 "INTEGER" 5004 1 10 "10" 61 "" "" "" "" 248 0)
  (249 "data_name" 6049 1 0 "" 61 "" "" "" "" 250 248)
   (245 "ud_name" 5012 3 "EW-THIS-FLIGHT-NUMBER" "EW-THIS-FLIGHT-NUMBER"
       61 "" "" "" "" 249 0)
  (247 "picture" 6166 1 0 "" 61 "" "" "" "" 250 249)
   (246 "Int_Type" 6163 3 "9(7)" "" 61 "7" "" "" "" 247 0)
(257 "rec_des_entry" 6131 3 "RECORD" "" 62 "" "" "" "" 217 243)
 (255 "level_number" 6048 1 0 "" 62 "" "" "" "" 257 0)
  (251 "INTEGER" 5004 1 5 "05" 62 "" "" "" "" 255 0)
 (256 "data_name" 6049 1 0 "" 62 "" "" "" "" 257 255)
  (252 "ud_name" 5012 3 "EW-FLIGHT-NUMBER" "EW-FLIGHT-NUMBER" 62 "" "" ""
      "" 256 0)
 (254 "data_value" 6171 1 0 "" 62 "" "" "" "" 257 256)
  (253 "ZERO" 5004 1 0 "ZERO" 62 "" "" "" "" 254 0)
 (264 "rec_des_entry" 6131 3 "STRING" "" 63 "" "" "" "" 257 254)
  (262 "level_number" 6048 1 0 "" 63 "" "" "" "" 264 0)
   (258 "INTEGER" 5004 1 10 "10" 63 "" "" "" "" 262 0)
  (263 "data_name" 6049 1 0 "" 63 "" "" "" "" 264 262)
   (259 "ud_name" 5012 3 "EW-CREATION" "EW-CREATION" 63 "" "" "" "" 263 0)
  (261 "picture" 6166 1 0 "" 63 "" "" "" "" 264 263)
   (260 "Text_Type" 6165 3 "X" "" 63 "1" "" "" "" 261 0)
 (271 "rec_des_entry" 6131 3 "INTEGER" "" 64 "" "" "" "" 257 264)
  (269 "level_number" 6048 1 0 "" 64 "" "" "" "" 271 0)
   (265 "INTEGER" 5004 1 10 "10" 64 "" "" "" "" 269 0)
  (270 "data_name" 6049 1 0 "" 64 "" "" "" "" 271 269)
   (266 "ud_name" 5012 3 "EW-FLIGHT" "EW-FLIGHT" 64 "" "" "" "" 270 0)
  (268 "picture" 6166 1 0 "" 64 "" "" "" "" 271 270)
   (267 "Int_Type" 6163 3 "9(6)" "" 64 "6" "" "" "" 268 0)
(278 "rec_des_entry" 6131 3 "RECORD" "" 65 "" "" "" "" 217 257)
 (276 "level_number" 6048 1 0 "" 65 "" "" "" "" 278 0)
  (272 "INTEGER" 5004 1 5 "05" 65 "" "" "" "" 276 0)
 (277 "data_name" 6049 1 0 "" 65 "" "" "" "" 278 276)
```

```
      (273 "ud_name" 5012 3 "EW-FN" "EW-FN" 65 "" "" "" "" 277 0)
     (275 "data_redef" 6162 1 0 "" 65 "" "" "" "" 278 277)
      (274 "ud_name" 5012 3 "EW-FLIGHT-NUMBER" "EW-FLIGHT-NUMBER" 65 "" "" ""
           "" 275 0)
     (285 "rec_des_entry" 6131 3 "INTEGER" "" 66 "" "" "" "" 278 275)
      (283 "level_number" 6048 1 0 "" 66 "" "" "" "" 285 0)
       (279 "INTEGER" 5004 1 10 "10" 66 "" "" "" "" 283 0)
      (284 "data_name" 6049 1 0 "" 66 "" "" "" "" 285 283)
       (280 "ud_name" 5012 3 "EW-LAST-FLIGHT-NUMBER" "EW-LAST-FLIGHT-NUMBER"
            66 "" "" "" "" 284 0)
      (282 "picture" 6166 1 0 "" 66 "" "" "" "" 285 284)
       (281 "Int_Type" 6163 3 "9(7)" "" 66 "7" "" "" "" 282 0)
(286 "Division" 1002 3 "PROCEDURE" "" 68 "" "" "" "" 914 35)
 (287 "section" 1003 3 "RESERVATIONS-PROGRAM" "" 83 "" "" "" "" 286 0)
  (288 "paragraph" 1004 3 "RESERVATIONS-PROGRAM-ENTRY" "" 84 "" "" "" "" 287
       0)
   (294 "Open_st" 2028 1 0 "OPEN INPUT TRANSACTION-FILE  OUTPUT
        RESERVATIONS-FILE  PRINT-FILE " 85 "" "" "" "" 288 0)
    (290 "INPUT" 4022 1 0 "INPUT TRANSACTION-FILE " 85 "" "" "" "" 294 0)
     (289 "ud_name" 5012 3 "TRANSACTION-FILE" "TRANSACTION-FILE " 85 "" "" ""
          "" 290 0)
    (293 "OUTPUT" 4022 1 0 "OUTPUT RESERVATIONS-FILE  PRINT-FILE " 86 "" "" ""
         "" 294 290)
     (291 "ud_name" 5012 3 "RESERVATIONS-FILE" "RESERVATIONS-FILE " 86 "" ""
          "" "" 293 0)
     (292 "ud_name" 5012 3 "PRINT-FILE" "PRINT-FILE " 86 "" "" "" "" 293 291)
   (304 "Perform_st" 2029 1 0 "PERFORM CRE-ATE BEFORE UNTIL RP-END-OF-TRANS =
        '1'  OR TR-CREATION-CODE = '1' " 87 "BEFORE" "" "" "" 288 294)
    (305 "Perform_body" 3501 1 0 "CRE-ATE" 87 "PROCEDURE" "" "" "" 304 0)
     (295 "ud_name" 5012 3 "CRE-ATE" "CRE-ATE" 87 "" "" "" "" 305 0)
    (303 "UNTIL" 3014 1 0 "UNTIL RP-END-OF-TRANS = '1'  OR TR-CREATION-CODE =
         '1' " 87 "" "" "" "" 304 305)
     (302 "OR" 5501 1 0 "RP-END-OF-TRANS = '1'  OR TR-CREATION-CODE = '1' "
          87 "" "" "" "" 303 0)
      (298 "Rel_op" 3504 3 "=" "RP-END-OF-TRANS = '1' " 87 "" "" "" "" 302 0)
       (296 "ud_name" 5012 3 "RP-END-OF-TRANS" "RP-END-OF-TRANS" 87 "" "" ""
            "" 298 0)
       (297 "LITERAL" 5001 3 "'1'" "'1'" 87 "" "" "" "" 298 296)
      (301 "Rel_op" 3504 3 "=" "TR-CREATION-CODE = '1' " 87 "" "" "" "" 302
           298)
       (299 "ud_name" 5012 3 "TR-CREATION-CODE" "TR-CREATION-CODE" 87 "" "" ""
            "" 301 0)
   (300 "LITERAL" 5001 3 "'1'" "'1'" 87 "" "" "" "" 301 299)
(307 "Close_st" 2006 1 0 "CLOSE RESERVATIONS-FILE" 88 "" "" "" "" 288 304)
 (306 "ud_name" 5012 3 "RESERVATIONS-FILE" "RESERVATIONS-FILE" 88 "" "" ""
      "" 307 0)
(312 "Open_st" 2028 1 0 "OPEN INPUT RESERVATIONS-FILE  OUTPUT
     NEW-RESERVATIONS-FILE " 89 "" "" "" "" 288 307)
 (309 "INPUT" 4022 1 0 "INPUT RESERVATIONS-FILE " 89 "" "" "" "" 312 0)
  (308 "ud_name" 5012 3 "RESERVATIONS-FILE" "RESERVATIONS-FILE " 89 "" ""
       "" "" 309 0)
 (311 "OUTPUT" 4022 1 0 "OUTPUT NEW-RESERVATIONS-FILE " 90 "" "" "" "" 312
      309)
  (310 "ud_name" 5012 3 "NEW-RESERVATIONS-FILE" "NEW-RESERVATIONS-FILE " 90
       "" "" "" "" 311 0)
(314 "Perform_st" 2029 1 0 "PERFORM READ-RESERVATION" 91 "" "" "" "" 288
     312)
 (315 "Perform_body" 3501 1 0 "READ-RESERVATION" 91 "PROCEDURE" "" "" ""
      314 0)
  (313 "ud_name" 5012 3 "READ-RESERVATION" "READ-RESERVATION" 91 "" "" ""
       "" 315 0)
(325 "Perform_st" 2029 1 0 "PERFORM UP-DATE BEFORE UNTIL RP-END-OF-TRANS =
     '1'  AND RP-END-OF-RESERV = '1' " 92 "BEFORE" "" "" "" 288 314)
 (326 "Perform_body" 3501 1 0 "UP-DATE" 92 "PROCEDURE" "" "" "" 325 0)
  (316 "ud_name" 5012 3 "UP-DATE" "UP-DATE" 92 "" "" "" "" 326 0)
 (324 "UNTIL" 3014 1 0 "UNTIL RP-END-OF-TRANS = '1'  AND RP-END-OF-RESERV =
      '1' " 92 "" "" "" "" 325 326)
  (323 "AND" 5503 1 0 "RP-END-OF-TRANS = '1'  AND RP-END-OF-RESERV = '1' "
       92 "" "" "" "" 324 0)
   (319 "Rel_op" 3504 3 "=" "RP-END-OF-TRANS = '1' " 92 "" "" "" "" 323 0)
    (317 "ud_name" 5012 3 "RP-END-OF-TRANS" "RP-END-OF-TRANS" 92 "" "" ""
         "" 319 0)
    (318 "LITERAL" 5001 3 "'1'" "'1'" 92 "" "" "" "" 319 317)
   (322 "Rel_op" 3504 3 "=" "RP-END-OF-RESERV = '1' " 92 "" "" "" "" 323
        319)
    (320 "ud_name" 5012 3 "RP-END-OF-RESERV" "RP-END-OF-RESERV" 92 "" "" ""
         "" 322 0)
    (321 "LITERAL" 5001 3 "'1'" "'1'" 92 "" "" "" "" 322 320)
(330 "Close_st" 2006 1 0 "CLOSE RESERVATIONS-FILE NEW-RESERVATIONS-FILE
     TRANSACTION-FILE" 93 "" "" "" "" 288 325)
 (327 "ud_name" 5012 3 "RESERVATIONS-FILE" "RESERVATIONS-FILE" 93 "" "" ""
      "" 330 0)
 (328 "ud_name" 5012 3 "NEW-RESERVATIONS-FILE" "NEW-RESERVATIONS-FILE" 94
      "" "" "" "" 330 327)
 (329 "ud_name" 5012 3 "TRANSACTION-FILE" "TRANSACTION-FILE" 95 "" "" "" ""
      330 328)
(333 "Open_st" 2028 1 0 "OPEN INPUT NEW-RESERVATIONS-FILE " 96 "" "" "" ""
     288 330)
 (332 "INPUT" 4022 1 0 "INPUT NEW-RESERVATIONS-FILE " 96 "" "" "" "" 333 0)
  (331 "ud_name" 5012 3 "NEW-RESERVATIONS-FILE" "NEW-RESERVATIONS-FILE " 96
       "" "" "" "" 332 0)
(338 "Move_st" 2026 1 0 "MOVE ZERO TO RP-END-OF-RESERV" 97 "" "" "" "" 288
     333)
 (336 "Sourcd_d" 3503 1 0 "ZERO" 97 "" "" "" "" 338 0)
  (334 "ZERO" 5004 1 0 "ZERO" 97 "" "" "" "" 336 0)
 (337 "TO" 3005 1 0 "TO RP-END-OF-RESERV" 97 "" "" "" "" 338 336)
  (335 "ud_name" 5012 3 "RP-END-OF-RESERV" "RP-END-OF-RESERV" 97 "" "" ""
       "" 337 0)
(343 "Move_st" 2026 1 0 "MOVE SPACES TO PRINT-RECORD" 98 "" "" "" "" 288
     338)
 (341 "Sourcd_d" 3503 1 0 "SPACES" 98 "" "" "" "" 343 0)
  (339 "SPACE" 5001 3 " " "SPACES" 98 "" "" "" "" 341 0)
 (342 "TO" 3005 1 0 "TO PRINT-RECORD" 98 "" "" "" "" 343 341)
  (340 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 98 "" "" "" "" 342 0)
(348 "Move_st" 2026 1 0 "MOVE 'UPDATED RESERVATION' TO PRT-MESSAGE" 99 ""
     "" "" "" 288 343)
 (346 "Sourcd_d" 3503 1 0 "'UPDATED RESERVATION'" 99 "" "" "" "" 348 0)
  (344 "LITERAL" 5001 3 "'UPDATED RESERVATION'" "'UPDATED RESERVATION'" 99
       "" "" "" "" 346 0)
 (347 "TO" 3005 1 0 "TO PRT-MESSAGE" 99 "" "" "" "" 348 346)
  (345 "ud_name" 5012 3 "PRT-MESSAGE" "PRT-MESSAGE" 99 "" "" "" "" 347 0)
(354 "Perform_st" 2029 1 0 "PERFORM RE-PORT BEFORE UNTIL RP-END-OF-RESERV =
     '1' " 100 "BEFORE" "" "" "" 288 348)
 (355 "Perform_body" 3501 1 0 "RE-PORT" 100 "PROCEDURE" "" "" "" 354 0)
  (349 "ud_name" 5012 3 "RE-PORT" "RE-PORT" 100 "" "" "" "" 355 0)
 (353 "UNTIL" 3014 1 0 "UNTIL RP-END-OF-RESERV = '1' " 100 "" "" "" "" 354
      355)
  (352 "Rel_op" 3504 3 "=" "RP-END-OF-RESERV = '1' " 100 "" "" "" "" 353 0)
   (350 "ud_name" 5012 3 "RP-END-OF-RESERV" "RP-END-OF-RESERV" 100 "" "" ""
        "" 352 0)
   (351 "LITERAL" 5001 3 "'1'" "'1'" 100 "" "" "" "" 352 350)
(358 "Close_st" 2006 1 0 "CLOSE NEW-RESERVATIONS-FILE PRINT-FILE" 101 "" ""
     "" "" 288 354)
 (356 "ud_name" 5012 3 "NEW-RESERVATIONS-FILE" "NEW-RESERVATIONS-FILE" 101
      "" "" "" "" 358 0)
 (357 "ud_name" 5012 3 "PRINT-FILE" "PRINT-FILE" 101 "" "" "" "" 358 356)
(359 "Stop_st" 2046 1 0 "STOP RUN" 102 "" "" "" "" 288 358)
```

```
(360 "paragraph" 1004 3 "RESERVATIONS-PROGRAM-EXIT" "" 103 "" "" "" "" 287
     288)
 (361 "Exit_st" 2017 1 0 "EXIT" 104 "" "" "" "" 360 0)
(362 "section" 1003 3 "CRE-ATE" "" 113 "" "" "" "" 286 287)
 (363 "paragraph" 1004 3 "CRE-ATE-ENTRY" "" 114 "" "" "" "" 362 0)
  (365 "Perform_st" 2029 1 0 "PERFORM READ-TRANSACTION" 115 "" "" "" "" 363
       0)
   (366 "Perform_body" 3501 1 0 "READ-TRANSACTION" 115 "PROCEDURE" "" "" ""
        365 0)
    (364 "ud_name" 5012 3 "READ-TRANSACTION" "READ-TRANSACTION" 115 "" "" ""
         "" 366 0)
  (387 "If_st" 2021 1 0 "IF RP-END-OF-TRANS = ZERO  AND RP-INVALID-RECORD =
       ZERO  AND TR-CREATION-CODE NOT = '1' THEN ELSE" 116 "" "" "" "" 363
       365)
   (378 "AND" 5503 1 0 "RP-END-OF-TRANS = ZERO  AND RP-INVALID-RECORD = ZERO
        AND TR-CREATION-CODE NOT = '1'" 116 "" "" "" "" 387 0)
    (369 "Rel_op" 3504 3 "=" "RP-END-OF-TRANS = ZERO " 116 "" "" "" "" 378 0)
     (367 "ud_name" 5012 3 "RP-END-OF-TRANS" "RP-END-OF-TRANS" 116 "" "" ""
          "" 369 0)
     (368 "ZERO" 5004 1 0 "ZERO" 116 "" "" "" "" 369 367)
    (377 "AND" 5503 1 0 "RP-INVALID-RECORD = ZERO  AND TR-CREATION-CODE NOT =
         '1'" 116 "" "" "" "" 378 369)
     (372 "Rel_op" 3504 3 "=" "RP-INVALID-RECORD = ZERO " 116 "" "" "" "" 377
          0)
      (370 "ud_name" 5012 3 "RP-INVALID-RECORD" "RP-INVALID-RECORD" 116 "" ""
           "" "" 372 0)
      (371 "ZERO" 5004 1 0 "ZERO" 116 "" "" "" "" 372 370)
     (376 "NOT" 5500 1 0 "TR-CREATION-CODE NOT = '1'" 117 "" "" "" "" 377
          372)
      (375 "Rel_op" 3504 3 "=" "TR-CREATION-CODE = '1'" 117 "" "" "" "" 376
           0)
       (373 "ud_name" 5012 3 "TR-CREATION-CODE" "TR-CREATION-CODE" 117 "" ""
            "" "" 375 0)
       (374 "LITERAL" 5001 3 "'1'" "'1'" 117 "" "" "" "" 375 373)
   (386 "THEN" 3000 1 0 "THEN" 118 "" "" "" "" 387 378)
    (383 "Write_st" 2053 1 0 "WRITE RESERVATIONS-RECORD FROM TR-REC" 118 ""
         "" "" "" 386 0)
     (382 "Source_rec" 3503 1 0 "RESERVATIONS-RECORD" 118 "" "" "" "" 383 0)
      (379 "ud_name" 5012 3 "RESERVATIONS-RECORD" "RESERVATIONS-RECORD" 118
           "" "" "" "" 382 0)
     (381 "FROM" 3004 1 0 "FROM TR-REC" 118 "" "" "" "" 383 382)
      (380 "ud_name" 5012 3 "TR-REC" "TR-REC" 118 "" "" "" "" 381 0)
 (385 "ELSE" 3001 1 0 "ELSE" 120 "" "" "" "" 387 386)
  (384 "NEXT_SENTENCE" 6202 1 0 "NEXT SENTENCE" 120 "" "" "" "" 385 0)
(388 "paragraph" 1004 3 "CRE-ATE-EXIT" "" 121 "" "" "" "" 362 363)
 (389 "Exit_st" 2017 1 0 "EXIT" 122 "" "" "" "" 388 0)
(390 "section" 1003 3 "READ-TRANSACTION" "" 133 "" "" "" "" 286 362)
(391 "paragraph" 1004 3 "READ-TRANSACTION-ENTRY" "" 134 "" "" "" "" 390 0)
 (403 "Read_st" 2035 1 0 "READ TRANSACTION-FILE" 135 "" "" "" "" 391 0)
  (402 "Source_file" 3503 1 0 "TRANSACTION-FILE" 135 "" "" "" "" 403 0)
   (392 "ud_name" 5012 3 "TRANSACTION-FILE" "TRANSACTION-FILE" 135 "" "" ""
        "" 402 0)
  (401 "AT END" 3016 1 0 "AT END" 135 "" "" "" "" 403 402)
   (397 "Move_st" 2026 1 0 "MOVE '1' TO RP-END-OF-TRANS" 135 "" "" "" "" 401
        0)
    (395 "Sourcd_d" 3503 1 0 "'1'" 135 "" "" "" "" 397 0)
     (393 "LITERAL" 5001 3 "'1'" "'1'" 135 "" "" "" "" 395 0)
    (396 "TO" 3005 1 0 "TO RP-END-OF-TRANS" 135 "" "" "" "" 397 395)
     (394 "ud_name" 5012 3 "RP-END-OF-TRANS" "RP-END-OF-TRANS" 135 "" "" ""
          "" 396 0)
   (400 "Goto_st" 2019 1 0 "GO TO READ-TRANSACTION-EXIT" 136 "" "" "" "" 401
        397)
    (399 "Procedure_d" 3503 1 0 "READ-TRANSACTION-EXIT" 136 "" "" "" "" 400
         0)
     (398 "ud_name" 5012 3 "READ-TRANSACTION-EXIT" "READ-TRANSACTION-EXIT"
          136 "" "" "" "" 399 0)
 (408 "Move_st" 2026 1 0 "MOVE ZERO TO RP-INVALID-RECORD" 137 "" "" "" ""
      391 403)
  (406 "Sourcd_d" 3503 1 0 "ZERO" 137 "" "" "" "" 408 0)
   (404 "ZERO" 5004 1 0 "ZERO" 137 "" "" "" "" 406 0)
  (407 "TO" 3005 1 0 "TO RP-INVALID-RECORD" 137 "" "" "" "" 408 406)
   (405 "ud_name" 5012 3 "RP-INVALID-RECORD" "RP-INVALID-RECORD" 137 "" ""
        "" "" 407 0)
 (410 "Perform_st" 2029 1 0 "PERFORM EDIT" 138 "" "" "" "" 391 408)
  (411 "Perform_body" 3501 1 0 "EDIT" 138 "PROCEDURE" "" "" "" 410 0)
   (409 "ud_name" 5012 3 "EDIT" "EDIT" 138 "" "" "" "" 411 0)
 (436 "If_st" 2021 1 0 "IF RP-INVALID-RECORD = '1' THEN ELSE" 139 "" "" ""
      "" 391 410)
  (414 "Rel_op" 3504 3 "=" "RP-INVALID-RECORD = '1' " 139 "" "" "" "" 436 0)
   (412 "ud_name" 5012 3 "RP-INVALID-RECORD" "RP-INVALID-RECORD" 139 "" ""
        "" "" 414 0)
   (413 "LITERAL" 5001 3 "'1'" "'1'" 139 "" "" "" "" 414 412)
  (435 "THEN" 3000 1 0 "THEN" 140 "" "" "" "" 436 414)
   (419 "Move_st" 2026 1 0 "MOVE SPACES TO PRINT-RECORD" 140 "" "" "" "" 435
        0)
      (417 "Sourcd_d" 3503 1 0 "SPACES" 140 "" "" "" "" 419 0)
       (415 "SPACE" 5001 3 " " "SPACES" 140 "" "" "" "" 417 0)
      (418 "TO" 3005 1 0 "TO PRINT-RECORD" 140 "" "" "" "" 419 417)
       (416 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 140 "" "" "" "" 418
            0)
     (424 "Move_st" 2026 1 0 "MOVE 'INLAID TRANSACTION' TO PRT-MESSAGE" 141 ""
          "" "" "" 435 419)
      (422 "Sourcd_d" 3503 1 0 "'INLAID TRANSACTION'" 141 "" "" "" "" 424 0)
       (420 "LITERAL" 5001 3 "'INLAID TRANSACTION'" "'INLAID TRANSACTION'" 141
            "" "" "" "" 422 0)
      (423 "TO" 3005 1 0 "TO PRT-MESSAGE" 141 "" "" "" "" 424 422)
       (421 "ud_name" 5012 3 "PRT-MESSAGE" "PRT-MESSAGE" 141 "" "" "" "" 423
            0)
     (429 "Move_st" 2026 1 0 "MOVE TRANSACTION-RECORD TO PRT-REC" 142 "" "" ""
          "" 435 424)
      (427 "Sourcd_d" 3503 1 0 "TRANSACTION-RECORD" 142 "" "" "" "" 429 0)
       (425 "ud_name" 5012 3 "TRANSACTION-RECORD" "TRANSACTION-RECORD" 142 ""
            "" "" "" 427 0)
      (428 "TO" 3005 1 0 "TO PRT-REC" 142 "" "" "" "" 429 427)
       (426 "ud_name" 5012 3 "PRT-REC" "PRT-REC" 142 "" "" "" "" 428 0)
     (432 "Write_st" 2053 1 0 "WRITE PRINT-RECORD" 143 "" "" "" "" 435 429)
      (431 "Source_rec" 3503 1 0 "PRINT-RECORD" 143 "" "" "" "" 432 0)
       (430 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 143 "" "" "" "" 431
            0)
    (434 "ELSE" 3001 1 0 "ELSE" 145 "" "" "" "" 436 435)
     (433 "NEXT_SENTENCE" 6202 1 0 "NEXT SENTENCE" 145 "" "" "" "" 434 0)
  (437 "paragraph" 1004 3 "READ-TRANSACTION-EXIT" "" 146 "" "" "" "" 390 391)
   (438 "Exit_st" 2017 1 0 "EXIT" 147 "" "" "" "" 437 0)
 (439 "section" 1003 3 "EDIT" "" 154 "" "" "" "" 286 390)
  (440 "paragraph" 1004 3 "EDIT-ENTRY" "" 155 "" "" "" "" 439 0)
   (479 "If_st" 2021 1 0 "IF TR-FLIGHT-NUMBER IS NOT NUMERIC THEN ELSE" 156 ""
        "" "" "" 440 0)
    (444 "IS NOT" 5500 1 0 "TR-FLIGHT-NUMBER IS NOT NUMERIC" 156 "" "" "" ""
         479 0)
     (443 "Class_type" 3505 1 0 "TR-FLIGHT-NUMBER NUMERIC" 156 "" "" "" "" 444
          0)
      (441 "ud_name" 5012 3 "TR-FLIGHT-NUMBER" "TR-FLIGHT-NUMBER" 156 "" "" ""
           "" 443 0)
      (442 "CLASS_NODE" 5013 3 "NUMERIC" "NUMERIC" 156 "" "" "" "" 443 441)
    (478 "THEN" 3000 1 0 "THEN" 157 "" "" "" "" 479 444)
     (449 "Move_st" 2026 1 0 "MOVE '1' TO RP-INVALID-RECORD" 157 "" "" "" ""
          478 0)
  (447 "Sourcd_d" 3503 1 0 "'1'" 157 "" "" "" "" 449 0)
   (445 "LITERAL" 5001 3 "'1'" "'1'" 157 "" "" "" "" 447 0)
  (448 "TO" 3005 1 0 "TO RP-INVALID-RECORD" 157 "" "" "" "" 449 447)
   (446 "ud_name" 5012 3 "RP-INVALID-RECORD" "RP-INVALID-RECORD" 157 "" ""
        "" "" 448 0)
(477 "ELSE" 3001 1 0 "ELSE" 159 "" "" "" "" 479 478)
 (454 "Move_st" 2026 1 0 "MOVE TR-CREATION-CODE TO EW-THIS-CREATION" 159
      "" "" "" "" 477 0)
  (452 "Sourcd_d" 3503 1 0 "TR-CREATION-CODE" 159 "" "" "" "" 454 0)
   (450 "ud_name" 5012 3 "TR-CREATION-CODE" "TR-CREATION-CODE" 159 "" ""
        "" "" 452 0)
  (453 "TO" 3005 1 0 "TO EW-THIS-CREATION" 159 "" "" "" "" 454 452)
   (451 "ud_name" 5012 3 "EW-THIS-CREATION" "EW-THIS-CREATION" 159 "" ""
        "" "" 453 0)
 (459 "Move_st" 2026 1 0 "MOVE TR-FLIGHT-NUMBER TO EW-THIS-FLIGHT" 160 ""
      "" "" "" 477 454)
  (457 "Sourcd_d" 3503 1 0 "TR-FLIGHT-NUMBER" 160 "" "" "" "" 459 0)
   (455 "ud_name" 5012 3 "TR-FLIGHT-NUMBER" "TR-FLIGHT-NUMBER" 160 "" ""
        "" "" 457 0)
  (458 "TO" 3005 1 0 "TO EW-THIS-FLIGHT" 160 "" "" "" "" 459 457)
   (456 "ud_name" 5012 3 "EW-THIS-FLIGHT" "EW-THIS-FLIGHT" 160 "" "" "" ""
        458 0)
 (476 "If_st" 2021 1 0 "IF EW-THIS-FLIGHT-NUMBER NOT >
      EW-LAST-FLIGHT-NUMBER THEN ELSE" 161 "" "" "" "" 477 459)
  (463 "NOT" 5500 1 0 "EW-THIS-FLIGHT-NUMBER NOT > EW-LAST-FLIGHT-NUMBER"
       161 "" "" "" "" 476 0)
   (462 "Rel_op" 3504 3 ">" "EW-THIS-FLIGHT-NUMBER >
        EW-LAST-FLIGHT-NUMBER" 161 "" "" "" "" 463 0)
    (460 "ud_name" 5012 3 "EW-THIS-FLIGHT-NUMBER" "EW-THIS-FLIGHT-NUMBER"
         161 "" "" "" "" 462 0)
    (461 "ud_name" 5012 3 "EW-LAST-FLIGHT-NUMBER" "EW-LAST-FLIGHT-NUMBER"
         161 "" "" "" "" 462 460)
  (475 "THEN" 3000 1 0 "THEN" 162 "" "" "" "" 476 463)
   (468 "Move_st" 2026 1 0 "MOVE '1' TO RP-INVALID-RECORD" 162 "" "" "" ""
        475 0)
    (466 "Sourcd_d" 3503 1 0 "'1'" 162 "" "" "" "" 468 0)
     (464 "LITERAL" 5001 3 "'1'" "'1'" 162 "" "" "" "" 466 0)
    (467 "TO" 3005 1 0 "TO RP-INVALID-RECORD" 162 "" "" "" "" 468 466)
     (465 "ud_name" 5012 3 "RP-INVALID-RECORD" "RP-INVALID-RECORD" 162 ""
          "" "" "" 467 0)
  (474 "ELSE" 3001 1 0 "ELSE" 164 "" "" "" "" 476 475)
   (473 "Move_st" 2026 1 0 "MOVE EW-THIS-FLIGHT-NUMBER TO
        EW-LAST-FLIGHT-NUMBER" 164 "" "" "" "" 474 0)
      (471 "Sourcd_d" 3503 1 0 "EW-THIS-FLIGHT-NUMBER" 164 "" "" "" "" 473
           0)
       (469 "ud_name" 5012 3 "EW-THIS-FLIGHT-NUMBER" "EW-THIS-FLIGHT-NUMBER"
            164 "" "" "" "" 471 0)
      (472 "TO" 3005 1 0 "TO EW-LAST-FLIGHT-NUMBER" 164 "" "" "" "" 473 471)
       (470 "ud_name" 5012 3 "EW-LAST-FLIGHT-NUMBER" "EW-LAST-FLIGHT-NUMBER"
            164 "" "" "" "" 472 0)
 (480 "paragraph" 1004 3 "EDIT-EXIT" "" 165 "" "" "" "" 439 440)
  (481 "Exit_st" 2017 1 0 "EXIT" 166 "" "" "" "" 480 0)
(482 "section" 1003 3 "READ-RESERVATION" "" 176 "" "" "" "" 286 439)
 (483 "paragraph" 1004 3 "READ-RESERVATION-ENTRY" "" 177 "" "" "" "" 482 0)
  (492 "Read_st" 2035 1 0 "READ RESERVATIONS-FILE" 178 "" "" "" "" 483 0)
   (491 "Source_file" 3503 1 0 "RESERVATIONS-FILE" 178 "" "" "" "" 492 0)
    (484 "ud_name" 5012 3 "RESERVATIONS-FILE" "RESERVATIONS-FILE" 178 "" ""
         "" "" 491 0)
   (490 "AT END" 3016 1 0 "AT END" 178 "" "" "" "" 492 491)
    (489 "Move_st" 2026 1 0 "MOVE '1' TO RP-END-OF-RESERV" 178 "" "" "" ""
         490 0)
     (487 "Sourcd_d" 3503 1 0 "'1'" 178 "" "" "" "" 489 0)
      (485 "LITERAL" 5001 3 "'1'" "'1'" 178 "" "" "" "" 487 0)
     (488 "TO" 3005 1 0 "TO RP-END-OF-RESERV" 178 "" "" "" "" 489 487)
      (486 "ud_name" 5012 3 "RP-END-OF-RESERV" "RP-END-OF-RESERV" 178 "" ""
           "" "" 488 0)
 (493 "paragraph" 1004 3 "READ-RESERVATION-EXIT" "" 179 "" "" "" "" 482 483)
  (494 "Exit_st" 2017 1 0 "EXIT" 180 "" "" "" "" 493 0)
(495 "section" 1003 3 "UP-DATE" "" 191 "" "" "" "" 286 482)
 (496 "paragraph" 1004 3 "UP-DATE-ENTRY" "" 192 "" "" "" "" 495 0)
  (544 "If_st" 2021 1 0 "IF RP-END-OF-TRANS = '1'  AND RP-END-OF-RESERV =
       ZERO THEN ELSE" 193 "" "" "" "" 496 0)
   (503 "AND" 5503 1 0 "RP-END-OF-TRANS = '1'  AND RP-END-OF-RESERV = ZERO "
        193 "" "" "" "" 544 0)
    (499 "Rel_op" 3504 3 "=" "RP-END-OF-TRANS = '1' " 193 "" "" "" "" 503 0)
     (497 "ud_name" 5012 3 "RP-END-OF-TRANS" "RP-END-OF-TRANS" 193 "" "" ""
          "" 499 0)
     (498 "LITERAL" 5001 3 "'1'" "'1'" 193 "" "" "" "" 499 497)
    (502 "Rel_op" 3504 3 "=" "RP-END-OF-RESERV = ZERO " 193 "" "" "" "" 503
         499)
     (500 "ud_name" 5012 3 "RP-END-OF-RESERV" "RP-END-OF-RESERV" 193 "" "" ""
          "" 502 0)
     (501 "ZERO" 5004 1 0 "ZERO" 193 "" "" "" "" 502 500)
   (543 "THEN" 3000 1 0 "THEN" 194 "" "" "" "" 544 503)
    (509 "Perform_st" 2029 1 0 "PERFORM FINISH-RESERVATION BEFORE UNTIL
    RP-END-OF-RESERV = '1' " 194 "BEFORE" "" "" "" 543 0)
  (510 "Perform_body" 3501 1 0 "FINISH-RESERVATION" 194 "PROCEDURE" "" ""
       "" 509 0)
   (504 "ud_name" 5012 3 "FINISH-RESERVATION" "FINISH-RESERVATION" 194 ""
        "" "" "" 510 0)
  (508 "UNTIL" 3014 1 0 "UNTIL RP-END-OF-RESERV = '1' " 194 "" "" "" ""
       509 510)
   (507 "Rel_op" 3504 3 "=" "RP-END-OF-RESERV = '1' " 194 "" "" "" "" 508
        0)
    (505 "ud_name" 5012 3 "RP-END-OF-RESERV" "RP-END-OF-RESERV" 194 "" ""
         "" "" 507 0)
    (506 "LITERAL" 5001 3 "'1'" "'1'" 194 "" "" "" "" 507 505)
(542 "ELSE" 3001 1 0 "ELSE" 195 "" "" "" "" 544 543)
 (541 "If_st" 2021 1 0 "IF RP-END-OF-TRANS = ZERO  AND RP-END-OF-RESERV =
      '1' THEN ELSE" 195 "" "" "" "" 542 0)
  (517 "AND" 5503 1 0 "RP-END-OF-TRANS = ZERO  AND RP-END-OF-RESERV = '1'"
       195 "" "" "" "" 541 0)
   (513 "Rel_op" 3504 3 "=" "RP-END-OF-TRANS = ZERO " 195 "" "" "" "" 517
        0)
    (511 "ud_name" 5012 3 "RP-END-OF-TRANS" "RP-END-OF-TRANS" 195 "" "" ""
         "" 513 0)
    (512 "ZERO" 5004 1 0 "ZERO" 195 "" "" "" "" 513 511)
   (516 "Rel_op" 3504 3 "=" "RP-END-OF-RESERV = '1' " 196 "" "" "" "" 517
        513)
    (514 "ud_name" 5012 3 "RP-END-OF-RESERV" "RP-END-OF-RESERV" 196 "" ""
         "" "" 516 0)
    (515 "LITERAL" 5001 3 "'1'" "'1'" 196 "" "" "" "" 516 514)
  (540 "THEN" 3000 1 0 "THEN" 197 "" "" "" "" 541 517)
   (523 "Perform_st" 2029 1 0 "PERFORM FINISH-TRANSACTION BEFORE UNTIL
        RP-END-OF-TRANS = '1' " 197 "BEFORE" "" "" "" 540 0)
    (524 "Perform_body" 3501 1 0 "FINISH-TRANSACTION" 197 "PROCEDURE" ""
         "" "" 523 0)
     (518 "ud_name" 5012 3 "FINISH-TRANSACTION" "FINISH-TRANSACTION" 197
          "" "" "" "" 524 0)
    (522 "UNTIL" 3014 1 0 "UNTIL RP-END-OF-TRANS = '1' " 197 "" "" "" ""
         523 524)
     (521 "Rel_op" 3504 3 "=" "RP-END-OF-TRANS = '1' " 197 "" "" "" "" 522
          0)
      (519 "ud_name" 5012 3 "RP-END-OF-TRANS" "RP-END-OF-TRANS" 197 "" ""
           "" "" 521 0)
      (520 "LITERAL" 5001 3 "'1'" "'1'" 197 "" "" "" "" 521 519)
  (539 "ELSE" 3001 1 0 "ELSE" 198 "" "" "" "" 541 540)
```

```
      (538 "If_st" 2021 1 0 "IF RP-END-OF-TRANS = ZERO  AND RP-END-OF-RESERV
           = ZERO THEN ELSE" 198 "" "" "" "" 539 0)
       (531 "AND" 5503 1 0 "RP-END-OF-TRANS = ZERO  AND RP-END-OF-RESERV =
            ZERO " 198 "" "" "" "" 538 0)
        (527 "Rel_op" 3504 3 "=" "RP-END-OF-TRANS = ZERO " 198 "" "" "" ""
             531 0)
         (525 "ud_name" 5012 3 "RP-END-OF-TRANS" "RP-END-OF-TRANS" 198 "" ""
              "" "" 527 0)
         (526 "ZERO" 5004 1 0 "ZERO" 198 "" "" "" "" 527 525)
        (530 "Rel_op" 3504 3 "=" "RP-END-OF-RESERV = ZERO " 199 "" "" "" ""
             531 527)
         (528 "ud_name" 5012 3 "RP-END-OF-RESERV" "RP-END-OF-RESERV" 199 ""
              "" "" "" 530 0)
         (529 "ZERO" 5004 1 0 "ZERO" 199 "" "" "" "" 530 528)
       (537 "THEN" 3000 1 0 "THEN" 200 "" "" "" "" 538 531)
        (533 "Perform_st" 2029 1 0 "PERFORM MATCH-FLIGHT" 200 "" "" "" "" 537
             0)
         (534 "Perform_body" 3501 1 0 "MATCH-FLIGHT" 200 "PROCEDURE" "" "" ""
              533 0)
          (532 "ud_name" 5012 3 "MATCH-FLIGHT" "MATCH-FLIGHT" 200 "" "" "" ""
               534 0)
       (536 "ELSE" 3001 1 0 "ELSE" 202 "" "" "" "" 538 537)
        (535 "NEXT_SENTENCE" 6202 1 0 "NEXT SENTENCE" 202 "" "" "" "" 536 0)
 (545 "paragraph" 1004 3 "UP-DATE-EXIT" "" 203 "" "" "" "" 495 496)
  (546 "Exit_st" 2017 1 0 "EXIT" 204 "" "" "" "" 545 0)
(547 "section" 1003 3 "FINISH-RESERVATION" "" 215 "" "" "" "" 286 495)
 (548 "paragraph" 1004 3 "FINISH-RESERVATION-ENTRY" "" 216 "" "" "" "" 547 0)
  (550 "Perform_st" 2029 1 0 "PERFORM WRITE-NEW-RESER" 217 "" "" "" "" 548 0)
   (551 "Perform_body" 3501 1 0 "WRITE-NEW-RESER" 217 "PROCEDURE" "" "" ""
        550 0)
    (549 "ud_name" 5012 3 "WRITE-NEW-RESER" "WRITE-NEW-RESER" 217 "" "" "" ""
         551 0)
  (553 "Perform_st" 2029 1 0 "PERFORM READ-RESERVATION" 218 "" "" "" "" 548
       550)
   (554 "Perform_body" 3501 1 0 "READ-RESERVATION" 218 "PROCEDURE" "" "" ""
        553 0)
    (552 "ud_name" 5012 3 "READ-RESERVATION" "READ-RESERVATION" 218 "" "" ""
         "" 554 0)
 (555 "paragraph" 1004 3 "FINISH-RESERVATION-EXIT" "" 219 "" "" "" "" 547
      548)
  (556 "Exit_st" 2017 1 0 "EXIT" 220 "" "" "" "" 555 0)
(557 "section" 1003 3 "WRITE-NEW-RESER" "" 230 "" "" "" "" 286 547)
```

```
(558 "paragraph" 1004 3 "WRITE-NEW-RESER-ENTRY" "" 231 "" "" "" "" 557 0)
 (563 "Move_st" 2026 1 0 "MOVE RR-REC TO NRR-REC" 232 "" "" "" "" 558 0)
  (561 "Sourcd_d" 3503 1 0 "RR-REC" 232 "" "" "" "" 563 0)
   (559 "ud_name" 5012 3 "RR-REC" "RR-REC" 232 "" "" "" "" 561 0)
  (562 "TO" 3005 1 0 "TO NRR-REC" 232 "" "" "" "" 563 561)
   (560 "ud_name" 5012 3 "NRR-REC" "NRR-REC" 232 "" "" "" "" 562 0)
 (566 "Write_st" 2053 1 0 "WRITE NEW-RESERVATIONS-RECORD" 233 "" "" "" ""
     558 563)
  (565 "Source_rec" 3503 1 0 "NEW-RESERVATIONS-RECORD" 233 "" "" "" "" 566
      0)
   (564 "ud_name" 5012 3 "NEW-RESERVATIONS-RECORD" "NEW-RESERVATIONS-RECORD"
       233 "" "" "" "" 565 0)
(567 "paragraph" 1004 3 "WRITE-NEW-RESER-EXIT" "" 234 "" "" "" "" 557 558)
 (568 "Exit_st" 2017 1 0 "EXIT" 235 "" "" "" "" 567 0)
(569 "section" 1003 3 "FINISH-TRANSACTION" "" 246 "" "" "" "" 286 557)
(570 "paragraph" 1004 3 "FINISH-TRANSACTION-ENTRY" "" 247 "" "" "" "" 569 0)
 (605 "If_st" 2021 1 0 "IF RP-INVALID-RECORD = ZERO THEN ELSE" 248 "" "" ""
     "" 570 0)
  (573 "Rel_op" 3504 3 "=" "RP-INVALID-RECORD = ZERO " 248 "" "" "" "" 605
      0)
   (571 "ud_name" 5012 3 "RP-INVALID-RECORD" "RP-INVALID-RECORD" 248 "" ""
       "" "" 573 0)
   (572 "ZERO" 5004 1 0 "ZERO" 248 "" "" "" "" 573 571)
  (604 "THEN" 3000 1 0 "THEN" 249 "" "" "" "" 605 573)
   (601 "If_st" 2021 1 0 "IF TR-TRANSACTION-CODE = '1' THEN ELSE" 249 "" ""
       "" "" 604 0)
    (576 "Rel_op" 3504 3 "=" "TR-TRANSACTION-CODE = '1' " 249 "" "" "" ""
        601 0)
     (574 "ud_name" 5012 3 "TR-TRANSACTION-CODE" "TR-TRANSACTION-CODE" 249
         "" "" "" "" 576 0)
     (575 "LITERAL" 5001 3 "'1'" "'1'" 249 "" "" "" "" 576 574)
    (600 "THEN" 3000 1 0 "THEN" 250 "" "" "" "" 601 576)
     (578 "Perform_st" 2029 1 0 "PERFORM WRITE-TRANS-TO-RESER" 250 "" "" ""
         "" 600 0)
      (579 "Perform_body" 3501 1 0 "WRITE-TRANS-TO-RESER" 250 "PROCEDURE" ""
          "" "" 578 0)
       (577 "ud_name" 5012 3 "WRITE-TRANS-TO-RESER" "WRITE-TRANS-TO-RESER"
           250 "" "" "" "" 579 0)
     (584 "Move_st" 2026 1 0 "MOVE SPACES TO PRINT-RECORD" 251 "" "" "" ""
         600 578)
      (582 "Sourcd_d" 3503 1 0 "SPACES" 251 "" "" "" "" 584 0)
       (580 "SPACE" 5001 3 " " "SPACES" 251 "" "" "" "" 582 0)
```

