1996

A Reverse Engineering Methodology for Extracting Parallelism From Design Abstractions.

Ravi Chandra Erraguntla
Louisiana State University and Agricultural & Mechanical College

INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

A REVERSE ENGINEERING METHODOLOGY FOR EXTRACTING PARALLELISM FROM DESIGN ABSTRACTIONS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Ravi Chandra Erraguntla
B.E., Andhra University, 1985
M.E., Bharatiar University, 1989
M.S., Louisiana State University, 1993
December 1996

Dedicated to my parents
Prof. Erraguntla Venkata Rao, D.Sc., Ph.D.
Erraguntla Kameswari, Ph.D.

Acknowledgments

I express my sincere gratitude to Dr. Doris L. Carver, my advisor and friend who introduced me to the field of Software Engineering. Her knowledge and versatility in the different areas of Software Engineering helped me gain invaluable insight into software systems. She has been a constant source of inspiration and none of this work could have been possible without her support and encouragement. She always listened to what I had to say and provided advice on many issues. I am fortunate for having been associated with her on research projects at Thermalscan Inc. and Medical Thermal Diagnostics. I thank her for appointing me as the Laboratory Manager of the Software Engineering Laboratory. I thank her for providing the facilities of the Software Engineering Laboratory.

I thank Dr. Mark L. Williams for introducing me to the field of Nuclear Engineering and for graciously agreeing to serve on my doctoral advisory committee. I extend my appreciation to Dr. Donald H. Kraft, Dr. J. Bush Jones, and Dr. Suresh Rai, members of my doctoral advisory committee, for reviewing my dissertation and for offering their invaluable suggestions.

I owe thanks to the Graduate Assistantship committee of the Department of Computer Science for providing financial assistance from August 1992. I thank Dr. Sitarama S. Iyengar for