```
        (583 "TO" 3005 1 0 "TO PRINT-RECORD" 251 "" "" "" "" 584 582)
         (581 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 251 "" "" "" ""
              583 0)
       (589 "Move_st" 2026 1 0 "MOVE 'RESERVATION ADDED' TO PRT-MESSAGE" 252
            "" "" "" "" 600 584)
        (587 "Sourcd_d" 3503 1 0 "'RESERVATION ADDED'" 252 "" "" "" "" 589 0)
         (585 "LITERAL" 5001 3 "'RESERVATION ADDED'" "'RESERVATION ADDED'" 252
              "" "" "" "" 587 0)
        (588 "TO" 3005 1 0 "TO PRT-MESSAGE" 252 "" "" "" "" 589 587)
         (586 "ud_name" 5012 3 "PRT-MESSAGE" "PRT-MESSAGE" 252 "" "" "" "" 588
              0)
       (594 "Move_st" 2026 1 0 "MOVE TRANSACTION-RECORD TO PRT-REC" 253 "" ""
            "" "" 600 589)
        (592 "Sourcd_d" 3503 1 0 "TRANSACTION-RECORD" 253 "" "" "" "" 594 0)
         (590 "ud_name" 5012 3 "TRANSACTION-RECORD" "TRANSACTION-RECORD" 253
              "" "" "" "" 592 0)
        (593 "TO" 3005 1 0 "TO PRT-REC" 253 "" "" "" "" 594 592)
         (591 "ud_name" 5012 3 "PRT-REC" "PRT-REC" 253 "" "" "" "" 593 0)
       (597 "Write_st" 2053 1 0 "WRITE PRINT-RECORD" 254 "" "" "" "" 600 594)
        (596 "Source_rec" 3503 1 0 "PRINT-RECORD" 254 "" "" "" "" 597 0)
         (595 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 254 "" "" "" ""
              596 0)
      (599 "ELSE" 3001 1 0 "ELSE" 256 "" "" "" "" 601 600)
       (598 "NEXT_SENTENCE" 6202 1 0 "NEXT SENTENCE" 256 "" "" "" "" 599 0)
    (603 "ELSE" 3001 1 0 "ELSE" 258 "" "" "" "" 605 604)
     (602 "NEXT_SENTENCE" 6202 1 0 "NEXT SENTENCE" 258 "" "" "" "" 603 0)
   (607 "Perform_st" 2029 1 0 "PERFORM READ-TRANSACTION" 259 "" "" "" "" 570
        605)
    (608 "Perform_body" 3501 1 0 "READ-TRANSACTION" 259 "PROCEDURE" "" "" ""
         607 0)
     (606 "ud_name" 5012 3 "READ-TRANSACTION" "READ-TRANSACTION" 259 "" "" ""
          "" 608 0)
  (609 "paragraph" 1004 3 "FINISH-TRANSACTION-EXIT" "" 260 "" "" "" "" 569
       570)
   (610 "Exit_st" 2017 1 0 "EXIT" 261 "" "" "" "" 609 0)
 (611 "section" 1003 3 "WRITE-TRANS-TO-RESER" "" 271 "" "" "" "" 286 569)
  (612 "paragraph" 1004 3 "WRITE-TRANS-TO-RESER-ENTRY" "" 272 "" "" "" "" 611
       0)
   (617 "Move_st" 2026 1 0 "MOVE TR-REC TO NRR-REC" 273 "" "" "" "" 612 0)
    (615 "Sourcd_d" 3503 1 0 "TR-REC" 273 "" "" "" "" 617 0)
     (613 "ud_name" 5012 3 "TR-REC" "TR-REC" 273 "" "" "" "" 615 0)
    (616 "TO" 3005 1 0 "TO NRR-REC" 273 "" "" "" "" 617 615)
     (614 "ud_name" 5012 3 "NRR-REC" "NRR-REC" 273 "" "" "" "" 616 0)
 (620 "Write_st" 2053 1 0 "WRITE NEW-RESERVATIONS-RECORD" 274 "" "" "" ""
      612 617)
  (619 "Source_rec" 3503 1 0 "NEW-RESERVATIONS-RECORD" 274 "" "" "" "" 620
       0)
   (618 "ud_name" 5012 3 "NEW-RESERVATIONS-RECORD" "NEW-RESERVATIONS-RECORD"
        274 "" "" "" "" 619 0)
(621 "paragraph" 1004 3 "WRITE-TRANS-TO-RESER-EXIT" "" 275 "" "" "" "" 611
     612)
 (622 "Exit_st" 2017 1 0 "EXIT" 276 "" "" "" "" 621 0)
(623 "section" 1003 3 "MATCH-FLIGHT" "" 289 "" "" "" "" 286 611)
(624 "paragraph" 1004 3 "MATCH-FLIGHT-ENTRY" "" 290 "" "" "" "" 623 0)
 (691 "If_st" 2021 1 0 "IF RP-INVALID-RECORD = '1' THEN ELSE" 291 "" "" ""
      "" 624 0)
  (627 "Rel_op" 3504 3 "=" "RP-INVALID-RECORD = '1' " 291 "" "" "" "" 691 0)
   (625 "ud_name" 5012 3 "RP-INVALID-RECORD" "RP-INVALID-RECORD" 291 "" ""
        "" "" 627 0)
   (626 "LITERAL" 5001 3 "'1'" "'1'" 291 "" "" "" "" 627 625)
  (690 "THEN" 3000 1 0 "THEN" 292 "" "" "" "" 691 627)
   (629 "Perform_st" 2029 1 0 "PERFORM READ-TRANSACTION" 292 "" "" "" "" 690
        0)
    (630 "Perform_body" 3501 1 0 "READ-TRANSACTION" 292 "PROCEDURE" "" "" ""
         629 0)
     (628 "ud_name" 5012 3 "READ-TRANSACTION" "READ-TRANSACTION" 292 "" ""
          "" "" 630 0)
  (689 "ELSE" 3001 1 0 "ELSE" 293 "" "" "" "" 691 690)
   (688 "If_st" 2021 1 0 "IF RR-FLIGHT-NUMBER < TR-FLIGHT-NUMBER THEN ELSE"
        293 "" "" "" "" 689 0)
    (633 "Rel_op" 3504 3 "<" "RR-FLIGHT-NUMBER < TR-FLIGHT-NUMBER " 293 ""
         "" "" "" 688 0)
     (631 "ud_name" 5012 3 "RR-FLIGHT-NUMBER" "RR-FLIGHT-NUMBER" 293 "" ""
          "" "" 633 0)
     (632 "ud_name" 5012 3 "TR-FLIGHT-NUMBER" "TR-FLIGHT-NUMBER" 293 "" ""
          "" "" 633 631)
    (687 "THEN" 3000 1 0 "THEN" 294 "" "" "" "" 688 633)
     (635 "Perform_st" 2029 1 0 "PERFORM WRITE-NEW-RESER" 294 "" "" "" ""
          687 0)
      (636 "Perform_body" 3501 1 0 "WRITE-NEW-RESER" 294 "PROCEDURE" "" ""
           "" 635 0)
       (634 "ud_name" 5012 3 "WRITE-NEW-RESER" "WRITE-NEW-RESER" 294 "" ""
            "" "" 636 0)
     (638 "Perform_st" 2029 1 0 "PERFORM READ-RESERVATION" 295 "" "" "" ""
          687 635)
  (639 "Perform_body" 3501 1 0 "READ-RESERVATION" 295 "PROCEDURE" "" ""
      "" 638 0)
   (637 "ud_name" 5012 3 "READ-RESERVATION" "READ-RESERVATION" 295 "" ""
       "" "" 639 0)
(686 "ELSE" 3001 1 0 "ELSE" 296 "" "" "" "" 688 687)
 (685 "If_st" 2021 1 0 "IF RR-FLIGHT-NUMBER > TR-FLIGHT-NUMBER THEN
     ELSE" 296 "" "" "" "" 686 0)
  (642 "Rel_op" 3504 3 ">" "RR-FLIGHT-NUMBER > TR-FLIGHT-NUMBER " 296 ""
      "" "" "" 685 0)
   (640 "ud_name" 5012 3 "RR-FLIGHT-NUMBER" "RR-FLIGHT-NUMBER" 296 "" ""
       "" "" 642 0)
   (641 "ud_name" 5012 3 "TR-FLIGHT-NUMBER" "TR-FLIGHT-NUMBER" 296 "" ""
       "" "" 642 640)
  (684 "THEN" 3000 1 0 "THEN" 297 "" "" "" "" 685 642)
   (647 "Move_st" 2026 1 0 "MOVE SPACES TO PRINT-RECORD" 297 "" "" "" ""
       684 0)
    (645 "Sourcd_d" 3503 1 0 "SPACES" 297 "" "" "" "" 647 0)
     (643 "SPACE" 5001 3 " " "SPACES" 297 "" "" "" "" 645 0)
    (646 "TO" 3005 1 0 "TO PRINT-RECORD" 297 "" "" "" "" 647 645)
     (644 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 297 "" "" "" ""
         646 0)
   (652 "Move_st" 2026 1 0 "MOVE 'RESERVATION RECORD ADDED' TO
       PRT-MESSAGE" 298 "" "" "" "" 684 647)
    (650 "Sourcd_d" 3503 1 0 "'RESERVATION RECORD ADDED'" 298 "" "" ""
        "" 652 0)
     (648 "LITERAL" 5001 3 "'RESERVATION RECORD ADDED'" "'RESERVATION
         RECORD ADDED'" 298 "" "" "" "" 650 0)
    (651 "TO" 3005 1 0 "TO PRT-MESSAGE" 298 "" "" "" "" 652 650)
     (649 "ud_name" 5012 3 "PRT-MESSAGE" "PRT-MESSAGE" 298 "" "" "" ""
         651 0)
   (657 "Move_st" 2026 1 0 "MOVE TRANSACTION-RECORD TO PRT-REC" 299 ""
       "" "" "" 684 652)
    (655 "Sourcd_d" 3503 1 0 "TRANSACTION-RECORD" 299 "" "" "" "" 657 0)
     (653 "ud_name" 5012 3 "TRANSACTION-RECORD" "TRANSACTION-RECORD" 299
         "" "" "" "" 655 0)
    (656 "TO" 3005 1 0 "TO PRT-REC" 299 "" "" "" "" 657 655)
     (654 "ud_name" 5012 3 "PRT-REC" "PRT-REC" 299 "" "" "" "" 656 0)
   (660 "Write_st" 2053 1 0 "WRITE PRINT-RECORD" 300 "" "" "" "" 684
       657)
    (659 "Source_rec" 3503 1 0 "PRINT-RECORD" 300 "" "" "" "" 660 0)
     (658 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 300 "" "" "" ""
         659 0)
   (662 "Perform_st" 2029 1 0 "PERFORM WRITE-TRANS-TO-RESER" 301 "" ""
      "" "" 684 660)
   (663 "Perform_body" 3501 1 0 "WRITE-TRANS-TO-RESER" 301 "PROCEDURE"
        "" "" "" 662 0)
    (661 "ud_name" 5012 3 "WRITE-TRANS-TO-RESER" "WRITE-TRANS-TO-RESER"
         301 "" "" "" "" 663 0)
  (665 "Perform_st" 2029 1 0 "PERFORM READ-TRANSACTION" 302 "" "" "" ""
       684 662)
   (666 "Perform_body" 3501 1 0 "READ-TRANSACTION" 302 "PROCEDURE" ""
        "" "" 665 0)
    (664 "ud_name" 5012 3 "READ-TRANSACTION" "READ-TRANSACTION" 302 ""
         "" "" "" 666 0)
 (683 "ELSE" 3001 1 0 "ELSE" 303 "" "" "" "" 685 684)
  (682 "If_st" 2021 1 0 "IF RR-FLIGHT-NUMBER = TR-FLIGHT-NUMBER THEN
       ELSE" 303 "" "" "" "" 683 0)
   (669 "Rel_op" 3504 3 "=" "RR-FLIGHT-NUMBER = TR-FLIGHT-NUMBER " 303
        "" "" "" "" 682 0)
    (667 "ud_name" 5012 3 "RR-FLIGHT-NUMBER" "RR-FLIGHT-NUMBER" 303 ""
         "" "" "" 669 0)
    (668 "ud_name" 5012 3 "TR-FLIGHT-NUMBER" "TR-FLIGHT-NUMBER" 303 ""
         "" "" "" 669 667)
   (681 "THEN" 3000 1 0 "THEN" 304 "" "" "" "" 682 669)
    (671 "Perform_st" 2029 1 0 "PERFORM UPDATE-RESERVATION" 304 "" ""
         "" "" 681 0)
     (672 "Perform_body" 3501 1 0 "UPDATE-RESERVATION" 304 "PROCEDURE"
          "" "" "" 671 0)
      (670 "ud_name" 5012 3 "UPDATE-RESERVATION" "UPDATE-RESERVATION"
           304 "" "" "" "" 672 0)
    (674 "Perform_st" 2029 1 0 "PERFORM READ-RESERVATION" 305 "" "" ""
         "" 681 671)
     (675 "Perform_body" 3501 1 0 "READ-RESERVATION" 305 "PROCEDURE" ""
          "" "" 674 0)
      (673 "ud_name" 5012 3 "READ-RESERVATION" "READ-RESERVATION" 305
           "" "" "" "" 675 0)
    (677 "Perform_st" 2029 1 0 "PERFORM READ-TRANSACTION" 306 "" "" ""
         "" 681 674)
     (678 "Perform_body" 3501 1 0 "READ-TRANSACTION" 306 "PROCEDURE" ""
          "" "" 677 0)
      (676 "ud_name" 5012 3 "READ-TRANSACTION" "READ-TRANSACTION" 306
           "" "" "" "" 678 0)
   (680 "ELSE" 3001 1 0 "ELSE" 308 "" "" "" "" 682 681)
    (679 "NEXT_SENTENCE" 6202 1 0 "NEXT SENTENCE" 308 "" "" "" "" 680
         0)
```

```
(692 "paragraph" 1004 3 "MATCH-FLIGHT-EXIT" "" 309 "" "" "" "" 623 624)
 (693 "Exit_st" 2017 1 0 "EXIT" 310 "" "" "" "" 692 0)
(694 "section" 1003 3 "UPDATE-RESERVATION" "" 319 "" "" "" "" 286 623)
 (695 "paragraph" 1004 3 "UPDATE-RESERVATION-ENTRY" "" 320 "" "" "" "" 694 0)
  (784 "If_st" 2021 1 0 "IF TR-TRANSACTION-CODE = ZERO THEN ELSE" 321 "" ""
      "" "" 695 0)
   (698 "Rel_op" 3504 3 "=" "TR-TRANSACTION-CODE = ZERO " 321 "" "" "" "" 784
       0)
    (696 "ud_name" 5012 3 "TR-TRANSACTION-CODE" "TR-TRANSACTION-CODE" 321 ""
        "" "" "" 698 0)
    (697 "ZERO" 5004 1 0 "ZERO" 321 "" "" "" "" 698 696)
   (783 "THEN" 3000 1 0 "THEN" 322 "" "" "" "" 784 698)
    (700 "Perform_st" 2029 1 0 "PERFORM MOD-IFY" 322 "" "" "" "" 783 0)
     (701 "Perform_body" 3501 1 0 "MOD-IFY" 322 "PROCEDURE" "" "" "" 700 0)
      (699 "ud_name" 5012 3 "MOD-IFY" "MOD-IFY" 322 "" "" "" "" 701 0)
    (732 "If_st" 2021 1 0 "IF RR-PASSENGER-NAME1 = SPACES  AND
        RR-PASSENGER-NAME2 = SPACES THEN ELSE" 323 "" "" "" "" 783 700)
     (708 "AND" 5503 1 0 "RR-PASSENGER-NAME1 = SPACES  AND
         RR-PASSENGER-NAME2 = SPACES " 323 "" "" "" "" 732 0)
      (704 "Rel_op" 3504 3 "=" "RR-PASSENGER-NAME1 = SPACES " 323 "" "" "" ""
          708 0)
       (702 "ud_name" 5012 3 "RR-PASSENGER-NAME1" "RR-PASSENGER-NAME1" 323 ""
           "" "" "" 704 0)
       (703 "SPACE" 5001 3 " " "SPACES" 323 "" "" "" "" 704 702)
      (707 "Rel_op" 3504 3 "=" "RR-PASSENGER-NAME2 = SPACES " 324 "" "" "" ""
          708 704)
       (705 "ud_name" 5012 3 "RR-PASSENGER-NAME2" "RR-PASSENGER-NAME2" 324 ""
           "" "" "" 707 0)
       (706 "SPACE" 5001 3 " " "SPACES" 324 "" "" "" "" 707 705)
     (731 "THEN" 3000 1 0 "THEN" 325 "" "" "" "" 732 708)
      (713 "Move_st" 2026 1 0 "MOVE SPACES TO PRINT-RECORD" 325 "" "" "" ""
          731 0)
       (711 "Sourcd_d" 3503 1 0 "SPACES" 325 "" "" "" "" 713 0)
        (709 "SPACE" 5001 3 " " "SPACES" 325 "" "" "" "" 711 0)
       (712 "TO" 3005 1 0 "TO PRINT-RECORD" 325 "" "" "" "" 713 711)
        (710 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 325 "" "" "" ""
            712 0)
      (718 "Move_st" 2026 1 0 "MOVE 'RESERVATION RECORD DELETED' TO
          PRT-MESSAGE" 326 "" "" "" "" 731 713)
       (716 "Sourcd_d" 3503 1 0 "'RESERVATION RECORD DELETED'" 326 "" "" ""
           "" 718 0)
        (714 "LITERAL" 5001 3 "'RESERVATION RECORD DELETED'" "'RESERVATION
          RECORD DELETED'" 326 "" "" "" "" 716 0)
    (717 "TO" 3005 1 0 "TO PRT-MESSAGE" 326 "" "" "" "" 718 716)
     (715 "ud_name" 5012 3 "PRT-MESSAGE" "PRT-MESSAGE" 326 "" "" "" "" 717
          0)
   (723 "Move_st" 2026 1 0 "MOVE RESERVATIONS-RECORD TO PRT-REC" 327 "" ""
        "" "" 731 718)
    (721 "Sourcd_d" 3503 1 0 "RESERVATIONS-RECORD" 327 "" "" "" "" 723 0)
     (719 "ud_name" 5012 3 "RESERVATIONS-RECORD" "RESERVATIONS-RECORD" 327
          "" "" "" "" 721 0)
    (722 "TO" 3005 1 0 "TO PRT-REC" 327 "" "" "" "" 723 721)
     (720 "ud_name" 5012 3 "PRT-REC" "PRT-REC" 327 "" "" "" "" 722 0)
   (726 "Write_st" 2053 1 0 "WRITE PRINT-RECORD" 328 "" "" "" "" 731 723)
    (725 "Source_rec" 3503 1 0 "PRINT-RECORD" 328 "" "" "" "" 726 0)
     (724 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 328 "" "" "" ""
          725 0)
  (730 "ELSE" 3001 1 0 "ELSE" 330 "" "" "" "" 732 731)
   (728 "Perform_st" 2029 1 0 "PERFORM WRITE-NEW-RESER" 330 "" "" "" ""
        730 0)
    (729 "Perform_body" 3501 1 0 "WRITE-NEW-RESER" 330 "PROCEDURE" "" ""
         "" 728 0)
     (727 "ud_name" 5012 3 "WRITE-NEW-RESER" "WRITE-NEW-RESER" 330 "" ""
          "" "" 729 0)
(782 "ELSE" 3001 1 0 "ELSE" 331 "" "" "" "" 784 783)
 (781 "If_st" 2021 1 0 "IF RR-PASSENGER-NAME1 = SPACES  OR
      RR-PASSENGER-NAME2 = SPACES THEN ELSE" 331 "" "" "" "" 782 0)
  (739 "OR" 5501 1 0 "RR-PASSENGER-NAME1 = SPACES  OR RR-PASSENGER-NAME2 =
       SPACES " 331 "" "" "" "" 781 0)
   (735 "Rel_op" 3504 3 "=" "RR-PASSENGER-NAME1 = SPACES " 331 "" "" "" ""
        739 0)
    (733 "ud_name" 5012 3 "RR-PASSENGER-NAME1" "RR-PASSENGER-NAME1" 331 ""
         "" "" "" 735 0)
    (734 "SPACE" 5001 3 " " "SPACES" 331 "" "" "" "" 735 733)
   (738 "Rel_op" 3504 3 "=" "RR-PASSENGER-NAME2 = SPACES " 332 "" "" "" ""
        739 735)
    (736 "ud_name" 5012 3 "RR-PASSENGER-NAME2" "RR-PASSENGER-NAME2" 332 ""
         "" "" "" 738 0)
    (737 "SPACE" 5001 3 " " "SPACES" 332 "" "" "" "" 738 736)
  (780 "THEN" 3000 1 0 "THEN" 333 "" "" "" "" 781 739)
   (741 "Perform_st" 2029 1 0 "PERFORM MOD-IFY" 333 "" "" "" "" 780 0)
    (742 "Perform_body" 3501 1 0 "MOD-IFY" 333 "PROCEDURE" "" "" "" 741 0)
     (740 "ud_name" 5012 3 "MOD-IFY" "MOD-IFY" 333 "" "" "" "" 742 0)
   (747 "Move_st" 2026 1 0 "MOVE SPACES TO PRINT-RECORD" 334 "" "" "" ""
     780 741)
  (745 "Sourcd_d" 3503 1 0 "SPACES" 334 "" "" "" "" 747 0)
   (743 "SPACE" 5001 3 " " "SPACES" 334 "" "" "" "" 745 0)
  (746 "TO" 3005 1 0 "TO PRINT-RECORD" 334 "" "" "" "" 747 745)
   (744 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 334 "" "" "" ""
        746 0)
 (752 "Move_st" 2026 1 0 "MOVE 'RESERVATION ADDED' TO PRT-MESSAGE" 335
      "" "" "" "" 780 747)
  (750 "Sourcd_d" 3503 1 0 "'RESERVATION ADDED'" 335 "" "" "" "" 752 0)
   (748 "LITERAL" 5001 3 "'RESERVATION ADDED'" "'RESERVATION ADDED'" 335
        "" "" "" "" 750 0)
  (751 "TO" 3005 1 0 "TO PRT-MESSAGE" 335 "" "" "" "" 752 750)
   (749 "ud_name" 5012 3 "PRT-MESSAGE" "PRT-MESSAGE" 335 "" "" "" "" 751
        0)
 (757 "Move_st" 2026 1 0 "MOVE TRANSACTION-RECORD TO PRT-REC" 336 "" ""
      "" "" 780 752)
  (755 "Sourcd_d" 3503 1 0 "TRANSACTION-RECORD" 336 "" "" "" "" 757 0)
   (753 "ud_name" 5012 3 "TRANSACTION-RECORD" "TRANSACTION-RECORD" 336
        "" "" "" "" 755 0)
  (756 "TO" 3005 1 0 "TO PRT-REC" 336 "" "" "" "" 757 755)
   (754 "ud_name" 5012 3 "PRT-REC" "PRT-REC" 336 "" "" "" "" 756 0)
 (760 "Write_st" 2053 1 0 "WRITE PRINT-RECORD" 337 "" "" "" "" 780 757)
  (759 "Source_rec" 3503 1 0 "PRINT-RECORD" 337 "" "" "" "" 760 0)
   (758 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 337 "" "" "" ""
        759 0)
(779 "ELSE" 3001 1 0 "ELSE" 339 "" "" "" "" 781 780)
 (765 "Move_st" 2026 1 0 "MOVE SPACES TO PRINT-RECORD" 339 "" "" "" ""
      779 0)
  (763 "Sourcd_d" 3503 1 0 "SPACES" 339 "" "" "" "" 765 0)
   (761 "SPACE" 5001 3 " " "SPACES" 339 "" "" "" "" 763 0)
  (764 "TO" 3005 1 0 "TO PRINT-RECORD" 339 "" "" "" "" 765 763)
   (762 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 339 "" "" "" ""
        764 0)
 (770 "Move_st" 2026 1 0 "MOVE 'NO ROOM ON FLIGHT' TO PRT-MESSAGE" 340
      "" "" "" "" 779 765)
  (768 "Sourcd_d" 3503 1 0 "'NO ROOM ON FLIGHT'" 340 "" "" "" "" 770 0)
   (766 "LITERAL" 5001 3 "'NO ROOM ON FLIGHT'" "'NO ROOM ON FLIGHT'" 340
        "" "" "" "" 768 0)
  (769 "TO" 3005 1 0 "TO PRT-MESSAGE" 340 "" "" "" "" 770 768)
   (767 "ud_name" 5012 3 "PRT-MESSAGE" "PRT-MESSAGE" 340 "" "" "" "" 769
        0)
 (775 "Move_st" 2026 1 0 "MOVE TRANSACTION-RECORD TO PRT-REC" 341 "" ""
      "" "" 779 770)
```

```
        (773 "Sourcd_d" 3503 1 0 "TRANSACTION-RECORD" 341 "" "" "" "" 775 0)
         (771 "ud_name" 5012 3 "TRANSACTION-RECORD" "TRANSACTION-RECORD" 341
              "" "" "" "" 773 0)
        (774 "TO" 3005 1 0 "TO PRT-REC" 341 "" "" "" "" 775 773)
         (772 "ud_name" 5012 3 "PRT-REC" "PRT-REC" 341 "" "" "" "" 774 0)
       (778 "Write_st" 2053 1 0 "WRITE PRINT-RECORD" 342 "" "" "" "" 779 775)
        (777 "Source_rec" 3503 1 0 "PRINT-RECORD" 342 "" "" "" "" 778 0)
         (776 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 342 "" "" "" ""
              777 0)
  (785 "paragraph" 1004 3 "UPDATE-RESERVATION-EXIT" "" 343 "" "" "" "" 694
       695)
   (786 "Exit_st" 2017 1 0 "EXIT" 344 "" "" "" "" 785 0)
 (787 "section" 1003 3 "MOD-IFY" "" 354 "" "" "" "" 286 694)
  (788 "paragraph" 1004 3 "MOD-IFY-ENTRY" "" 355 "" "" "" "" 787 0)
   (834 "If_st" 2021 1 0 "IF TR-TRANSACTION-CODE = ZERO THEN ELSE" 356 "" ""
        "" "" 788 0)
    (791 "Rel_op" 3504 3 "=" "TR-TRANSACTION-CODE = ZERO " 356 "" "" "" "" 834
         0)
     (789 "ud_name" 5012 3 "TR-TRANSACTION-CODE" "TR-TRANSACTION-CODE" 356 ""
          "" "" "" 791 0)
     (790 "ZERO" 5004 1 0 "ZERO" 356 "" "" "" "" 791 789)
    (833 "THEN" 3000 1 0 "THEN" 357 "" "" "" "" 834 791)
     (830 "If_st" 2021 1 0 "IF TR-PASSENGER-NAME1 = RR-PASSENGER-NAME1  OR
          TR-PASSENGER-NAME2 = RR-PASSENGER-NAME1 THEN ELSE" 357 "" "" "" ""
          833 0)
      (798 "OR" 5501 1 0 "TR-PASSENGER-NAME1 = RR-PASSENGER-NAME1  OR
           TR-PASSENGER-NAME2 = RR-PASSENGER-NAME1 " 357 "" "" "" "" 830 0)
       (794 "Rel_op" 3504 3 "=" "TR-PASSENGER-NAME1 = RR-PASSENGER-NAME1 "
            357 "" "" "" "" 798 0)
        (792 "ud_name" 5012 3 "TR-PASSENGER-NAME1" "TR-PASSENGER-NAME1" 357 ""
             "" "" "" 794 0)
        (793 "ud_name" 5012 3 "RR-PASSENGER-NAME1" "RR-PASSENGER-NAME1" 357 ""
             "" "" "" 794 792)
       (797 "Rel_op" 3504 3 "=" "TR-PASSENGER-NAME2 = RR-PASSENGER-NAME1 " 358
            "" "" "" "" 798 794)
        (795 "ud_name" 5012 3 "TR-PASSENGER-NAME2" "TR-PASSENGER-NAME2" 358 ""
             "" "" "" 797 0)
        (796 "ud_name" 5012 3 "RR-PASSENGER-NAME1" "RR-PASSENGER-NAME1" 358 ""
             "" "" "" 797 795)
      (829 "THEN" 3000 1 0 "THEN" 359 "" "" "" "" 830 798)
       (803 "Move_st" 2026 1 0 "MOVE SPACES TO RR-PASSENGER-NAME1" 359 "" ""
            "" "" 829 0)
 (801 "Sourcd_d" 3503 1 0 "SPACES" 359 "" "" "" "" 803 0)
  (799 "SPACE" 5001 3 " " "SPACES" 359 "" "" "" "" 801 0)
 (802 "TO" 3005 1 0 "TO RR-PASSENGER-NAME1" 359 "" "" "" "" 803 801)
  (800 "ud_name" 5012 3 "RR-PASSENGER-NAME1" "RR-PASSENGER-NAME1" 359
       "" "" "" "" 802 0)
(806 "Goto_st" 2019 1 0 "GO TO MOD-IFY-EXIT" 360 "" "" "" "" 829 803)
 (805 "Procedure_d" 3503 1 0 "MOD-IFY-EXIT" 360 "" "" "" "" 806 0)
  (804 "ud_name" 5012 3 "MOD-IFY-EXIT" "MOD-IFY-EXIT" 360 "" "" "" ""
       805 0)
(828 "ELSE" 3001 1 0 "ELSE" 361 "" "" "" "" 830 829)
 (827 "If_st" 2021 1 0 "IF TR-PASSENGER-NAME1 = RR-PASSENGER-NAME2  OR
     TR-PASSENGER-NAME2 = RR-PASSENGER-NAME2 THEN ELSE" 361 "" "" ""
     "" 828 0)
  (813 "OR" 5501 1 0 "TR-PASSENGER-NAME1 = RR-PASSENGER-NAME2  OR
      TR-PASSENGER-NAME2 = RR-PASSENGER-NAME2 " 361 "" "" "" "" 827 0)
   (809 "Rel_op" 3504 3 "=" "TR-PASSENGER-NAME1 = RR-PASSENGER-NAME2 "
       361 "" "" "" "" 813 0)
    (807 "ud_name" 5012 3 "TR-PASSENGER-NAME1" "TR-PASSENGER-NAME1" 361
        "" "" "" "" 809 0)
    (808 "ud_name" 5012 3 "RR-PASSENGER-NAME2" "RR-PASSENGER-NAME2" 361
        "" "" "" "" 809 807)
   (812 "Rel_op" 3504 3 "=" "TR-PASSENGER-NAME2 = RR-PASSENGER-NAME2 "
       362 "" "" "" "" 813 809)
    (810 "ud_name" 5012 3 "TR-PASSENGER-NAME2" "TR-PASSENGER-NAME2" 362
        "" "" "" "" 812 0)
    (811 "ud_name" 5012 3 "RR-PASSENGER-NAME2" "RR-PASSENGER-NAME2" 362
        "" "" "" "" 812 810)
  (826 "THEN" 3000 1 0 "THEN" 363 "" "" "" "" 827 813)
   (818 "Move_st" 2026 1 0 "MOVE SPACES TO RR-PASSENGER-NAME2" 363 "" ""
       "" "" 826 0)
    (816 "Sourcd_d" 3503 1 0 "SPACES" 363 "" "" "" "" 818 0)
     (814 "SPACE" 5001 3 " " "SPACES" 363 "" "" "" "" 816 0)
    (817 "TO" 3005 1 0 "TO RR-PASSENGER-NAME2" 363 "" "" "" "" 818 816)
     (815 "ud_name" 5012 3 "RR-PASSENGER-NAME2" "RR-PASSENGER-NAME2" 363
         "" "" "" "" 817 0)
   (821 "Goto_st" 2019 1 0 "GO TO MOD-IFY-EXIT" 364 "" "" "" "" 826 818)
    (820 "Procedure_d" 3503 1 0 "MOD-IFY-EXIT" 364 "" "" "" "" 821 0)
     (819 "ud_name" 5012 3 "MOD-IFY-EXIT" "MOD-IFY-EXIT" 364 "" "" "" ""
         820 0)
  (825 "ELSE" 3001 1 0 "ELSE" 366 "" "" "" "" 827 826)
   (824 "Goto_st" 2019 1 0 "GO TO MOD-IFY-EXIT" 366 "" "" "" "" 825 0)
    (823 "Procedure_d" 3503 1 0 "MOD-IFY-EXIT" 366 "" "" "" "" 824 0)
     (822 "ud_name" 5012 3 "MOD-IFY-EXIT" "MOD-IFY-EXIT" 366 "" "" "" ""
          823 0)
 (832 "ELSE" 3001 1 0 "ELSE" 368 "" "" "" "" 834 833)
  (831 "NEXT_SENTENCE" 6202 1 0 "NEXT SENTENCE" 368 "" "" "" "" 832 0)
(887 "If_st" 2021 1 0 "IF RR-PASSENGER-NAME1 = SPACES THEN ELSE" 369 "" ""
     "" "" 788 834)
 (837 "Rel_op" 3504 3 "=" "RR-PASSENGER-NAME1 = SPACES " 369 "" "" "" ""
      887 0)
  (835 "ud_name" 5012 3 "RR-PASSENGER-NAME1" "RR-PASSENGER-NAME1" 369 "" ""
      "" "" 837 0)
  (836 "SPACE" 5001 3 " " "SPACES" 369 "" "" "" "" 837 835)
 (886 "THEN" 3000 1 0 "THEN" 370 "" "" "" "" 887 837)
  (842 "Move_st" 2026 1 0 "MOVE TR-PASSENGER-NAME1 TO RR-PASSENGER-NAME1"
      370 "" "" "" "" 886 0)
   (840 "Sourcd_d" 3503 1 0 "TR-PASSENGER-NAME1" 370 "" "" "" "" 842 0)
    (838 "ud_name" 5012 3 "TR-PASSENGER-NAME1" "TR-PASSENGER-NAME1" 370 ""
        "" "" "" 840 0)
   (841 "TO" 3005 1 0 "TO RR-PASSENGER-NAME1" 370 "" "" "" "" 842 840)
    (839 "ud_name" 5012 3 "RR-PASSENGER-NAME1" "RR-PASSENGER-NAME1" 370 ""
        "" "" "" 841 0)
 (885 "ELSE" 3001 1 0 "ELSE" 372 "" "" "" "" 887 886)
  (847 "Move_st" 2026 1 0 "MOVE TR-PASSENGER-NAME1 TO RR-PASSENGER-NAME2"
      372 "" "" "" "" 885 0)
   (845 "Sourcd_d" 3503 1 0 "TR-PASSENGER-NAME1" 372 "" "" "" "" 847 0)
    (843 "ud_name" 5012 3 "TR-PASSENGER-NAME1" "TR-PASSENGER-NAME1" 372 ""
        "" "" "" 845 0)
   (846 "TO" 3005 1 0 "TO RR-PASSENGER-NAME2" 372 "" "" "" "" 847 845)
    (844 "ud_name" 5012 3 "RR-PASSENGER-NAME2" "RR-PASSENGER-NAME2" 372 ""
        "" "" "" 846 0)
  (884 "If_st" 2021 1 0 "IF TR-PASSENGER-NAME2 NOT = SPACES THEN ELSE" 373
      "" "" "" "" 885 847)
   (851 "NOT" 5500 1 0 "TR-PASSENGER-NAME2 NOT = SPACES" 373 "" "" "" ""
       884 0)
    (850 "Rel_op" 3504 3 "=" "TR-PASSENGER-NAME2 = SPACES" 373 "" "" "" ""
        851 0)
     (848 "ud_name" 5012 3 "TR-PASSENGER-NAME2" "TR-PASSENGER-NAME2" 373 ""
         "" "" "" 850 0)
     (849 "SPACE" 5001 3 " " "SPACES" 373 "" "" "" "" 850 848)
   (883 "THEN" 3000 1 0 "THEN" 374 "" "" "" "" 884 851)
    (880 "If_st" 2021 1 0 "IF RR-PASSENGER-NAME2 = SPACES THEN ELSE" 374
        "" "" "" "" 883 0)
     (854 "Rel_op" 3504 3 "=" "RR-PASSENGER-NAME2 = SPACES " 374 "" "" ""
         "" 880 0)
      (852 "ud_name" 5012 3 "RR-PASSENGER-NAME2" "RR-PASSENGER-NAME2" 374
        "" "" "" "" 854 0)
      (853 "SPACE" 5001 3 " " "SPACES" 374 "" "" "" "" 854 852)
     (879 "THEN" 3000 1 0 "THEN" 375 "" "" "" "" 880 854)
      (859 "Move_st" 2026 1 0 "MOVE TR-PASSENGER-NAME2 TO
           RR-PASSENGER-NAME2" 375 "" "" "" "" 879 0)
       (857 "Sourcd_d" 3503 1 0 "TR-PASSENGER-NAME2" 375 "" "" "" "" 859 0)
        (855 "ud_name" 5012 3 "TR-PASSENGER-NAME2" "TR-PASSENGER-NAME2" 375
             "" "" "" "" 857 0)
       (858 "TO" 3005 1 0 "TO RR-PASSENGER-NAME2" 375 "" "" "" "" 859 857)
        (856 "ud_name" 5012 3 "RR-PASSENGER-NAME2" "RR-PASSENGER-NAME2" 375
             "" "" "" "" 858 0)
     (878 "ELSE" 3001 1 0 "ELSE" 377 "" "" "" "" 880 879)
      (864 "Move_st" 2026 1 0 "MOVE SPACES TO PRINT-RECORD" 377 "" "" "" ""
           878 0)
       (862 "Sourcd_d" 3503 1 0 "SPACES" 377 "" "" "" "" 864 0)
        (860 "SPACE" 5001 3 " " "SPACES" 377 "" "" "" "" 862 0)
       (863 "TO" 3005 1 0 "TO PRINT-RECORD" 377 "" "" "" "" 864 862)
        (861 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 377 "" "" "" ""
             863 0)
      (869 "Move_st" 2026 1 0 "MOVE 'NO ROOM FOR 2ND PASSENGER' TO
           PRT-MESSAGE" 378 "" "" "" "" 878 864)
       (867 "Sourcd_d" 3503 1 0 "'NO ROOM FOR 2ND PASSENGER'" 378 "" "" ""
            "" 869 0)
        (865 "LITERAL" 5001 3 "'NO ROOM FOR 2ND PASSENGER'" "'NO ROOM FOR
             2ND PASSENGER'" 378 "" "" "" "" 867 0)
       (868 "TO" 3005 1 0 "TO PRT-MESSAGE" 378 "" "" "" "" 869 867)
        (866 "ud_name" 5012 3 "PRT-MESSAGE" "PRT-MESSAGE" 378 "" "" "" ""
             868 0)
      (874 "Move_st" 2026 1 0 "MOVE TRANSACTION-RECORD TO PRT-REC" 379 ""
           "" "" "" 878 869)
       (872 "Sourcd_d" 3503 1 0 "TRANSACTION-RECORD" 379 "" "" "" "" 874 0)
        (870 "ud_name" 5012 3 "TRANSACTION-RECORD" "TRANSACTION-RECORD" 379
             "" "" "" "" 872 0)
       (873 "TO" 3005 1 0 "TO PRT-REC" 379 "" "" "" "" 874 872)
        (871 "ud_name" 5012 3 "PRT-REC" "PRT-REC" 379 "" "" "" "" 873 0)
      (877 "Write_st" 2053 1 0 "WRITE PRINT-RECORD" 380 "" "" "" "" 878
           874)
       (876 "Source_rec" 3503 1 0 "PRINT-RECORD" 380 "" "" "" "" 877 0)
        (875 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 380 "" "" "" ""
             876 0)
   (882 "ELSE" 3001 1 0 "ELSE" 382 "" "" "" "" 884 883)
    (881 "NEXT_SENTENCE" 6202 1 0 "NEXT SENTENCE" 382 "" "" "" "" 882 0)
(888 "paragraph" 1004 3 "MOD-IFY-EXIT" "" 383 "" "" "" "" 787 788)
```

```
  (889 "Exit_st" 2017 1 0 "EXIT" 384 "" "" "" "" 888 0)
(890 "section" 1003 3 "RE-PORT" "" 392 "" "" "" "" 286 787)
 (891 "paragraph" 1004 3 "RE-PORT-ENTRY" "" 393 "" "" "" "" 890 0)
  (903 "Read_st" 2035 1 0 "READ NEW-RESERVATIONS-FILE" 394 "" "" "" "" 891 0)
   (902 "Source_file" 3503 1 0 "NEW-RESERVATIONS-FILE" 394 "" "" "" "" 903 0)
    (892 "ud_name" 5012 3 "NEW-RESERVATIONS-FILE" "NEW-RESERVATIONS-FILE" 394
         "" "" "" "" 902 0)
   (901 "AT END" 3016 1 0 "AT END" 395 "" "" "" "" 903 902)
    (897 "Move_st" 2026 1 0 "MOVE '1' TO RP-END-OF-RESERV" 395 "" "" "" ""
         901 0)
     (895 "Sourcd_d" 3503 1 0 "'1'" 395 "" "" "" "" 897 0)
      (893 "LITERAL" 5001 3 "'1'" "'1'" 395 "" "" "" "" 895 0)
     (896 "TO" 3005 1 0 "TO RP-END-OF-RESERV" 395 "" "" "" "" 897 895)
      (894 "ud_name" 5012 3 "RP-END-OF-RESERV" "RP-END-OF-RESERV" 395 "" ""
           "" "" 896 0)
    (900 "Goto_st" 2019 1 0 "GO TO RE-PORT-EXIT" 396 "" "" "" "" 901 897)
     (899 "Procedure_d" 3503 1 0 "RE-PORT-EXIT" 396 "" "" "" "" 900 0)
      (898 "ud_name" 5012 3 "RE-PORT-EXIT" "RE-PORT-EXIT" 396 "" "" "" "" 899
           0)
  (908 "Move_st" 2026 1 0 "MOVE NEW-RESERVATIONS-RECORD TO PRT-REC" 397 "" ""
       "" "" 891 903)
   (906 "Sourcd_d" 3503 1 0 "NEW-RESERVATIONS-RECORD" 397 "" "" "" "" 908 0)
    (904 "ud_name" 5012 3 "NEW-RESERVATIONS-RECORD" "NEW-RESERVATIONS-RECORD"
         397 "" "" "" "" 906 0)
   (907 "TO" 3005 1 0 "TO PRT-REC" 397 "" "" "" "" 908 906)
    (905 "ud_name" 5012 3 "PRT-REC" "PRT-REC" 397 "" "" "" "" 907 0)
  (911 "Write_st" 2053 1 0 "WRITE PRINT-RECORD" 398 "" "" "" "" 891 908)
   (910 "Source_rec" 3503 1 0 "PRINT-RECORD" 398 "" "" "" "" 911 0)
    (909 "ud_name" 5012 3 "PRINT-RECORD" "PRINT-RECORD" 398 "" "" "" "" 910
         0)
 (912 "paragraph" 1004 3 "RE-PORT-EXIT" "" 399 "" "" "" "" 890 891)
  (913 "Exit_st" 2017 1 0 "EXIT" 400 "" "" "" "" 912 0)
```

B.3 Generated Tables

B.3.1 Data Declaration Table

```
TABLE(DATA_DECLARATION,11)
ID:INTEGER,NAME:STRING,TYPE:STRING,PARENT:INTEGER,POSITION:INTEGER,
   LENGTH:INTEGER,FORMAT:STRING,ALIAS:STRING,VALUE:STRING,SCOPE:STRING,
   LINE_NO:INTEGER
```

```
1,"RESERVATIONS-FILE","FILE",0,1,0,,,,"EXTERNAL",20,
2,"RESERVATIONS-RECORD","RECORD",1,1,0,"0","0",,"EXTERNAL",23,
3,"RR-REC","RECORD",2,1,0,"0","0",,"EXTERNAL",24,
4,"RR-FLIGHT-NUMBER","INTEGER",3,1,6,"9(6)","0",,"EXTERNAL",25,
5,"RR-PASSENGER-NAME1","STRING",3,2,30,"X(30)","0",,"EXTERNAL",26,
6,"RR-PASSENGER-NAME2","STRING",3,3,30,"X(30)","0",,"EXTERNAL",27,
7,"PRINT-FILE","FILE",0,2,0,,,,"EXTERNAL",28,
8,"PRINT-RECORD","RECORD",7,1,0,"0","0",,"EXTERNAL",31,
9,"PRT-MESSAGE","STRING",8,1,30,"X(30)","0",,"EXTERNAL",32,
10,"PRT-REC","STRING",8,2,68,"X(68)","0",,"EXTERNAL",33,
11,"FILLER","STRING",8,3,34,"X(34)","0",,"EXTERNAL",34,
12,"TRANSACTION-FILE","FILE",0,3,0,,,,"EXTERNAL",35,
13,"TRANSACTION-RECORD","RECORD",12,1,0,"0","0",,"EXTERNAL",38,
14,"TR-REC","RECORD",13,1,0,"0","0",,"EXTERNAL",39,
15,"TR-FLIGHT-NUMBER","INTEGER",14,1,6,"9(6)","0",,"EXTERNAL",40,
16,"TR-PASSENGER-NAME1","STRING",14,2,30,"X(30)","0",,"EXTERNAL",41,
17,"TR-PASSENGER-NAME2","STRING",14,3,30,"X(30)","0",,"EXTERNAL",42,
18,"TR-CREATION-CODE","STRING",13,2,1,"X","0",,"EXTERNAL",43,
19,"TR-TRANSACTION-CODE","STRING",13,3,1,"X","0",,"EXTERNAL",44,
20,"FILLER","STRING",13,4,12,"X(12)","0",,"EXTERNAL",45,
21,"NEW-RESERVATIONS-FILE","FILE",0,4,0,,,,"EXTERNAL",46,
22,"NEW-RESERVATIONS-RECORD","RECORD",21,1,0,"0","0",,"EXTERNAL",49,
23,"NRR-REC","STRING",22,1,66,"X(66)","0",,"EXTERNAL",50,
24,"RESERVATIONS-PROGRAM-WORK","RECORD",0,1,0,"0","0",,"INTERNAL",52,
25,"RP-END-OF-TRANS","STRING",24,1,1,"X","0","0","INTERNAL",53,
26,"RP-END-OF-RESERV","STRING",24,2,1,"X","0","0","INTERNAL",54,
27,"RP-INVALID-RECORD","STRING",24,3,1,"X","0","0","INTERNAL",55,
28,"EDIT-WORK","RECORD",0,2,0,"0","0",,"INTERNAL",56,
29,"EW-THIS-FLIGHT-NUM","RECORD",28,1,0,"0","0",,"INTERNAL",57,
30,"EW-THIS-CREATION","STRING",29,1,1,"X","0",,"INTERNAL",58,
31,"EW-THIS-FLIGHT","INTEGER",29,2,6,"9(6)","0",,"INTERNAL",59,
32,"EW-THIS-FN","RECORD",28,2,0,"0","EW-THIS-FLIGHT-NUM",,"INTERNAL",60,
33,"EW-THIS-FLIGHT-NUMBER","INTEGER",32,1,7,"9(7)","0",,"INTERNAL",61,
34,"EW-FLIGHT-NUMBER","RECORD",28,3,0,"0","0","0","INTERNAL",62,
35,"EW-CREATION","STRING",34,1,1,"X","0",,"INTERNAL",63,
36,"EW-FLIGHT","INTEGER",34,2,6,"9(6)","0",,"INTERNAL",64,
37,"EW-FN","RECORD",28,4,0,"0","EW-FLIGHT-NUMBER",,"INTERNAL",65,
38,"EW-LAST-FLIGHT-NUMBER","INTEGER",37,1,7,"9(7)","0",,"INTERNAL",66,
```

B.3.2 Data Reference Table

```
TABLE(DATA_REFERENCE,3)
```

```
LINE_NO:INTEGER,DATA_NAME:STRING,NODE_ID:INTEGER
87,"RP-END-OF-TRANS",303,
87,"'1'",303,
87,"TR-CREATION-CODE",303,
87,"'1'",303,
92,"RP-END-OF-TRANS",324,
92,"'1'",324,
92,"RP-END-OF-RESERV",324,
92,"'1'",324,
98," ",343,
99,"'UPDATED RESERVATION'",348,
100,"RP-END-OF-RESERV",353,
100,"'1'",353,
116,"RP-END-OF-TRANS",387,
116,"RP-INVALID-RECORD",387,
117,"TR-CREATION-CODE",387,
117,"'1'",387,
118,"RESERVATIONS-RECORD",383,
118,"TR-REC",383,
135,"TRANSACTION-FILE",403,
135,"'1'",397,
136,"READ-TRANSACTION-EXIT",397,
139,"RP-INVALID-RECORD",436,
139,"'1'",436,
140," ",419,
141,"'INLAID TRANSACTION'",424,
142,"TRANSACTION-RECORD",429,
143,"PRINT-RECORD",432,
156,"TR-FLIGHT-NUMBER",479,
157,"'1'",449,
159,"TR-CREATION-CODE",454,
160,"TR-FLIGHT-NUMBER",459,
161,"EW-THIS-FLIGHT-NUMBER",476,
161,"EW-LAST-FLIGHT-NUMBER",476,
162,"'1'",468,
164,"EW-THIS-FLIGHT-NUMBER",473,
178,"RESERVATIONS-FILE",492,
178,"'1'",489,
193,"RP-END-OF-TRANS",544,
193,"'1'",544,
193,"RP-END-OF-RESERV",544,
194,"RP-END-OF-RESERV",508,
194,"'1'",508,
195,"RP-END-OF-TRANS",541,
196,"RP-END-OF-RESERV",541,
196,"'1'",541,
197,"RP-END-OF-TRANS",522,
197,"'1'",522,
198,"RP-END-OF-TRANS",538,
199,"RP-END-OF-RESERV",538,
232,"RR-REC",563,
233,"NEW-RESERVATIONS-RECORD",566,
248,"RP-INVALID-RECORD",605,
249,"TR-TRANSACTION-CODE",601,
249,"'1'",601,
251," ",584,
252,"'RESERVATION ADDED'",589,
253,"TRANSACTION-RECORD",594,
254,"PRINT-RECORD",597,
273,"TR-REC",617,
274,"NEW-RESERVATIONS-RECORD",620,
291,"RP-INVALID-RECORD",691,
291,"'1'",691,
293,"RR-FLIGHT-NUMBER",688,
293,"TR-FLIGHT-NUMBER",688,
296,"RR-FLIGHT-NUMBER",685,
296,"TR-FLIGHT-NUMBER",685,
297," ",647,
298,"'RESERVATION RECORD ADDED'",652,
299,"TRANSACTION-RECORD",657,
300,"PRINT-RECORD",660,
303,"RR-FLIGHT-NUMBER",682,
303,"TR-FLIGHT-NUMBER",682,
321,"TR-TRANSACTION-CODE",784,
323,"RR-PASSENGER-NAME1",732,
323," ",732,
324,"RR-PASSENGER-NAME2",732,
324," ",732,
325," ",713,
326,"'RESERVATION RECORD DELETED'",718,
327,"RESERVATIONS-RECORD",723,
328,"PRINT-RECORD",726,
331,"RR-PASSENGER-NAME1",781,
331," ",781,
332,"RR-PASSENGER-NAME2",781,
332," ",781,
334," ",747,
335,"'RESERVATION ADDED'",752,
336,"TRANSACTION-RECORD",757,
337,"PRINT-RECORD",760,
339," ",765,
340,"'NO ROOM ON FLIGHT'",770,
341,"TRANSACTION-RECORD",775,
342,"PRINT-RECORD",778,
356,"TR-TRANSACTION-CODE",834,
357,"TR-PASSENGER-NAME1",830,
357,"RR-PASSENGER-NAME1",830,
358,"TR-PASSENGER-NAME2",830,
358,"RR-PASSENGER-NAME1",830,
359," ",803,
360,"MOD-IFY-EXIT",803,
361,"TR-PASSENGER-NAME1",827,
361,"RR-PASSENGER-NAME2",827,
362,"TR-PASSENGER-NAME2",827,
362,"RR-PASSENGER-NAME2",827,
363," ",818,
364,"MOD-IFY-EXIT",818,
366,"MOD-IFY-EXIT",818,
369,"RR-PASSENGER-NAME1",887,
369," ",887,
370,"TR-PASSENGER-NAME1",842,
372,"TR-PASSENGER-NAME1",847,
373,"TR-PASSENGER-NAME2",884,
373," ",884,
374,"RR-PASSENGER-NAME2",880,
374," ",880,
375,"TR-PASSENGER-NAME2",859,
377," ",864,
378,"'NO ROOM FOR 2ND PASSENGER'",869,
379,"TRANSACTION-RECORD",874,
380,"PRINT-RECORD",877,
394,"NEW-RESERVATIONS-FILE",903,
395,"'1'",897,
396,"RE-PORT-EXIT",897,
397,"NEW-RESERVATIONS-RECORD",908,
398,"PRINT-RECORD",911,
```

B.3.3 Data Definition Table

```
TABLE(DATA_DEFINITION,5)
LINE_NO:INTEGER,DEST_NAME:STRING,NODE_ID:INTEGER,SRC_TYPE:STRING,SRC_NAME:STRING
```

```
97,"RP-END-OF-RESERV",338,"CONSTANT","0",
98,"PRINT-RECORD",343,"CONSTANT"," ",
99,"PRT-MESSAGE",348,"CONSTANT","'UPDATED RESERVATION'",
118,"RESERVATIONS-RECORD",383,"VARIABLE","TR-REC",
118,"RESERVATIONS-FILE",383,"VARIABLE","RESERVATIONS-RECORD",
135,"TRANSACTION-RECORD",403,"FILE","TRANSACTION-FILE",
135,"RP-END-OF-TRANS",397,"CONSTANT","'1'",
137,"RP-INVALID-RECORD",408,"CONSTANT","0",
140,"PRINT-RECORD",419,"CONSTANT"," ",
141,"PRT-MESSAGE",424,"CONSTANT","'INLAID TRANSACTION'",
142,"PRT-REC",429,"VARIABLE","TRANSACTION-RECORD",
143,"PRINT-FILE",432,"VARIABLE","PRINT-RECORD",
157,"RP-INVALID-RECORD",449,"CONSTANT","'1'",
159,"EW-THIS-CREATION",454,"VARIABLE","TR-CREATION-CODE",
160,"EW-THIS-FLIGHT",459,"VARIABLE","TR-FLIGHT-NUMBER",
162,"RP-INVALID-RECORD",468,"CONSTANT","'1'",
164,"EW-LAST-FLIGHT-NUMBER",473,"VARIABLE","EW-THIS-FLIGHT-NUMBER",
178,"RESERVATIONS-RECORD",492,"FILE","RESERVATIONS-FILE",
178,"RP-END-OF-RESERV",489,"CONSTANT","'1'",
232,"NRR-REC",563,"VARIABLE","RR-REC",
233,"NEW-RESERVATIONS-FILE",566,"VARIABLE","NEW-RESERVATIONS-RECORD",
251,"PRINT-RECORD",584,"CONSTANT"," ",
252,"PRT-MESSAGE",589,"CONSTANT","'RESERVATION ADDED'",
253,"PRT-REC",594,"VARIABLE","TRANSACTION-RECORD",
254,"PRINT-FILE",597,"VARIABLE","PRINT-RECORD",
273,"NRR-REC",617,"VARIABLE","TR-REC",
274,"NEW-RESERVATIONS-FILE",620,"VARIABLE","NEW-RESERVATIONS-RECORD",
297,"PRINT-RECORD",647,"CONSTANT"," ",
298,"PRT-MESSAGE",652,"CONSTANT","'RESERVATION RECORD ADDED'",
299,"PRT-REC",657,"VARIABLE","TRANSACTION-RECORD",
300,"PRINT-FILE",660,"VARIABLE","PRINT-RECORD",
325,"PRINT-RECORD",713,"CONSTANT"," ",
326,"PRT-MESSAGE",718,"CONSTANT","'RESERVATION RECORD DELETED'",
327,"PRT-REC",723,"VARIABLE","RESERVATIONS-RECORD",
328,"PRINT-FILE",726,"VARIABLE","PRINT-RECORD",
334,"PRINT-RECORD",747,"CONSTANT"," ",
335,"PRT-MESSAGE",752,"CONSTANT","'RESERVATION ADDED'",
336,"PRT-REC",757,"VARIABLE","TRANSACTION-RECORD",
337,"PRINT-FILE",760,"VARIABLE","PRINT-RECORD",
339,"PRINT-RECORD",765,"CONSTANT"," ",
340,"PRT-MESSAGE",770,"CONSTANT","'NO ROOM ON FLIGHT'",
341,"PRT-REC",775,"VARIABLE","TRANSACTION-RECORD",
342,"PRINT-FILE",778,"VARIABLE","PRINT-RECORD",
359,"RR-PASSENGER-NAME1",803,"CONSTANT"," ",
363,"RR-PASSENGER-NAME2",818,"CONSTANT"," ",
370,"RR-PASSENGER-NAME1",842,"VARIABLE","TR-PASSENGER-NAME1",
372,"RR-PASSENGER-NAME2",847,"VARIABLE","TR-PASSENGER-NAME1",
375,"RR-PASSENGER-NAME2",859,"VARIABLE","TR-PASSENGER-NAME2",
377,"PRINT-RECORD",864,"CONSTANT"," ",
378,"PRT-MESSAGE",869,"CONSTANT","'NO ROOM FOR 2ND PASSENGER'",
379,"PRT-REC",874,"VARIABLE","TRANSACTION-RECORD",
380,"PRINT-FILE",877,"VARIABLE","PRINT-RECORD",
394,"NEW-RESERVATIONS-RECORD",903,"FILE","NEW-RESERVATIONS-FILE",
395,"RP-END-OF-RESERV",897,"CONSTANT","'1'",
397,"PRT-REC",908,"VARIABLE","NEW-RESERVATIONS-RECORD",
398,"PRINT-FILE",911,"VARIABLE","PRINT-RECORD",
```

B.3.4 Program Structure Table

```
TABLE(PROGRAM_STRUCTURE,6)
ID:INTEGER,LINE_NO:INTEGER,NAME:STRING,POSITION:INTEGER,
  PARENT:INTEGER,NODE_ID:INTEGER
1,2,"RESVPG",1,0,914,
2,83,"RESERVATIONS-PROGRAM",1,1,287,
3,84,"RESERVATIONS-PROGRAM-ENTRY",1,2,288,
4,103,"RESERVATIONS-PROGRAM-EXIT",2,2,360,
5,113,"CRE-ATE",2,1,362,
6,114,"CRE-ATE-ENTRY",1,5,363,
7,121,"CRE-ATE-EXIT",2,5,388,
8,133,"READ-TRANSACTION",3,1,390,
9,134,"READ-TRANSACTION-ENTRY",1,8,391,
10,146,"READ-TRANSACTION-EXIT",2,8,437,
11,154,"EDIT",4,1,439,
12,155,"EDIT-ENTRY",1,11,440,
13,165,"EDIT-EXIT",2,11,480,
14,176,"READ-RESERVATION",5,1,482,
15,177,"READ-RESERVATION-ENTRY",1,14,483,
16,179,"READ-RESERVATION-EXIT",2,14,493,
17,191,"UP-DATE",6,1,495,
18,192,"UP-DATE-ENTRY",1,17,496,
19,203,"UP-DATE-EXIT",2,17,545,
20,215,"FINISH-RESERVATION",7,1,547,
21,216,"FINISH-RESERVATION-ENTRY",1,20,548,
22,219,"FINISH-RESERVATION-EXIT",2,20,555,
23,230,"WRITE-NEW-RESER",8,1,557,
24,231,"WRITE-NEW-RESER-ENTRY",1,23,558,
25,234,"WRITE-NEW-RESER-EXIT",2,23,567,
26,246,"FINISH-TRANSACTION",9,1,569,
27,247,"FINISH-TRANSACTION-ENTRY",1,26,570,
28,260,"FINISH-TRANSACTION-EXIT",2,26,609,
29,271,"WRITE-TRANS-TO-RESER",10,1,611,
30,272,"WRITE-TRANS-TO-RESER-ENTRY",1,29,612,
31,275,"WRITE-TRANS-TO-RESER-EXIT",2,29,621,
32,289,"MATCH-FLIGHT",11,1,623,
33,290,"MATCH-FLIGHT-ENTRY",1,32,624,
34,309,"MATCH-FLIGHT-EXIT",2,32,692,
35,319,"UPDATE-RESERVATION",12,1,694,
36,320,"UPDATE-RESERVATION-ENTRY",1,35,695,
37,343,"UPDATE-RESERVATION-EXIT",2,35,785,
38,354,"MOD-IFY",13,1,787,
39,355,"MOD-IFY-ENTRY",1,38,788,
40,383,"MOD-IFY-EXIT",2,38,888,
41,392,"RE-PORT",14,1,890,
42,393,"RE-PORT-ENTRY",1,41,891,
43,399,"RE-PORT-EXIT",2,41,912,
```

B.3.5 Calling Structure Table

```
TABLE(CALLING_STRUCTURE,4)
LINE_NO:INTEGER,CALLING_BLOCK:STRING,CALLED_BLOCK:STRING,LAST_BLOCK:STRING
87,"RESERVATIONS-PROGRAM-ENTRY","CRE-ATE",,
91,"RESERVATIONS-PROGRAM-ENTRY","READ-RESERVATION",,
92,"RESERVATIONS-PROGRAM-ENTRY","UP-DATE",,
100,"RESERVATIONS-PROGRAM-ENTRY","RE-PORT",,
115,"CRE-ATE-ENTRY","READ-TRANSACTION",,
138,"READ-TRANSACTION-ENTRY","EDIT",,
194,"UP-DATE-ENTRY","FINISH-RESERVATION",,
197,"UP-DATE-ENTRY","FINISH-TRANSACTION",,
200,"UP-DATE-ENTRY","MATCH-FLIGHT",,
217,"FINISH-RESERVATION-ENTRY","WRITE-NEW-RESER",,
218,"FINISH-RESERVATION-ENTRY","READ-RESERVATION",,
250,"FINISH-TRANSACTION-ENTRY","WRITE-TRANS-TO-RESER",,
259,"FINISH-TRANSACTION-ENTRY","READ-TRANSACTION",,
292,"MATCH-FLIGHT-ENTRY","READ-TRANSACTION",,
294,"MATCH-FLIGHT-ENTRY","WRITE-NEW-RESER",,
295,"MATCH-FLIGHT-ENTRY","READ-RESERVATION",,
301,"MATCH-FLIGHT-ENTRY","WRITE-TRANS-TO-RESER",,
302,"MATCH-FLIGHT-ENTRY","READ-TRANSACTION",,
304,"MATCH-FLIGHT-ENTRY","UPDATE-RESERVATION",,
305,"MATCH-FLIGHT-ENTRY","READ-RESERVATION",,
306,"MATCH-FLIGHT-ENTRY","READ-TRANSACTION",,
322,"UPDATE-RESERVATION-ENTRY","MOD-IFY",,
330,"UPDATE-RESERVATION-ENTRY","WRITE-NEW-RESER",,
333,"UPDATE-RESERVATION-ENTRY","MOD-IFY",,
```

B.4 CPM Graphs

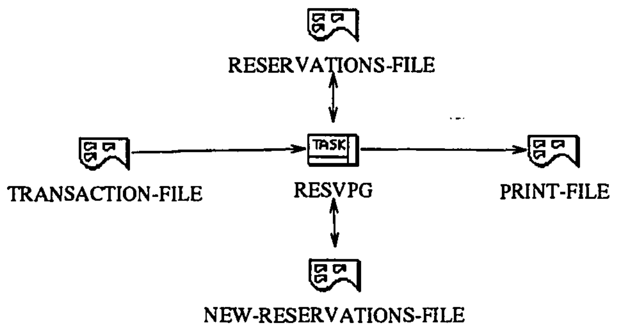


Figure B.1: CPM - OVERVIEW

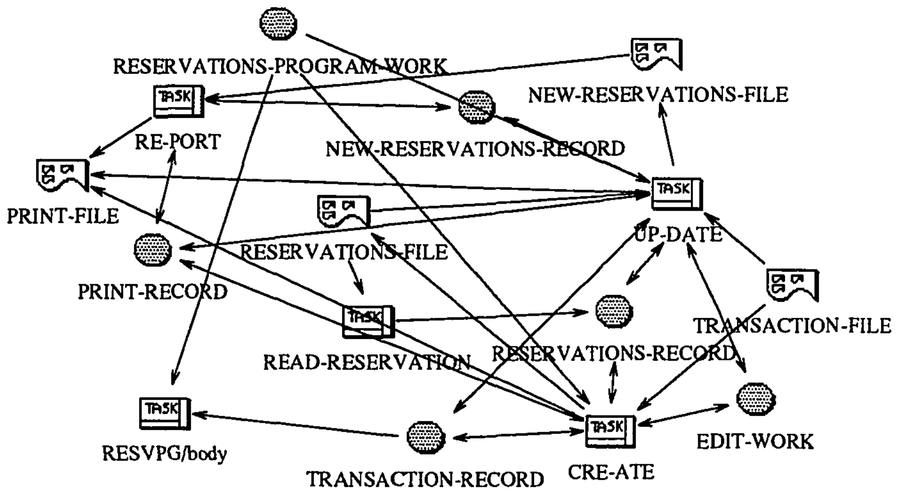


Figure B.2: CPM - RESVPG

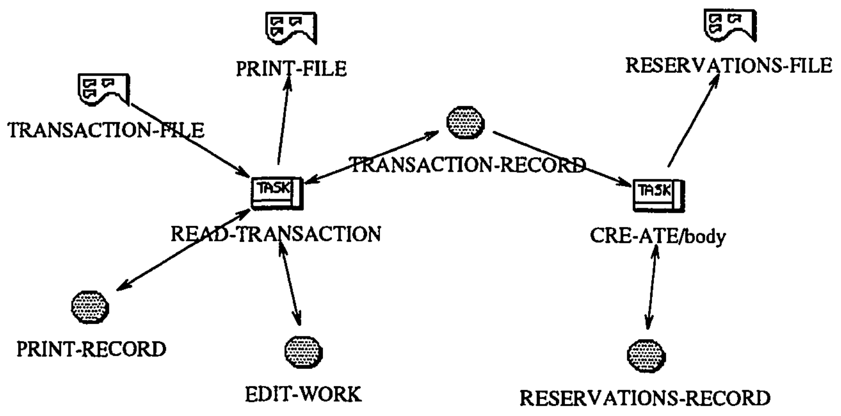


Figure B.3: CPM - CRE-ATE

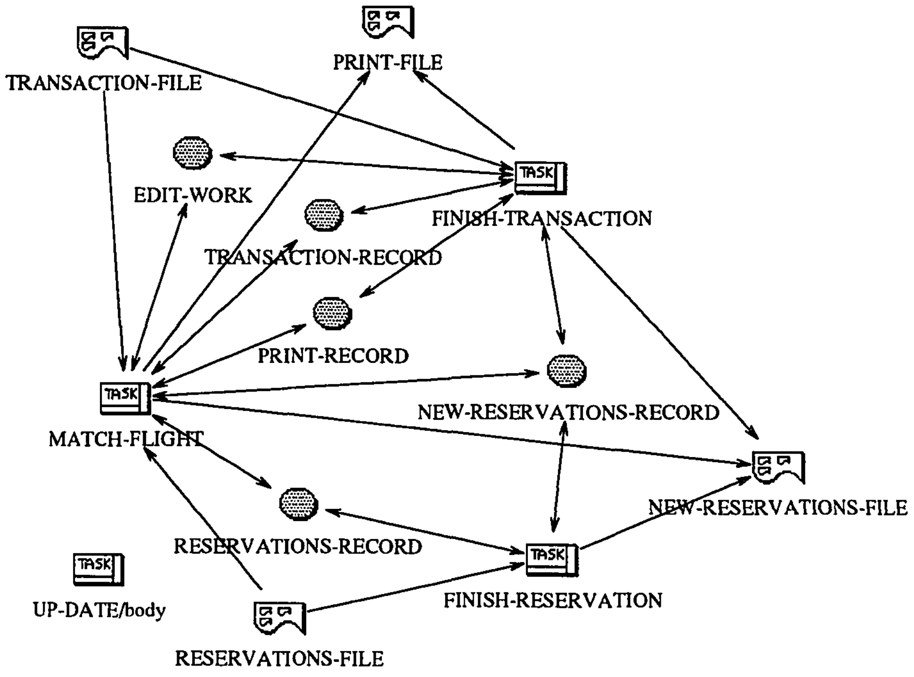


Figure B.4: CPM - UP-DATE

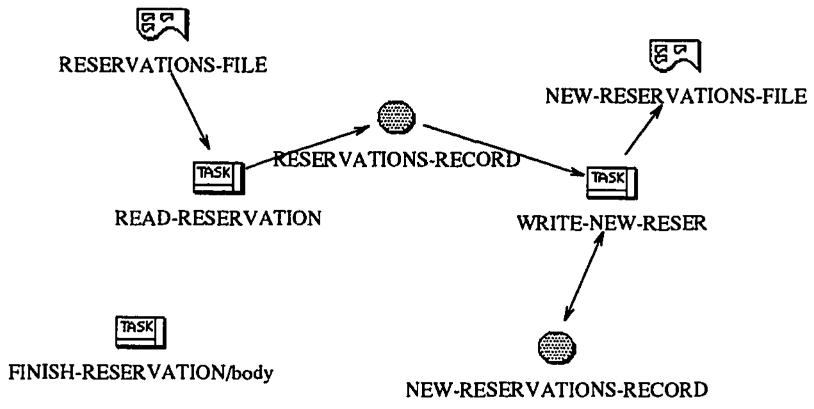


Figure B.5: CPM – FINISH-RESERVATION

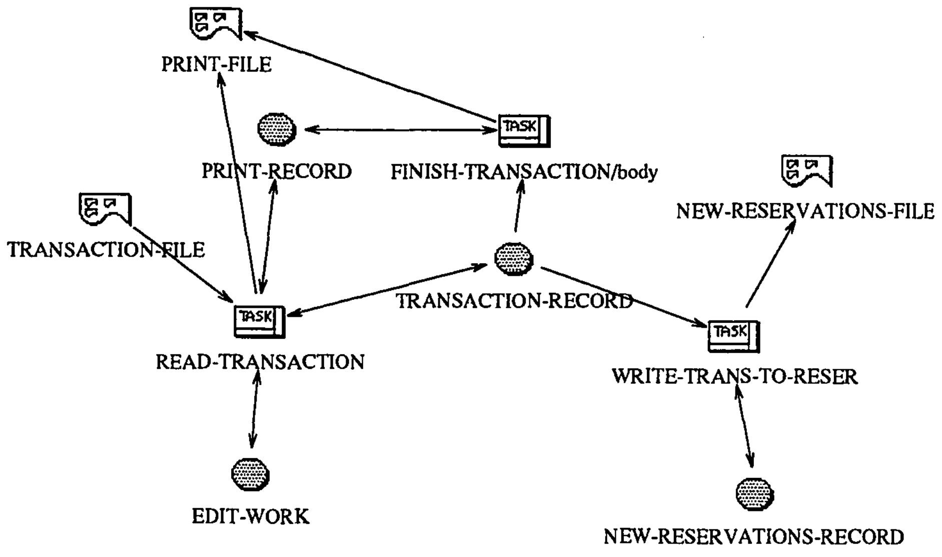


Figure B.6: CPM – FINISH-TRANSACTION

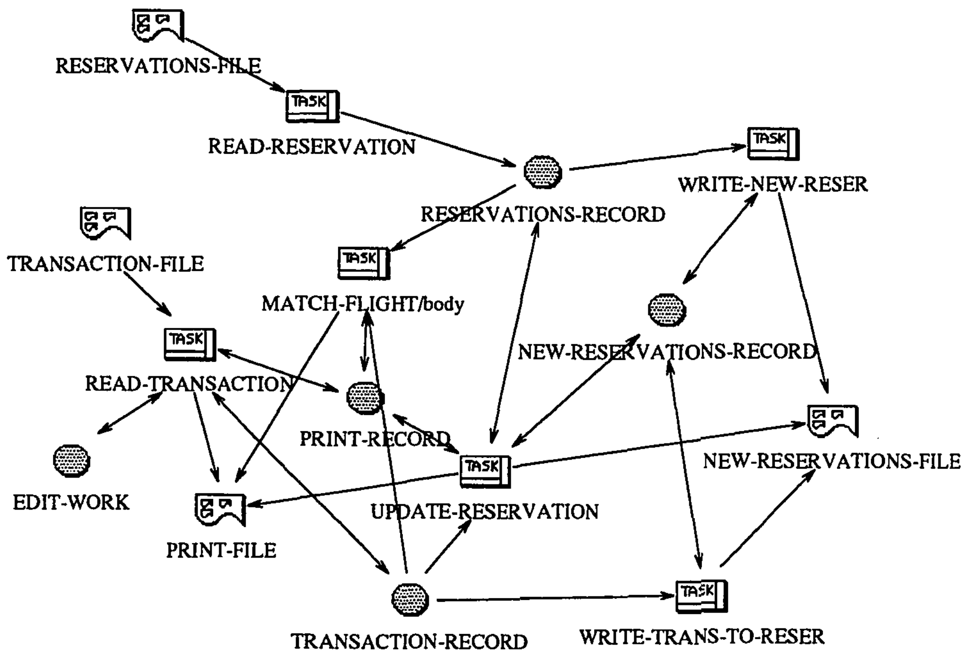


Figure B.7: CPM – MATCH-FLIGHT

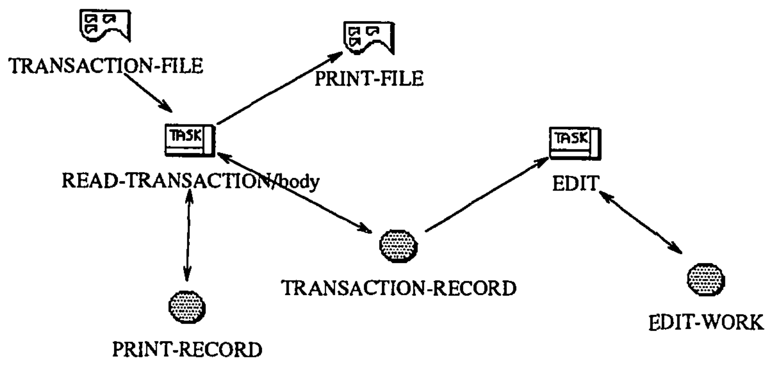


Figure B.8: CPM – READ-TRANSACTION

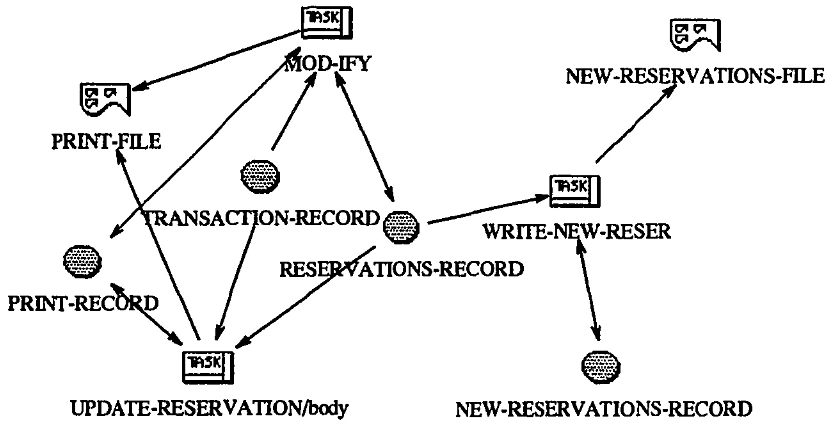


Figure B.9: CPM – UPDATE-RESERVATION

B.5 IOPM2 Graphs

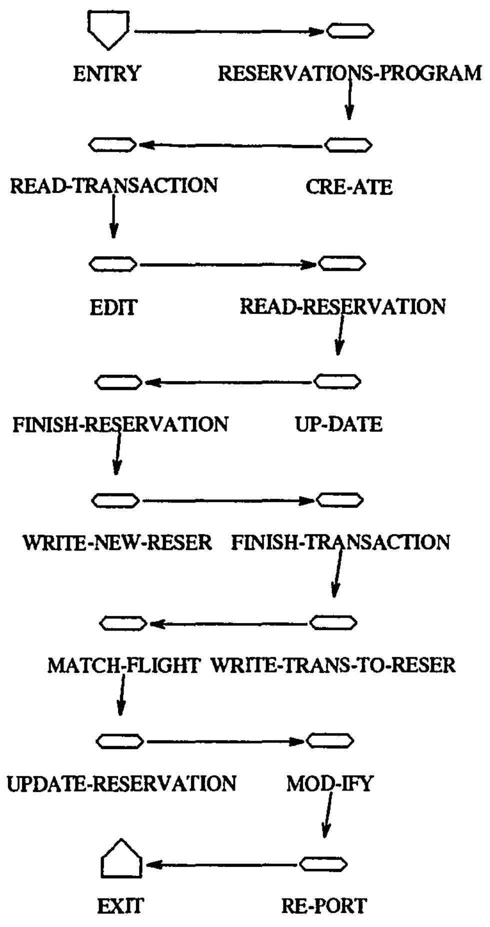
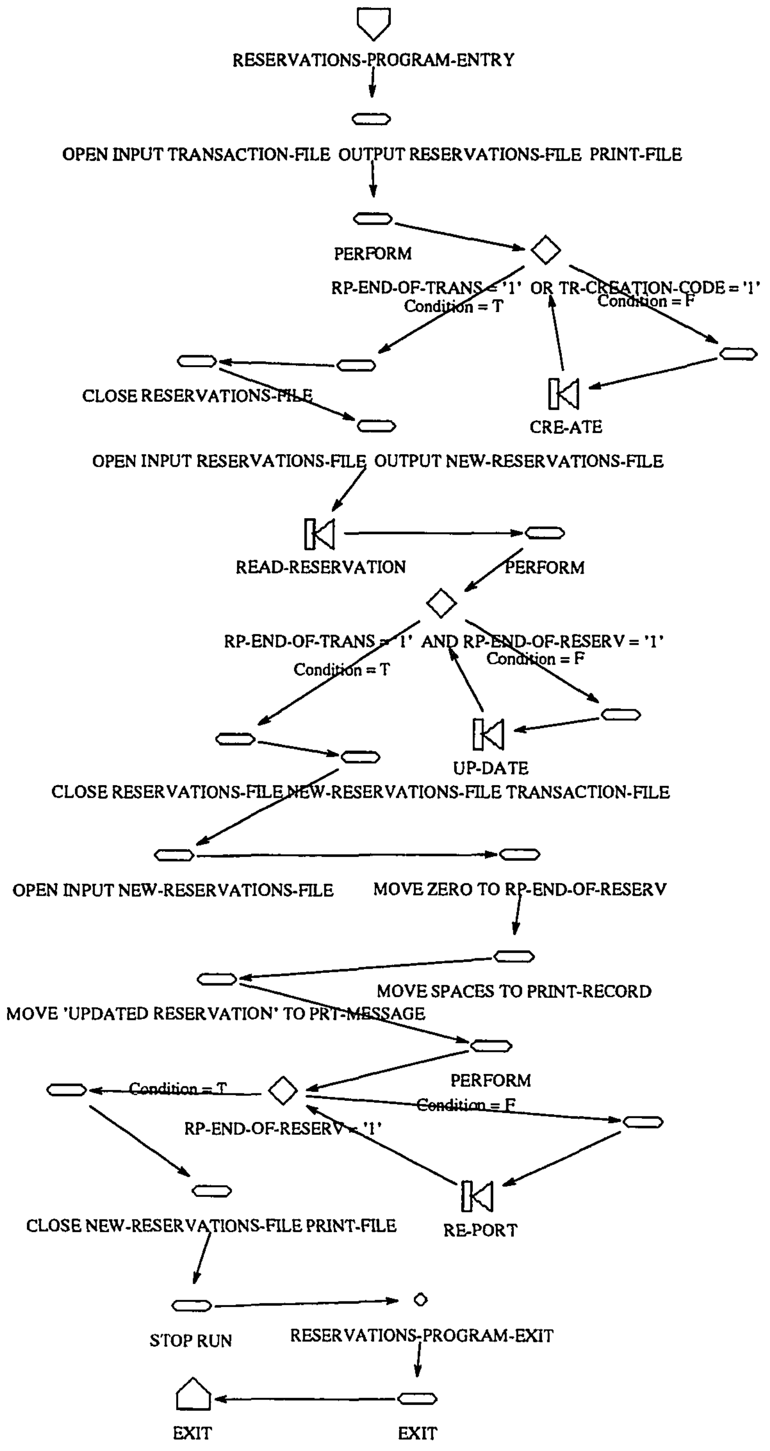


Figure B.10: IOPM2 – RESVPG



RESERVATIONS-PROGRAM-ENTRY
OPEN INPUT TRANSACTION-FILE OUTPUT RESERVATIONS-FILE PRINT-FILE
PERFORM
RP-END-OF-TRANS = '1' OR TR-CREATION-CODE = '1'
Condition = T
Condition = F
CLOSE RESERVATIONS-FILE
CRE-ATE
OPEN INPUT RESERVATIONS-FILE OUTPUT NEW-RESERVATIONS-FILE
READ-RESERVATION
PERFORM
RP-END-OF-TRANS = '1' AND RP-END-OF-RESERV = '1'
Condition = T
Condition = F
UP-DATE
CLOSE RESERVATIONS-FILE NEW-RESERVATIONS-FILE TRANSACTION-FILE
OPEN INPUT NEW-RESERVATIONS-FILE
MOVE ZERO TO RP-END-OF-RESERV
MOVE SPACES TO PRINT-RECORD
MOVE 'UPDATED RESERVATION' TO PRT-MESSAGE
PERFORM
Condition = T
Condition = F
RP-END-OF-RESERV = '1'
CLOSE NEW-RESERVATIONS-FILE PRINT-FILE
RE-PORT
STOP RUN
RESERVATIONS-PROGRAM-EXIT
EXIT
EXIT

Figure B.11: IOPM2 - RESERVATION-PROGRAM

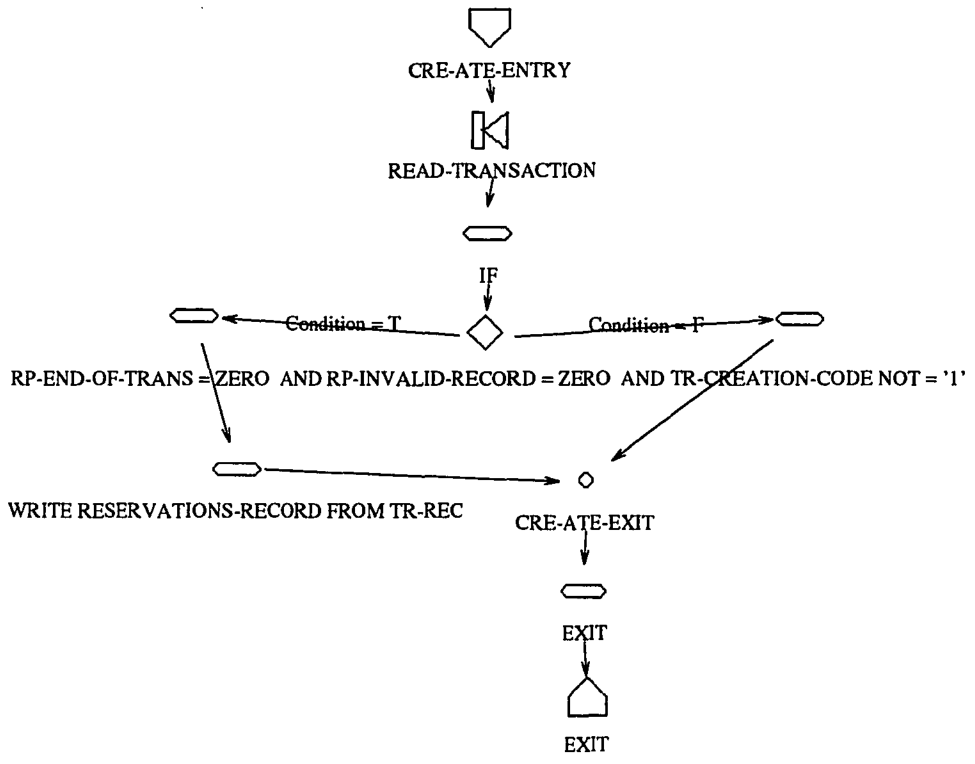

CRE-ATE-ENTRY
READ-TRANSACTION
IF
Condition = T
Condition = F
RP-END-OF-TRANS = ZERO AND RP-INVALID-RECORD = ZERO AND TR-CREATION-CODE NOT = '1'
WRITE RESERVATIONS-RECORD FROM TR-REC
CRE-ATE-EXIT
EXIT
EXIT

Figure B.12: IOPM2 - CRE-ATE

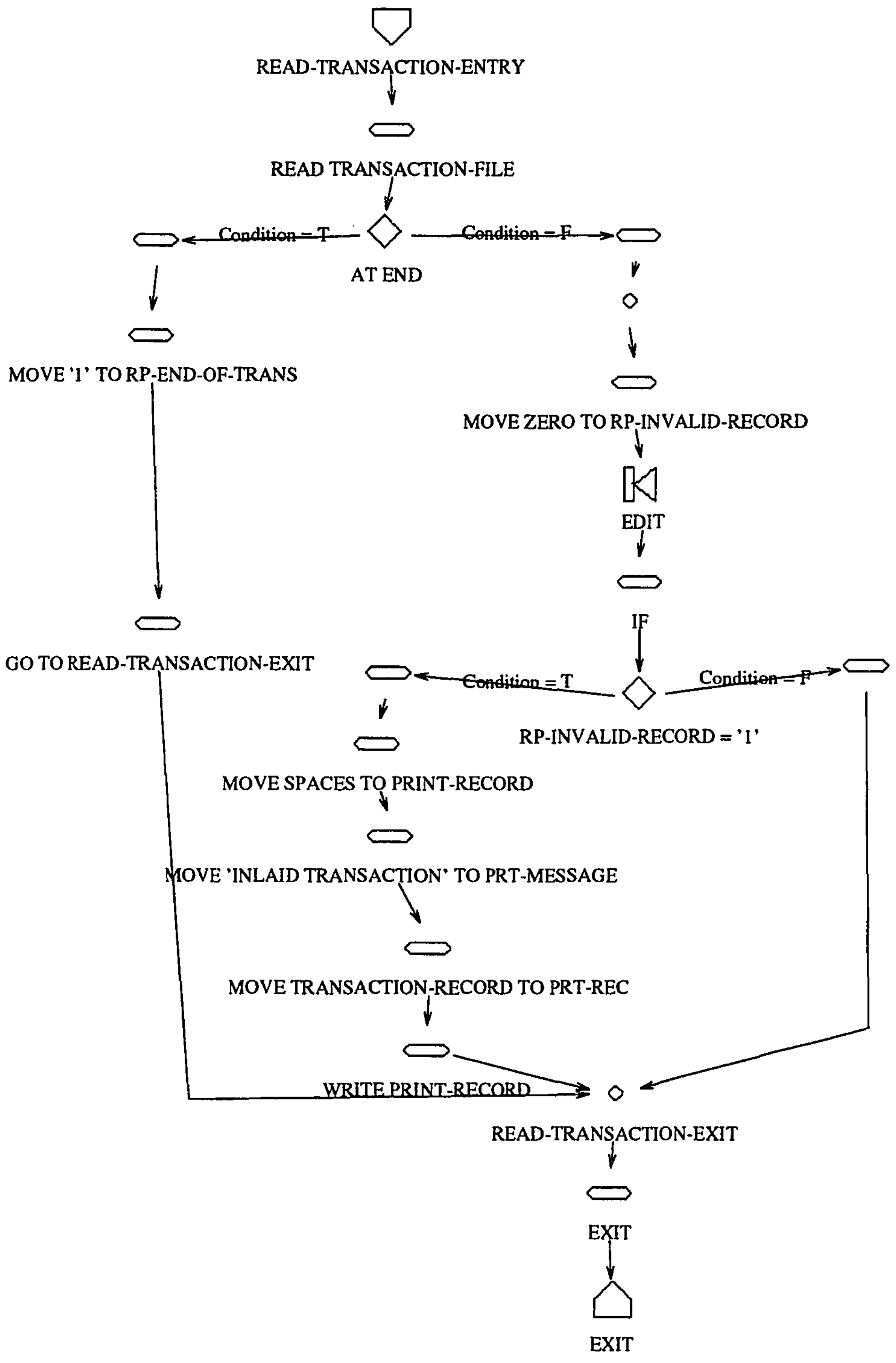

READ-TRANSACTION-ENTRY
READ TRANSACTION-FILE
Condition = T
AT END
Condition = F
MOVE '1' TO RP-END-OF-TRANS
MOVE ZERO TO RP-INVALID-RECORD
EDIT
IF
GO TO READ-TRANSACTION-EXIT
Condition = T
Condition = F
RP-INVALID-RECORD = '1'
MOVE SPACES TO PRINT-RECORD
MOVE 'INLAID TRANSACTION' TO PRT-MESSAGE
MOVE TRANSACTION-RECORD TO PRT-REC
WRITE PRINT-RECORD
READ-TRANSACTION-EXIT
EXIT
EXIT

Figure B.13: IOPM2 – READ-TRANSACTION

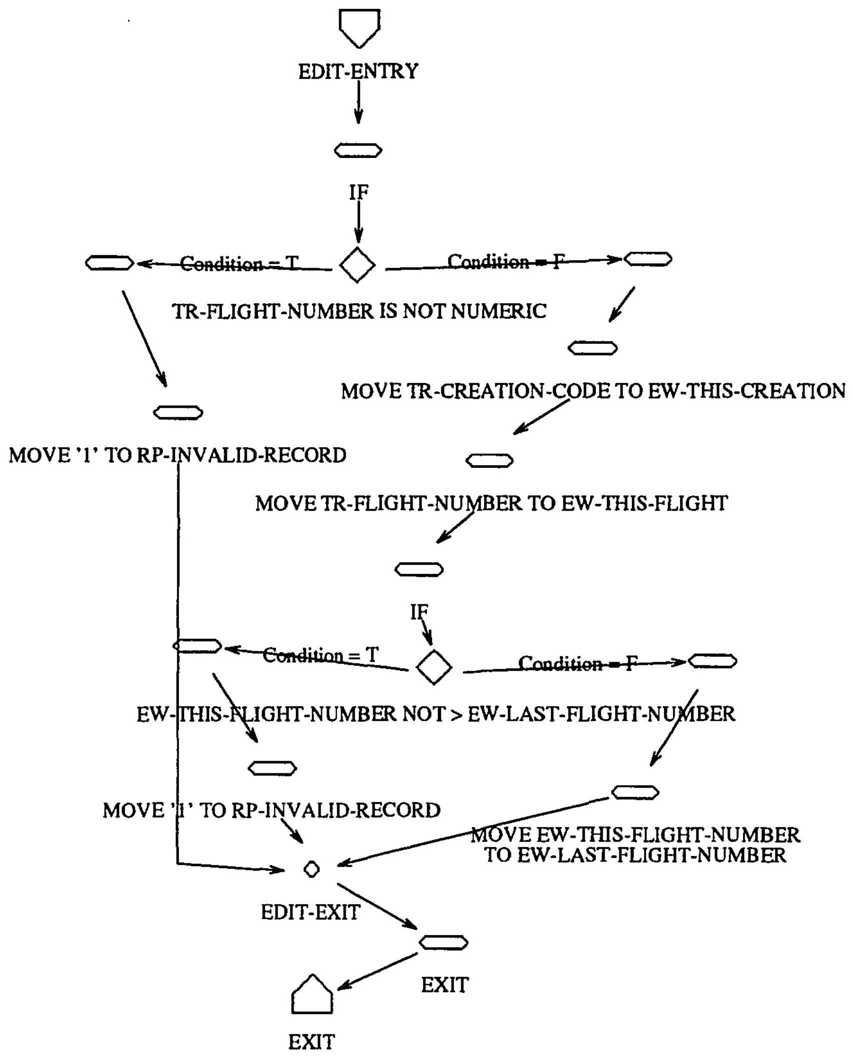

EDIT-ENTRY
IF
Condition = T
Condition = F
TR-FLIGHT-NUMBER IS NOT NUMERIC
MOVE TR-CREATION-CODE TO EW-THIS-CREATION
MOVE '1' TO RP-INVALID-RECORD
MOVE TR-FLIGHT-NUMBER TO EW-THIS-FLIGHT
IF
Condition = T
Condition = F
EW-THIS-FLIGHT-NUMBER NOT > EW-LAST-FLIGHT-NUMBER
MOVE '1' TO RP-INVALID-RECORD
MOVE EW-THIS-FLIGHT-NUMBER TO EW-LAST-FLIGHT-NUMBER
EDIT-EXIT
EXIT
EXIT

Figure B.14: IOPM2 - EDIT

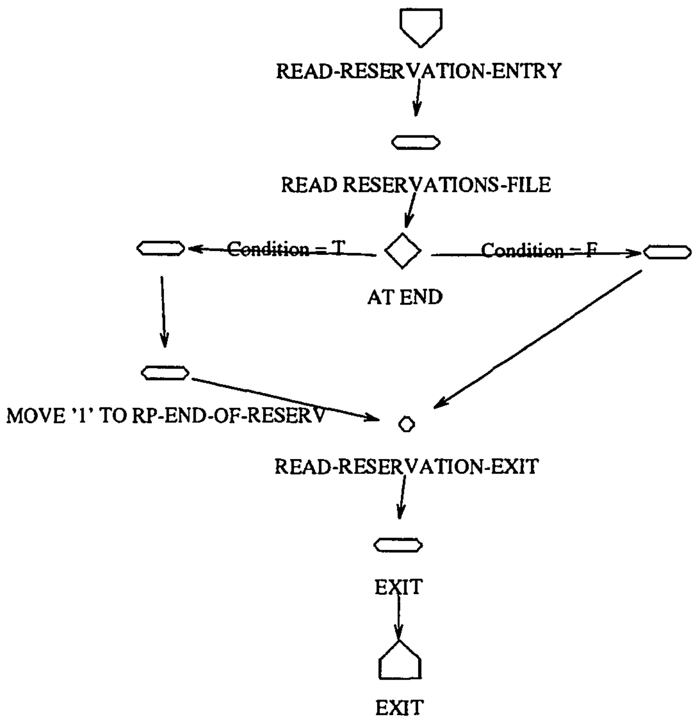


Figure B.15: IOPM2 – READ-RESERVATION

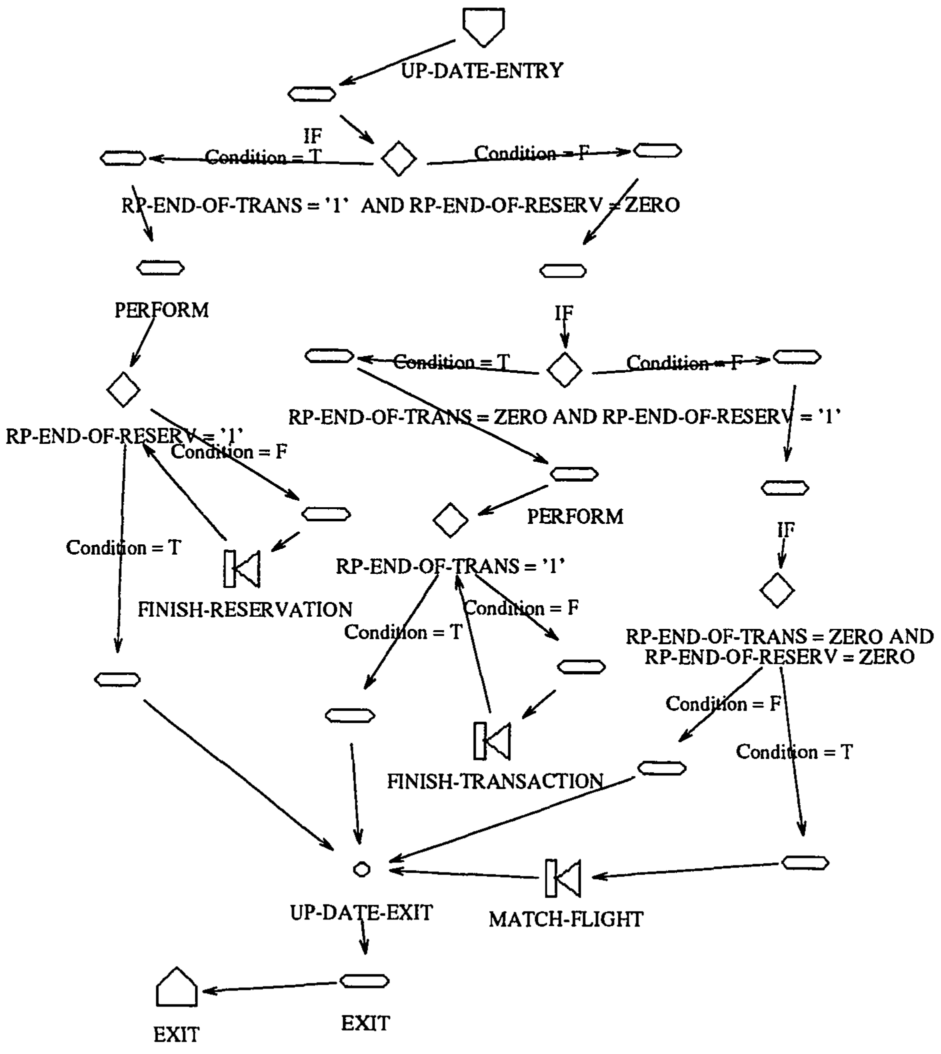


Figure B.16: IOPM2 - UP-DATE

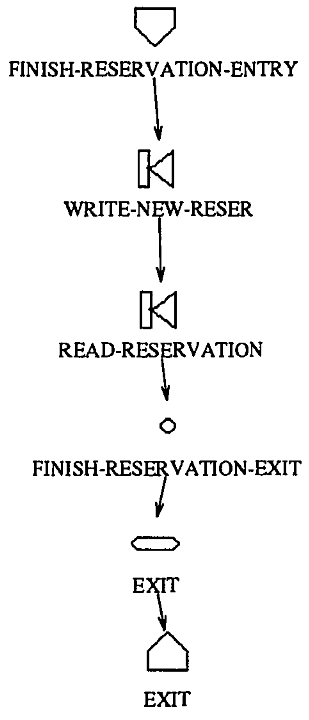


Figure B.17: IOPM2 – FINISH-RESERVATION

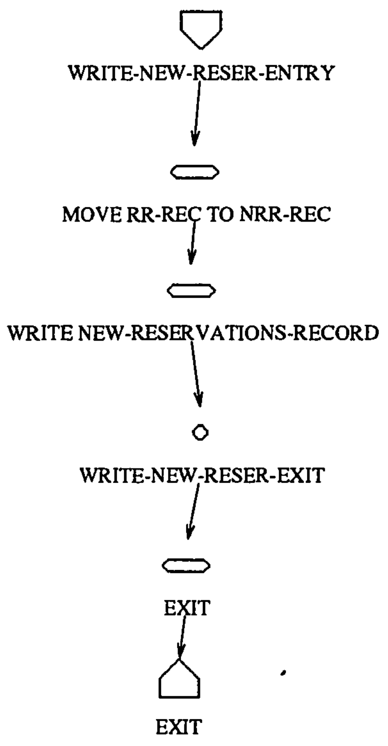


Figure B.18: IOPM2 – WRITE-NEW-RESER

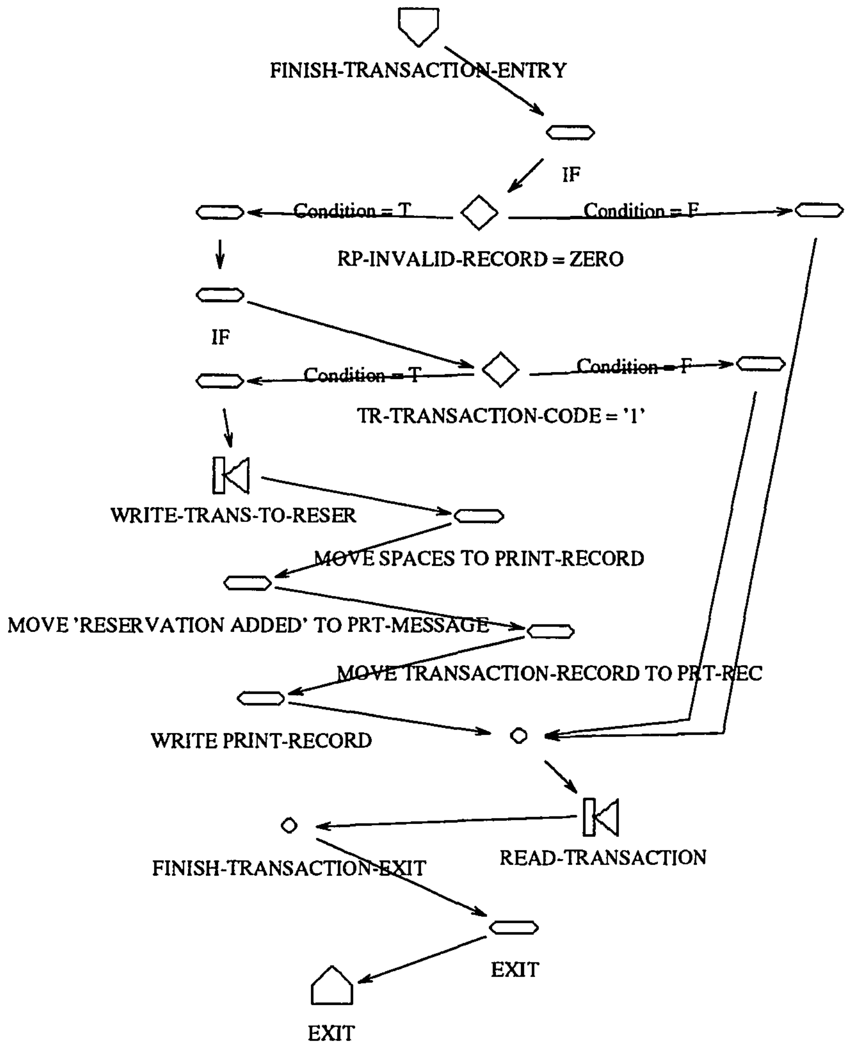


Figure B.19: IOPM2 - FINISH-TRANSACTION

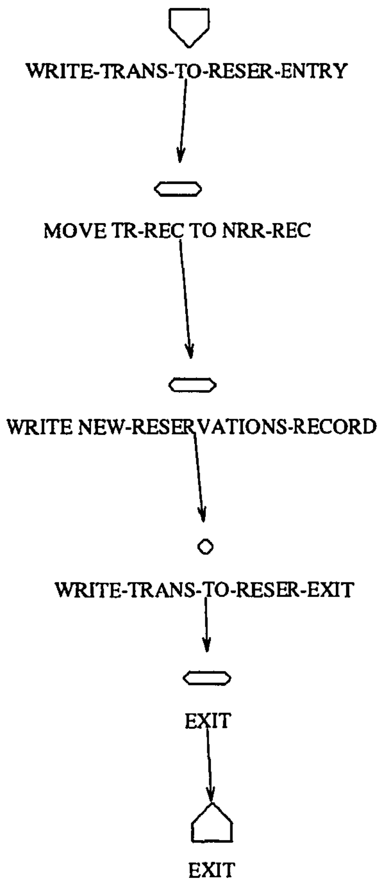

WRITE-TRANS-TO-RESER-ENTRY
MOVE TR-REC TO NRR-REC
WRITE NEW-RESERVATIONS-RECORD
WRITE-TRANS-TO-RESER-EXIT
EXIT
EXIT

Figure B.20: IOPM2 – WRITE-TRANS-TO-RESER

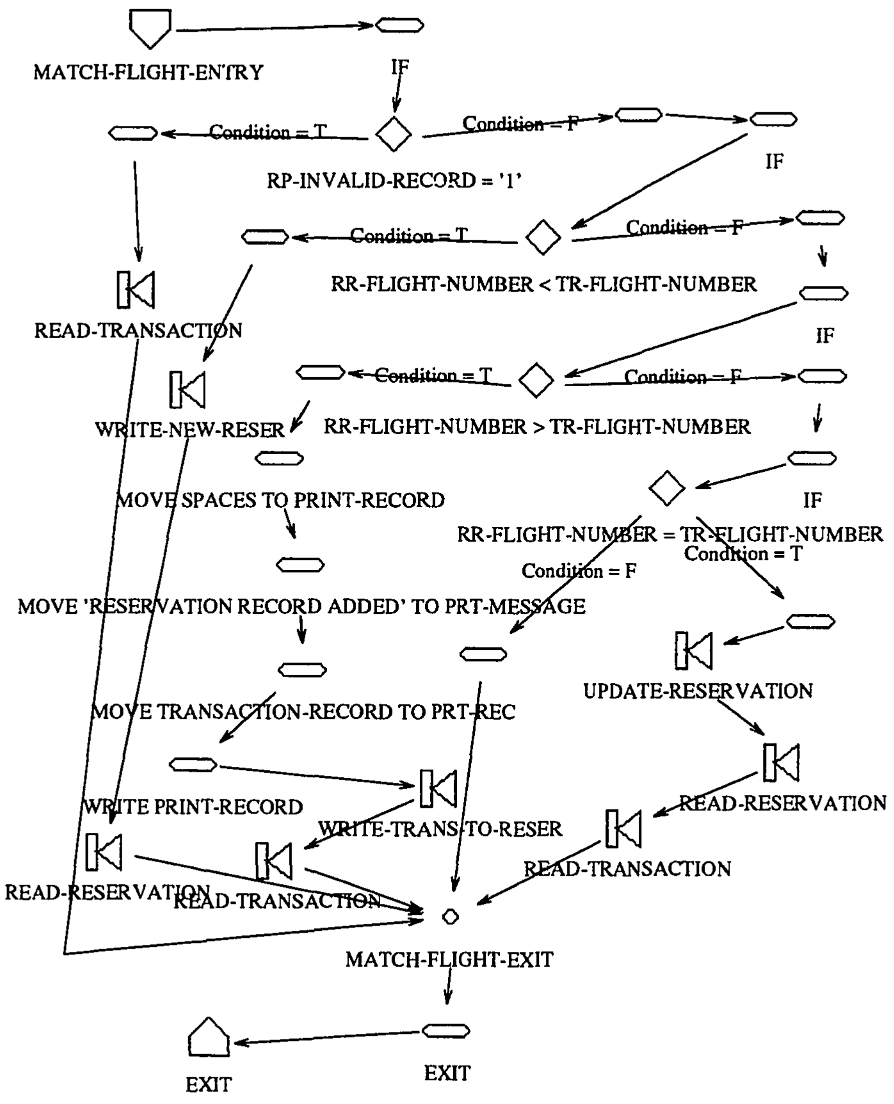


Figure B.21: IOPM2 – MATCH-FLIGHT

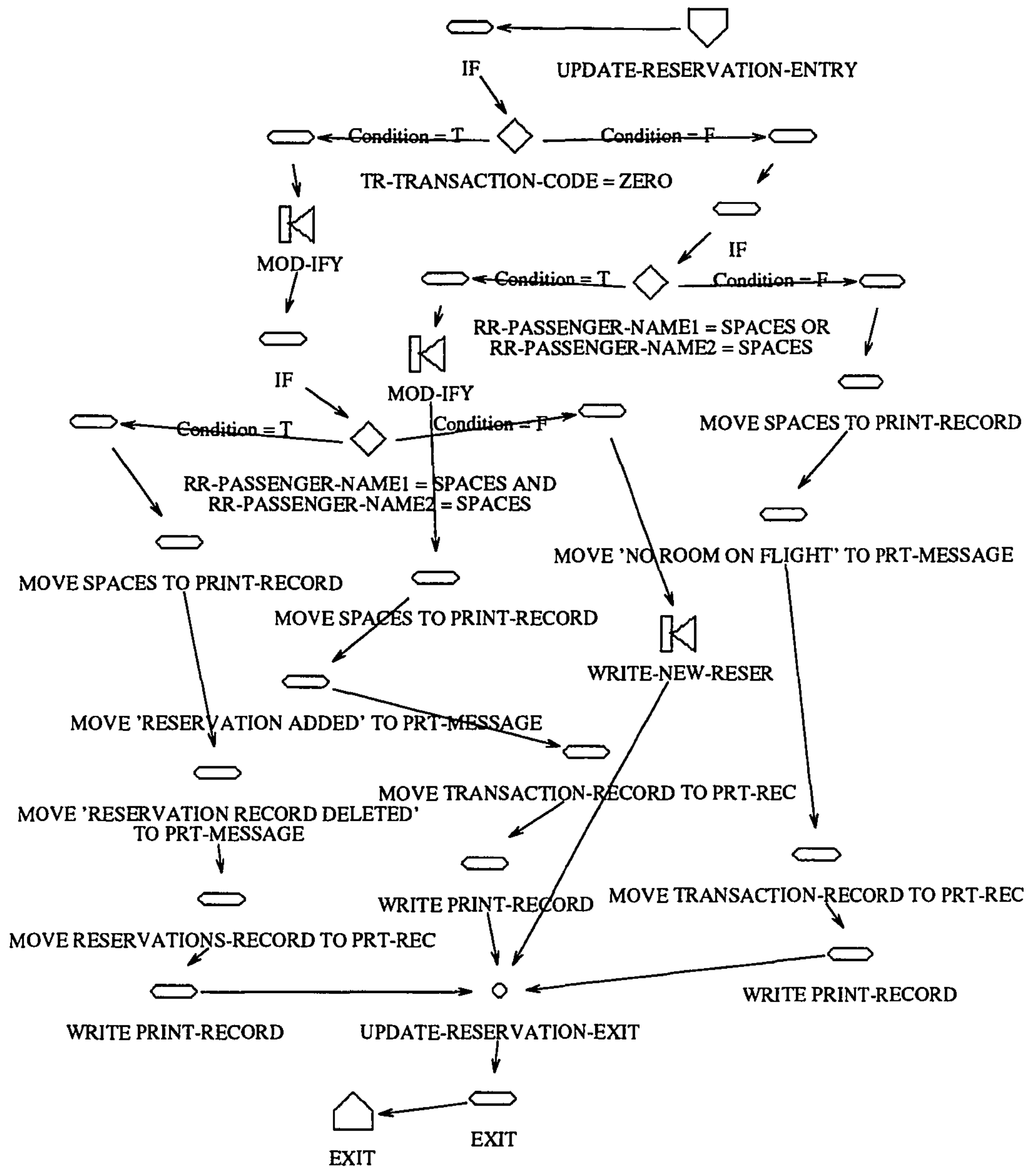

UPDATE-RESERVATION-ENTRY
IF
Condition = T
Condition = F
TR-TRANSACTION-CODE = ZERO
MOD-IFY
IF
RR-PASSENGER-NAME1 = SPACES OR RR-PASSENGER-NAME2 = SPACES
MOD-IFY
IF
RR-PASSENGER-NAME1 = SPACES AND RR-PASSENGER-NAME2 = SPACES
MOVE SPACES TO PRINT-RECORD
MOVE 'NO ROOM ON FLIGHT' TO PRT-MESSAGE
WRITE-NEW-RESER
MOVE 'RESERVATION ADDED' TO PRT-MESSAGE
MOVE TRANSACTION-RECORD TO PRT-REC
MOVE 'RESERVATION RECORD DELETED' TO PRT-MESSAGE
WRITE PRINT-RECORD
MOVE RESERVATIONS-RECORD TO PRT-REC
UPDATE-RESERVATION-EXIT
EXIT

Figure B.22: IOPM2 – UPDATE-RESERVATION

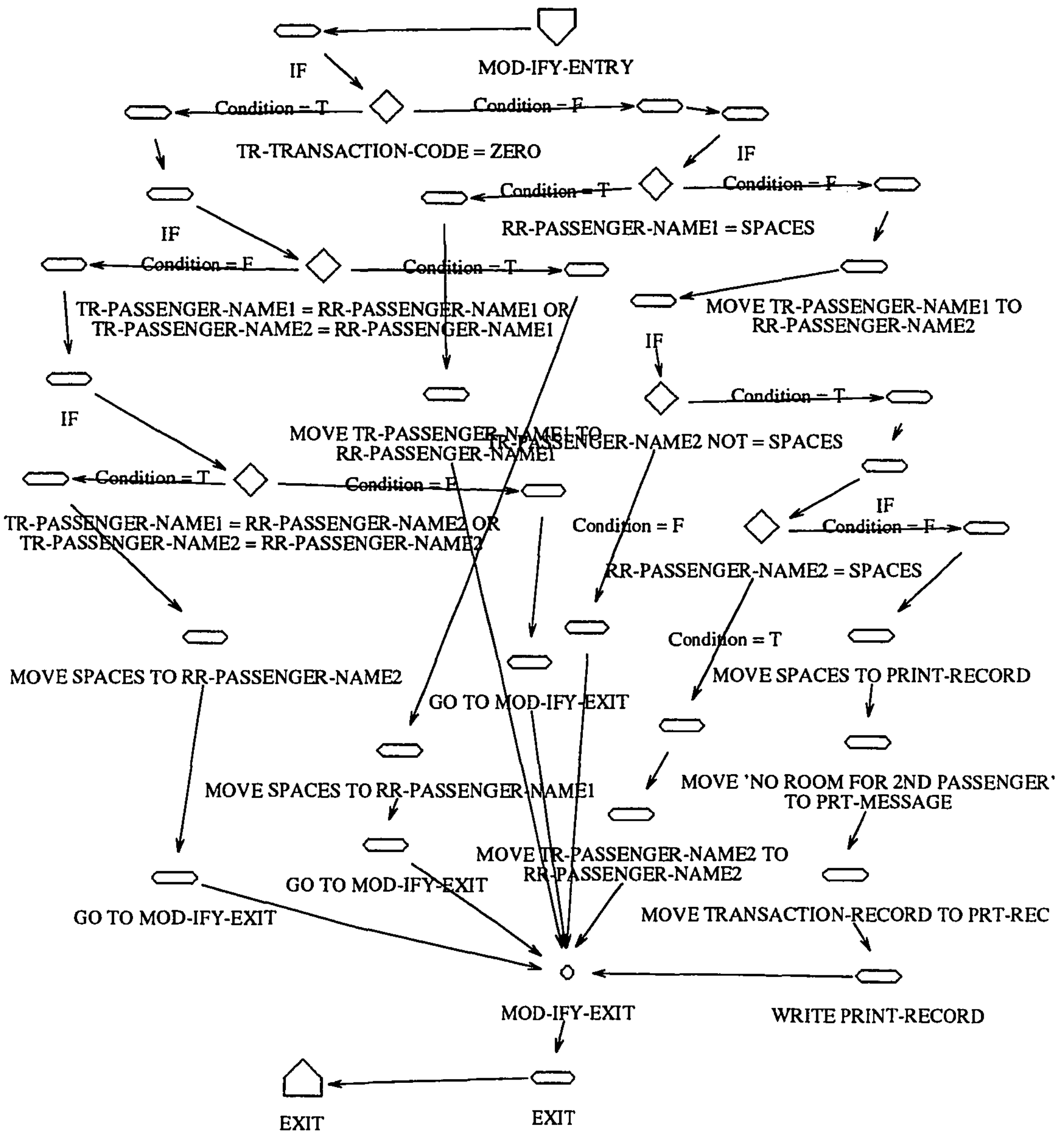

MOD-IFY-ENTRY
IF
Condition = T
Condition = F
TR-TRANSACTION-CODE = ZERO
IF
RR-PASSENGER-NAME1 = SPACES
IF
TR-PASSENGER-NAME1 = RR-PASSENGER-NAME1 OR TR-PASSENGER-NAME2 = RR-PASSENGER-NAME1
MOVE TR-PASSENGER-NAME1 TO RR-PASSENGER-NAME2
IF
MOVE TR-PASSENGER-NAME1 TO RR-PASSENGER-NAME1
TR-PASSENGER-NAME2 NOT = SPACES
IF
TR-PASSENGER-NAME1 = RR-PASSENGER-NAME2 OR TR-PASSENGER-NAME2 = RR-PASSENGER-NAME2
IF
RR-PASSENGER-NAME2 = SPACES
MOVE SPACES TO RR-PASSENGER-NAME2
GO TO MOD-IFY-EXIT
MOVE SPACES TO PRINT-RECORD
MOVE SPACES TO RR-PASSENGER-NAME1
MOVE 'NO ROOM FOR 2ND PASSENGER' TO PRT-MESSAGE
MOVE TR-PASSENGER-NAME2 TO RR-PASSENGER-NAME2
GO TO MOD-IFY-EXIT
GO TO MOD-IFY-EXIT
MOVE TRANSACTION-RECORD TO PRT-REC
MOD-IFY-EXIT
WRITE PRINT-RECORD
EXIT
EXIT

Figure B.23: IOPM2 – MOD-IFY

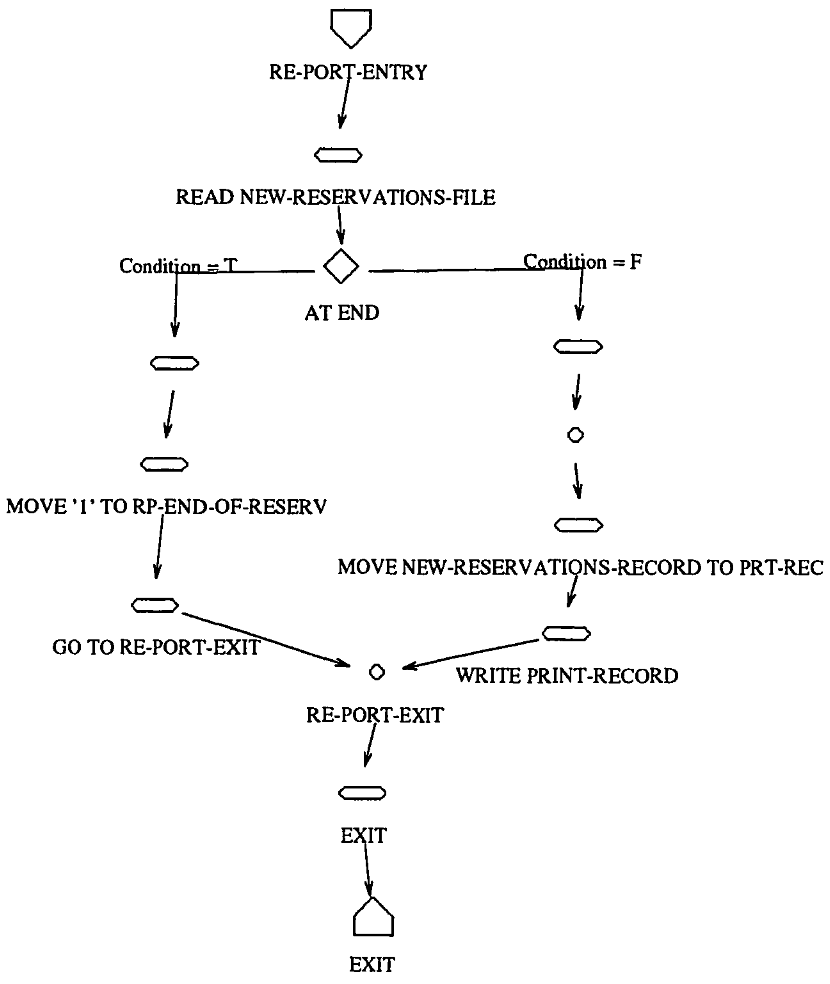


Figure B.24: IOPM2 – RE-PORT

Appendix C

Grammars of Generation Rules

C.1 Common Definitions used in Generation Rules

Tokens

There are four classes of tokens: keywords, constants, operators, and other separators. Blank, tabs, newlines, commas, and comments are ignored except when they serve as separator tokens. No other words are allowed except when they are surrounded by double quotes, which are considered string constant.

Comments

The characters /* introduce a comment, which terminates with the characters */. Comments do not nest.

Constants There are three kinds of constants, as listed below.

- An integer constant consists of a sequence of digits and may start with a sign. The range of integers allowed is between 65535 and -65535.
- A real constant consists of an integer part, a decimal point, a fraction part, an e or E, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing.
- A string is a sequence of characters. If the string contains white space (blank or newline) or the string is the same as one of the key words, it must be surrounded by double quotes. The length of a string should not exceed 256 characters.

Operators There are two kinds of operators, as listed below:

Comparison Operators =, <>, >=, <=, >, and <.

Arithmetic Operators +, −, ∗, and /.

C.2 Table Generation Rules

Lexicon of Table Generation Rules

Keywords The following are the keywords used in *table generation rules.*

ANCESTOR	AND	ANY	CONCAT
CONDITION	ELDER	ENTRY	FIELD
FLAG	GLOBAL-ID	GROUP	GROUP-ID
INTEGER-TYPE	NODE-ELEMENT	NODE-ID	NODE-KEY-1
NODE-KEY-2	NODE-KEY-3	NODE-KEY-4	NODE-LINE-NO
NODE-NAME	NODE-TYPE	NODE-VALUE	OFFSPRING
OR	PATH	PROCESS	PUT-FIELD
PUT-VARIABLE	RULE	STRING-TYPE	TABLE
TABLE-CONTENT	TABLE-ID	VARIABLE	YOUNGER

Constants Integer and string constants are used in *table generation rules.*

Operators Both comparison and arithmetic operators are used in *table generation rules.*

Grammar of Table Generation Rules

```
table-generation-rule    : table-definition table-rule*
table-definition         : TABLE '(' table-name number-of-field
                           field-definition+ ')'
table-name               : STRING
number-of-field          : INTEGER
field-definition         : FIELD '(' field-name field-type ')'
field-name               : STRING
field-type               : INTEGER-TYPE
                         | STRING-TYPE
```

```
table-rule            : RULE '(' trigger+ table-action* ')'
trigger               : CONDITION '(' node-type ( AND condition)* ')'
node-type             : STRING
condition             : '(' expression comparison-operator expression ')'
                      | node
                      | flag
node                  : path
flag                  : FLAG '(' STRING ')'
table-action          : entry-action
                      | put-field-action
                      | process-action
                      | group-action
                      | put-variable-action
entry-action          : ENTRY '(' entry-id ')'
entry-id              : INTEGER
put-field-action      : PUT-FIELD '(' [ entry-id ] content-list ')'
content-list          : '(' field-name content [ concat-option ] ')'
content               : GLOBAL-ID
                      | GROUP-ID
                      | TABLE-ID '(' check-field check-content ')'
                      | TABLE-CONTENT '(' check-field check-content
                        value-field ')'
                      | node-content
                      | STRING
                      | INTEGER
                      | VARIABLE '(' variable-id ')'
concat-option         : CONCAT '(' STRING ')'
check-field           : field-name
check-content         : expression
value-field           : field-name
variable-id           : INTEGER
process-action        :  PROCESS '(' node [ flag ] ')'
group-action          : GROUP '(' start-node [ stop-node ] sub-action* ')'
start-node            : path
stop-node             : path
sub-action            : entry-action
                      | put-field-action
                      | process-action
                      | put-variable-action
put-variable-action   : PUT-VARIABLE '(' variable-id content ')'
path                  : PATH '(' [ '/' ] path-element ( '/' path-element )+ ')'
```

```
path-element        : path-descriptor [ path-condition ]
path-descriptor     : INTEGER
                    | node-types
                    | ..
                    | ELDER [ '(' node-types ')' ]
                    | YOUNGER [ '(' node-types ')' ]
                    | ANCESTOR '(' node-types ')'
                    | OFFSPRING '(' node-types ')'
node-types          : node-type ( OR node-type )*
                    | ANY
path-condition      : '[' node-content '=' expression ']'
node-content        : [ path ':' ] node-element
node-element        : NODE-ID
                    | NODE-NAME
                    | NODE-TYPE
                    | NODE-VALUE
                    | NODE-ELEMENT
                    | NODE-KEY-1
                    | NODE-KEY-2
                    | NODE-KEY-3
                    | NODE-KEY-4
                    | NODE-LINE-NO
expression          : STRING
                    | term
                    | '+' term
                    | '−' term
                    | expression '+' term
                    | expression '−' term
term                : factor
                    | term '*' factor
                    | term '/' factor
factor              : node-content
                    | INTEGER
                    | REAL
                    | VARIABLE '(' variable-id ')'
                    | '(' expression ')'
```

C.3 Model Generating Rules

Lexicon of Generating Rules

Keywords The following are the keywords used in *model generation rules.*

ANCESTOR	AND	ANY	ARGUMENT
ATTRIBUTE	BREAK	CONDITION	CONNECT
ELDER	ELSE	ENTITY	ENTRY
EXIT	FLAG	GRAPH	GROUP
IF	ITEM	MACRO	NAME
NODE-ELEMENT	NODE-ID	NODE-KEY-1	NODE-KEY-2
NODE-KEY-3	NODE-KEY-4	NODE-LINE-NO	NODE-NAME
NODE-TYPE	NODE-VALUE	OFFSPRING	OLD-OBJ
OR	PATH	PROCESS	RELATION
RESERVED	RETURN	REVERSE	RULE
SAME	TYPE	UNIQUE	VARIABLE
YOUNGER			

Constants Integer, real, and string constants are used in *model generation rules.*

Operators Both comparison and arithmetic operators are used in *model generation rules.*

Grammar of Model Generation Rules

model-generation-rule	: relation-type-definition* macro-definition* model-rule*
relation-type-definition	: RELATION '(' type source-types destination-types ')'
type	: STRING
source-types	: '(' type-list ')'
destination-type	: '(' type-list ')'
type-list	: type (OR type)*
macro-definition	: MACRO '(' macro-name argument-list sub-action* ')'
macro-name	: STRING
argument-list	: ARGUMENT '(' INTEGER* ')'
model-rule	: RULE '(' model-trigger+ model-action* ')'
trigger	: CONDITION '(' node-type (AND condition) * [return-list] ')'

```
node-type              : STRING
condition              : '(' expression comparison-operator expression ')'
                       | node
                       | flag
node                   : path
flag                   : FLAG '(' STRING ')'
return-list            : RETURN '(' INTEGER* ')'
model-action           : graph-action
                       | entity-action
                       | relation-action
                       | attribute-action
                       | process-action
                       | group-action
                       | if-action
                       | break-action
                       | exit-action
                       | variable-action
                       | macro-action
sub-action             : model-action
                       | item-action
                       | connect-action
graph-action           : GRAPH '(' graph-name graph-formalism
                         entry-object-id exit-object-id ')'
graph-name             : name
graph-formalism        : STRING
entry-object-id        : INTEGER
exit-object-id         : INTEGER
entity-action          : ENTITY '(' object-id entity-type entity-name
                         [ RESERVE ] [ UNIQUE ] attribute* ')'
object-id              : INTEGER
entity-type            : STRING
entity-name            : name
item-action            : ITEM '(' object-id entity-type entity-name
                         [ RESERVED ] [ UNIQUE ] attribute* ')'
relation-action        : RELATION '(' object-id relation-type relation-name
                         source-object destination-object [ RESERVED ]
                         attribute* ')'
relation-type          : STRING
relation-name          : name
source-object          : object-path
destination-object     : object-path
```

```
object-path            : object-id
                       | graph-description ':' object-description
object-id              : INTEGER
graph-description      : name
                       | SAME
                       | ANY
object-description     : [ type-list ':' ] name
                       | ENTRY
                       | EXIT
type-list              : STRING ( OR STRING )*
connect-action         : CONNECT '(' object-id relation-type relation-name
                         [ REVERSE ] connect-source connect-destination
                         [ RESERVED ] attribute* ')'
connect-source         : INTEGER
connect-destination    : INTEGER
attribute-action       : ATTRIBUTE '(' object-id attribute-type
                         attribute-value ')'
attribute-type         : STRING
attribute-value        : name
process-action         : PROCESS '(' node [ return-list ] [ flag ] ')'
node                   : path
flag                   : FLAG '(' STRING ')'
group-action           : GROUP '(' start-node [stop-node] sub-action* ')'
start-node             : path
stop-node              : path
if-action              : IF '(' if-conditions true-actions [ false-actions ] ')'
if-conditions          : if-condition ( OR if-condition )*
if-condition           : condition-element ( AND if-element )*
if-element             : condition
                       | '(' object-element [ OLD-OBJ ] '(' object-id ')'
                         comparison-operator expression ')'
object-element         : TYPE
                       | NAME
                       | STRING
true-actions           : sub-action*
false-actions          : ELSE sub-action *
break-action           : BREAK
exit-action            : EXIT
variable-action        : VARIABLE '(' variable-id variable-value ')'
variable-value         : name
macro-action           : MACRO '(' macro-name argument-list ')'
```

```
attribute           : ATTRIBUTE ’(’ attribute-type attribute-name ’)’
name                : NAME ’(’ name-element ( ’+’ name-element )* ’)’
name-element        : STRING
                    | variable
                    | node-content
                    | loop-count
variable            : VARIABLE ’(’ variable-id ’)’
variable-id         : INTEGER
loop-count          : initial-value ’:’ increment-value
initial-value       : INTEGER
increment-value     : INTEGER
path                : PATH ’(’ [ ’/’ ] path-element ( ’/’ path-element )+ ’)’
path-element        : path-descriptor [ path-condition ]
path-descriptor     : INTEGER
                    | node-types
                    | ..
                    | ELDER [ ’(’ node-types ’)’ ]
                    | YOUNGER [ ’(’ node-types ’)’ ]
                    | ANCESTOR ’(’ node-types ’)’
                    | OFFSPRING ’(’ node-types ’)’
node-types          : type-list
                    | ANY
path-condition      : ’[’ node-content ’=’ expression ’]’
node-content        : [ path ’:’ ] node-element
node-element        : NODE-ID
                    | NODE-NAME
                    | NODE-TYPE
                    | NODE-VALUE
                    | NODE-ELEMENT
                    | NODE-KEY-1
                    | NODE-KEY-2
                    | NODE-KEY-3
                    | NODE-KEY-4
                    | NODE-LINE-NO
expression          : STRING
                    | term
                    | ’+’ term
                    | ’−’ term
                    | expression ’+’ term
                    | expression ’−’ term
term                : factor
```

```
                          | term '*' factor
                          | term '/' factor
factor                    : node-content
                          | INTEGER
                          | REAL
                          | variable
                          | '(' expression ')'
```

C.4 Linkage Generation Rules

Lexicon of Linkage Generation Rules

Keywords The following are keywords used in *linkage generation rules.*

ANY	ATTRIBUTE	DST	ENTITY
GRAPH	LINK	NAME	OR
RELATION	SAME	SRC	TABLE
TYPE			

Constants String constants are used in *linkage generation rules.*

Operators Both comparison and arithmetic operators are used in *linkage generation rules.*

Grammar of Linkage Generation Rules

```
linkage-generation-rule   : linkage-table* linkage-rule*
linkage-table             : TABLE '(' table-type [ table-file ] ')'
table-type                : STRING
table-file                : STRING
linkage-rule              : link '('graph-formalisms linkage-action* ')'
graph-formalisms          : '(' source-graph destination-graph ')'
source-graph              : STRING
destination-graph         : STRING
linkage-action            : '(' link-type source-object destination-object
                            condition attribute* ')'
link-type                 : STRING
```

```
source-object           : object
destination-object      : object
object                  : GRAPH
                        | ANY
                        | ENTITY
                        | RELATION
                        | type-list
type-list               : STRING ( OR STRING )*
condition               : logical-fact
                        | condition '|' logical-fact
logical-fact            : comparison
                        | logical-fact '&' comparison
                        | '(' condition ')'
comparison              : expression comparison-operator expression
expression              : term
                        | '−' term
                        | expression '+' term
                        | expression '−' term
term                    : factor
                        | term '*' factor
                        | term '/' factor
factor                  : object ':' element
                        | table ':' field
                        | STRING
                        | INTEGER
                        | '(' expression ')'
object                  : SRC
                        | DST
element                 : GRAPH
                        | NAME
                        | TYPE
                        | STRING
table                   : STRING
                        | SAME
field                   : STRING
attribute               : ATTRIBUTE '(' attribute-type attribute-value ')'
attribute-type          : STRING
attribute-value         : expression
```

Appendix D

Information Used for COBOL Reverse Engineering

D.1 Node Definition File

This section shows the node definition file for COBOL programming language.

```
/*
 * cobol_node.h
 *
 * node type definition for COBOL
 *
 */

#define PROGRAM_NODE          1000
#define NESTED_PROG_NODE      1001
#define DIVISION_NODE         1002
#define SECTION_NODE          1003
#define PARAGRAPH_NODE        1004

#define ACCEPT_ST             2001
#define ADD_ST                2002
#define ALTER_ST              2003
#define CALL_ST               2004
#define CANCEL_ST             2005
#define CLOSE_ST              2006
#define COMPUTE_ST            2007
#define COPY_ST               2008
#define CONTINUE_ST           2009
#define DELETE_ST             2010
#define DISABLE_ST            2011
#define DISPLAY_ST            2012
#define DIVIDE_ST             2013
#define ENABLE_ST             2014
#define ENTER_ST              2015
```

```
#define EVALUATE_ST            2016
#define EXIT_ST                2017
#define GENERATE_ST            2018
#define GOTO_ST                2019
#define GOTO_DEPEND_ST         2020
#define IF_ST                  2021
#define INITIALIZE_ST          2022
#define INITIATE_ST            2023
#define INSPECT_ST             2024
#define MERGE_ST               2025
#define MOVE_ST                2026
#define MULTIPLY_ST            2027
#define OPEN_ST                2028
#define PERFORM_ST             2029
#define PERFORM_UNTIL_NODE     2030
#define PERFORM_TIMES_NODE     2031
#define PERFORM_BEFORE_NODE    2032
#define PERFORM_AFTER_NODE     2033
#define PURGE_ST               2034
#define READ_ST                2035
#define RECEIVE_ST             2036
#define RETURN_ST              2037
#define RELEASE_ST             2038
#define REPLACE_ST             2039
#define REWRITE_ST             2040
#define SEARCH_ST              2041
#define SEND_ST                2042
#define SET_ST                 2043
#define SORT_ST                2044
#define START_ST               2045
#define STOP_ST                2046
#define STRING_ST              2047
#define SUBTRACT_ST            2048
#define SUPPRESS_ST            2049
#define TERMINATE_ST           2050
#define UNSTRING_ST            2051
#define USE_ST                 2052
#define WRITE_ST               2053
#define GOBACK_ST              2054

#define THEN_NODE              3000
#define ELSE_NODE              3001
#define END_NODE               3002
#define NOT_END_NODE           3003
#define FROM_NODE              3004
```

```
#define TO_NODE                 3005
#define GIVING_NODE             3006
#define INTO_NODE               3007
#define BY_NODE                 3008
#define FOR_NODE                3009
#define ROUNDED_NODE            3010
#define REMAINDER_NODE          3011
#define CORRESPONDING_NODE      3012
#define TIMES_NODE              3013
#define UNTIL_NODE              3014
#define VARYING_NODE            3015
#define AT_END_NODE             3016
#define NOT_AT_END_NODE         3017
#define ON_SIZE_ERROR_NODE      3018
#define NOT_ON_SIZE_ERROR_NODE  3019
#define ON_OVERFLOW_NODE        3020
#define NOT_ON_OVERFLOW_NODE    3021
#define ON_EXCEPTION_NODE       3022
#define NOT_ON_EXCEPTION_NODE   3023
#define INVALID_KEY_NODE        3024
#define NOT_INVALID_KEY_NODE    3025
#define END_OF_PAGE_NODE        3026
#define NOT_END_OF_PAGE_NODE    3027
#define UPON_NODE               3028
#define WHEN_OTHER_NODE         3029
#define WHEN_NODE               3030
#define THROUGH_NODE            3031
#define DEPENDING_ON_NODE       3032
#define REPLACING_NODE          3033
#define TALLYING_NODE           3034
#define CONVERTING_NODE         3035
#define COLLATING_SEQ_NODE      3036
#define OUTPUT_PROC_NODE        3037
#define NO_DATA_NODE            3038
#define DATA_NODE               3039
#define BEFORE_ADVANCING_NODE   3040
#define AFTER_ADVANCING_NODE    3041
#define UP_BY_NODE              3042
#define DOWN_BY_NODE            3043
#define KEY_NODE                3044
#define USING_NODE              3045
#define WHEN_BODY_NODE          3046
#define IN_NODE                 3047
#define OF_NODE                 3048
#define SUBSCRIPT_NODE          3049
```

```
#define DEST_FILE_NODE          3050

#define PERFORM_BODY_NODE       3501
#define VARY_LOOP_NODE          3502
#define DUM_SOURCE_NODE         3503
#define REL_OP_NODE             3504
#define CLASS_TYPE_NODE         3505
#define SIGN_TYPE_NODE          3506
#define LHS_NODE                3507
#define RHS_NODE                3508
#define LANG_NAME_NODE          3509
#define ROUTINE_NAME_NODE       3510
#define COND_NODE               3511
#define CASE_NODE               3512
#define WHEN_COND_NODE          3513
#define CONST_COND_NODE         3514

#define DATE_NODE               4001
#define DAY_NODE                4002
#define DAY_OF_WEEK_NODE        4003
#define TIME_NODE               4004
#define INPUT_NODE              4005
#define INPUT_PROCEDURE_NODE    4006
#define INPUT_TERMINAL_NODE     4007
#define IO_TERMINAL_NODE        4008
#define OUTPUT_NODE             4009
#define OUTPUT_PROCEDURE_NODE   4010
#define EXTEND_NODE             4011
#define CHARACTERS_NODE         4012
#define LEADING_NODE            4013
#define CHARACTERS_BY_NODE      4014
#define ALL_NODE                4015
#define FIRST_NODE              4016
#define POINTER_NODE            4017
#define WITH_MISC_NODE          4018
#define DELIMIT_NODE            4019
#define COUNT_IN_NODE           4020
#define WITH_NODE               4021
#define IO_NODE                 4022

#define LITERAL_NODE            5001
#define ID_NODE                 5002
#define STRING_NODE             5003
#define INTEGER_NODE            5004
#define FLOATING_NODE           5005
#define SPACE_NODE              5006
```

```
#define ZERO_NODE                5007
#define HIGH_VALUE_NODE          5008
#define LOW_VALUE_NODE           5009
#define NINES_NODE               5010
#define QUOTE_NODE               5011
#define UD_NAME_NODE             5012
#define NUMERIC_NODE_TYPE        5013
#define ALPHABETIC_NODE_TYPE     5014
#define ALPHA_LOWER_NODE_TYPE    5015
#define ALPHA_UPPER_NODE_TYPE    5016
#define PARAMETER_NODE           5017
#define PROGRAM_NAME             5018
#define STD_IN                   5019
#define STD_OUT                  5020
#define FILLER_NODE              5021

#define NOT_NODE                 5500
#define OR_NODE                  5501
#define OR_ALL_NODE              5502
#define AND_NODE                 5503
#define ARITHMATIC_NODE          5504

#define CONFIG_SECTION_NODE      6000
#define MEMORY_SIZE_NODE         6003
#define COLLATING_SEQUENCE_NODE 6004
#define IMPLEMENT_LIST_NODE      6007
#define ALPHABET_LIST_NODE       6008
#define SYMBOLIC_LIST_NODE       6009
#define CLASS_LIST_NODE          6010
#define ALPHA_THRU_ELEM_NODE     6011
#define SYMBOL_IN_NODE           6013
#define SYMBOL_LIST_NODE         6014
#define F_NAME_LIST_NODE         6016
#define FILE_CNTL_ENTRY_NODE     6017
#define ALT_KEY_LIST_NODE        6018
#define DELIMITER_TERM_NODE      6019
#define RERUN_LIST_NODE          6020
#define SAME_LIST_NODE           6021
#define MULTI_FILE_NODE          6022
#define IO_CNTL_LIST_NODE        6023
#define CLASS_REST_NODE          6025
#define DECIMAL_POINT_NODE       6027
#define RESERVE_PART_NODE        6029
#define PADDING_PART_NODE        6030
#define DELIMITER_NODE           6031
#define FILE_STATUS_NODE         6032
```

```
#define REL_ACCESS_NODE         6034
#define RERUN_ON_NODE           6037
#define RERUN_COND_NODE         6038
#define ALPHA_THRU_LIST_NODE    6039
#define FILE_NAME_NODE          6040
#define SEQ_ORG_NODE            6041
#define SEQ_ACCESS_NODE         6042
#define REL_ORG_NODE            6043
#define INDEX_ORG_NODE          6044
#define F_SIZE_NODE             6045
#define F_TOP_NODE              6046
#define F_BOTTOM_NODE           6047
#define LEVEL_NUMBER_NODE       6048
#define DATA_NAME_NODE          6049
#define ALPHABET_SET_NODE       6050
#define OBJ_COMP_NAME_NODE      6051
#define RESET_ON_NODE           6052

#define FILE_SECTION_NODE       6100
#define WORK_SECTION_NODE       6101
#define LINK_SECTION_NODE       6102
#define REPORT_SEC_NODE         6103
#define FILE_DES_FILE_NODE      6105
#define FILE_VALUE_NODE         6109
#define FILE_DATA_NODE          6110
#define FILE_LINAGE_NODE        6111
#define REPORT_LIST_NODE        6112
#define REP_DES_NODE            6113
#define REP_PAGE_NODE           6114
#define REP_GROUP_NODE          6115
#define REP_NEXT_G_NODE         6117
#define REP_TYPE_NODE           6118
#define REP_T_C_REST_NODE       6119
#define FILE_BLOCK_NODE         6122
#define FILE_RECORD_NODE        6123
#define FILE_CODE_NODE          6124
#define F_LINE_FOOT_NODE        6129
#define REC_DES_ENTRY_NODE      6131
#define D_SIGN_SEP_NODE         6132
#define DATA_OCCUR_NODE         6133
#define D_O_KEY_LIST_NODE       6134
#define D_OCCUR_INDEX_NODE      6135
#define REP_CODE_NODE           6136
#define REP_FOOTING_NODE        6140
#define REP_DES_LIST_NODE       6141
```

```
#define REP_NAME_NODE          6142
#define REP_LINE_NODE          6143
#define REP_COLUMN_NODE        6144
#define REP_SEL_NODE           6145
#define REP_SUM_LIST_NODE      6146
#define ACCESS_MODE_NODE       6148
#define LEVEL_66_NODE          6150
#define LEVEL_88_NODE          6151
#define DATA_USAGE_NODE        6152
#define DATA_SIGN_NODE         6153
#define REP_CNTL_NODE          6154
#define EXT_GLOBAL_NODE        6156
#define RECORD_FROM_NODE       6157
#define RECORD_TO_NODE         6158
#define RECORD_VARYING_NODE    6159
#define FILE_LABEL_NODE        6160
#define DATA_88_LIST_NODE      6161
#define DATA_REDEF_NODE        6162
#define INT_TYPE_NODE          6163
#define REAL_TYPE_NODE         6164
#define TEXT_TYPE_NODE         6165
#define PICTURE_NODE           6166
#define D_O_KEY_NODE           6167
#define DATA_SYNC_NODE         6168
#define DATA_JUST_NODE         6169
#define DATA_BLANK_NODE        6170
#define DATA_VALUE_NODE        6171
#define DATA_THRU_NODE         6172
#define CONTROL_FINAL_NODE     6175
#define REPORT_HEADING_NODE    6176
#define REP_FIRST_DETAIL_NODE  6177
#define REP_LAST_DETAIL_NODE   6178
#define REP_USAGE_NODE         6179
#define USE_ST_NODE            6180
#define USE_REST_NODE          6181
#define        RECORDING_NODE  6182

#define CLOSE_OPT              6200
#define STR_DEL_LIST           6201
#define NEXT_SEN_NODE          6202
```

D.2 Table Generation Rules

This section shows the table generation rules defined for generating of *data declaration table*, *data definition table*, *data reference table*, *program structure table*, and *calling structure table* from the syntax tree produced from source code written in the COBOL programming language.

Data Declaration Generation Rules

The following rules are used to generate *data declaration tables*.

```
TABLE("DATA_DECLARATION" 11
      FIELD("ID"  INTEGER-TYPE)
      FIELD("NAME"  STRING-TYPE)
      FIELD("TYPE" STRING-TYPE)
      FIELD("PARENT"  INTEGER-TYPE)
      FIELD("POSITION"  INTEGER-TYPE)
      FIELD("LENGTH" INTEGER-TYPE)
      FIELD("FORMAT"  STRING-TYPE)
      FIELD("ALIAS"  STRING-TYPE)
      FIELD("VALUE" STRING-TYPE)
      FIELD("SCOPE" STRING-TYPE)
      FIELD("LINE_NO" INTEGER-TYPE))
RULE(CONDITION("PROGRAM_NODE")
     CONDITION("DIVISION_NODE" AND (NODE-VALUE = "DATA"))
     CONDITION("SECTION_NODE" AND (NODE-VALUE = "FILE"))
     PUT-VARIABLE(0 "EXTERNAL")
     GROUP(PATH(1)
   PROCESS(PATH(0))))
RULE(CONDITION("SECTION_NODE" AND (NODE-VALUE = "WORKING STORAGE"))
     PUT-VARIABLE(0 "INTERNAL")
     GROUP(PATH(1)
   PROCESS(PATH(0))))
RULE(CONDITION("SECTION_NODE" AND (NODE-VALUE = "LINKAGE"))
     PUT-VARIABLE(0 "PARAMETER")
     GROUP(PATH(1)
   PROCESS(PATH(0))))
RULE(CONDITION("FILE_DES_FILE_NODE")
     ENTRY(1)
     PUT-FIELD(("ID" GLOBAL-ID)
       ("NAME" PATH("FILE_NAME_NODE"/1):NODE-VALUE)
```

```
        ("TYPE" NODE-VALUE)
        ("PARENT" 0)
        ("POSITION" GROUP-ID)
        ("LENGTH" 0)
        ("SCOPE" VARIABLE(0))
        ("LINE_NO" NODE-LINE-NO))
      GROUP(PATH(1)
    PROCESS(PATH(0))))
RULE(CONDITION("REC_DES_ENTRY_NODE")
      ENTRY(1)
      PUT-FIELD(("ID" GLOBAL-ID)
        ("TYPE" NODE-VALUE)
        ("PARENT" TABLE-ID("LINE_NO" PATH(..):NODE-LINE-NO))
        ("POSITION" GROUP-ID)
        ("LENGTH" PATH("PICTURE_NODE"/1):NODE-KEY-1)
        ("FORMAT" PATH("PICTURE_NODE"/1):NODE-VALUE)
        ("ALIAS" PATH("DATA_REDEF_NODE"/1):NODE-VALUE)
        ("SCOPE" VARIABLE(0))
        ("LINE_NO" NODE-LINE-NO))
      PROCESS(PATH("DATA_NAME_NODE" OR "FILLER_NODE") FLAG("FIELD"))
      PROCESS(PATH("DATA_VALUE_NODE") FLAG("FIELD"))
      GROUP(PATH(1)
            PROCESS(PATH(0))))
RULE(CONDITION("DATA_NAME_NODE" AND FLAG("FIELD"))
      PUT-FIELD(("NAME" PATH(1):NODE-VALUE)))
RULE(CONDITION("FILLER_NODE" AND FLAG("FIELD"))
      PUT-FIELD(("NAME" "FILLER")))
RULE(CONDITION("DATA_VALUE_NODE" AND FLAG("FIELD"))
      PUT-FIELD(("VALUE" PATH(1):NODE-VALUE)))
```

Data Definition Table Generation Rules

The following rules are used to generate *data definition tables*.

```
TABLE("DATA_DEFINITION" 5
       FIELD("LINE_NO" INTEGER-TYPE)
       FIELD("DEST_NAME" STRING-TYPE)
       FIELD("NODE_ID"  INTEGER-TYPE)
       FIELD("SRC_TYPE" STRING-TYPE)
       FIELD("SRC_NAME"  STRING-TYPE))
RULE(CONDITION("PROGRAM_NODE")
     CONDITION("DIVISION_NODE" AND (NODE-VALUE = "PROCEDURE"))
     CONDITION("SECTION_NODE")
     CONDITION("PARAGRAPH_NODE")
     GROUP(PATH(1)
   PUT-VARIABLE(0 NODE-ID)
           PROCESS(PATH(0))))
RULE(CONDITION("THEN_NODE")
     CONDITION("ELSE_NODE")
     CONDITION("PERFORM_BODY_NODE" AND(PATH(0):NODE-KEY-1 = "STATEMENT_LIST"))
     CONDITION("WHEN_BODY_NODE")
     CONDITION("AT_END_NODE")
     CONDITION("NOT_AT_END_NODE")
     CONDITION("ON_SIZE_ERROR_NODE")
     CONDITION("NOT_ON_SIZE_ERROR_NODE")
     CONDITION("ON_OVERFLOW_NODE")
     CONDITION("NOT_ON_OVERFLOW_NODE")
     CONDITION("ON_EXCEPTION_NODE")
     CONDITION("NOT_ON_EXCEPTION_NODE")
     CONDITION("INVALID_KEY_NODE")
     CONDITION("NOT_INVALID_KEY_NODE")
     CONDITION("END_OF_PAGE_NODE")
     CONDITION("NOT_END_OF_PAGE_NODE")
     CONDITION("DATA_NODE")
     CONDITION("NO_DATA_NODE")
     CONDITION("DUM_SOURCE_NODE" AND FLAG("SOURCE"))
     CONDITION("TO_NODE" AND FLAG("SOURCE"))
     GROUP(PATH(1)
           PROCESS(PATH(0))))
RULE(CONDITION("ACCEPT_ST")
     ENTRY(1)
     PUT-FIELD(("LINE_NO" NODE-LINE-NO)
       ("DEST_NAME" PATH("DUM_SOURCE_NODE"/1):NODE-VALUE)
       ("NODE_ID" NODE-ID))
     PROCESS(PATH("FROM_NODE"/1) FLAG ("SOURCE")))
RULE(CONDITION("ADD_ST" AND PATH("GIVING_NODE"))
     PROCESS(PATH("ON_SIZE_ERROR_NODE"))
     PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE")))
RULE(CONDITION("ADD_ST")
     ENTRY(1)
     PUT-FIELD(("LINE_NO"      NODE-LINE-NO)
       ("NODE_ID" NODE-ID))
     PROCESS(PATH("TO_NODE"/1)  FLAG("DEST"))
     PROCESS(PATH("DUM_SOURCE_NODE"/1)   FLAG("SOURCE"))
     PROCESS(PATH("ON_SIZE_ERROR_NODE"))
     PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE")))
RULE(CONDITION("CALL_ST")
     PROCESS(PATH("ON_OVERFLOW_NODE"))
     PROCESS(PATH("ON_EXCEPTION_NODE"))
     PROCESS(PATH("NOT_ON_EXCEPTION_NODE")))
RULE(CONDITION("COMPUTE_ST")
     GROUP(PATH("LHS_NODE"/1)
   ENTRY(1)
   PUT-FIELD(("LINE_NO" NODE-LINE-NO)
     ("DEST_NAME" NODE-VALUE)
     ("NODE_ID" PATH(../..):NODE-ID)))
     PROCESS(PATH("RHS_NODE"/1) FLAG("SOURCE"))
     PROCESS(PATH("ON_SIZE_ERROR_NODE"))
     PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE")))
RULE(CONDITION("DELETE_ST")
     PROCESS(PATH("INVALID_KEY_NODE"))
     PROCESS(PATH("NOT_INVALID_KEY_NODE")))
RULE(CONDITION("DIVIDE_ST" AND PATH("GIVING_NODE"))
     PROCESS(PATH("REMAINDER_NODE"))
     PROCESS(PATH("ON_SIZE_ERROR_NODE"))
     PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE")))
RULE(CONDITION("DIVIDE_ST")
     PROCESS(PATH("REMAINDER_NODE"))
     PROCESS(PATH("ON_SIZE_ERROR_NODE"))
     PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE")))
RULE(CONDITION("EVALUATE_ST")
     GROUP(PATH(2)
           PROCESS(PATH("WHEN_BODY_NODE"))))
RULE(CONDITION("GENERATE_ST"))
RULE(CONDITION("IF_ST")
PROCESS(PATH("THEN_NODE"))
     PROCESS(PATH("ELSE_NODE")))
RULE(CONDITION("INITIALIZE_ST"))
RULE(CONDITION("INITIATE_ST"))
RULE(CONDITION("INSPECT_ST")
     ENTRY(1)
     PUT-FIELD(("LINE_NO"       NODE-LINE-NO)
               ("NODE_ID"       NODE-ID))
     PROCESS(PATH("DUM_SOURCE_NODE"/1)  FLAG("DEST"))
     GROUP(PATH("REPLACING_NODE"/1)
   PROCESS(PATH(1)      FLAG("SOURCE"))
   PROCESS(PATH("LITERAL_NODE"/1)    FLAG("SOURCE"))))
```

```
RULE(CONDITION("LITERAL_NODE" AND FLAG("SOURCE"))
     PUT-FIELD(("SRC_NAME"        NODE-VALUE)
               ("SRC_TYPE"        "CONSTANT")))
RULE(CONDITION("MERGE_ST"))
RULE(CONDITION("MOVE_ST")
     GROUP(PATH("TO_NODE"/1)
       ENTRY(1)
       PUT-FIELD(("LINE_NO" PATH(../..):NODE-LINE-NO)
 ("NODE_ID" PATH(../..):NODE-ID))
       PROCESS(PATH(0)  FLAG("DEST"))
       PROCESS(PATH(../../"DUM_SOURCE_NODE"/1) FLAG("SOURCE"))))
RULE(CONDITION("MULTIPLY_ST")
     PROCESS(PATH("ON_SIZE_ERROR_NODE"))
     PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE")))
RULE(CONDITION("PERFORM_ST")
     PROCESS(PATH("PERFORM_BODY_NODE")))
RULE(CONDITION("READ_ST" AND PATH("INTO_NODE"))
     ENTRY(1)
     PUT-FIELD(("LINE_NO" NODE-LINE-NO)
       ("NODE_ID" NODE-ID)
       ("SRC_TYPE" "FILE")
               ("SRC_NAME"  PATH("DUM_SOURCE_NODE"/1):NODE-VALUE))
     PROCESS(PATH("INTO_NODE"/1) FLAG("DEST"))
     PROCESS(PATH("INVALID_KEY_NODE"))
     PROCESS(PATH("NOT_INVALID_KEY_NODE"))
     PROCESS(PATH("AT_END_NODE"))
     PROCESS(PATH("NOT_AT_END_NODE")))
RULE(CONDITION("SUBSCRIPT_NODE" AND FLAG("DEST"))
     PUT-FIELD(("DEST_NAME"   PATH("UD_NAME_NODE"):NODE-VALUE)))
RULE(CONDITION("SUBSCRIPT_NODE" AND FLAG("SOURCE"))
     PUT-FIELD(("SRC_NAME"     PATH("UD_NAME_NODE"):NODE-VALUE)
("SRC_TYPE" "VARIABLE")))
RULE(CONDITION("READ_ST")
     ENTRY(1)
     PUT-FIELD(("LINE_NO" NODE-LINE-NO)
       ("DEST_NAME" PATH(/OFFSPRING("FILE_DES_FILE_NODE")
     [PATH("FILE_NAME_NODE"/1):NODE-VALUE =
     PATH("DUM_SOURCE_NODE"/1):NODE-VALUE]/
     "FILE_DATA_NODE"/1):NODE-VALUE)
       ("NODE_ID" NODE-ID)
       ("SRC_TYPE" "FILE")
       ("SRC_NAME" PATH("DUM_SOURCE_NODE"/1):NODE-VALUE))
     PROCESS(PATH("INVALID_KEY_NODE"))
     PROCESS(PATH("NOT_INVALID_KEY_NODE"))
     PROCESS(PATH("AT_END_NODE"))
     PROCESS(PATH("NOT_AT_END_NODE")))
RULE(CONDITION("RECEIVE_ST")
     PROCESS(PATH("DATA_NODE"))
     PROCESS(PATH("NO_DATA_NODE")))
RULE(CONDITION("RELEASE_ST"))
RULE(CONDITION("RETURN_ST")
     PROCESS(PATH("AT_END_NODE"))
     PROCESS(PATH("NOT_AT_END_NODE")))
RULE(CONDITION("REWRITE_ST")
     PROCESS(PATH("INVALID_KEY_NODE"))
     PROCESS(PATH("NOT_INVALID_KEY_NODE")))
RULE(CONDITION("SEARCH_ST")
     GROUP(PATH(2)
           PROCESS(PATH("WHEN_BODY_NODE"))))
RULE(CONDITION("SEND_ST"))
RULE(CONDITION("SET_ST"))
RULE(CONDITION("SORT_ST"))
RULE(CONDITION("START_ST")
     PROCESS(PATH("INVALID_KEY_NODE"))
     PROCESS(PATH("NOT_INVALID_KEY_NODE")))
RULE(CONDITION("STRING_ST")
     ENTRY(1)
     PUT-FIELD(("LINE_NO" NODE-LINE-NO)
("NODE_ID" NODE-ID))
     PROCESS(PATH("DUM_SOURCE_NODE"/1) FLAG("SOURCE"))
     PROCESS(PATH("INTO_NODE"/1) FLAG("DEST"))
     PROCESS(PATH("ON_OVERFLOW_NODE"))
     PROCESS(PATH("NOT_ON_OVERFLOW_NODE")))
RULE(CONDITION("STR_DEL_LIST" AND FLAG("SOURCE"))
     GROUP(PATH(1)
PROCESS(PATH(0) FLAG("SOURCE"))))
RULE(CONDITION("SUBTRACT_ST")
     PROCESS(PATH("ON_SIZE_ERROR_NODE"))
     PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE")))
RULE(CONDITION("UNSTRING_ST")
     PROCESS(PATH("ON_OVERFLOW_NODE"))
     PROCESS(PATH("NOT_ON_OVERFLOW_NODE")))
RULE(CONDITION("WRITE_ST" AND PATH("FROM_NODE"))
     ENTRY(1)
     PUT-FIELD(("LINE_NO" NODE-LINE-NO)
       ("DEST_NAME" PATH("DUM_SOURCE_NODE"/1):NODE-VALUE)
```

```
          ("NODE_ID" NODE-ID))
        PROCESS(PATH("FROM_NODE"/1) FLAG("SOURCE"))
        ENTRY(2)
        PUT-FIELD(("LINE_NO" NODE-LINE-NO)
          ("DEST_NAME" PATH(/OFFSPRING("FILE_DES_FILE_NODE")
        [PATH("FILE_DATA_NODE"/1):NODE-VALUE =
        PATH("DUM_SOURCE_NODE"/1):NODE-VALUE]/
        "FILE_NAME_NODE"/1):NODE-VALUE)
          ("NODE_ID" NODE-ID)
          ("SRC_TYPE" "VARIABLE")
          ("SRC_NAME" PATH("DUM_SOURCE_NODE"/1):NODE-VALUE))
        PROCESS(PATH("INVALID_KEY_NODE"))
        PROCESS(PATH("NOT_INVALID_KEY_NODE"))
        PROCESS(PATH("END_OF_PAGE_NODE"))
        PROCESS(PATH("NOT_END_OF_PAGE_NODE")))
RULE(CONDITION("WRITE_ST")
        ENTRY(1)
        PUT-FIELD(("LINE_NO" NODE-LINE-NO)
          ("DEST_NAME" PATH(/OFFSPRING("FILE_DES_FILE_NODE")
        [PATH("FILE_DATA_NODE"/1):NODE-VALUE =
        PATH("DUM_SOURCE_NODE"/1):NODE-VALUE]/
        "FILE_NAME_NODE"/1):NODE-VALUE)
          ("NODE_ID" NODE-ID)
          ("SRC_TYPE" "VARIABLE")
          ("SRC_NAME" PATH("DUM_SOURCE_NODE"/1):NODE-VALUE))
        PROCESS(PATH("INVALID_KEY_NODE"))
        PROCESS(PATH("NOT_INVALID_KEY_NODE"))
        PROCESS(PATH("END_OF_PAGE_NODE"))
        PROCESS(PATH("NOT_END_OF_PAGE_NODE")))
RULE(CONDITION("UD_NAME_NODE", AND FLAG("DEST"))
        PUT-FIELD(("DEST_NAME"  NODE-VALUE)))
RULE(CONDITION("UD_NAME_NODE" AND FLAG("SOURCE"))
        PUT-FIELD(("SRC_NAME"  NODE-VALUE)
          ("SRC_TYPE" "VARIABLE")))
RULE(CONDITION("LITERAL_NODE" AND FLAG("SOURCE"))
        CONDITION("INTEGER_NODE" AND FLAG("SOURCE"))
        CONDITION("FLOATING_NODE" AND FLAG("SOURCE"))
        CONDITION("SPACE_NODE" AND FLAG("SOURCE"))
        CONDITION("ZERO_NODE" AND FLAG("SOURCE"))
        CONDITION("HIGH_VALUE_NODE" AND FLAG("SOURCE"))
        CONDITION("LOW_VALUE_NODE" AND FLAG("SOURCE"))
        CONDITION("QUOTE_NODE" AND FLAG("SOURCE"))
        PUT-FIELD(("SRC_TYPE" "CONSTANT")
       ("SRC_NAME" NODE-VALUE)))
RULE(CONDITION("DATE_NODE" AND FLAG("SOURCE"))
     CONDITION("DAY_NODE" AND FLAG("SOURCE"))
     CONDITION("DAY_OF_WEEK_NODE" AND FLAG("SOURCE"))
     CONDITION("TIME_NODE" AND FLAG("SOURCE"))
     PUT-FIELD(("SRC_TYPE" "CONSTANT")
       ("SRC_NAME" NODE-NAME)))
RULE(CONDITION("ARITHMATIC_NODE" AND FLAG("SOURCE"))
     CONDITION("AND_NODE" AND FLAG("SOURCE"))
     CONDITION("OR_NODE" AND FLAG("SOURCE"))
     CONDITION("REL_OP_NODE" AND FLAG("SOURCE"))
     CONDITION("CLASS_TYPE_NODE" AND FLAG("SOURCE"))
     PROCESS(PATH(1) FLAG("SOURCE"))
     PROCESS(PATH(2) FLAG("SOURCE")))
```

Data Reference Table Generation Rules

The following rules are used to generate *data reference tables*.

```
TABLE("DATA_REFERENCE" 3
      FIELD("LINE_NO"   INTEGER-TYPE)
      FIELD("DATA_NAME" STRING-TYPE)
      FIELD("NODE_ID"   INTEGER-TYPE))
RULE(CONDITION("PROGRAM_NODE")
     CONDITION("DIVISION_NODE" AND (NODE-VALUE = "PROCEDURE"))
     CONDITION("SECTION_NODE")
     CONDITION("PARAGRAPH_NODE")
     GROUP(PATH(1)
   PUT-VARIABLE(0 NODE-ID)
           PROCESS(PATH(0))))
RULE(CONDITION("THEN_NODE")
     CONDITION("ELSE_NODE")
     CONDITION("PERFORM_BODY_NODE" AND(PATH(0):NODE-KEY-1 = "STATEMENT_LIST"))
     CONDITION("WHEN_BODY_NODE")
     CONDITION("AT_END_NODE")
     CONDITION("NOT_AT_END_NODE")
     CONDITION("ON_SIZE_ERROR_NODE")
     CONDITION("NOT_ON_SIZE_ERROR_NODE")
     CONDITION("ON_OVERFLOW_NODE")
     CONDITION("NOT_ON_OVERFLOW_NODE")
     CONDITION("ON_EXCEPTION_NODE")
     CONDITION("NOT_ON_EXCEPTION_NODE")
     CONDITION("INVALID_KEY_NODE")
     CONDITION("NOT_INVALID_KEY_NODE")
     CONDITION("END_OF_PAGE_NODE")
     CONDITION("NOT_END_OF_PAGE_NODE")
     CONDITION("DATA_NODE")
     CONDITION("NO_DATA_NODE")
     CONDITION("DUM_SOURCE_NODE")
     GROUP(PATH(1)
           PROCESS(PATH(0))))
RULE(CONDITION("ACCEPT_ST")
     PUT-VARIABLE(0 NODE-ID)
     PROCESS(PATH("FROM_NODE"/1)))
RULE(CONDITION("ADD_ST")
     PROCESS(PATH("ON_SIZE_ERROR_NODE"))
     PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE")))
RULE(CONDITION("CALL_ST")
     PROCESS(PATH("ON_OVERFLOW_NODE"))
     PROCESS(PATH("ON_EXCEPTION_NODE"))
     PROCESS(PATH("NOT_ON_EXCEPTION_NODE")))
RULE(CONDITION("COMPUTE_ST")
     PROCESS(PATH("RHS_NODE"/1))
     PROCESS(PATH("ON_SIZE_ERROR_NODE"))
     PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE")))
RULE(CONDITION("DELETE_ST")
     PROCESS(PATH("INVALID_KEY_NODE"))
     PROCESS(PATH("NOT_INVALID_KEY_NODE")))
RULE(CONDITION("DIVIDE_ST")
     PROCESS(PATH("ON_SIZE_ERROR_NODE"))
     PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE")))
RULE(CONDITION("EVALUATE_ST")
     PUT-VARIABLE(0 NODE-ID)
     GROUP(PATH(1/1)
           PROCESS(PATH(1)))
     GROUP(PATH(2)
   PROCESS(PATH("WHEN_COND_NODE"/1))
           PROCESS(PATH("WHEN_BODY_NODE"))))
RULE(CONDITION("WHEN_COND_NODE")
GROUP(PATH(1)
PROCESS(PATH(1))))
RULE(CONDITION("GOTO_ST")
     PROCESS(PATH(1))
     PROCESS(PATH("DEPENDING_ON_NODE"/1)))
```

```
RULE(CONDITION("IF_ST")
     PUT-VARIABLE(0 NODE-ID)
     PROCESS(PATH(1))
     PROCESS(PATH("THEN_NODE"))
     PROCESS(PATH("ELSE_NODE")))
RULE(CONDITION("INITIALIZE_ST"))
RULE(CONDITION("INSPECT_ST")
     PUT-VARIABLE(0 NODE-ID)
     GROUP(PATH("REPLACING_NODE"/1)
     PROCESS(PATH(1))
     PROCESS(PATH("LITERAL_NODE"/1))))
RULE(CONDITION("LITERAL_NODE")
     ENTRY(1)
     PUT-FIELD(("LINE_NO"     NODE-LINE-NO)
       ("DATA_NAME"   NODE-VALUE)
       ("NODE_ID"     VARIABLE(0))))
RULE(CONDITION("MERGE_ST"))
RULE(CONDITION("MOVE_ST")
     PUT-VARIABLE(0 NODE-ID)
     PROCESS(PATH("DUM_SOURCE_NODE"/1)))
RULE(CONDITION("MULTIPLY_ST")
     PROCESS(PATH("ON_SIZE_ERROR_NODE"))
     PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE")))
RULE(CONDITION("PERFORM_ST")
     PUT-VARIABLE(0 PATH("TIME_NODE"):NODE-ID)
     PROCESS(PATH("TIMES_NODE"/1))
     PUT-VARIABLE(0 PATH("UNTIL_NODE"):NODE-ID)
     PROCESS(PATH("UNTIL_NODE"/1))
     PUT-VARIABLE(0 PATH("VARYING_NODE"):NODE-ID)
     PROCESS(PATH("VARYING_NODE"))
     PUT-VARIABLE(0 NODE-ID)
     PROCESS(PATH("PERFORM_BODY_NODE")))
RULE(CONDITION("READ_ST")
     PUT-VARIABLE(0 NODE-ID)
     PROCESS(PATH("DUM_SOURCE_NODE"/1))
     PROCESS(PATH("INVALID_KEY_NODE"))
     PROCESS(PATH("NOT_INVALID_KEY_NODE"))
     PROCESS(PATH("AT_END_NODE"))
     PROCESS(PATH("NOT_AT_END_NODE")))
RULE(CONDITION("RECEIVE_ST")
     PROCESS(PATH("DATA_NODE"))
     PROCESS(PATH("NO_DATA_NODE")))
RULE(CONDITION("RELEASE_ST"))
RULE(CONDITION("RETURN_ST")
     PROCESS(PATH("AT_END_NODE"))
     PROCESS(PATH("NOT_AT_END_NODE")))
RULE(CONDITION("REWRITE_ST")
     PROCESS(PATH("INVALID_KEY_NODE"))
     PROCESS(PATH("NOT_INVALID_KEY_NODE")))
RULE(CONDITION("SEARCH_ST")
     GROUP(PATH(2)
           PROCESS(PATH("WHEN_BODY_NODE"))))
RULE(CONDITION("SEND_ST"))
RULE(CONDITION("SET_ST"))
RULE(CONDITION("SORT_ST"))
RULE(CONDITION("START_ST")
     PROCESS(PATH("INVALID_KEY_NODE"))
     PROCESS(PATH("NOT_INVALID_KEY_NODE")))
RULE(CONDITION("STRING_ST")
     PUT-VARIABLE(0 NODE-ID)
     GROUP(PATH("DUM_SOURCE_NODE"/1)
           PROCESS(PATH(0)))
     PROCESS(PATH("ON_OVERFLOW_NODE"))
     PROCESS(PATH("NOT_ON_OVERFLOW_NODE")))
RULE(CONDITION("STR_DEL_LIST")
     GROUP(PATH(1)
           PROCESS(PATH(0))))
RULE(CONDITION("DELIMIT_NODE")
           PROCESS(PATH(1)))
RULE(CONDITION("SUBTRACT_ST")
     PROCESS(PATH("ON_SIZE_ERROR_NODE"))
     PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE")))
RULE(CONDITION("UNSTRING_ST")
     PROCESS(PATH("ON_OVERFLOW_NODE"))
     PROCESS(PATH("NOT_ON_OVERFLOW_NODE")))
RULE(CONDITION("WRITE_ST")
     PUT-VARIABLE(0 NODE-ID)
     PROCESS(PATH("DUM_SOURCE_NODE"/1))
     PROCESS(PATH("FROM_NODE"/1))
     PROCESS(PATH("INVALID_KEY_NODE"))
     PROCESS(PATH("NOT_INVALID_KEY_NODE"))
     PROCESS(PATH("END_OF_PAGE_NODE"))
     PROCESS(PATH("NOT_END_OF_PAGE_NODE")))
RULE(CONDITION("SUBSCRIPT_NODE")
     PROCESS(PATH(1)))
RULE(CONDITION("NOT_NODE")
     CONDITION("SIGN_TYPE_NODE")
     PROCESS(PATH(1)))
RULE(CONDITION("AND_NODE")
     CONDITION("OR_NODE")
     CONDITION("REL_OP_NODE")
     CONDITION("ARITHMATIC_NODE")
     CONDITION("CLASS_TYPE_NODE")
     PROCESS(PATH(1))
     PROCESS(PATH(2)))
RULE(CONDITION("UD_NAME_NODE")
     ENTRY(1)
     PUT-FIELD(("LINE_NO" NODE-LINE-NO)
       ("DATA_NAME" NODE-VALUE)
       ("NODE_ID" VARIABLE(0))))
RULE(CONDITION("VARYING_NODE")
     PUT-VARIABLE(0 NODE-ID)
     GROUP(PATH(1)
PROCESS(PATH(0))))
```

Program Structure Table Generation Rules

The following rules are used to generate *program structure tables*.

```
TABLE("PROGRAM_STRUCTURE" 6
      FIELD("ID"  INTEGER-TYPE)
      FIELD("LINE_NO"  INTEGER-TYPE)
      FIELD("NAME"  STRING-TYPE)
      FIELD("POSITION"  INTEGER-TYPE)
      FIELD("PARENT"  INTEGER-TYPE)
      FIELD("NODE_ID" INTEGER-TYPE))
RULE(CONDITION("PROGRAM_NODE")
     ENTRY(1)
     PUT-FIELD(("ID"  GLOBAL-ID)
               ("LINE_NO"  PATH(1/1):NODE-LINE-NO)
               ("NAME"  PATH(1/1):NODE-VALUE)
               ("POSITION" GROUP-ID)
               ("PARENT"  0)
       ("NODE_ID" NODE-ID))
     GROUP(PATH(1)
           PROCESS(PATH(0))))
RULE(CONDITION("DIVISION_NODE" AND (NODE-VALUE = "PROCEDURE"))
     GROUP(PATH(1)
           PROCESS(PATH(0))))
RULE(CONDITION("SECTION_NODE")
     CONDITION("PARAGRAPH_NODE")
     ENTRY(1)
     PUT-FIELD(("ID"  GLOBAL-ID)
               ("LINE_NO"  NODE-LINE-NO)
               ("NAME"  NODE-VALUE)
               ("POSITION" GROUP-ID)
               ("PARENT"  TABLE-ID("NODE_ID"
 PATH(ANCESTOR("PROGRAM_NODE" OR
       "SECTION_NODE")):
 NODE-ID))
       ("NODE_ID" NODE-ID))
     GROUP(PATH(1)
           PROCESS(PATH(0))))
```

Calling Structure Table Generation Rules

The following rules are used to generate *calling structure tables*.

```
TABLE("CALLING_STRUCTURE" 4
      FIELD("LINE_NO"  INTEGER-TYPE)
      FIELD("CALLING_BLOCK"  STRING-TYPE)
      FIELD("CALLED_BLOCK"  STRING-TYPE)
      FIELD("LAST_BLOCK" STRING-TYPE))
RULE(CONDITION("PROGRAM_NODE")
     CONDITION("DIVISION_NODE" AND (NODE-VALUE = "PROCEDURE"))
     CONDITION("SECTION_NODE")
     CONDITION("PARAGRAPH_NODE")
     GROUP(PATH(1)
           PROCESS(PATH(0))))
RULE(CONDITION("ADD_ST")
     CONDITION("COMPUTE_ST")
     CONDITION("DIVIDE_ST")
     CONDITION("MULTIPLY_ST")
     CONDITION("SUBTRACT_ST")
     PROCESS(PATH("ON_SIZE_ERROR_NODE"))
     PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE")))
RULE(CONDITION("CALL_ST")
     CONDITION("STRING_ST")
     CONDITION("UNSTRING_ST")
     PROCESS(PATH("ON_EXCEPTION_NODE"))
     PROCESS(PATH("NOT_ON_EXCEPTION_NODE"))
     PROCESS(PATH("ON_OVERFLOW_NODE"))
     PROCESS(PATH("NOT_ON_OVERFLOW_NODE")))
RULE(CONDITION("DELETE_ST")
     CONDITION("READ_ST")
     CONDITION("RETURN_ST")
     CONDITION("REWRITE_ST")
     CONDITION("START_ST")
     CONDITION("WRITE_ST")
     PROCESS(PATH("AT_END_NODE"))
     PROCESS(PATH("NOT_AT_END_NODE"))
     PROCESS(PATH("END_OF_PAGE_NODE"))
     PROCESS(PATH("NOT_END_OF_PAGE_NODE"))
     PROCESS(PATH("INVALID_KEY_NODE"))
     PROCESS(PATH("NOT_INVALID_KEY_NODE")))
RULE(CONDITION("EVALUATE_ST")
     GROUP(PATH(2)
           PROCESS(PATH("WHEN_BODY_NODE"))))
RULE(CONDITION("IF_ST")
     PROCESS(PATH("THEN_NODE"))
     PROCESS(PATH("ELSE_NODE")))
RULE(CONDITION("PERFORM_ST")
     PROCESS(PATH("PERFORM_BODY_NODE")))
RULE(CONDITION("RECEIVE_ST")
     PROCESS(PATH("DATA_NODE"))
     PROCESS(PATH("NO_DATA_NODE")))
RULE(CONDITION("SEARCH_ST")
     GROUP(PATH(2)
           PROCESS(PATH("WHEN_BODY_NODE"))))
RULE(CONDITION("THEN_NODE")
     CONDITION("ELSE_NODE")
     CONDITION("PERFORM_BODY_NODE" AND (PATH(0):NODE-KEY-1 = "STATEMENT_LIST"))
     CONDITION("WHEN_BODY_NODE")
     CONDITION("AT_END_NODE")
     CONDITION("NOT_AT_END_NODE")
     CONDITION("ON_SIZE_ERROR_NODE")
     CONDITION("NOT_ON_SIZE_ERROR_NODE")
     CONDITION("ON_OVERFLOW_NODE")
     CONDITION("NOT_ON_OVERFLOW_NODE")
     CONDITION("ON_EXCEPTION_NODE")
     CONDITION("NOT_ON_EXCEPTION_NODE")
     CONDITION("INVALID_KEY_NODE")
     CONDITION("NOT_INVALID_KEY_NODE")
     CONDITION("END_OF_PAGE_NODE")
     CONDITION("NOT_END_OF_PAGE_NODE")
     CONDITION("DATA_NODE")
     CONDITION("NO_DATA_NODE")
     GROUP(PATH(1)
           PROCESS(PATH(0))))
RULE(CONDITION("PERFORM_BODY_NODE" AND(PATH(0):NODE-KEY-1 = "PROCEDURE"))
     PROCESS(PATH(1)))
RULE(CONDITION("THROUGH_NODE")
     ENTRY(1)
     PUT-FIELD(("LINE_NO" NODE-LINE-NO)
       ("CALLING_BLOCK" PATH(ANCESTOR("PARAGRAPH_NODE" OR
      "SECTION_NODE")):NODE-VALUE)
       ("CALLED_BLOCK"  PATH(1):NODE-VALUE)
       ("LAST_BLOCK"  PATH(2):NODE-VALUE)))
RULE(CONDITION("UD_NAME_NODE")
     CONDITION("INTEGER_NODE")
     ENTRY(1)
     PUT-FIELD(("LINE_NO"  NODE-LINE-NO)
       ("CALLING_BLOCK" PATH(ANCESTOR("PARAGRAPH_NODE" OR
      "SECTION_NODE")):NODE-VALUE)
       ("CALLED_BLOCK"  NODE-VALUE)
       ("LAST_BLOCK"  "")))
```

D.3 Model Generation Rules

This section shows the table generation rules defined for generating the following RMA graphs from the syntax tree produced from source code written in the COBOL programming language: DM2, EOPM, FSM, IOPM, IOPM2, and UIM.

DM2 Generation Rule

The following rules are used to generate DM2.

```
RULE(CONDITION("PROGRAM_NODE")
     GROUP(PATH(1)
   PROCESS(PATH(0))))
```

```
RULE(CONDITION("DIVISION_NODE" AND (NODE-VALUE = "DATA"))
      GROUP(PATH(1)
   PROCESS(PATH(0))))
RULE(CONDITION("SECTION_NODE" AND (NODE-VALUE = "FILE"))
      GRAPH(NAME("EXTERNAL_DATA") "DM2" 1 1)
      ENTITY(1 "space" NAME("external_data"))
      GROUP(PATH(1)
            PROCESS(PATH(0) RETURN(2))
   RELATION(3 "belong" NAME("") 2 1)))
RULE(CONDITION("SECTION_NODE" AND (NODE-VALUE = "WORKING STORAGE"))
      GRAPH(NAME("INTERNAL_DATA") "DM2" 1 1)
      ENTITY(1 "space" NAME ("internal_data"))
      GROUP(PATH(1)
            PROCESS(PATH(0) RETURN(2))
   RELATION(3 "belong" NAME("") 2 1)))
RULE(CONDITION("SECTION_NODE" AND (NODE-VALUE = "LINKAGE"))
      GRAPH(NAME("LINKAGE_DATA") "DM2" 1 1)
      ENTITY(1 "space" NAME ("linkage_data"))
      GROUP(PATH(1)
   PROCESS(PATH(0) RETURN(2))
   RELATION(3 "belong" NAME("") 2 1)))
RULE(CONDITION("FILE_DES_FILE_NODE" RETURN(1))
      ENTITY(1 "file" NAME(PATH("FILE_NAME_NODE"/1):NODE-VALUE))
      GROUP(PATH(1)
   PROCESS(PATH(0) RETURN(2))
            RELATION(3 "define" NAME("") 2 1)))
RULE(CONDITION("REC_DES_ENTRY_NODE" AND (NODE-VALUE = "RECORD") RETURN(1))
      PROCESS(PATH("DATA_NAME_NODE" OR "FILLER_NODE"))
      ENTITY(1 "record" NAME(VARIABLE(2)))
      GROUP(PATH(1)
   PROCESS(PATH(0) RETURN(2))
            RELATION(3 "contain" NAME("") 1 2)))
RULE(CONDITION("REC_DES_ENTRY_NODE" AND (NODE-VALUE <> "RECORD") RETURN(1))
      PROCESS(PATH("DATA_NAME_NODE" OR "FILLER_NODE"))
      ENTITY(1 "data_item" NAME(VARIABLE(2))
     ATTRIBUTE("data_format" NAME(PATH("PICTURE_NODE"/1):NODE-VALUE))))
RULE(CONDITION("DATA_NAME_NODE")
      VARIABLE(2 NAME(PATH(1):NODE-VALUE)))
RULE(CONDITION("FILLER_NODE" AND FLAG("FIELD"))
      VARIABLE(2 NAME("FILLER")))
RULE(CONDITION("DIVISION_NODE" AND (NODE-VALUE = "PROCEDURE"))
      GRAPH(NAME("SYSTEM_DATA") "DM2" 1 1)
     ENTITY(1 "space" NAME("system_data"))
     GROUP(PATH(1)
           PROCESS(PATH(0))))
RULE(CONDITION("SECTION_NODE")
     CONDITION("PARAGRAPH_NODE")
     CONDITION("THEN_NODE")
     CONDITION("ELSE_NODE")
     CONDITION("PERFORM_BODY_NODE" AND(PATH(0):NODE-KEY-1 = "STATEMENT_LIST"))
     CONDITION("WHEN_BODY_NODE")
     CONDITION("AT_END_NODE")
     CONDITION("NOT_AT_END_NODE")
     CONDITION("ON_SIZE_ERROR_NODE")
     CONDITION("NOT_ON_SIZE_ERROR_NODE")
     CONDITION("ON_OVERFLOW_NODE")
     CONDITION("NOT_ON_OVERFLOW_NODE")
     CONDITION("ON_EXCEPTION_NODE")
     CONDITION("NOT_ON_EXCEPTION_NODE")
     CONDITION("INVALID_KEY_NODE")
     CONDITION("NOT_INVALID_KEY_NODE")
     CONDITION("END_OF_PAGE_NODE")
     CONDITION("NOT_END_OF_PAGE_NODE")
     CONDITION("DATA_NODE")
     CONDITION("NO_DATA_NODE")
     CONDITION("DUM_SOURCE_NODE")
     CONDITION("VARYING_NODE")
     CONDITION("STR_DEL_LIST")
     GROUP(PATH(1)
           PROCESS(PATH(0))))
RULE(CONDITION("ACCEPT_ST")
     PROCESS(PATH("FROM_NODE"/1)))
RULE(CONDITION("ADD_ST")
     CONDITION("DIVIDE_ST")
     CONDITION("MULTIPLY_ST")
     CONDITION("SUBTRACT_ST")
     PROCESS(PATH("DUM_SOURCE_NODE"))
     PROCESS(PATH("TO_NODE"))
     PROCESS(PATH("INTO_NODE"))
     PROCESS(PATH("BY_NODE"))
     PROCESS(PATH("FROM_NODE"))
     PROCESS(PATH("RHS_NODE"/1))
     PROCESS(PATH("ON_SIZE_ERROR_NODE"))
     PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE")))
RULE(CONDITION("CALL_ST")
     PROCESS(PATH("ON_OVERFLOW_NODE"))
     PROCESS(PATH("ON_EXCEPTION_NODE"))
     PROCESS(PATH("NOT_ON_EXCEPTION_NODE")))
RULE(CONDITION("DELETE_ST")
     CONDITION("READ_ST")
     CONDITION("RETURN_ST")
     CONDITION("REWRITE_ST")
     CONDITION("START_ST")
     PROCESS(PATH("DUM_SOURCE_NODE"/1))
     PROCESS(PATH("AT_END_NODE"))
     PROCESS(PATH("NOT_AT_END_NODE"))
     PROCESS(PATH("INVALID_KEY_NODE"))
     PROCESS(PATH("NOT_INVALID_KEY_NODE")))
RULE(CONDITION("EVALUATE_ST")
     GROUP(PATH(1/1)
           PROCESS(PATH(1)))
     GROUP(PATH(2)
   PROCESS(PATH("WHEN_COND_NODE"/1))
           PROCESS(PATH("WHEN_BODY_NODE"))))
RULE(CONDITION("WHEN_COND_NODE")
GROUP(PATH(1)
PROCESS(PATH(1))))
RULE(CONDITION("GOTO_ST")
     PROCESS(PATH(1))
     PROCESS(PATH("DEPENDING_ON_NODE"/1)))
RULE(CONDITION("IF_ST")
     PROCESS(PATH(1))
     PROCESS(PATH("THEN_NODE"))
     PROCESS(PATH("ELSE_NODE")))
RULE(CONDITION("INSPECT_ST")
     GROUP(PATH("REPLACING_NODE"/1)
           PROCESS(PATH(1))
   PROCESS(PATH("LITERAL_NODE"/1))))
RULE(CONDITION("LITERAL_NODE" RETURN())
     CONDITION("INTEGER_NODE")
     CONDITION("FLOATING_NODE")
     CONDITION("SPACE_NODE")
     CONDITION("ZERO_NODE")
     CONDITION("HIGH_VALUE_NODE")
     CONDITION("LOW_VALUE_NODE")
     CONDITION("QUOTE_NODE")
     ENTITY(1 "data_item" NAME(PATH(0):NODE-VALUE) UNIQUE
    ATTRIBUTE("data_value" NAME(NODE-VALUE)))
```

```
     RELATION(2 "define" NAME("") 1 SAME:"space":NAME("system_data")))
RULE(CONDITION("MOVE_ST")
     PROCESS(PATH("DUM_SOURCE_NODE"/1)))
RULE(CONDITION("PERFORM_ST")
     PROCESS(PATH("TIMES_NODE"/1))
     PROCESS(PATH("UNTIL_NODE"/1))
     PROCESS(PATH("VARYING_NODE"))
     PROCESS(PATH("PERFORM_BODY_NODE")))
RULE(CONDITION("RECEIVE_ST")
     PROCESS(PATH("DATA_NODE"))
     PROCESS(PATH("NO_DATA_NODE")))
RULE(CONDITION("SEARCH_ST")
     GROUP(PATH(2)
           PROCESS(PATH("WHEN_BODY_NODE"))))
RULE(CONDITION("STRING_ST")
     GROUP(PATH("DUM_SOURCE_NODE"/1)
           PROCESS(PATH(0)))
     PROCESS(PATH("ON_OVERFLOW_NODE"))
     PROCESS(PATH("NOT_ON_OVERFLOW_NODE")))
RULE(CONDITION("UNSTRING_ST")
     PROCESS(PATH("ON_OVERFLOW_NODE"))
     PROCESS(PATH("NOT_ON_OVERFLOW_NODE")))
RULE(CONDITION("WRITE_ST")
     PROCESS(PATH("DUM_SOURCE_NODE"/1))
     PROCESS(PATH("FROM_NODE"/1))
     PROCESS(PATH("INVALID_KEY_NODE"))
     PROCESS(PATH("NOT_INVALID_KEY_NODE"))
     PROCESS(PATH("END_OF_PAGE_NODE"))
     PROCESS(PATH("NOT_END_OF_PAGE_NODE")))
RULE(CONDITION("SUBSCRIPT_NODE")
     CONDITION("NOT_NODE")
     CONDITION("SIGN_TYPE_NODE")
     CONDITION("DELIMIT_NODE")
     PROCESS(PATH(1)))
RULE(CONDITION("AND_NODE")
     CONDITION("OR_NODE")
     CONDITION("REL_OP_NODE")
     CONDITION("ARITHMATIC_NODE")
     CONDITION("CLASS_TYPE_NODE")
     PROCESS(PATH(1))
     PROCESS(PATH(2)))
```

EOPM Generation Rule

The following rules are used to generate EOPM.

```
RULE(CONDITION("PROGRAM_NODE" RETURN(1 2))
   GRAPH(NAME(PATH(1/1):NODE-VALUE) "EOPM" 1 2)
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2)) ) )
RULE(CONDITION("DIVISION_NODE" AND (NODE-VALUE = "PROCEDURE") RETURN(1 2))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2))
      CONNECT(3 "becomes" NAME("") 2 1) ) )
RULE(CONDITION("SECTION_NODE" RETURN(1 1))
   ENTITY(1 "state" NAME(PATH(0):NODE-VALUE))
   GRAPH(NAME(PATH(0):NODE-VALUE) "EOPM" 2 3)
   ENTITY(2 "state" NAME(PATH(0):NODE-VALUE))
   ENTITY(3 "state" NAME("Exit"))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(4 5))
      CONNECT(6 "becomes" NAME("") 5 4) )
   RELATION(7 "becomes" NAME("") 2 4)
   RELATION(8 "becomes" NAME("") 5 3) )
RULE(CONDITION("PARAGRAPH_NODE" RETURN(2 3))
   ENTITY(1 "dummy" NAME(PATH(0):NODE-VALUE))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(2 3))
      CONNECT(4 "becomes" NAME("") 3 2))
   RELATION(5 "becomes" NAME("") 1 2))
RULE(CONDITION("EVALUATE_ST" RETURN(1 4))
   ENTITY(1 "state" NAME(PATH(1/1):NODE-ELEMENT))
   ENTITY(4 "dummy" NAME(""))
   GROUP(PATH(2)
      PROCESS(PATH("WHEN_BODY_NODE") RETURN(2 3))
      RELATION(5 "becomes" NAME("") 1 2
 ATTRIBUTE("events" NAME(PATH(1):NODE-ELEMENT)))
      RELATION(6 "becomes" NAME("") 3 4) ) )
RULE(CONDITION("GOTO_ST"AND(NODE-KEY-1 = "") RETURN(1))
   ENTITY(1 "state" NAME(PATH(0):NODE-ELEMENT))
   RELATION(2 "becomes" NAME("") 1 ANY:NAME(PATH(1/1):NODE-VALUE)) )
RULE(CONDITION("GOTO_ST"AND(NODE-KEY-1 = "DEPENDING ON") RETURN(1))
   ENTITY(1 "state" NAME("GO TO DEPENDING ON "+PATH(2/1):NODE-VALUE))
   GROUP(PATH(1/1)
     RELATION(2 "becomes" NAME("") 1 ANY:NAME(PATH(1/1):NODE-VALUE) RESERVED
 ATTRIBUTE("events" NAME(1:1))) ) )
RULE(CONDITION("IF_ST" RETURN(1 6))
   ENTITY(1 "state" NAME(PATH(1):NODE-ELEMENT))
   ENTITY(6 "dummy" NAME(""))
   PROCESS(PATH("THEN_NODE") RETURN(2 3))
   PROCESS(PATH("ELSE_NODE") RETURN(4 5))
   RELATION(7 "becomes" NAME("") 1 2 RESERVED
      ATTRIBUTE("events" NAME("true")))
   RELATION(8 "becomes" NAME("") 1 4 RESERVED
      ATTRIBUTE("events" NAME("false")))
   RELATION(9 "becomes" NAME("") 3 6)
   RELATION(10 "becomes" NAME("") 5 6) )
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "") RETURN(1 2))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(1 2)) )
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "TIMES") RETURN(1 4))
   ENTITY(1 "state" NAME("Loop "+PATH("TIMES_NODE"/1):NODE-VALUE))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(2 3))
   ENTITY(4 "dummy" NAME(""))
   RELATION(5 "becomes" NAME("") 1 4 RESERVED
      ATTRIBUTE("events" NAME("true")))
   RELATION(6 "becomes" NAME("") 1 2 RESERVED
      ATTRIBUTE("events" NAME("false")))
   RELATION(7 "becomes" NAME("") 3 1) )
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "BEFORE")
   AND(NODE-KEY-2 <> "VARYING") RETURN(1 4))
     CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "AFTER")
   AND(NODE-KEY-2 <> "VARYING") RETURN(2 4))
   ENTITY(1 "state" NAME(PATH("UNTIL_NODE"):NODE-ELEMENT))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(2 3))
   ENTITY(4 "dummy" NAME(""))
   RELATION(5 "becomes" NAME("") 1 4 RESERVED
      ATTRIBUTE("events" NAME("true")))
   RELATION(6 "becomes" NAME("") 1 2 RESERVED
      ATTRIBUTE("events" NAME("false")))
   RELATION(7 "becomes" NAME("") 3 1) )
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "BEFORE")
   AND(NODE-KEY-2 = "VARYING") RETURN(1 2))
     CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "AFTER")
   AND(NODE-KEY-2 = "VARYING") RETURN(1 2))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(1 2)) )
RULE(CONDITION("ADD_ST" RETURN(1 28))
      CONDITION("CALL_ST" RETURN(1 28))
      CONDITION("COMPUTE_ST" RETURN(1 28))
      CONDITION("DELETE_ST" RETURN(1 28))
      CONDITION("DIVIDE_ST" RETURN(1 28))
      CONDITION("MULTIPLY_ST" RETURN(1 28))
      CONDITION("READ_ST" RETURN(1 28))
      CONDITION("RECEIVE_ST" RETURN(1 28))
      CONDITION("RETURN_ST" RETURN(1 28))
      CONDITION("REWRITE_ST" RETURN(1 28))
      CONDITION("START_ST" RETURN(1 28))
      CONDITION("STRING_ST" RETURN(1 28))
      CONDITION("SUBTRACT_ST" RETURN(1 28))
      CONDITION("UNSTRING_ST" RETURN(1 28))
      CONDITION("WRITE_ST" RETURN(1 28))
    PROCESS(PATH("INVALID_KEY_NODE") RETURN(1 2))
    PROCESS(PATH("NOT_INVALID_KEY_NODE") RETURN(3 4))
    RELATION(29 "becomes" NAME("") 1 2)
    RELATION(30 "becomes" NAME("") 2 3)
    PROCESS(PATH("AT_END_NODE") RETURN(5 6))
    PROCESS(PATH("NOT_AT_END_NODE") RETURN(7 8))
    RELATION(31 "becomes" NAME("") 4 5)
    RELATION(32 "becomes" NAME("") 6 7)
    PROCESS(PATH("DATA_NODE") RETURN(9 10))
    PROCESS(PATH("NO_DATA_NODE") RETURN(11 12))
    RELATION(33 "becomes" NAME("") 8 9)
    RELATION(34 "becomes" NAME("") 10 11)
    PROCESS(PATH("ON_OVERFLOW_NODE") RETURN(13 14))
    PROCESS(PATH("NOT_ON_OVERFLOW_NODE") RETURN(15 16))
    RELATION(35 "becomes" NAME("") 12 13)
    RELATION(36 "becomes" NAME("") 14 15)
    PROCESS(PATH("END_OF_PAGE_NODE") RETURN(17 18))
    PROCESS(PATH("NOT_END_OF_PAGE_NODE") RETURN(19 20))
    RELATION(37 "becomes" NAME("") 16 17)
    RELATION(38 "becomes" NAME("") 18 19)
    PROCESS(PATH("ON_SIZE_ERROR_NODE") RETURN(21 22))
    PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE") RETURN(23 24))
    RELATION(39 "becomes" NAME("") 20 21)
    RELATION(40 "becomes" NAME("") 22 23)
    PROCESS(PATH("ON_EXCEPTION_NODE") RETURN(25 26))
    PROCESS(PATH("NOT_ON_EXCEPTION_NODE") RETURN(27 28))
    RELATION(41 "becomes" NAME("") 24 25)
    RELATION(42 "becomes" NAME("") 26 27) )
RULE(CONDITION("SEARCH_ST" RETURN(1 2))
   ENTITY(1 "state" NAME("Search_st"))
   ENTITY(2 "dummy" NAME(""))
   GROUP(PATH(2)
      PROCESS(PATH("WHEN_BODY_NODE") RETURN(3 4))
      RELATION(5 "becomes" NAME("") 1 3
 ATTRIBUTE("events" NAME(PATH(0):NODE-ELEMENT)))
      RELATION(6 "becomes" NAME("") 4 2) ) )
RULE(CONDITION("THEN_NODE" RETURN(1 2))
     CONDITION("ELSE_NODE" RETURN(1 2))
     CONDITION("WHEN_BODY_NODE" RETURN(1 2))
     CONDITION("PERFORM_BODY_NODE"AND(NODE-KEY-1 = "STATEMENT_LIST")
       RETURN(1 2))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2))
      CONNECT(3 "becomes" NAME("") 2 1) ) )
RULE(CONDITION("AT_END_NODE" RETURN(1 4))
     CONDITION("NOT_AT_END_NODE" RETURN(1 4))
     CONDITION("ON_SIZE_ERROR_NODE" RETURN(1 4))
     CONDITION("NOT_ON_SIZE_ERROR_NODE" RETURN(1 4))
     CONDITION("ON_OVERFLOW_NODE" RETURN(1 4))
     CONDITION("NOT_ON_OVERFLOW_NODE" RETURN(1 4))
     CONDITION("ON_EXCEPTION_NODE" RETURN(1 4))
     CONDITION("NOT_ON_EXCEPTION_NODE" RETURN(1 4))
     CONDITION("INVALID_KEY_NODE" RETURN(1 4))
     CONDITION("NOT_INVALID_KEY_NODE" RETURN(1 4))
     CONDITION("END_OF_PAGE_NODE" RETURN(1 4))
     CONDITION("NOT_END_OF_PAGE_NODE" RETURN(1 4))
     CONDITION("DATA_NODE" RETURN(1 4))
     CONDITION("NO_DATA_NODE" RETURN(1 4))
   ENTITY(1 "state" NAME(PATH(0):NODE-NAME))
   ENTITY(4 "dummy" NAME(""))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(2 3))
      CONNECT(5 "becomes" NAME("") 3 2) )
   RELATION(6 "becomes" NAME("") 1 2 RESERVED
      ATTRIBUTE("events" NAME("true")))
   RELATION(7 "becomes" NAME("") 1 4 RESERVED
      ATTRIBUTE("events" NAME("false")))
   RELATION(8 "becomes" NAME("") 3 4) )
RULE(CONDITION("PERFORM_BODY_NODE"AND(NODE-KEY-1 = "PROCEDURE") RETURN(1 2))
   PROCESS(PATH(1) RETURN(1 2)) )
RULE(CONDITION("THROUGH_NODE" RETURN(1 1))
   ENTITY(1 "state" NAME(PATH(1):NODE-VALUE+" through "+PATH(2):NODE-VALUE)))
```

```
RULE(CONDITION("UD_NAME_NODE" RETURN(1 1))
     CONDITION("INTEGER_NODE" RETURN(1 1))
   ENTITY(1 "state" NAME(PATH(0):NODE-VALUE)) )
```

FSM Generation Rule

The following rules are used to generate FSM.

```
RULE(CONDITION("PROGRAM_NODE")
   GRAPH(NAME(PATH(1/1):NODE-VALUE) "FSM" 1 3)
   ENTITY(1 "actor" NAME(PATH(1/1):NODE-VALUE))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(2 3)) ) )
RULE(CONDITION("DIVISION_NODE" AND (NODE-VALUE = "PROCEDURE") RETURN(1 2))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2))
      CONNECT(3 "is_followed_by" NAME("") 1 2) ) )
RULE(CONDITION("SECTION_NODE" RETURN(1 1))
   ENTITY(1 "action" NAME(PATH(0):NODE-VALUE))
   RELATION(5 "takes" NAME("") SAME:ENTRY 1)
   GRAPH(NAME(PATH(0):NODE-VALUE) "FSM" 2 4)
   ENTITY(2 "actor" NAME(PATH(0):NODE-VALUE))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(3 4))
      CONNECT(6 "is_followed_by" NAME("") 4 3) ) )
RULE(CONDITION("PARAGRAPH_NODE" RETURN(1 2))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2))
      CONNECT(3 "is_followed_by" NAME("") 2 1) ) )
RULE(CONDITION("CLOSE_ST" RETURN(1 2))
   GROUP(PATH(1)
      ENTITY(1 "action" NAME("CLOSE"))
      ENTITY(2 "action" NAME("CLOSE"))
      ENTITY(3 "object" NAME(PATH(0):NODE-VALUE) UNIQUE)
      RELATION(4 "affects" NAME("") 1 3)
      RELATION(5 "takes" NAME("") SAME:ENTRY 1)
      RELATION(6 "dummy" NAME("") 1 2)
      CONNECT(7 "is_followed_by" NAME("") 2 1 RESERVED) ) )
RULE(CONDITION("OPEN_ST" RETURN(1 2))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2))
      CONNECT(3 "is_followed_by" NAME("") 2 1 RESERVED) ) )
RULE(CONDITION("INPUT_NODE" RETURN(1 2))
   GROUP(PATH(1)
      ENTITY(1 "action" NAME("OPEN INPUT"))
      ENTITY(2 "action" NAME("OPEN INPUT"))
      ENTITY(3 "object" NAME(PATH(0):NODE-VALUE) UNIQUE)
      RELATION(4 "affects" NAME("") 1 3)
      RELATION(5 "takes" NAME("") SAME:ENTRY 1)
      RELATION(6 "dummy" NAME("") 1 2)
      CONNECT(7 "is_followed_by" NAME("") 2 1 RESERVED) ) )
RULE(CONDITION("OUTPUT_NODE" RETURN(1 2))
   GROUP(PATH(1)
      ENTITY(1 "action" NAME("OPEN_OUTPUT"))
      ENTITY(2 "action" NAME("OPEN OUTPUT"))
      ENTITY(3 "object" NAME(PATH(0):NODE-VALUE) UNIQUE)
      RELATION(4 "affects" NAME("") 1 3)
      RELATION(5 "takes" NAME("") SAME:ENTRY 1)
      RELATION(6 "dummy" NAME("") 1 2)
      CONNECT(7 "is_followed_by" NAME("") 2 1 RESERVED) ) )
RULE(CONDITION("IO_NODE" RETURN(1 1))
   GROUP(PATH(1)
      ENTITY(1 "action" NAME("OPEN I-O"))
      ENTITY(2 "object" NAME(PATH(0):NODE-VALUE) UNIQUE)
      RELATION(3 "affects" NAME("") 1 2)
      RELATION(4 "takes" NAME("") SAME:ENTRY 1) ) )
RULE(CONDITION("ADD_ST")
     CONDITION("COMPUTE_ST")
     CONDITION("DIVIDE_ST")
     CONDITION("MULTIPLY_ST")
     CONDITION("SUBTRACT_ST")
   PROCESS(PATH("ON_SIZE_ERROR_NODE"))
   PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE")))
RULE(CONDITION("CALL_ST")
   ENTITY(1 "action" NAME(PATH(0):NODE-ELEMENT))
   RELATION(2 "takes" NAME("") SAME:ENTRY 1)
   PROCESS(PATH("ON_OVERFLOW_NODE"))
   PROCESS(PATH("ON_EXCEPTION_NODE"))
   PROCESS(PATH("NOT_ON_EXCEPTION_NODE")))
RULE(CONDITION("DELETE_ST" RETURN(1 1))
     CONDITION("REWRITE_ST"RETURN(1 1))
     CONDITION("START_ST"RETURN(1 1))
   ENTITY(1 "action" NAME(PATH(0):NODE-ELEMENT))
   RELATION(2 "takes" NAME("") SAME:ENTRY 1)
   PROCESS(PATH("INVALID_KEY_NODE"))
   PROCESS(PATH("NOT_INVALID_KEY_NODE")))
RULE(CONDITION("EVALUATE_ST")
   GROUP(PATH(2)
      PROCESS(PATH("WHEN_BODY_NODE"))))
RULE(CONDITION("IF_ST")
   PROCESS(PATH("THEN_NODE"))
   PROCESS(PATH("ELSE_NODE")))
RULE(CONDITION("PERFORM_ST" RETURN(1 2))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(1 2)))
RULE(CONDITION("READ_ST" RETURN(1 1))
   ENTITY(1 "action" NAME("READ"))
   ENTITY(2 "object" NAME(PATH(1/1):NODE-VALUE) UNIQUE)
   RELATION(3 "affects" NAME("") 1 2)
   RELATION(14 "takes" NAME("") SAME:ENTRY 1)
   PROCESS(PATH("INVALID_KEY_NODE"))
   PROCESS(PATH("NOT_INVALID_KEY_NODE"))
   PROCESS(PATH("AT_END_NODE"))
   PROCESS(PATH("NOT_AT_END_NODE")))
RULE(CONDITION("RECEIVE_ST")
   PROCESS(PATH("DATA_NODE"))
   PROCESS(PATH("NO_DATA_NODE")))
RULE(CONDITION("RETURN_ST")
   PROCESS(PATH("AT_END_NODE"))
   PROCESS(PATH("NOT_AT_END_NODE")) )
RULE(CONDITION("SEARCH_ST")
   GROUP(PATH(2)
      PROCESS(PATH("WHEN_BODY_NODE"))))
RULE(CONDITION("STRING_ST")
     CONDITION("UNSTRING_ST")
   PROCESS(PATH("ON_OVERFLOW_NODE"))
   PROCESS(PATH("NOT_ON_OVERFLOW_NODE")))
RULE(CONDITION("WRITE_ST" RETURN(1 1))
   ENTITY(1 "action" NAME("WRITE"))
   ENTITY(2 "object" NAME(PATH(1/1):NODE-VALUE) UNIQUE)
   RELATION(13 "affects" NAME("") 1 2)
   RELATION(14 "takes" NAME("") SAME:ENTRY 1)
   PROCESS(PATH("INVALID_KEY_NODE"))
   PROCESS(PATH("NOT_INVALID_KEY_NODE"))
   PROCESS(PATH("END_OF_PAGE_NODE"))
   PROCESS(PATH("NOT_END_OF_PAGE_NODE")))
```

```
RULE(CONDITION("THEN_NODE" RETURN(1 2))
     CONDITION("ELSE_NODE" RETURN(1 2))
     CONDITION("PERFORM_BODY_NODE"AND(NODE-KEY-1 = "STATEMENT_LIST")
       RETURN(1 2))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2))
      CONNECT(3 "is_followed_by" NAME("") 2 1) ) )
RULE(CONDITION("AT_END_NODE")
     CONDITION("NOT_AT_END_NODE")
     CONDITION("ON_SIZE_ERROR_NODE")
     CONDITION("NOT_ON_SIZE_ERROR_NODE")
     CONDITION("ON_OVERFLOW_NODE")
     CONDITION("NOT_ON_OVERFLOW_NODE")
     CONDITION("ON_EXCEPTION_NODE")
     CONDITION("NOT_ON_EXCEPTION_NODE")
     CONDITION("INVALID_KEY_NODE")
     CONDITION("NOT_INVALID_KEY_NODE")
     CONDITION("END_OF_PAGE_NODE")
     CONDITION("NOT_END_OF_PAGE_NODE")
     CONDITION("DATA_NODE")
     CONDITION("NO_DATA_NODE")
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2))
      CONNECT(3 "is_followed_by" NAME("") 2 1) ) )
RULE(CONDITION("PERFORM_BODY_NODE"AND(NODE-KEY-1 = "PROCEDURE") RETURN(1 2))
   PROCESS(PATH(1) RETURN(1 2)) )
RULE(CONDITION("THROUGH_NODE" RETURN(2 2))
   ENTITY(2 "action" NAME(PATH(1):NODE-VALUE+" through "+PATH(1):NODE-VALUE))
   RELATION(1 "takes" NAME("") SAME:ENTRY 2) )
RULE(CONDITION("UD_NAME_NODE" RETURN(2 2))
   ENTITY(2 "action" NAME(PATH(0):NODE-VALUE))
   RELATION(1 "takes" NAME("") SAME:ENTRY 2) )
```

IOPM Generation Rule

The following rules are used to generate IOPM.

```
RULE(CONDITION("PROGRAM_NODE" RETURN(1 2))
   GRAPH(NAME(PATH(1/1):NODE-VALUE) "IOPM" 1 2)
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2)) ) )
RULE(CONDITION("DIVISION_NODE" AND (NODE-VALUE = "PROCEDURE") RETURN(1 2))
   ENTITY(1 "entry" NAME("ENTRY") RESERVED
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   ENTITY(2 "exit" NAME("EXIT") RESERVED
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(3 4))
      CONNECT(11 "dummy" NAME("") 4 3) )
   RELATION(12 "dummy" NAME("") 1 3)
   RELATION(13 "dummy" NAME("") 4 2) )
RULE(CONDITION("SECTION_NODE" RETURN(1 3))
   ENTITY(1 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   ENTITY(2 "process" NAME(PATH(0):NODE-VALUE)
      ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
   ENTITY(3 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   RELATION(11 "consuming_arc" NAME("") 1 2)
   RELATION(12 "producing_arc" NAME("") 2 3)
   GRAPH(NAME(PATH(0):NODE-VALUE) "IOPM" 4 5)
   ENTITY(4 "entry" NAME("") RESERVED
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   ENTITY(5 "exit" NAME("EXIT") RESERVED
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(6 7))
      CONNECT(13 "dummy" NAME("") 7 6) )
   RELATION(14 "dummy" NAME("") 4 6)
   RELATION(15 "dummy" NAME("") 7 5) )
RULE(CONDITION("PARAGRAPH_NODE" RETURN(1 3))
   ENTITY(1 "place" NAME(PATH(0):NODE-VALUE)
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(2 3))
      CONNECT(11 "dummy" NAME("") 3 2) )
   RELATION(12 "dummy" NAME("") 1 2) )
RULE(CONDITION("ACCEPT_ST" RETURN(1 3))
     CONDITION("ALTER_ST" RETURN(1 3))
     CONDITION("CANCEL_ST" RETURN(1 3))
     CONDITION("CLOSE_ST" RETURN(1 3))
```

```
      CONDITION("CONTINUE_ST" RETURN(1 3))
      CONDITION("DISABLE_ST" RETURN(1 3))
      CONDITION("DISPLAY_ST" RETURN(1 3))
      CONDITION("ENABLE_ST" RETURN(1 3))
      CONDITION("ENTER_ST" RETURN(1 3))
      CONDITION("EXIT_ST" RETURN(1 3))
      CONDITION("GENERATE_ST" RETURN(1 3))
      CONDITION("INITIALIZE_ST" RETURN(1 3))
      CONDITION("INITIATE_ST" RETURN(1 3))
      CONDITION("INSPECT_ST" RETURN(1 3))
      CONDITION("MERGE_ST" RETURN(1 3))
      CONDITION("MOVE_ST" RETURN(1 3))
      CONDITION("OPEN_ST" RETURN(1 3))
      CONDITION("PURGE_ST" RETURN(1 3))
      CONDITION("RELEASE_ST" RETURN(1 3))
      CONDITION("REPLACE_ST" RETURN(1 3))
      CONDITION("SEND_ST" RETURN(1 3))
      CONDITION("SET_ST" RETURN(1 3))
      CONDITION("SORT_ST" RETURN(1 3))
      CONDITION("STOP_ST" RETURN(1 3))
      CONDITION("SUPPRESS_ST" RETURN(1 3))
      CONDITION("TERMINATE_ST" RETURN(1 3))
    ENTITY(1 "place" NAME("")
       ATTRIBUTE("initial_no_tokens" NAME("0")))
    ENTITY(2 "process" NAME(PATH(0):NODE-ELEMENT)
       ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
    ENTITY(3 "place" NAME("")
       ATTRIBUTE("initial_no_tokens" NAME("0")))
    RELATION(11 "consuming_arc" NAME("") 1 2)
    RELATION(12 "producing_arc" NAME("") 2 3) )
RULE(CONDITION("ADD_ST" RETURN(1 31))
      CONDITION("CALL_ST" RETURN(1 31))
      CONDITION("COMPUTE_ST" RETURN(1 31))
      CONDITION("DELETE_ST" RETURN(1 31))
      CONDITION("DIVIDE_ST" RETURN(1 31))
      CONDITION("MULTIPLY_ST" RETURN(1 31))
      CONDITION("READ_ST" RETURN(1 31))
      CONDITION("RECEIVE_ST" RETURN(1 31))
      CONDITION("RETURN_ST" RETURN(1 31))
      CONDITION("REWRITE_ST" RETURN(1 31))
      CONDITION("START_ST" RETURN(1 31))
      CONDITION("STRING_ST" RETURN(1 31))
    CONDITION("SUBTRACT_ST" RETURN(1 31))
    CONDITION("UNSTRING_ST" RETURN(1 31))
    CONDITION("WRITE_ST" RETURN(1 31))
  ENTITY(1 "place" NAME("")
     ATTRIBUTE("initial_no_tokens" NAME("0")))
  ENTITY(2 "process" NAME(PATH(0):NODE-ELEMENT)
     ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
  ENTITY(3 "place" NAME("")
     ATTRIBUTE("initial_no_tokens" NAME("0")))
  RELATION(41 "consuming_arc" NAME("") 1 2)
  RELATION(42 "producing_arc" NAME("") 2 3)
  PROCESS(PATH("ON_SIZE_ERROR_NODE") RETURN(4 5))
  PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE") RETURN(6 7))
  RELATION(43 "dummy" NAME("") 3 4)
  RELATION(44 "dummy" NAME("") 5 6)
  PROCESS(PATH("ON_OVERFLOW_NODE") RETURN(8 9))
  PROCESS(PATH("NOT_ON_OVERFLOW_NODE") RETURN(10 11))
  RELATION(45 "dummy" NAME("") 7 8)
  RELATION(46 "dummy" NAME("") 9 11)
  PROCESS(PATH("ON_EXCEPTION_NODE") RETURN(12 13))
  PROCESS(PATH("NOT_ON_EXCEPTION_NODE") RETURN(14 15))
  RELATION(47 "dummy" NAME("") 11 12)
  RELATION(48 "dummy" NAME("") 13 14)
  PROCESS(PATH("INVALID_KEY_NODE") RETURN(16 17))
  PROCESS(PATH("NOT_INVALID_KEY_NODE") RETURN(18 19))
  RELATION(49 "dummy" NAME("") 15 16)
  RELATION(50 "dummy" NAME("") 17 18)
  PROCESS(PATH("AT_END_NODE") RETURN(20 21))
  PROCESS(PATH("NOT_AT_END_NODE") RETURN(22 23))
  RELATION(51 "dummy" NAME("") 19 20)
  RELATION(52 "dummy" NAME("") 21 22)
  PROCESS(PATH("DATA_NODE") RETURN(24 25))
  PROCESS(PATH("NO_DATA_NODE") RETURN(26 27))
  RELATION(53 "dummy" NAME("") 23 24)
  RELATION(54 "dummy" NAME("") 25 26)
  PROCESS(PATH("END_OF_PAGE_NODE") RETURN(28 29))
  PROCESS(PATH("NOT_END_OF_PAGE_NODE") RETURN(30 31))
  RELATION(55 "dummy" NAME("") 27 28)
  RELATION(56 "dummy" NAME("") 29 30) )
RULE(CONDITION("EVALUATE_ST" RETURN(1 3))
  ENTITY(1 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   ENTITY(2 "branch" NAME(PATH(1):NODE-ELEMENT)
         ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
   ENTITY(3 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   GROUP(PATH(2)
      PROCESS(PATH("WHEN_BODY_NODE") RETURN(4 5))
      RELATION(12 "condition" NAME("") 2 4 RESERVED
 ATTRIBUTE("condition" NAME(PATH(1):NODE-ELEMENT))
 ATTRIBUTE("probability" NAME("0.5")))
      RELATION(13 "dummy" NAME("") 5 3) )
   RELATION(11 "consuming_arc" NAME("") 1 2) )
RULE(CONDITION("GOTO_ST"AND(NODE-KEY-1 = "") RETURN(1))
   ENTITY(1 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   ENTITY(2 "process" NAME(PATH(0):NODE-ELEMENT)
      ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
   ENTITY(3 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   RELATION(11 "consuming_arc" NAME("") 1 2)
   RELATION(12 "producing_arc" NAME("") 2 3)
   RELATION(13 "dummy" NAME("") 3 ANY:NAME(PATH(1/1):NODE-VALUE)) )
RULE(CONDITION("GOTO_ST"AND(NODE-KEY-1 = "DEPENDING ON") RETURN(1))
   ENTITY(1 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   ENTITY(2 "process" NAME(PATH(0):NODE-ELEMENT)
      ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
   RELATION(3 "consuming_arc" NAME("") 1 2)
   GROUP(PATH(1/1)
      RELATION(4 "condition" NAME("") 2 ANY:NAME(PATH(1/1):NODE-VALUE) RESERVED
 ATTRIBUTE("condition" NAME(1:1))
 ATTRIBUTE("probability" NAME("0.5"))) ) )
RULE(CONDITION("IF_ST" RETURN(1 9))
   ENTITY(1 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   ENTITY(2 "branch" NAME(PATH(1):NODE-ELEMENT)
      ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
   ENTITY(3 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   ENTITY(4 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   ENTITY(9 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   PROCESS(PATH("THEN_NODE") RETURN(5 6))
   PROCESS(PATH("ELSE_NODE") RETURN(7 8))
   RELATION(11 "consuming_arc" NAME("") 1 2)
   RELATION(12 "condition" NAME("") 2 3 RESERVED
      ATTRIBUTE("condition" NAME("true"))
      ATTRIBUTE("probability" NAME("0.5")))
   RELATION(13 "condition" NAME("") 2 4 RESERVED
      ATTRIBUTE("condition" NAME("false"))
      ATTRIBUTE("probability" NAME("0.5")))
   RELATION(15 "dummy" NAME("") 3 5)
   RELATION(16 "dummy" NAME("") 6 9)
   RELATION(17 "dummy" NAME("") 4 7)
   RELATION(18 "dummy" NAME("") 8 9) )
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "") RETURN(1 4))
   ENTITY(1 "place" NAME("PERFORM_ST")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(2 3))
   ENTITY(4 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   RELATION(11 "dummy" NAME("") 1 2)
   RELATION(12 "dummy" NAME("") 3 4) )
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "TIMES") RETURN(1 5))
   ENTITY(1 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   ENTITY(2 "branch" NAME("Loop "+PATH("TIMES_NODE"/1):NODE-VALUE)
      ATTRIBUTE("node_id" NAME(PATH("TIME_NODE"):NODE-ID)))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(3 4))
   ENTITY(5 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   ENTITY(6 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   RELATION(11 "consuming_arc" NAME("") 1 2)
   RELATION(12 "condition" NAME("") 2 5 RESERVED
      ATTRIBUTE("condition" NAME("true"))
      ATTRIBUTE("probability" NAME("0.5")))
   RELATION(13 "condition" NAME("") 2 6 RESERVED
      ATTRIBUTE("condition" NAME("false"))
      ATTRIBUTE("probability" NAME("0.5")))
   RELATION(14 "dummy" NAME("") 6 3)
   RELATION(15 "dummy" NAME("") 4 1) )
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "BEFORE")
   AND(NODE-KEY-2 <> "VARYING") RETURN(1 5))
   ENTITY(1 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   ENTITY(2 "branch" NAME(PATH("UNTIL_NODE"):NODE-ELEMENT)
      ATTRIBUTE("node_id" NAME(PATH("UNTIL_NODE"):NODE-ID)))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(3 4))
   ENTITY(5 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   ENTITY(6 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   RELATION(11 "consuming_arc" NAME("") 1 2)
   RELATION(12 "condition" NAME("") 2 5 RESERVED
      ATTRIBUTE("condition" NAME("true"))
      ATTRIBUTE("probability" NAME("0.5")))
   RELATION(13 "condition" NAME("") 2 6 RESERVED
      ATTRIBUTE("condition" NAME("false"))
      ATTRIBUTE("probability" NAME("0.5")))
   RELATION(14 "dummy" NAME("") 6 3)
   RELATION(15 "dummy" NAME("") 4 1) )
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "AFTER")
   AND(NODE-KEY-2 <> "VARYING") RETURN(1 5))
   ENTITY(1 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   ENTITY(2 "branch" NAME(PATH("UNTIL_NODE"):NODE-ELEMENT)
      ATTRIBUTE("node_id" NAME(PATH("UNTIL_NODE"):NODE-ID)))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(3 4))
   ENTITY(5 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   RELATION(11 "consuming_arc" NAME("") 4 2)
   RELATION(12 "condition" NAME("") 2 5 RESERVED
      ATTRIBUTE("condition" NAME("true"))
      ATTRIBUTE("probability" NAME("0.5")))
   RELATION(13 "condition" NAME("") 2 1 RESERVED
      ATTRIBUTE("condition" NAME("false"))
      ATTRIBUTE("probability" NAME("0.5")))
   RELATION(14 "dummy" NAME("") 1 3) )
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "BEFORE")
   AND(NODE-KEY-2 = "VARYING") RETURN(1 9))
   ENTITY(1 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
    PROCESS(PATH("PERFORM_BODY_NODE") RETURN(2 3))
    GROUP(PATH("VARYING_NODE"/1)
      ENTITY(4 "place" NAME("")
 ATTRIBUTE("initial_no_tokens" NAME("0")))
      ENTITY(5 "process" NAME(PATH("DUM_SOURCE_NODE"/1):NODE-VALUE+" sets to "
       +PATH("FROM_NODE"/1):NODE-VALUE)
        ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
      ENTITY(6 "place" NAME("")
 ATTRIBUTE("initial_no_tokens" NAME("0")))
      ENTITY(7 "branch" NAME("Until "+PATH("UNTIL_NODE"/1):NODE-VALUE)
        ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
      ENTITY(8 "place" NAME("")
 ATTRIBUTE("initial_no_tokens" NAME("0")))
      ENTITY(9 "place" NAME("")
 ATTRIBUTE("initial_no_tokens" NAME("0")))
      ENTITY(10 "place" NAME("")
 ATTRIBUTE("initial_no_tokens" NAME("0")))
      ENTITY(11 "process" NAME(PATH("DUM_SOURCE_NODE"/1):NODE-VALUE+
       " increases "+PATH("BY_NODE"/1):NODE-VALUE)
        ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
      RELATION(21 "consuming_arc" NAME("") 4 5)
      RELATION(22 "producing_arc" NAME("") 5 6)
      RELATION(23 "consuming_arc" NAME("") 6 7)
      RELATION(24 "condition" NAME("") 7 9 RESERVED
 ATTRIBUTE("condition" NAME("true"))
 ATTRIBUTE("probability" NAME("0.5")))
      RELATION(25 "condition" NAME("") 7 8 RESERVED
 ATTRIBUTE("condition" NAME("false"))
 ATTRIBUTE("probability" NAME("0.5")))
      RELATION(26 "consuming_arc" NAME("") 10 11)
      RELATION(27 "producing_arc" NAME("") 11 6)
      CONNECT(28 "dummy" NAME("") 8 4)
      CONNECT(29 "dummy" NAME("") REVERSE 9 10) )
    RELATION(30 "dummy" NAME("") 8 2)
    RELATION(31 "dummy" NAME("") 3 10)
    RELATION(32 "dummy" NAME("") 1 4) )
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "AFTER")
    AND(NODE-KEY-2 = "VARYING") RETURN(1 9))
    ENTITY(1 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
```

```
  PROCESS(PATH("PERFORM_BODY_NODE") RETURN(2 3))
  GROUP(PATH("VARYING_NODE"/1)
     ENTITY(4 "place" NAME("")
ATTRIBUTE("initial_no_tokens" NAME("0")))
     ENTITY(5 "process" NAME(PATH("DUM_SOURCE_NODE"/1):NODE-VALUE+" sets to "
      +PATH("FROM_NODE"/1):NODE-VALUE)
        ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
     ENTITY(6 "place" NAME("")
ATTRIBUTE("initial_no_tokens" NAME("0")))
     ENTITY(7 "branch" NAME("Until "+PATH("UNTIL_NODE"/1):NODE-VALUE)
        ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
     ENTITY(8 "place" NAME("")
ATTRIBUTE("initial_no_tokens" NAME("0")))
     ENTITY(9 "place" NAME("")
ATTRIBUTE("initial_no_tokens" NAME("0")))
     ENTITY(10 "place" NAME("")
ATTRIBUTE("initial_no_tokens" NAME("0")))
     ENTITY(11 "process" NAME(PATH("DUM_SOURCE_NODE"/1):NODE-VALUE+
      " increases "+PATH("BY_NODE"/1):NODE-VALUE)
        ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
     RELATION(12 "consuming_arc" NAME("") 4 5)
     RELATION(13 "producing_arc" NAME("") 5 6)
     RELATION(14 "consuming_arc" NAME("") 10 7)
     RELATION(15 "condition" NAME("") 7 9 RESERVED
ATTRIBUTE("condition" NAME("true"))
ATTRIBUTE("probability" NAME("0.5")))
     RELATION(20 "condition" NAME("") 7 8 RESERVED
ATTRIBUTE("condition" NAME("false"))
ATTRIBUTE("probability" NAME("0.5")))
     RELATION(21 "consuming_arc" NAME("") 8 11)
     RELATION(22 "producing_arc" NAME("") 11 6)
     CONNECT(23 "dummy" NAME("") 6 4)
     CONNECT(24 "dummy" NAME("") REVERSE 9 10) )
  RELATION(25 "dummy" NAME("") 6 2)
  RELATION(26 "dummy" NAME("") 3 10)
  RELATION(27 "dummy" NAME("") 1 4) )
RULE(CONDITION("SEARCH_ST" RETURN(1 2))
  ENTITY(1 "place" NAME("")
     ATTRIBUTE("initial_no_tokens" NAME("0")))
  ENTITY(2 "place" NAME("")
     ATTRIBUTE("initial_no_tokens" NAME("0")))
```

```
   GROUP(PATH(2)
      ENTITY(3 "branch" NAME(PATH(0):NODE-ELEMENT)
         ATTRIBUTE("node_id" NAME(PATH(..):NODE-ID)))
      ENTITY(4 "place" NAME("")
 ATTRIBUTE("initial_no_tokens" NAME("0")))
      ENTITY(5 "place" NAME("")
 ATTRIBUTE("initial_no_tokens" NAME("0")))
      PROCESS(PATH("WHEN_BODY_NODE") RETURN(6 7))
      RELATION(9 "condition" NAME("") 3 4 RESERVED
 ATTRIBUTE("condition" NAME("true"))
 ATTRIBUTE("probability" NAME("0.5")))
      RELATION(10 "condition" NAME("") 3 5 RESERVED
 ATTRIBUTE("condition" NAME("false"))
 ATTRIBUTE("probability" NAME("0.5")))
      RELATION(11 "dummy" NAME("") 4 6)
      RELATION(12 "dummy" NAME("") 7 2)
      CONNECT(13 "consuming_arc" NAME("") 5 3) )
   ENTITY(8 "process" NAME("increase index")
      ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
   RELATION(14 "consuming_arc" NAME("") 5 8)
   RELATION(15 "producing_arc" NAME("") 8 1)
   RELATION(16 "consuming_arc" NAME("") 1 3) )
RULE(CONDITION("THEN_NODE" RETURN(1 2))
     CONDITION("ELSE_NODE" RETURN(1 2))
     CONDITION("WHEN_BODY_NODE" RETURN(1 2))
     CONDITION("PERFORM_BODY_NODE"AND(NODE-KEY-1 = "STATEMENT_LIST")
      RETURN(1 2))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2))
      CONNECT(3 "dummy" NAME("") 2 1) ) )
RULE(CONDITION("AT_END_NODE" RETURN(1 6))
     CONDITION("NOT_AT_END_NODE" RETURN(1 6))
     CONDITION("ON_SIZE_ERROR_NODE" RETURN(1 6))
     CONDITION("NOT_ON_SIZE_ERROR_NODE" RETURN(1 6))
     CONDITION("ON_OVERFLOW_NODE" RETURN(1 6))
     CONDITION("NOT_ON_OVERFLOW_NODE" RETURN(1 6))
     CONDITION("ON_EXCEPTION_NODE" RETURN(1 6))
     CONDITION("NOT_ON_EXCEPTION_NODE" RETURN(1 6))
     CONDITION("INVALID_KEY_NODE" RETURN(1 6))
     CONDITION("NOT_INVALID_KEY_NODE" RETURN(1 6))
     CONDITION("END_OF_PAGE_NODE" RETURN(1 6))
     CONDITION("NOT_END_OF_PAGE_NODE" RETURN(1 6))
```

```
    CONDITION("DATA_NODE" RETURN(1 6))
    CONDITION("NO_DATA_NODE" RETURN(1 6))
  ENTITY(1 "place" NAME("")
     ATTRIBUTE("initial_no_tokens" NAME("0")))
  ENTITY(2 "branch" NAME(PATH(0):NODE-NAME)
     ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
  ENTITY(3 "place" NAME("")
     ATTRIBUTE("initial_no_tokens" NAME("0")))
  ENTITY(6 "place" NAME("")
     ATTRIBUTE("initial_no_tokens" NAME("0")))
  GROUP(PATH(1)
     PROCESS(PATH(0) RETURN(4 5))
     CONNECT(7 "dummy" NAME("") 5 4) )
  RELATION(8 "consuming_arc" NAME("") 1 2)
  RELATION(9 "condition" NAME("") 2 3 RESERVED
     ATTRIBUTE("condition" NAME("true"))
     ATTRIBUTE("probability" NAME("0.5")))
  RELATION(10 "condition" NAME("") 2 6 RESERVED
     ATTRIBUTE("condition" NAME("false"))
     ATTRIBUTE("probability" NAME("0.5")))
  RELATION(11 "dummy" NAME("") 3 4)
  RELATION(12 "dummy" NAME("") 5 6) )
RULE(CONDITION("PERFORM_BODY_NODE"AND(NODE-KEY-1 = "PROCEDURE") RETURN(1 2))

  PROCESS(PATH(1) RETURN(1 2)) )
RULE(CONDITION("THROUGH_NODE" RETURN(1 3))
  ENTITY(1 "place" NAME("")
     ATTRIBUTE("initial_no_tokens" NAME("0")))
  ENTITY(2 "process" NAME(PATH(1):NODE-VALUE+" through "+PATH(2):NODE-VALUE)
     ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
  ENTITY(3 "place" NAME("")
     ATTRIBUTE("initial_no_tokens" NAME("0")))
  RELATION(4 "consuming_arc" NAME("") 1 2)
  RELATION(5 "producing_arc" NAME("") 2 3) )
RULE(CONDITION("UD_NAME_NODE" RETURN(1 3))
    CONDITION("INTEGER_NODE" RETURN(1 3))
  ENTITY(1 "place" NAME("")
     ATTRIBUTE("initial_no_tokens" NAME("0")))
  ENTITY(2 "process" NAME(PATH(0):NODE-VALUE)
     ATTRIBUTE("node_id" NAME(PATH(0):NODE-ID)))
  ENTITY(3 "place" NAME("")
      ATTRIBUTE("initial_no_tokens" NAME("0")))
   RELATION(4 "consuming_arc" NAME("") 1 2)
   RELATION(5 "producing_arc" NAME("") 2 3) )
```

IOPM2 Generation Rule

The following rules are used to generate IOPM2.

```
RELATION("Consuming_arc" ("Branch" OR "Ready_indicator" OR "Entry" OR "Exit")
                  ("Process" OR "Invoke"))
RELATION("Producing_arc" ("Process" OR "Invoke")
                  ("Branch" OR "Ready_indicator" OR "Entry" OR "Exit"))
RELATION("dummy" ("Branch")
         ("Ready_indicator" OR "Exit"))
RELATION("dummy" ("Ready_indicator")
                  ("Branch" OR "Ready_indicator" OR "Entry" OR "Exit"))
RELATION("dummy" ("Entry")
                  ("Ready_indicator" OR "Entry" OR "Exit"))
MACRO("test" ARGUMENT(1 2 3 4)
      IF ((TYPE OLD-OBJ(1) = "Ready_indicator")
  CONNECT(3 "Producing_arc" NAME("") 2 1)
      ELSE
  CONNECT(4 "Exec_next" NAME("") 2 1)))
RULE(CONDITION("PROGRAM_NODE" RETURN(1 2))
   GRAPH(NAME(PATH(1/1):NODE-VALUE) "IOPM2" 1 2)
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2)) ) )
RULE(CONDITION("DIVISION_NODE" AND (NODE-VALUE = "PROCEDURE") RETURN(1 2))
   VARIABLE(4 NAME(PATH("USING_NODE"):NODE-ELEMENT))
   ENTITY(1 "Entry" NAME("ENTRY") RESERVED
      ATTRIBUTE("Initial_no_tokens" NAME("0")))
   ENTITY(2 "Exit" NAME("EXIT") RESERVED
      ATTRIBUTE("Initial_no_tokens" NAME("0")))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(3 4))
      CONNECT(11 "Exec_next" NAME("") 4 3 ))
   RELATION(12 "Consuming_arc" NAME("") 1 3 )
   RELATION(13 "Producing_arc" NAME("") 4 2 ) )
RULE(CONDITION("SECTION_NODE" RETURN(1 1))
   ENTITY(1 "Process" NAME(NODE-VALUE)
      ATTRIBUTE("node_id" NAME(NODE-ID)))
   GRAPH(NAME(NODE-VALUE) "IOPM2" 4 5)
   ENTITY(2 "Entry" NAME(VARIABLE(4)) RESERVED
      ATTRIBUTE("Initial_no_tokens" NAME("0")))
   ENTITY(3 "Exit" NAME("EXIT") RESERVED
      ATTRIBUTE("Initial_no_tokens" NAME("0")))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(4 5))
      MACRO("test" ARGUMENT(4 5 9 6)))
   RELATION(7 "Consuming_arc" NAME("") 2 4)
   RELATION(8 "Producing_arc" NAME("") 5 3)
   VARIABLE(4 NAME("")))
RULE(CONDITION("PARAGRAPH_NODE" RETURN(2 4))
   ENTITY(2 "Ready_indicator" NAME(NODE-VALUE)
      ATTRIBUTE("Initial_no_tokens" NAME("0")))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(3 4))
      CONNECT(11 "Exec_next" NAME("") 4 3))
   RELATION(13 "Consuming_arc" NAME("") 2 3))
RULE(CONDITION("ACCEPT_ST" RETURN(1 1))
     CONDITION("ALTER_ST" RETURN(1 1))
     CONDITION("CANCEL_ST" RETURN(1 1))
     CONDITION("CLOSE_ST" RETURN(1 1))
     CONDITION("CONTINUE_ST" RETURN(1 1))
     CONDITION("DISABLE_ST" RETURN(1 1))
     CONDITION("DISPLAY_ST" RETURN(1 1))
     CONDITION("ENABLE_ST" RETURN(1 1))
     CONDITION("ENTER_ST" RETURN(1 1))
     CONDITION("EXIT_ST" RETURN(1 1))
     CONDITION("GENERATE_ST" RETURN(1 1))
     CONDITION("INITIALIZE_ST" RETURN(1 1))
     CONDITION("INITIATE_ST" RETURN(1 1))
     CONDITION("INSPECT_ST" RETURN(1 1))
     CONDITION("MERGE_ST" RETURN(1 1))
     CONDITION("MOVE_ST" RETURN(1 1))
     CONDITION("OPEN_ST" RETURN(1 1))
     CONDITION("PURGE_ST" RETURN(1 1))
     CONDITION("RELEASE_ST" RETURN(1 1))
     CONDITION("REPLACE_ST" RETURN(1 1))
     CONDITION("SEND_ST" RETURN(1 1))
     CONDITION("SET_ST" RETURN(1 1))
     CONDITION("SORT_ST" RETURN(1 1))
     CONDITION("STOP_ST" RETURN(1 1))
     CONDITION("SUPPRESS_ST" RETURN(1 1))
     CONDITION("TERMINATE_ST" RETURN(1 1))
   ENTITY(1 "Process" NAME(NODE-ELEMENT)
      ATTRIBUTE("node_id" NAME(NODE-ID))))
RULE(CONDITION("ADD_ST" RETURN(1 31))
     CONDITION("COMPUTE_ST" RETURN(1 31))
     CONDITION("DELETE_ST" RETURN(1 31))
     CONDITION("DIVIDE_ST" RETURN(1 31))
     CONDITION("MULTIPLY_ST" RETURN(1 31))
     CONDITION("READ_ST" RETURN(1 31))
     CONDITION("RECEIVE_ST" RETURN(1 31))
     CONDITION("RETURN_ST" RETURN(1 31))
     CONDITION("REWRITE_ST" RETURN(1 31))
     CONDITION("START_ST" RETURN(1 31))
     CONDITION("STRING_ST" RETURN(1 31))
     CONDITION("SUBTRACT_ST" RETURN(1 31))
     CONDITION("UNSTRING_ST" RETURN(1 31))
     CONDITION("WRITE_ST" RETURN(1 31))
   ENTITY(1 "Process" NAME(NODE-ELEMENT)
      ATTRIBUTE("node_id" NAME(NODE-ID)))
   PROCESS(PATH("ON_SIZE_ERROR_NODE") RETURN(4 5))
   PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE") RETURN(6 7))
   RELATION(43 "Exec_next" NAME("") 1 4 )
   RELATION(44 "Exec_next" NAME("") 5 6 )
   PROCESS(PATH("ON_OVERFLOW_NODE") RETURN(8 9))
   PROCESS(PATH("NOT_ON_OVERFLOW_NODE") RETURN(10 11))
   RELATION(45 "Exec_next" NAME("") 7 8 )
   RELATION(46 "Exec_next" NAME("") 9 11 )
   PROCESS(PATH("ON_EXCEPTION_NODE") RETURN(12 13))
   PROCESS(PATH("NOT_ON_EXCEPTION_NODE") RETURN(14 15))
   RELATION(47 "Exec_next" NAME("") 11 12 )
   RELATION(48 "Exec_next" NAME("") 13 14 )
   PROCESS(PATH("INVALID_KEY_NODE") RETURN(16 17))
   PROCESS(PATH("NOT_INVALID_KEY_NODE") RETURN(18 19))
   RELATION(49 "Exec_next" NAME("") 15 16 )
   RELATION(110 "Exec_next" NAME("") 17 18 )
   PROCESS(PATH("AT_END_NODE") RETURN(20 21))
   PROCESS(PATH("NOT_AT_END_NODE") RETURN(22 23))
   RELATION(111 "Exec_next" NAME("") 19 20 )
   RELATION(112 "Exec_next" NAME("") 21 22 )
   PROCESS(PATH("DATA_NODE") RETURN(24 25))
   PROCESS(PATH("NO_DATA_NODE") RETURN(26 27))
   RELATION(53 "Exec_next" NAME("") 23 24 )
   RELATION(54 "Exec_next" NAME("") 25 26 )
   PROCESS(PATH("END_OF_PAGE_NODE") RETURN(28 29))
   PROCESS(PATH("NOT_END_OF_PAGE_NODE") RETURN(30 31))
   RELATION(55 "Exec_next" NAME("") 27 28 )
   RELATION(56 "Exec_next" NAME("") 29 30 ) )
RULE(CONDITION("CALL_ST" RETURN(1 1))
   ENTITY(1 "Invoke" NAME(NODE-ELEMENT)
    ATTRIBUTE("node_id" NAME(NODE-ID))))
RULE(CONDITION("EVALUATE_ST" RETURN(1 8))
   ENTITY(1 "Process" NAME("EVALUATE-"+NODE-LINE-NO))
   ENTITY(2 "Ready_indicate" NAME("")
    ATTRIBUTE("Initial_no_tokens" NAME("0")))
   GROUP(PATH(2)
      ENTITY(3 "Branch" NAME(PATH(../1):NODE-ELEMENT + "=" +
 PATH(1):NODE-ELEMENT) RESERVED
        ATTRIBUTE("node_id" NAME(NODE-ID))
        ATTRIBUTE("Branch_type" NAME("EVALUATE")))
      PROCESS(PATH("WHEN_BODY_NODE") RETURN(4 5))
      ENTITY(6 "Process" NAME("") RESERVED)
      ENTITY(7 "Process" NAME("") RESERVED)
      RELATION(11 "Consuming_arc" NAME("") 3 6 RESERVED
         ATTRIBUTE("Condition" NAME("T")))
      RELATION(12 "Consuming" NAME("") 3 7 RESERVED
         ATTRIBUTE("Condition" NAME("F")))
      RELATION(13 "Exec_next" NAME("") 6 4)
      RELATION(14 "Exec_next" NAME("") 5 2)
      CONNECT(15 "Producing_arc" NAME("") 7 3 RESERVED))
   RELATION(16 "Producing_are" NAME("") 1 3)
   RELATION(17 "Exec_next" NAME("") 7 2)
   ENTITY(8 "Process" NAME("END-EVALUATE-"+NODE-LINE-NO) RESERVED)
   RELATION(18 "Consuming_arc" NAME("") 2 8 RESERVED))
RULE(CONDITION("GOTO_ST"AND(NODE-KEY-1 = "") RETURN(1))
   ENTITY(1 "Process" NAME(NODE-ELEMENT)
      ATTRIBUTE("node_id" NAME(NODE-ID)))
   RELATION(2 "Exec_next" NAME("") 2 ANY:NAME(PATH(1/1):NODE-VALUE)))
RULE(CONDITION("GOTO_ST"AND(NODE-KEY-1 = "DEPENDING ON") RETURN(1))
   ENTITY(1 "Process" NAME("GOTO DEPENDING ON"))
   ENTITY(2 "Branch" NAME("") RESERVED
    ATTRIBUTE("node_id" NAME(NODE-ID))
    ATTRIBUTE("Branch_type"
      NAME(PATH("DEPENDING_ON_NODE"/1):NODE-VALUE)))
   GROUP(PATH(1/1)
      ENTITY(3 "Process" NAME(""))
      RELATION(11 "Consuming_arc" NAME("") 2 3 RESERVED
         ATTRIBUTE("Condition" NAME(1:1)))
      RELATION(12 "Exec_next" NAME("") 3 ANY:NAME(PATH(1/1):NODE-VALUE)))
   RELATION(13 "Producing_arc" NAME("") 1 2 RESERVED))
RULE(CONDITION("IF_ST" RETURN(1 6))
   ENTITY(1 "Process" NAME("IF"))
   ENTITY(2 "Branch" NAME(PATH(1):NODE-ELEMENT) RESERVED
    ATTRIBUTE("node_id" NAME(NODE-ID))
    ATTRIBUTE("Branch_type" NAME("IF")))
   ENTITY(3 "Process" NAME(""))
   ENTITY(4 "Process" NAME(""))
   ENTITY(5 "Ready_indicator" NAME("")
    ATTRIBUTE("Initial_no_tokens" NAME("0")))
   ENTITY(6 "dummy" NAME(""))
   RELATION(11 "Producing_arc" NAME("") 1 2 RESERVED)
   RELATION(12 "Consuming_arc" NAME("") 2 3 RESERVED
      ATTRIBUTE("Condition" NAME("T")))
   RELATION(13 "Consuming_arc" NAME("") 2 4 RESERVED
      ATTRIBUTE("Condition" NAME("F")))
   PROCESS(PATH("THEN_NODE") RETURN(7 8))
   PROCESS(PATH("ELSE_NODE") RETURN(9 10))
   RELATION(14 "Exec_next" NAME("") 3 7)
   RELATION(15 "Exec_next" NAME("") 4 9)
   RELATION(16 "Producing_arc" NAME("") 8 5 RESERVED)
   RELATION(17 "Producing_arc" NAME("") 10 5 RESERVED)
   RELATION(18 "Consuming_arc" NAME("") 5 6 RESERVED))
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "") RETURN(1 2))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(1 2)))
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "TIMES") RETURN(1 4))
   ENTITY(1 "Process" NAME("PERFORM"))
   ENTITY(2 "Branch" NAME("Loop "+PATH("TIMES_NODE"/1):NODE-VALUE) RESERVED
    ATTRIBUTE("node_id" NAME(PATH("TIME_NODE"):NODE-ID))
    ATTRIBUTE("Branch_type" NAME("PERFORM TIMES")))
   ENTITY(3 "Process" NAME(""))
   ENTITY(4 "Process" NAME(""))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(5 6))
   RELATION(11 "Producing_arc" NAME("") 1 2 RESERVED)
   RELATION(12 "Consuming_arc" NAME("") 2 3 RESERVED
      ATTRIBUTE("Condition" NAME("F")))
   RELATION(13 "Consuming_arc" NAME("") 2 4 RESERVED
      ATTRIBUTE("Condition" NAME("T")))
   RELATION(14 "Exec_next" NAME("") 3 5)
   RELATION(15 "Producing_arc" NAME("") 6 2 RESERVED))
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "BEFORE")
   AND(NODE-KEY-2 <> "VARYING") RETURN(1 3))
   ENTITY(1 "Process" NAME("PERFORM"))
   ENTITY(2 "Branch" NAME(PATH("UNTIL_NODE"/1):NODE-ELEMENT) RESERVED
    ATTRIBUTE("node_id" NAME(PATH("UNTIL_NODE"):NODE-ID))
    ATTRIBUTE("Branch_type" NAME("PERFORM TEST BEFORE")))
   ENTITY(3 "Process" NAME(""))
   ENTITY(4 "Process" NAME(""))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(5 6))
   RELATION(11 "Producing_arc" NAME("") 1 2 RESERVED)
   RELATION(12 "Consuming_arc" NAME("") 2 3 RESERVED
      ATTRIBUTE("Condition" NAME("T")))
   RELATION(13 "Consuming_arc" NAME("") 2 4 RESERVED
      ATTRIBUTE("Condition" NAME("F")))
   RELATION(14 "Exec_next" NAME("") 4 5)
   RELATION(15 "Producing_arc" NAME("") 6 2 RESERVED) )
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "AFTER")
   AND(NODE-KEY-2 <> "VARYING") RETURN(1 4))
   ENTITY(1 "Process" NAME("PERFORM"))
   ENTITY(2 "Ready_indicator" NAME("")
    ATTRIBUTE("Initial_no_tokens" NAME("0")))
   ENTITY(3 "Branch" NAME(PATH("UNTIL_NODE"/1):NODE-ELEMENT) RESERVED
    ATTRIBUTE("node_id" NAME(PATH("UNTIL_NODE"):NODE-ID))
    ATTRIBUTE("Branch_type" NAME("PERFORM TEST AFTER")))
   ENTITY(4 "Process" NAME(""))
   ENTITY(5 "Process" NAME(""))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(6 7))
   RELATION(11 "Producing_arc" NAME("") 1 2 RESERVED)
   RELATION(12 "Consuming_arc" NAME("") 2 6 RESERVED)
   RELATION(13 "Producing_arc" NAME("") 7 3 RESERVED)
   RELATION(14 "Consuming_arc" NAME("") 3 4 RESERVED
      ATTRIBUTE("Condition" NAME("T")))
   RELATION(15 "Consuming_arc" NAME("") 3 5 RESERVED
      ATTRIBUTE("Condition" NAME("F")))
   RELATION(16 "Producing_arc" NAME("") 5 2 RESERVED))
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "BEFORE")
   AND(NODE-KEY-2 = "VARYING") RETURN(3 5))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(1 2))
   GROUP(PATH("VARYING_NODE"/1)
      ENTITY(3 "Process" NAME(PATH("DUM_SOURCE_NODE"/1):NODE-VALUE+" sets to "
       +PATH("FROM_NODE"/1):NODE-VALUE)
         ATTRIBUTE("node_id" NAME(NODE-ID)))
      ENTITY(4 "Branch" NAME(PATH("DUM_SOURCE_NODE"/1):NODE-VALUE+">"
        +PATH("UNTIL_NODE"/1):NODE-VALUE)
       ATTRIBUTE("node_id" NAME(NODE-ID))
       ATTRIBUTE("Branch_type" NAME("PERFORM TEST BEFORE VARYING")))
      ENTITY(5 "Process" NAME(""))
      ENTITY(6 "Process" NAME(""))
      ENTITY(7 "Process" NAME(PATH("DUM_SOURCE_NODE"/1):NODE-VALUE+
       " increases "+PATH("BY_NODE"/1):NODE-VALUE)
         ATTRIBUTE("node_id" NAME(NODE-ID)))
      RELATION(11 "Producing_arc" NAME("") 3 4)
      RELATION(12 "Consuming_arc" NAME("") 4 5 RESERVED
  ATTRIBUTE("condition" NAME("T")))
      RELATION(13 "Consuming_arc" NAME("") 4 7 RESERVED
  ATTRIBUTE("condition" NAME("F")))
      RELATION(14 "Producing_arc" NAME("") 7 4 RESERVED)
      CONNECT(15 "Producing_arc" NAME("") 6 3 RESERVED)
      CONNECT(16 "Exec_next" NAME("") REVERSE 5 7))
   RELATION(17 "Exec_next" NAME("") 6 1 RESERVED)
   RELATION(18 "Exec_next" NAME("") 2 7))
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "AFTER")
   AND(NODE-KEY-2 = "VARYING") RETURN(3 6))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(1 2))
   GROUP(PATH("VARYING_NODE"/1)
      ENTITY(3 "Process" NAME(PATH("DUM_SOURCE_NODE"/1):NODE-VALUE+" sets to "
       +PATH("FROM_NODE"/1):NODE-VALUE) RESERVED
         ATTRIBUTE("node_id" NAME(NODE-ID)))
      ENTITY(4 "Ready_indicator" NAME("") RESERVED
         ATTRIBUTE("Initial_no_tokens" NAME("0")))
      ENTITY(5 "Branch" NAME(PATH("DUM_SOURCE_NODE"/1):NODE-VALUE + ">"
        + PATH("UNTIL_NODE"/1):NODE-VALUE) RESERVED
       ATTRIBUTE("node_id" NAME(NODE-ID))
       ATTRIBUTE("Branch_type" NAME("PERFORM TEST AFTER VARYING")))
      ENTITY(6 "Process" NAME(""))
      ENTITY(7 "Process" NAME(""))
      ENTITY(8 "Process" NAME(PATH("DUM_SOURCE_NODE"/1):NODE-VALUE+
       " increases "+PATH("BY_NODE"/1):NODE-VALUE) RESERVED
         ATTRIBUTE("node_id" NAME(NODE-ID)))
      RELATION(11 "Producing_arc" NAME("") 3 4)
      RELATION(12 "Consuming_arc" NAME("") 5 6 RESERVED
         ATTRIBUTE("Condition" NAME("T")))
      RELATION(13 "Consuming_arc" NAME("") 5 7 RESERVED
         ATTRIBUTE("Condition" NAME("F")))
      RELATION(14 "Exec_next" NAME("") 7 8)
      RELATION(15 "Producing_arc" NAME("") 8 4 RESERVED)
      CONNECT(16 "Consuming_arc" NAME("") 3 4 RESERVED)
      CONNECT(17 "Producing_arc" NAME("") REVERSE 6 5 RESERVED))
   RELATION(18 "dummy" NAME("") 4 1)
   RELATION(19 "dummy" NAME("") 2 5))
RULE(CONDITION("SEARCH_ST" RETURN(1 3))
   ENTITY(1 "Process" NAME("SEARCH-"+NODE-LINE-NO) RESERVED)
   ENTITY(2 "Ready_indicator" NAME(""))
   ENTITY(3 "Process" NAME("END-SEARCH-"+NODE-LINE-NO) RESERVED)
   GROUP(PATH(2)
      ENTITY(4 "Branch" NAME(NODE-ELEMENT) RESERVED
       ATTRIBUTE("node_id" NAME(PATH(..):NODE-ID))
       ATTRIBUTE("Branch_type" NAME("SEARCH")))
      ENTITY(5 "Process" NAME(""))
      ENTITY(6 "Process" NAME(""))
      PROCESS(PATH("WHEN_BODY_NODE") RETURN(7 8))
      RELATION(11 "Consuming_arc" NAME("") 4 5RESERVED
 ATTRIBUTE("condition" NAME("T")))
      RELATION(12 "Consuming_arc" NAME("") 4 6 RESERVED
 ATTRIBUTE("condition" NAME("F")))
      RELATION(13 "Exec_next" NAME("") 5 7)
      RELATION(14 "Producing_arc" NAME("") 8 2 RESERVED)
      CONNECT(15 "Producing_arc" NAME("") 6 4 RESERVED))
   ENTITY(9 "Process" NAME("increase index")
      ATTRIBUTE("node_id" NAME(NODE-ID)))
   RELATION(16 "Consuming_arc" NAME("") 6 9)
   RELATION(17 "Producing_arc" NAME("") 9 4)
   RELATION(18 "Producing_arc" NAME("") 1 4)
   RELATION(19 "Consuming_arc" NAME("") 2 3 RESERVED))
RULE(CONDITION("THEN_NODE" RETURN(1 2))
     CONDITION("ELSE_NODE" RETURN(1 2))
     CONDITION("WHEN_BODY_NODE" RETURN(1 2))
     CONDITION("PERFORM_BODY_NODE"AND(NODE-KEY-1 = "STATEMENT_LIST")
       RETURN(1 2))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2))
      CONNECT(11 "Exec_next" NAME("") 2 1 )))
RULE(CONDITION("AT_END_NODE" RETURN(2 6))
     CONDITION("NOT_AT_END_NODE" RETURN(2 6))
     CONDITION("ON_SIZE_ERROR_NODE" RETURN(2 6))
     CONDITION("NOT_ON_SIZE_ERROR_NODE" RETURN(2 6))
     CONDITION("ON_OVERFLOW_NODE" RETURN(2 6))
     CONDITION("NOT_ON_OVERFLOW_NODE" RETURN(2 6))
     CONDITION("ON_EXCEPTION_NODE" RETURN(2 6))
     CONDITION("NOT_ON_EXCEPTION_NODE" RETURN(2 6))
     CONDITION("INVALID_KEY_NODE" RETURN(2 6))
     CONDITION("NOT_INVALID_KEY_NODE" RETURN(2 6))
     CONDITION("END_OF_PAGE_NODE" RETURN(2 6))
     CONDITION("NOT_END_OF_PAGE_NODE" RETURN(2 6))
     CONDITION("DATA_NODE" RETURN(2 6))
     CONDITION("NO_DATA_NODE" RETURN(2 6))
   ENTITY(2 "Branch" NAME(NODE-NAME) RESERVED
      ATTRIBUTE("node_id" NAME(NODE-ID))
      ATTRIBUTE("Branch_type" NAME(NODE-NAME)))
   ENTITY(3 "Process" NAME(""))
   ENTITY(4 "Process" NAME(""))
   ENTITY(5 "Ready_indicator" NAME("")
      ATTRIBUTE("Initial_no_tokens" NAME("0")))
   ENTITY(6 "dummy" NAME(""))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(7 8))
      CONNECT(11 "Exec_next" NAME("") 8 7))
   RELATION(13 "Consuming_arc" NAME("") 2 3 RESERVED
      ATTRIBUTE("Condition" NAME("T")))
   RELATION(14 "Consuming_arc" NAME("") 2 4 RESERVED
      ATTRIBUTE("Condition" NAME("F")))
   RELATION(15 "Exec_next" NAME("") 3 7)
   RELATION(16 "Producing_arc" NAME("") 8 5)
   RELATION(17 "Producing_arc" NAME("") 4 5)
   RELATION(18 "Consuming_arc" NAME("") 5 6))
RULE(CONDITION("PERFORM_BODY_NODE"AND(NODE-KEY-1 = "PROCEDURE") RETURN(1 2))
   PROCESS(PATH(1) RETURN(1 2)) )
RULE(CONDITION("THROUGH_NODE" RETURN(1 1))
   ENTITY(1 "Invoke" NAME(PATH(1):NODE-VALUE+" through "+PATH(2):NODE-VALUE)
      ATTRIBUTE("node_id" NAME(NODE-ID))))
RULE(CONDITION("UD_NAME_NODE" RETURN(1 1))
     CONDITION("INTEGER_NODE" RETURN(1 1))
   ENTITY(1 "Invoke" NAME(NODE-VALUE)
      ATTRIBUTE("node_id" NAME(NODE-ID))))
```

UIM Generation Rule

The following rules are used to generate UIM.

```
RULE(CONDITION("PROGRAM_NODE" RETURN(1 2))
   GRAPH(NAME(PATH(1/1):NODE-VALUE) "UIM" 1 2)
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2)) ) )
RULE(CONDITION("DIVISION_NODE" AND (NODE-VALUE = "PROCEDURE") RETURN(1 2))
   ENTITY(1 "state" NAME("Start") RESERVED)
   ENTITY(2 "state" NAME("End") RESERVED)
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(3 4))
      CONNECT(11 "becomes" NAME("") 4 3) )
   RELATION(12 "becomes" NAME("") 1 3)
   RELATION(13 "becomes" NAME("") 4 2) )
RULE(CONDITION("SECTION_NODE" RETURN(1 1))
   ENTITY(1 "state" NAME(PATH(0):NODE-VALUE))
   GRAPH(NAME(PATH(0):NODE-VALUE) "UIM" 2 3)
   ENTITY(2 "state" NAME(PATH(0):NODE-VALUE))
   ENTITY(3 "state" NAME("Exit"))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(4 5))
      CONNECT(11 "becomes" NAME("") 5 4) )
   RELATION(12 "becomes" NAME("") 2 4)
   RELATION(13 "becomes" NAME("") 5 3) )
RULE(CONDITION("PARAGRAPH_NODE" RETURN(1 4))
   ENTITY(1 "state" NAME(PATH(0):NODE-VALUE))
   ENTITY(4 "dummy" NAME("Exit"))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(2 3))
      CONNECT(11 "becomes" NAME("") 3 2) )
   RELATION(12 "becomes" NAME("") 1 2)
   RELATION(13 "becomes" NAME("") 3 4) )
RULE(CONDITION("EVALUATE_ST" RETURN(1 4))
   ENTITY(1 "state" NAME(PATH(1/1):NODE-ELEMENT))
   ENTITY(4 "dummy" NAME(""))
   GROUP(PATH(2)
      PROCESS(PATH("WHEN_BODY_NODE") RETURN(2 3))
      RELATION(11 "becomes" NAME("") 1 2
 ATTRIBUTE("event" NAME(PATH(1):NODE-ELEMENT)))
      RELATION(12 "becomes" NAME("") 3 4) ) )
RULE(CONDITION("GOTO_ST"AND(NODE-KEY-1 = "") RETURN(1 -1))
   ENTITY(1 "state" NAME(PATH(0):NODE-ELEMENT))
   RELATION(11 "becomes" NAME("") 1 ANY:NAME(PATH(1/1):NODE-VALUE)) )
RULE(CONDITION("GOTO_ST"AND(NODE-KEY-1 = "DEPENDING ON") RETURN(1 -1))
   ENTITY(1 "state" NAME("GO TO DEPENDING ON "+PATH(2/1):NODE-VALUE))
   GROUP(PATH(1/1)
      RELATION(11 "becomes" NAME("") 1 ANY:NAME(PATH(1/1):NODE-VALUE) RESERVED
 ATTRIBUTE("event" NAME(1:1))) ) )
RULE(CONDITION("IF_ST" RETURN(1 6))
   ENTITY(1 "state" NAME(PATH(0):NODE-VALUE))
   ENTITY(6 "dummy" NAME(""))
   PROCESS(PATH("THEN_NODE") RETURN(2 3))
   PROCESS(PATH("ELSE_NODE") RETURN(4 5))
   RELATION(11 "becomes" NAME("") 1 2 RESERVED
      ATTRIBUTE("event" NAME("true")))
   RELATION(12 "becomes" NAME("") 1 4 RESERVED
      ATTRIBUTE("event" NAME("false")))
   RELATION(13 "becomes" NAME("") 3 6)
   RELATION(14 "becomes" NAME("") 5 6) )
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "") RETURN(1 2))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(1 2)) )
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "TIMES") RETURN(1 4))
   ENTITY(1 "state" NAME("Loop "+PATH("TIMES_NODE"/1):NODE-VALUE))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(2 3))
   ENTITY(4 "dummy" NAME(""))
   RELATION(11 "becomes" NAME("") 1 4 RESERVED
      ATTRIBUTE("event" NAME("true")))
   RELATION(12 "becomes" NAME("") 1 2 RESERVED
      ATTRIBUTE("event" NAME("false")))
   RELATION(13 "becomes" NAME("") 3 1) )
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "BEFORE")
   AND(NODE-KEY-2 <> "VARYING") RETURN(1 4))
     CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "AFTER")
   AND(NODE-KEY-2 <> "VARYING") RETURN(2 4))
   ENTITY(1 "state" NAME(PATH("UNTIL_NODE"):NODE-ELEMENT))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(2 3))
   ENTITY(4 "dummy" NAME(""))
   RELATION(11 "becomes" NAME("") 1 4 RESERVED
      ATTRIBUTE("event" NAME("true")))
   RELATION(12 "becomes" NAME("") 1 2 RESERVED
      ATTRIBUTE("event" NAME("false")))
   RELATION(13 "becomes" NAME("") 3 1) )
RULE(CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "BEFORE")
   AND(NODE-KEY-2 = "VARYING") RETURN(1 2))
     CONDITION("PERFORM_ST"AND(NODE-KEY-1 = "AFTER")
   AND(NODE-KEY-2 = "VARYING") RETURN(1 2))
   PROCESS(PATH("PERFORM_BODY_NODE") RETURN(1 2)) )
RULE(CONDITION("ADD_ST" RETURN(1 28))
     CONDITION("CALL_ST" RETURN(1 28))
     CONDITION("COMPUTE_ST" RETURN(1 28))
     CONDITION("DELETE_ST" RETURN(1 28))
     CONDITION("DIVIDE_ST" RETURN(1 28))
     CONDITION("MULTIPLY_ST" RETURN(1 28))
     CONDITION("READ_ST" RETURN(1 28))
     CONDITION("RECEIVE_ST" RETURN(1 28))
     CONDITION("RETURN_ST" RETURN(1 28))
     CONDITION("REWRITE_ST" RETURN(1 28))
     CONDITION("START_ST" RETURN(1 28))
     CONDITION("STRING_ST" RETURN(1 28))
     CONDITION("SUBTRACT_ST" RETURN(1 28))
     CONDITION("UNSTRING_ST" RETURN(1 28))
     CONDITION("WRITE_ST" RETURN(1 28))
   PROCESS(PATH("INVALID_KEY_NODE") RETURN(1 2))
   PROCESS(PATH("NOT_INVALID_KEY_NODE") RETURN(3 4))
   RELATION(41 "becomes" NAME("") 1 2)
   RELATION(42 "becomes" NAME("") 2 3)
   PROCESS(PATH("AT_END_NODE") RETURN(5 6))
   PROCESS(PATH("NOT_AT_END_NODE") RETURN(7 8))
   RELATION(43 "becomes" NAME("") 4 5)
   RELATION(44 "becomes" NAME("") 6 7)
   PROCESS(PATH("DATA_NODE") RETURN(9 10))
   PROCESS(PATH("NO_DATA_NODE") RETURN(11 12))
   RELATION(45 "becomes" NAME("") 8 9)
   RELATION(46 "becomes" NAME("") 10 11)
   PROCESS(PATH("ON_OVERFLOW_NODE") RETURN(13 14))
   PROCESS(PATH("NOT_ON_OVERFLOW_NODE") RETURN(15 16))
   RELATION(47 "becomes" NAME("") 12 13)
   RELATION(48 "becomes" NAME("") 14 15)
   PROCESS(PATH("END_OF_PAGE_NODE") RETURN(17 18))
   PROCESS(PATH("NOT_END_OF_PAGE_NODE") RETURN(19 20))
   RELATION(49 "becomes" NAME("") 16 17)
   RELATION(50 "becomes" NAME("") 18 19)
   PROCESS(PATH("ON_SIZE_ERROR_NODE") RETURN(21 22))
   PROCESS(PATH("NOT_ON_SIZE_ERROR_NODE") RETURN(23 24))
   RELATION(51 "becomes" NAME("") 20 21)
   RELATION(52 "becomes" NAME("") 22 23)
   PROCESS(PATH("ON_EXCEPTION_NODE") RETURN(25 26))
   PROCESS(PATH("NOT_ON_EXCEPTION_NODE") RETURN(27 28))
   RELATION(53 "becomes" NAME("") 24 25)
   RELATION(54 "becomes" NAME("") 26 27) )
RULE(CONDITION("SEARCH_ST" RETURN(1 2))
   ENTITY(1 "state" NAME("Search_st"))
   ENTITY(2 "dummy" NAME(""))
   GROUP(PATH(2)
      PROCESS(PATH("WHEN_BODY_NODE") RETURN(3 4))
      RELATION(11 "becomes" NAME("") 1 3
 ATTRIBUTE("event" NAME(PATH(0):NODE-ELEMENT)))
      RELATION(12 "becomes" NAME("") 4 2) ) )
RULE(CONDITION("THEN_NODE" RETURN(1 2))
     CONDITION("ELSE_NODE" RETURN(1 2))
     CONDITION("WHEN_BODY_NODE" RETURN(1 2))
     CONDITION("PERFORM_BODY_NODE"AND(NODE-KEY-1 = "STATEMENT_LIST") RETURN(1
  2))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(1 2))
      CONNECT(11 "becomes" NAME("") 2 1) ) )
RULE(CONDITION("AT_END_NODE" RETURN(1 4))
     CONDITION("NOT_AT_END_NODE" RETURN(1 4))
     CONDITION("ON_SIZE_ERROR_NODE" RETURN(1 4))
     CONDITION("NOT_ON_SIZE_ERROR_NODE" RETURN(1 4))
     CONDITION("ON_OVERFLOW_NODE" RETURN(1 4))
     CONDITION("NOT_ON_OVERFLOW_NODE" RETURN(1 4))
     CONDITION("ON_EXCEPTION_NODE" RETURN(1 4))
     CONDITION("NOT_ON_EXCEPTION_NODE" RETURN(1 4))
     CONDITION("INVALID_KEY_NODE" RETURN(1 4))
     CONDITION("NOT_INVALID_KEY_NODE" RETURN(1 4))
     CONDITION("END_OF_PAGE_NODE" RETURN(1 4))
     CONDITION("NOT_END_OF_PAGE_NODE" RETURN(1 4))
     CONDITION("DATA_NODE" RETURN(1 4))
     CONDITION("NO_DATA_NODE" RETURN(1 4))
   ENTITY(1 "state" NAME(PATH(0):NODE-NAME))
   ENTITY(4 "dummy" NAME(""))
   GROUP(PATH(1)
      PROCESS(PATH(0) RETURN(2 3))
      CONNECT(11 "becomes" NAME("") 3 2) )
   RELATION(12 "becomes" NAME("") 1 2 RESERVED
      ATTRIBUTE("event" NAME("true")))
   RELATION(13 "becomes" NAME("") 1 4 RESERVED
      ATTRIBUTE("event" NAME("false")))
   RELATION(14 "becomes" NAME("") 3 4) )
RULE(CONDITION("PERFORM_BODY_NODE"AND(NODE-KEY-1 = "PROCEDURE") RETURN(1 2))

   PROCESS(PATH(1) RETURN(1 2)) )
RULE(CONDITION("THROUGH_NODE" RETURN(1 1))
   ENTITY(1 "state" NAME(PATH(1):NODE-VALUE+" through "+PATH(2):NODE-VALUE))
 )
RULE(CONDITION("UD_NAME_NODE" RETURN(1 1))
     CONDITION("INTEGER_NODE" RETURN(1 1))
   ENTITY(1 "state" NAME(PATH(0):NODE-VALUE)) )
```

Appendix E

Information Used for Linkage Generation

E.1 Linkages Defined Among RMA Models

This part gives the definitions of the linkages between RMA models. There is a table for each pair of RMA models. Each table consists of five columns:

- The source object represents a particular type of object in the source model.
- The destination object represents a particular type of object in the destination model.
- The type of linkage represents what kind of linkage holds between the two objects. It takes following values:

  equate means that two objects being linked represent the same information.

  refer means that the sink object is referred by the source object.

  define means that the sink object is defined by the source object.

  relate means that two objects are related in some fashion.

  decomposite means that the sink object is the decomposition of the source object.

- The link criteria indicates under what conditions the two objects should be linked. It takes following values:

  same name means that the two objects should have the same value in their name fields.

  used in the statement means that the sink object is referred in the statement represented by the source object.

defined in the statement means that the statement represented by the source object assigns a value to the sink object.

used in the block means that the sink object is referred in the block represented by the source object.

defined in the block means that the block represented by the source object assigns a value to the sink object.

the state is an entry state means that a **state** entity is an entry point of the graph.

the state is an exit state means that a **state** entity is an exit point of the graph.

- The direction specifies the connectivity of the linkage. It takes following values:

  bi means the link is bi-direction.

  uni means the link is uni-direction.

There are ten types of RMA models generated by Proud – IOPM, IOPM2, EOPM, FM, FSM, CPM, DM, DM2, UIM, and PSTM. DD is also generated by Proud and linked with the models. Since DD is not a graph and each entry in DD has a corresponding entity in DM or DM2, only the linkages between DD and DM and between DD and DM2 are defined. IOPM2 is the updated version of IOPM and DM2 is the updated version of DM, so no linkages are defined between IOPM2 and IOPM, DM2 and DM, IOPM2 and DM, and IOPM and DM2.

Linkages between DM and DD

There is one type of linkage defined between DM and DD. An **equate** linkage indicates the two objects represent the same data item. The DM and DD linkage is defined in Table E.1.

Table E.1: Linkage between DM and DD

Source	*Destination*	*Type*	*Link Criteria*	*direction*
any entity	any entry	equate	same name	bi

Linkages between DM2 and DD

DM2 is a modified version of DM. The reason for keeping DM is for compatibility of some old data which were created before DM2 was defined. There is one type of linkage defined between DM2 and DD. An **equate** linkage indicates the two objects represent the same data item. The DM2 and DD linkage is defined in Table E.2.

Table E.2: Linkage between DM2 and DD

Source	*Destination*	*Type*	*Link Criteria*	*direction*
any entity	any entry	equate	same name	bi

Linkages between CPM and DM

There is one type of linkage defined between CPM and DM. An **equate** linkage indicates the two objects represent the same data item. The CPM and DM linkages are defined in Table E.3.

Table E.3: Linkage between CPM and DM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
internal_data	date_item	equate	same name	bi
internal_data	record	equate	same name	bi
internal_data	internal_data	equate	same name	bi
database_system	data_bank	equate	same name	bi
external_data	file	equate	same name	bi
external_data	report	equate	same name	bi
external_data	form	equate	same name	bi
external_data	graph	equate	same name	bi
external_data	monitor	equate	same name	bi
external_data_element	record	equate	same name	bi
external_data_element	data_item	equate	same name	bi
datastore	data_bank	equate	same name	bi
datastore	folder	equate	same name	bi

Linkages between FM and DM

There is one type of linkage defined between FM and DM. An **equate** linkage indicates the two objects represent the same data item. The FM and DM linkages are defined in Table E.4.

Linkages between FSM and DM

There is one type of linkage defined between FSM and DM. An **equate** linkage indicates the two objects represent the same data item. The FSM and DM linkages are defined in Table E.5.

Linkages between IOPM and DM

There are two types of linkage defined between IOPM and DM. A **refer** linkage indicates the data item is used (referred) by the **process** (statement). A **define** define linkage indicates a value is assigned to the data item by the **process** (statement). The IOPM and DM linkages are defined in Table E.6.

Linkages between EOPM and DM

There are two types of linkage defined between EOPM and DM. A **refer** linkage indicates the data item is used (referred) by the **state** (statement). A **define** define linkage indicates a value is assigned to the data item by the **state** (statement). The EOPM and DM linkages are defined in Table E.7.

Linkages between UIM and DM

There are two types of linkage defined between UIM and DM. A **refer** linkage indicates the data item is used (referred) by the **state** (statement). A **define** linkage indicates a value is assigned to the data item by the **state** (statement). The UIM and DM linkages are defined in Table E.8.

Table E.4: Linkage between FM and DM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
input_data_set	file	equate	same name	bi
input_data_set	report	equate	same name	bi
input_data_set	form	equate	same name	bi
input_data_set	graph	equate	same name	bi
input_data_set	monitor	equate	same name	bi
input_data_set	internal_data	equate	same name	bi
input_data_set	record	equate	same name	bi
input_data_set	data_item	equate	same name	bi
output_data_set	file	equate	same name	bi
output_data_set	report	equate	same name	bi
output_data_set	form	equate	same name	bi
output_data_set	graph	equate	same name	bi
output_data_set	monitor	equate	same name	bi
output_data_set	internal_data	equate	same name	bi
output_data_set	record	equate	same name	bi
output_data_set	data_item	equate	same name	bi
sub_input_data_set	file	equate	same name	bi
sub_input_data_set	report	equate	same name	bi
sub_input_data_set	form	equate	same name	bi
sub_input_data_set	graph	equate	same name	bi
sub_input_data_set	monitor	equate	same name	bi
sub_input_data_set	internal_data	equate	same name	bi
sub_input_data_set	record	equate	same name	bi
sub_input_data_set	data_item	equate	same name	bi
sub_output_data_set	file	equate	same name	bi
sub_output_data_set	report	equate	same name	bi
sub_output_data_set	form	equate	same name	bi
sub_output_data_set	graph	equate	same name	bi
sub_output_data_set	monitor	equate	same name	bi
sub_output_data_set	internal_data	equate	same name	bi
sub_output_data_set	record	equate	same name	bi
sub_output_data_set	data_item	equate	same name	bi
internal_data	record	equate	same name	bi
internal_data	data_item	equate	same name	bi

Table E.5: Linkage between FSM and DM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
object	file	equate	same name	bi
object	report	equate	same name	bi
object	form	equate	same name	bi
object	graph	equate	same name	bi
object	monitor	equate	same name	bi

Table E.6: Linkage between IOPM and DM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
process	file	refer	used in the statement	uni
process	report	refer	used in the statement	uni
process	form	refer	used in the statement	uni
process	graph	refer	used in the statement	uni
process	monitor	refer	used in the statement	uni
process	internal_data	refer	used in the statement	uni
process	record	refer	used in the statement	uni
process	data_item	refer	used in the statement	uni
branch	internal_data	refer	used in the statement	uni
branch	record	refer	used in the statement	uni
branch	data_item	refer	used in the statement	uni
process	file	define	defined in the statement	uni
process	report	define	defined in the statement	uni
process	form	define	defined in the statement	uni
process	graph	define	defined in the statement	uni
process	monitor	define	defined in the statement	uni
process	internal_data	define	defined in the statement	uni
process	record	define	defined in the statement	uni
process	data_item	define	defined in the statement	uni

Linkages between PSTM and DM

There are two types of linkage defined between PSTM and DM. A **refer** linkage indicates the data item is used (referred) by the **block** (procedure or function). A **define** linkage indicates a value is assigned to the data item by the **block** (procedure or function). The PSTM and DM linkages are defined in Table E.9.

Table E.7: Linkage between EOPM and DM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
state	file	refer	used in the statement	uni
state	report	refer	used in the statement	uni
state	form	refer	used in the statement	uni
state	graph	refer	used in the statement	uni
state	monitor	refer	used in the statement	uni
state	file	define	defined in the statement	uni
state	report	define	defined in the statement	uni
state	form	define	defined in the statement	uni
state	graph	define	defined in the statement	uni
state	monitor	define	defined in the statement	uni

Table E.8: Linkage between UIM and DM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
state	file	refer	used in the statement	uni
state	report	refer	used in the statement	uni
state	form	refer	used in the statement	uni
state	graph	refer	used in the statement	uni
state	monitor	refer	used in the statement	uni
state	file	define	defined in the statement	uni
state	report	define	defined in the statement	uni
state	form	define	defined in the statement	uni
state	graph	define	defined in the statement	uni
state	monitor	define	defined in the statement	uni

Linkages between CPM and DM2

There is one type of linkage defined between CPM and DM2. An **equate** linkage indicates the two objects represent the same data item. The CPM and DM2 linkages are defined in Table E.10.

Table E.9: Linkage between PSTM and DM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
block	file	refer	used in the block	uni
block	report	refer	used in the block	uni
block	form	refer	used in the block	uni
block	graph	refer	used in the block	uni
block	monitor	refer	used in the block	uni
block	internal_data	refer	used in the block	uni
block	record	refer	used in the block	uni
block	data_item	refer	used in the block	uni
block	file	define	defined in the block	uni
block	report	define	defined in the block	uni
block	form	define	defined in the block	uni
block	graph	define	defined in the block	uni
block	monitor	define	defined in the block	uni
block	internal_data	define	defined in the block	uni
block	record	define	defined in the block	uni
block	data_item	define	defined in the block	uni

Table E.10: Linkage between CPM and DM2

Source	*Destination*	*Type*	*Link Criteria*	*direction*
internal_data	date_item	equate	same name	bi
internal_data	record	equate	same name	bi
database_system	file	equate	same name	bi
external_data	file	equate	same name	bi
external_data_element	record	equate	same name	bi
external_data_element	data_item	equate	same name	bi

Linkages between FM and DM2

There is one type of linkage defined between FM and DM2. An **equate** linkage indicates the two objects represent the same data item. The FM and DM2 linkages are defined in Table E.11.

Table E.11: Linkage between FM and DM2

Source	*Destination*	*Type*	*Link Criteria*	*direction*
input_data_set	file	equate	same name	bi
input_data_set	record	equate	same name	bi
input_data_set	data_item	equate	same name	bi
output_data_set	file	equate	same name	bi
output_data_set	record	equate	same name	bi
output_data_set	data_item	equate	same name	bi
sub_input_data_set	file	equate	same name	bi
sub_input_data_set	record	equate	same name	bi
sub_input_data_set	data_item	equate	same name	bi
sub_output_data_set	file	equate	same name	bi
sub_output_data_set	record	equate	same name	bi
sub_output_data_set	data_item	equate	same name	bi
internal_data	record	equate	same name	bi
internal_data	data_item	equate	same name	bi

Linkages between FSM and DM2

There is one type of linkage defined between FSM and DM2. An **equate** linkage indicates the two objects represent the same file. The FSM and DM2 linkage is defined in Table E.12.

Table E.12: Linkage between FSM and DM2

Source	*Destination*	*Type*	*Link Criteria*	*direction*
object	file	equate	same name	bi

Linkages between IOPM2 and DM2

There are two types of linkage defined between IOPM and DM2. A **refer** linkage indicates the data item is used (referred) by the **Process** (statement). A **define** linkage indicates a value is assigned to the data item by the **Process** (statement). The IOPM2 and DM2 linkages are defined in Table E.13.

Table E.13: Linkage between IOPM2 and DM2

Source	*Destination*	*Type*	*Link Criteria*	*direction*
Process	file	refer	used in the statement	uni
Process	record	refer	used in the statement	uni
Process	data_item	refer	used in the statement	uni
Invoke	record	refer	used in the statement	uni
Invoke	data_item	refer	used in the statement	uni
Branch	record	refer	used in the statement	uni
Branch	data_item	refer	used in the statement	uni
Process	file	define	defined in the statement	uni
Process	record	define	defined in the statement	uni
Process	data_item	define	defined in the statement	uni

Linkages between EOPM and DM2

There are two types of linkage defined between EOPM and DM2. A **refer** linkage indicates the **state** (statement) reads the file, A **define** linkage indicates the **state** (statement) writes the file. The EOPM and DM2 linkages are defined in Table E.14.

Table E.14: Linkage between EOPM and DM2

Source	*Destination*	*Type*	*Link Criteria*	*direction*
state	file	refer	used in the statement	uni
state	file	define	defined in the statement	uni

Linkages between UIM and DM2

There are two types of linkage defined between UIM and DM2. A **refer** linkage indicates the **state** (statement) reads the file. A **define** linkage indicates the **state** (statement) writes the file. The UIM and DM2 linkages are defined in Table E.15.

Linkages between PSTM and DM2

There are two types of linkage defined between PSTM and DM2. A **refer** linkage indicates the data item is used (referred) by the **block** (procedure or function). A

Table E.15: Linkage between UIM and DM2

Source	*Destination*	*Type*	*Link Criteria*	*direction*
state	file	refer	used in the statement	uni
state	file	define	defined in the statement	uni

define linkage indicates a value is assigned to the data item by the **block** (procedure or function). The PSTM and DM2 linkages are defined in Table E.16.

Table E.16: Linkage between PSTM and DM2

Source	*Destination*	*Type*	*Link Criteria*	*direction*
block	file	refer	used in the block	uni
block	record	refer	used in the block	uni
block	data_item	refer	used in the block	uni
block	file	define	defined in the block	uni
block	record	define	defined in the block	uni
block	data_item	define	defined in the block	uni

Linkages between CPM and FM

There are two types of linkage defined between CPM and FM. An **equate** linkage indicates the two objects represent the same data item. A **relate** linkage indicates the **task** in CPM represents the same piece of code as the **function** in FM. The CPM and FM linkages are defined in Table E.17.

Linkages between CPM and FSM

There are two types of linkage defined between CPM and FSM. An **equate** linkage indicates the two objects represent the same data item. A **relate** linkage indicates the **task** in CPM represents the same piece of code as the **actor** in FSM. The CPM and FSM linkages are defined in Table E.18.

Table E.17: Linkage between CPM and FM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
task	function	relate	same name	bi
task	sub_function	relate	same name	bi
database_system	input_data_set	equate	same name	bi
database_system	output_data_set	equate	same name	bi
external_data	input_data_set	equate	same name	bi
external_data	output_data_set	equate	same name	bi
external_data_element	sub_input_data	equate	same name	bi
external_data_element	sub_output_data	equate	same name	bi
internal_data	input_data_set	equate	same name	bi
internal_data	output_data_set	equate	same name	bi
internal_data	sub_input_data	equate	same name	bi
internal_data	sub_output_data	equate	same name	bi
internal_data	internal_data	equate	same name	bi
datastore	input_data_set	equate	same name	bi
datastore	output_data_set	equate	same name	bi

Table E.18: Linkage between CPM and FSM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
task	actor	relate	same name	bi
database_system	object	equate	same name	bi
external_data	object	equate	same name	bi
internal_data	object	equate	same name	bi
datastore	object	equate	same name	bi

Linkages between CPM and IOPM

There is one type of linkage defined between CPM and IOPM. A **relate** linkage indicates the **task** in CPM represents the same piece of code as the IOPM graph. The CPM and IOPM linkage is defined in Table E.19.

Table E.19: Linkage between CPM and IOPM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
task	graph	relate	same name	bi

Linkages between CPM and IOPM2

There is one type of linkage defined between CPM and IOPM2. A **relate** linkage indicates the **task** in CPM represents the same piece of code as the IOPM2 graph. The CPM and IOPM2 linkage is defined in Table E.20.

Table E.20: Linkage between CPM and IOPM2

Source	*Destination*	*Type*	*Link Criteria*	*direction*
task	graph	relate	same name	bi

Linkages between CPM and EOPM

There is one type of linkage defined between CPM and EOPM. A **relate** linkage indicates the **task** in CPM represents the same piece of code as the EOPM graph. The CPM and EOPM linkage is defined in Table E.21.

Table E.21: Linkage between CPM and EOPM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
task	graph	relate	same name	bi

Linkages between CPM and UIM

There is one type of linkage defined between CPM and UIM. An **equate** linkage indicates the **task** in CPM represents the same piece of code as the UIM graph. The CPM and UIM linkage is defined in Table E.22.

Table E.22: Linkage between CPM and UIM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
task	graph	relate	same name	bi

Linkages between CPM and PSTM

The is one type of linkage defined between CPM and PSTM. An **equate** linkage indicates the **task** or the CPM graph represents the same piece of code as the **module** in PSTM. The CPM and PSTM linkages are defined in Table E.23.

Table E.23: Linkage between CPM and PSTM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
task	module	equate	same name	bi
graph	module	equate	same name	bi

Linkages between CPMs

There are two types of linkage defined between CPMs. An **equate** linkage indicates the two objects represent the same information. A **decomposite** linkage indicates the graph is the decomposition of the **task** in the source graph. The linkages between CPMs are defined in Table E.24.

Table E.24: Linkage between CPMs

Source	*Destination*	*Type*	*Link Criteria*	*direction*
task	graph	decomposite	same name	uni
section	section	equate	same name	bi
database_system	database_system	equate	same name	bi
external_data	external_data	equate	same name	bi
external_data_element	external_data_element	equate	same name	bi
internal_data	internal_data	equate	same name	bi
datastore	datastore	equate	same name	bi

Linkages between FM and FSM

There are two types of linkage defined between FM and FSM. An **equate** linkage indicates the two objects represent the same data item. A **relate** linkage indicates that the **function** or the **sub_function** represents the same piece of code as the **actor** in FSM or a FSM graph. The FM and FSM linkages are defined in Table E.25.

Table E.25: Linkage between FM and FSM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
function	actor	relate	same name	bi
sub_function	actor	relate	same name	bi
function	graph	relate	same name	bi
input_data_set	object	equate	same name	bi
output_data_set	object	equate	same name	bi

Linkages between FM and IOPM

There is one type of linkage defined between FM and IOPM. A **relate** linkage indicates the **function** in FM represents the same piece of code as the **process** in IOPM or the IOPM graph. The FM and IOPM linkages are defined in Table E.26.

Table E.26: Linkage between FM and IOPM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
function	process	relate	same name	bi
function	graph	relate	same name	bi

Linkages between FM and IOPM2

There is one type of linkage defined between FM and IOPM2. A **relate** linkage indicates the **function** in FM represents the same piece of code as the **Invoke** in IOPM2 or the IOPM2 graph. The FM and IOPM2 linkages are defined in Table E.27.

Table E.27: Linkage between FM and IOPM2

Source	*Destination*	*Type*	*Link Criteria*	*direction*
function	Invoke	relate	same name	bi
function	graph	relate	same name	bi

Linkages between FM and EOPM

There is one type of linkage defined between FM and EOPM. A **relate** linkage indicates the **function** in FM represents the same piece of code as the **state** in EOPM or the EOPM graph. The FM and EOPM linkages are defined in Table E.28.

Table E.28: Linkage between FM and EOPM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
function	state	relate	same name	bi
function	graph	relate	same name	bi

Linkages between FM and UIM

There is one type of linkage defined between FM and UIM. A **relate** linkage indicates the **function** in FM represents the same piece of code as the **state** in UIM or the UIM graph. The FM and UIM linkages are defined in Table E.29.

Table E.29: Linkage between FM and UIM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
function	state	relate	same name	bi
function	graph	relate	same name	bi

Linkages between FM and PSTM

There is one type of linkage defined between FM and PSTM. A **relate** linkage indicates the **function** in FM represents the same piece of code as the **module** in UIM. The FM and PSTM linkages are defined in Table E.30.

Table E.30: Linkage between FM and PSTM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
function	module	relate	same name	bi
sub_function	module	relate	same name	bi

Linkages between FMs

There are two types of linkage defined between FMs. An **equate** linkage indicates the two objects represent the same information. A **decomposite** linkage indicates the graph is the decomposition of the **function** in the source graph. The linkages between FMs are defined in Table E.31.

Table E.31: Linkage between FMs

Source	*Destination*	*Type*	*Link Criteria*	*direction*
function	graph	decomposite	same name	uni
sub_function	function	equate	same name	bi
input_data_set	input_data_set	equate	same name	bi
output_data_set	output_data_set	equate	same name	bi
sub_input_data	input_data_set	equate	same name	bi
sub_input_data	sub_input_data	equate	same name	bi
sub_output_data	output_data_set	equate	same name	bi
sub_output_data	sub_output_data	equate	same name	bi

Linkages between FSM and IOPM

There is one type of linkage defined between FSM and IOPM. A **relate** linkage indicates the **actor** in FSM represents the same piece of code as the **process** or **branch** in IOPM or the IOPM graph. The FSM and IOPM linkages are defined in Table E.32.

Table E.32: Linkage between FSM and IOPM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
action	process	relate	same name	bi
action	branch	relate	same name	bi
actor	graph	relate	same name	bi

Linkages between FSM and IOPM2

There is one linkages defined between FSM and IOPM2. A **relate** linkage indicates the **actor** in FSM represents the same piece of code as **Process**, **Invoke**, or **Branch** in IOPM2, or the IOPM2 graph. The FSM and IOPM2 linkages are defined in Table E.33.

Table E.33: Linkage between FSM and IOPM2

Source	*Destination*	*Type*	*Link Criteria*	*direction*
action	Process	relate	same name	bi
action	Invoke	relate	same name	bi
action	Branch	relate	same name	bi
actor	graph	relate	same name	bi

Linkages between FSM and EOPM

There is one type of linkage defined between FSM and EOPM. A **relate** linkage indicates the **actor** in FSM represents the same piece of code as the **state** in EOPM. The FSM and EOPM linkages are defined in Table E.34.

Table E.34: Linkage between FSM and EOPM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
action	state	relate	same name	bi
actor	graph	relate	same name	bi

Linkages between FSM and UIM

There is one type of linkage defined between FSM and UIM. An **relate** linkage indicates the **actor** in FSM represents the same piece of code as the **state** in UIM. The FSM and UIM linkage is defined in Table E.35.

Table E.35: Linkage between FSM and UIM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
action	state	relate	same name	bi

Linkages between FSM and PSTM

There is one type of linkage defined between FSM and PSTM. A **relate** linkage indicates the **actor** in FSM or the FSM graph represents the same piece of code as the **block** in PSTM. The FSM and PSTM linkages are defined in Table E.36.

Table E.36: Linkage between FSM and PSTM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
actor	block	relate	same name	bi
graph	block	relate	same name	bi

Linkages between FSMs

There are two types of linkage defined between FSMs. An **equate** linkage indicates the two objects represent the same information. A **decomposite** linkage indicates the graph is the decomposition of the **actor** in the source graph. The linkages between FSMs linkages are defined in Table E.37.

Table E.37: Linkage between FSMs

Source	*Destination*	*Type*	*Link Criteria*	*direction*
actor	graph	decomposite	same name	uni
actor	actor	equate	same name	bi
object	object	equate	same name	bi

Linkages between IOPM and EOPM

There is one type of linkage defined between IOPM and EOPM. A **relate** linkage between **branch** or **process** in IOPM represents the same piece of code represented by the **state** in EOPM. A **relate** linkage between an **entry** or **exit** and a **state** indicates both of them are entry or exit point of the graphs. The IOPM and EOPM linkages are defined in Table E.38.

Table E.38: Linkage between IOPM and EOPM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
branch	state	relate	same statement	bi
process	state	relate	same statement	bi
entry	state	relate	the state is an entry state	bi
exit	state	relate	the state is an exit state	bi

Linkages between IOPM and UIM

There is one type of linkage defined between IOPM and UIM. A **relate** linkage between **branch** or **process** in IOPM represents the same piece of code represented by the **state** in UIM. A **relate** linkage between **entry** or **exit** and a **state** indicates both of them are entry or exit point of the graphs. The IOPM and UIM linkages are defined in Table E.39.

Table E.39: Linkage between IOPM and UIM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
branch	state	relate	same statement	bi
process	state	relate	same statement	bi
entry	state	relate	the state is an entry state	bi
exit	state	relate	the state is an exit state	bi

Linkages between IOPM and PSTM

There is one type of linkage defined between IOPM and PSTM. A **relate** linkage indicates the **process** or the IOPM graph represents the same piece of code as the **block** in PSTM. The IOPM and PSTM linkages are defined in Table E.40.

Table E.40: Linkage between IOPM and PSTM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
process	block	relate	same name	bi
graph	block	relate	same name	bi

Linkages between IOPMs

There is one type of linkage defined between IOPMs. A **decomposite** linkage indicates the sink graph is the decomposition of the **process** in the source graph. The linkage between IOPMs is defined in Table E.41.

Table E.41: Linkage between IOPMs

Source	*Destination*	*Type*	*Link Criteria*	*direction*
process	graph	decomposite	same name	uni

Linkages between IOPM2 and EOPM

There is one type of linkage defined between IOPM2 and EOPM. A **relate** linkage between a **Branch**, **Process**, or **Invoke** in IOPM2 represents the same piece of code represented by the **state** in EOPM. A **relate** linkage between **Entry** or **Exit** and a **state** indicates both of them are entry or exit point of the graphs. The IOPM2 and EOPM linkages are defined in Table E.42.

Table E.42: Linkage between IOPM2 and EOPM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
Branch	state	relate	same statement	bi
Process	state	relate	same statement	bi
Invoke	state	relate	same statement	bi
Entry	state	relate	the state is the entry state	bi
Exit	state	relate	the state is the exit state	bi

Linkages between IOPM2 and UIM

There is one type of linkage defined between IOPM2 and UIM. A **relate** linkage between **Branch**, **Process**, or **Invoke** in IOPM2 represents the same piece of code represented by the **state** in UIM. A **relate** linkage between **Entry** or **Exit** and a **state** indicates both of them are entry or exit point of the graphs. The IOPM2 and UIM linkages are defined in Table E.43.

Table E.43: Linkage between IOPM2 and UIM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
Branch	state	relate	same statement	bi
Process	state	relate	same statement	bi
Invoke	state	relate	same statement	bi
Entry	state	relate	the state is the entry state	bi
Exit	state	relate	the state is the exit state	bi

Linkages between IOPM2 and PSTM

There is one type of linkage defined between IOPM2 and PSTM. A **relate** linkage indicates the **Invoke** or the IOPM2 graph represents the same piece of code as the **block** or the **module** in PSTM. The IOPM2 and PSTM linkages are defined in Table E.44.

Table E.44: Linkage between IOPM2 and PSTM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
Invoke	module	relate	same name	bi
graph	block	relate	same name	bi

Linkages between IOPM2s

There is one type of linkage defined between IOPMs. A **decomposite** linkage indicates the sink graph is the decomposition of the **Invoke** in the source graph. The linkage between IOPM2s is defined in Table E.45.

Table E.45: Linkage between IOPM2s

Source	*Destination*	*Type*	*Link Criteria*	*direction*
Invoke	graph	decomposite	same name	uni

Linkages between EOPM and UIM

There is one type of linkage defined between EOPM and UIM. A **relate** linkage indicates the **state** in EOPM represents the same piece of code as the **state** in UIM. The EOPM and UIM linkage is defined in Table E.46.

Table E.46: Linkage between EOPM and UIM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
state	state	relate	same name	bi

Linkages between EOPM and PSTM

There is one type of linkage defined between EOPM and PSTM. A **relate** linkage indicates the **state** in EOPM represents the same piece of code as the **block** in PSTM. The EOPM and PSTM linkages are defined in Table E.47.

Table E.47: Linkage between EOPM and PSTM

Source	*Destination*	*Type*	*Link Criteria*	*direction*
state	block	relate	same name	bi
graph	block	relate	same name	bi

Linkages between EOPMs

There is one type of linkage defined between EOPMs. A **decomposite** linkage indicates the graph is the decomposition of the **state** in the source graph. The linkage between EOPMs is defined in Table E.48.

Table E.48: Linkage between EOPMs

Source	*Destination*	*Type*	*Link Criteria*	*direction*
state	graph	decomposite	same name	uni

E.2 Linkage Generation Rule

This section shows the linkage generation rules defined for generating of the following linkages: CPM to DM, CPM to DM2, CPM to FM, CPM to PSTM, IOPM to DM, IOPM2 to DM2, PSTM to DM, and PSTM to DM2.

CPM to DM Linkage Generation Rule

The following rules are used to generate CPM to DM linkage. The first linkage rule represents an **equate** linkage from a **internal_data** in a CPM graph to a **internal_data**, a **record**, or a **data_item** in a DM graph if the names of the two entities are same. The second linkage represents an **equate** linkage from a **external_data** in CPM to a **report**, a **file**, a **form**, a **monitor**, or a **graph** in a DM graph if the names of the two entities are same. The third linkage rule represent an **equate** linkage from a **datastore** in CPM to a **data_bank** in a DM graph if the names of the two entities are same.

```
LINK(("CPM" "DM")
    ("equate","internal_data","internal_data" OR "record" OR "data_item",
      SRC:NAME = DST:NAME)
    ("equate","external_data","report" OR "file" OR "form" OR
     "monitor" OR "graph", SRC:NAME = DST:NAME)
    ("equate","external_data_element","record" OR "data_item",
      SRC:NAME = DST:NAME)
    ("equate" "datastore" "data_bank" SRC:NAME = DST:NAME))
```

CPM to DM2 Linkage Generation Rule

The following rules are used to generate CPM to DM2 linkage. The first linkage rule represents an **equate** linkage from an **internal_data** or an **external_data** in a CPM graph to a **data_unit**, a **record**, or a **data_item** in a DM2 graph if the names of the two entities are same. The second linkage rule represents an **equate** linkage from an **external_data** in a CPM graph to a **file** in a DM2 graph if the names of the two entities are same.

```
LINK(("CPM" "DM2")
     ("equate","internal_data" OR "external_data_element",
       "data_unit" OR "record" OR "data_item",
       SRC:NAME = DST:NAME)
     ("equate","external_data","file", SRC:NAME = DST:NAME))
```

CPM to FM Linkage Generation Rule

The following rules are used to generate CPM to FM linkage. The first linkage rule represent a **relate** linkage from a **task** in a CPM graph to a **function** or a **sub_function** in a FM graph if the names of the two entities are same. The second linkage rule represents an **equate** linkage from a **database_system**, a **a datastore**, an **external_data**, or an **internal_data** in a CPM graph to an **input_data_set** or an **external_data_set** in a FM graph if the names of the two entities are same. The third linkage rule represents an **equate** linkage from an **internal_data** or an **external_data_element** in a CPM graph to a **sub_input_data** or a **sub_output_data** in a FM graph if the names of the two entities are same.

```
LINK(("CPM" "FM")
     ("relate","task","function" or "sub_function", SRC:NAME = DST:NAME)
     ("equal", "database_system" or "datastore" or "external_data" or
       "internal_data", "input_data_set" or "external_data_set",
       SRC:NAME = DST:NAME)
     ("equal", "internal_data" or "external_data_element",
       "sub_input_data" or "sub_output_data",
       SRC:NAME = DST:NAME))
```

CPM to PSTM Linkage Generation Rule

The following rule is used to generate CPM to PSTM linkage. The linkage rule represents an **equate** linkage between a **task** in a CPM graph to a **module** in a PSTM graph if the names of the two entities are same.

```
LINK(("CPM" "PSTM")
     ("equate" "task" "module" SRC:NAME = DST:NAME))
```

IOPM to DM Linkage Generation Rule

The following rules are used to generate IOPM to DM linkage. There are two tables required to establish the linkages – a *data definition table* and a *data reference table.* The first linkage rule represents a **refer** linkage from a **process** or a **branch** in a IOPM graph to an entity in a DM graph if the value of the **node_id** attribute of the **process** or **branch** is same as the value of the NODE_ID field of a entry in the *data reference table* and the value of the DATA_NAME field of the entry is same as the name of the entity in the DM graph. The second linkage rule represents a **define** linkage from a **process** or a **branch** in a IOPM graph to an entity in a DM graph if the value of the **node_id** attribute of the **process** or **branch** is same as the value of the NODE_ID field of a entry in the *data definition table* and the value of the DEST_NAME field of the entry is same as the name of the entity in the DM graph.

```
TABLE(DEF)
TABLE(REF)

LINK(("IOPM", "DM")
     ("refer", "process" OR "branch", ENTITY,
      REF:NODE_ID = SRC:NODE_ID & SAME:DATA_NAME = DST:NAME)
     ("define", "process" OR "branch", ENTITY,
      DEF:NODE_ID = SRC:NODE_ID & SAME:DEST_NAME = DST:NAME))
```

IOPM2 to DM2 Linkage Generation Rule

The following rules are used to generate IOPM2 to DM2 linkage. There are two tables required to establish the linkages – a *data definition table* and a *data reference table.* The first linkage rule represents a **refer** linkage from a **Process** or a **Branch** in a IOPM2 graph to an entity in a DM2 graph if the value of the **node_id** attribute of the **Process** or **Branch** is same as the value of the NODE_ID field of a entry in the *data reference table* and the value of the DATA_NAME field of the entry is same as the name of the entity in the DM2 graph. The second linkage rule represents a **define** linkage from a **Process** or a **Branch** in a IOPM2 graph to an entity in a DM2 graph if the value of the **node_id** attribute of the **Process** or **Branch** is same as the value of the NODE_ID field of a entry in the *data definition table* and the value of the DEST_NAME field of the entry is same as the name of the entity in the DM2 graph.

```
TABLE(DEF)
TABLE(REF)

LINK(("IOPM2", "DM2")
     ("refer", "Process" OR "Branch", ENTITY,
      REF:NODE_ID = SRC:NODE_ID & SAME:DATA_NAME = DST:NAME)
     ("define", "Process" OR "Branch", ENTITY,
      DEF:NODE_ID = SRC:NODE_ID & SAME:DEST_NAME = DST:NAME))
```

PSTM to DM Linkage Generation Rule

The following rules are used to generate PSTM to DM linkage. There are two tables required to establish the linkages – a *data definition table* and a *data reference table.* The first linkage rule represents a **refer** linkage from a **block** in a PSTM graph to an entity in a DM graph if the value of the LINE_NO field of a entry in the *data reference table* is greater than the value of the **start** attribute of the **block** but less then the value of **end** attribute of the **block** and the value of the DATA_NAME field of the entry is same as the name of the entity in the DM graph. The second linkage rule represents a **define** linkage from a **block** in a PSTM graph to an entity in a DM graph if the value of the LINE_NO field of a entry in the *data definition table* is greater than the value of the **start** attribute of the **block** but less then the value of **end** attribute of the **block** and the value of the DEST_NAME field of the entry is same as the name of the entity in the DM graph.

```
TABLE(DEF)
TABLE(REF)

LINK(("PSTM", "DM")
     ("refer", "block", ENTITY, REF:LINE_NO > SRC:START &
      SAME:LINE_NO < SRC:END & SAME:DATA_NAME = DST:NAME)
     ("define", "block", ENTITY, DEF:LINE_NO > SRC:START &
      SAME:LINE_NO < SRC:END & SAME:DEST_NAME = DST:NAME))
```

PSTM to DM2 Linkage Generation Rule

The following rules are used to generate PSTM to DM2 linkage. There are two tables required to establish the linkages – a *data definition table* and a *data reference table.* The first linkage rule represents a **refer** linkage from a **block** in a PSTM graph to an entity in a DM2 graph if the value of the LINE_NO field of a entry in the *data reference table* is greater than the value of the **start** attribute of the **block** but less then the value of **end** attribute of the **block** and the value of the DATA_NAME field of the entry is same as the name of the entity in the DM2 graph. The second linkage rule represents a **define** linkage from a **block** in a PSTM graph to an entity in a DM2 graph if the value of the LINE_NO field of a entry in the *data definition table* is greater than the value of the **start** attribute of the **block** but less then the value of **end** attribute of the **block** and the value of the DEST_NAME field of the entry is same as the name of the entity in the DM2 graph.

```
TABLE(DEF)
TABLE(REF)

LINK(("PSTM", "DM2")
     ("refer", "block", ENTITY, REF:LINE_NO > SRC:START &
      SAME:LINE_NO < SRC:END & SAME:DATA_NAME = DST:NAME)
     ("define", "block", ENTITY, DEF:LINE_NO > SRC:START &
      SAME:LINE_NO < SRC:END & SAME:DEST_NAME = DST:NAME))
```

Bibliography

[1] G. Arango, I. Baxter, P. Freeman, and C. Pidgeon, "Software Maintenance by Transformation," *IEEE Software,* Vol. 3, No. 3, pp. 27–39, May 1986.

[2] Avron Barr and Edward A. Feigenbaum, *The Handbook of Artificial Intelligence*, Vol. 4, Stanford, CA, HeurisTech Press, 1981.

[3] P. Benedusi, A. Cimitile and U. De Carlini, "A Reverse Engineering Methodology to Reconstruct Hierarchical Data Flow Diagrams For Software Maintenance," *Proc. 1989 IEEE Conference on Software Maintenance*, pp. 180–189, Oct. 16–19, 1989.

[4] T. J. Biggerstaff, "Design Recovery for Maintenance and Reuse," *IEEE Computer*, Vol. 22, No. 7, pp. 36–49, July 1989.

[5] R. J. Brachman and J. G. Schmolze, "A Overview of the KL-ONE knowledge representation system," *Cognitive Science*, Vol. 9, NO. 2, pp. 171–216, 1985.

[6] Alan W. Brown and John A. McDermid, "Learning from IPSE's Mistakes", *IEEE Software*, Vol. 7, No. 2, pp. 23–28, March 1992.

[7] F. W. Calliss, M. Khalil, M. Munro, and M. Ward, "A Knowledge-Based System for Software Maintenance," *Proc. 1988 IEEE Conference on Software Maintenance*, pp. 319–323, Oct. 24–27, 1988.

[8] P. P. Chen, "The Entity Relationship Model - Toward a Unified View of Data," *ACM Transactions on Database System*, Vol. 1, No. 1, pp. 9–36, Mar. 1976.

[9] Y. F. Chen and C. V. Ramamoorthy, "The C information abstractor," *Proc. 10th International Computer Software and Applications Conference (COMPSAC)*, pp. 291–198, Oct. 8–10, 1986.

[10] Y. F. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy, "The C Information Abstraction System," *IEEE Transaction on Software Engineering*, Vol. 16, No. 3, pp. 325–334, Mar. 1990.

[11] E. J. Chikofsky and J. H. Cross II. "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, Vol. 7, No. 1, pp. 13–17, Jan. 1990.

[12] A. Cimitili and U. De Carlini, "Reverse Engineering: Algorithms for Program Graph Production," *Software – Practice and Experience*, Vol. 25, No. 5, pp. 519–537, May 1991.

[13] A. Colbrook and C. Smythe, "The Retrospective Introduction of Abstraction into Software," *Proc. 1989 IEEE Conference on Software Maintenance*, pp. 166–173, Oct. 16–19, 1989,

[14] B. K. Das, "A Knowledge-based Approach to the Analysis of Code and Program Design Language," *Proc. 1989 IEEE Conference on Software Maintenance*, pp. 290–295, Oct. 16–19, 1989.

[15] C. Desclauz and M. Ribault, "MACS: Maintenance Assistance Capability for Software A K.A.D.M.E.," *Proc. 1991 Conf. on Software Maintenance*, Sorrento, Italy, Oct. 15–17, 1991, pp. 2–12.

[16] A. Engberts, W. Kozaczynski, and J. Ning, "Concept Recognition-Based Program Transformation," *Proc. 1991 IEEE Conference on Software Maintenance*, pp. 73–82, Oct. 15–17, 1991.

[17] S. D. Fay and D. G. Holmes, "Help! I Have to Update an Undocumented Program," *Proc. 1985 IEEE Conference on Software Maintenance*, pp. 194–202, Nov. 11–13. 1985.

[18] R. W. Gray et al., "Eli: A complete, flexible, compiler construction system," *Commun. ACM*, Vol. 35, No. 2, pp. 121–130, Feb. 1992.

[19] D. P. Hale and D. A. Haworth, "Software Maintenance: A Profile of Fast Empirical Research," *Proc. 1988 IEEE Conference on Software Maintenance*, pp. 236–240, Oct. 24–27, 1988.

[20] M. T. Harandi and J. Q. Ning, "PAT: A Knowledge-based Program Analysis Tool," *Proc. 1988 IEEE Conference on Software Maintenance*, pp. 312–318, Oct. 24–27, 1988.

[21] M. T. Harandi and J. Q. Ning, "Knowledge-based Program Analysis," *IEEE Software*, Vol. 7 No. 1, pp. 74–81, Jan. 1990.

[22] H. Huang, "USER'S MANUAL For Reverse Engineering Tool Set, Version 2.2," Software Engineering Research Lab, Dept. of Information and Computer Sciences, University of Hawaii at Manoa, Honolulu, HI, Technical Report, Sept. 1992

[23] H. Huang, K. Sugihara, and I. Miyamoto, "Reconstructing Data Flow Diagram from COBOL Source Code," *Proceedings of International Conference for Young Computer Scientists*, Beijing, China, July 18–20, 1991, pp. 198–201.

[24] H. Huang, K. Sugihara, and I. Miyamoto, "A rule-based tool for reverse engineering from source code to graphical models," *Proceedings of the Fourth International Conference on Software Engineering and Knowledge Engineering*, Capri, Italy, June 1992, pp. 178–185.

[25] H. Huang, K. Sugihara, K. Takeda, K. Yamamoto, and I. Miyamoto, "Reverse engineering tools in Software Maintenance Assistant," *Proceedings of the Fifth International Conference on Software Engineering & Its Applications (Toulouse '92)*, Toulouse, France, December 1992, pp. 401–410.

[26] T. Ishii, "A Graphical Query Language for ERA Database," Software Engineering Research Lab, Department of Information and Computer Sciences, University of Hawaii at Manoa, Honolulu, HI, Technical Report 92–12, August 1992.

[27] W. L. Johnson and E. Soloway, "PROUST: Knowledge-Based Program Understanding," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, pp. 267–275, Mar. 1985.

[28] W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors*, Morgan Kaufmann, Palo Alto, California, 1986.

[29] W. Kozaczynski, J. Ning, and A. Engberts, "Program Concept Recognition and Transformation" *IEEE Transactions on Software Engineering*, Vol. 18, No. 12, pp. 1065–1075, Dec. 1992.

[30] D. R. Kuhu, "A Source Code Analyzer for Maintenance," *Proc. of 1987 IEEE Conference on Software Maintenance*, pp. 176–180, Sept. 21–24, 1987.

[31] S. Letovsky, "Cognitive Processes in Program Comprehension," in *Empirical Studies of Programmers*, Edited by E. Soloway and S. Iyengar, Ablex Publishing Corp., pp. 58–79, 1986.

[32] F. J. Lukey, "Understanding and Debugging Program," *International Journal of Man-Machine Studies*, Vol. 12, No. 2, pp. 189–202, Feb. 1980.

[33] J. Martin and C. McClure. *SOFTWARE MAINTENANCE The Problem and Its Solution*, Prentice-Hall, Inc. 1983.

[34] E. Merlo, K. Kontogiannis, and J. F. Girard. "Structural and Behavioral Code Representation for Program Understanding," *Proc. of 5th International Workshop on Computer-Aided Software Engineering*, pp. 106–108, July 6–10, 1992.

[35] J. Moad, "Maintaining the Competitive Edge," *Datamation*, Vol. 36, No. 4, pp. 61–66, Feb. 1990.

[36] I. Miyamoto, "An Analysis of Reverse Engineering Tools on Market," *SMA Project Memo 91-05*, Software Engineering Research Laboratory, 1991.

[37] Ja. Nielsen, "Noncommand user interfaces," *Commun. ACM*, Vol. 36, No. 4, pp. 83–99, April 1993.

[38] S. Paul and A. Prakash, "Source Code Retrieval Using Program Patterns," *Proc. of 1992 Conference on Software Maintenance*, pp. 95–105,

[39] M. G. Rekoff, Jr., "On Reverse Engineering," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-15, No. 2, pp. 244–252, Mar. 1985.

[40] D. J. Robson, K. H. Bennett, B. J. Cornelius, and M. Munro, "Approaches to Program Comprehension," *Journal of Systems and Software*, Vol. 18, No. 2, pp. 79-84, May 1992.

[41] W. C. Sasso, "An Empirical Study of Reengineering Behavior: Design Recovery by Experienced Professionals," *Software Engineering*, pp. 13–20, May/June 1990.

[42] N. F. Schneidewind, "The State of Software Maintenance," *Proc. of 1985 IEEE Conference on Software Maintenance*, pp. 28–34, Nov. 11–13, 1985.

[43] SMA Model Formalism Committee, "SMA Model Formalisms," Software Engineering Research Lab, Department of Information and Computer Sciences, University of Hawaii at Manoa, Honolulu, HI, July 1993.

[44] E. Swanson. "The Dimensions of Maintenance," *Proc. of 2nd International Conference on Software Engineering*, pp. 492–497, Oct. 13–15, 1976.

[45] K. Takeda, D. N. Chin, and I. Miyamoto. "MERA: Meta Language for Software Engineering," *Proc. of 4th International Conference on Software Engineering and Knowledge Engineering*, pp. 495–502, June 15–20, 1992.

[46] Ian Thomas and Brian A. Nejmeh, "Definitions of Tool Integration for Environments", *IEEE Software*, Vol. 7, No. 2, pp. 29–35, March 1992.

[47] J. D. Wedo, "Structured Program Analysis Applied to Software Maintenance," *Proc. of 1985 IEEE Conference on Software Maintenance*, pp. 28–34, Nov. 11–13, 1985.

[48] S. Wiedenbeck, "Processes in Computer Program Comprehension," in *Empirical Studies of Programmers*, Edited by E. Soloway and S. Iyengar, Ablex Publishing Corporation, pp. 48–57, 1968.

[49] R. Wilensky, "Some Problems and Proposals for Knowledge Representation," Computer Science Division, University of California, Berkeley, Report No. UCB/CSD 87/351.

[50] L. M. Wills, "Automated Program Recognition," Master's thesis, MIT, Cambridge, Mass. 1987.

[51] D. Yu "A View On Three R's (3Rs): Reuse, Re-engineering, and Reverse-engineering," *ACM SIGSOFT, Software Engineering Notes*, Vol. 16, No. 3, pp. 69, July 1991.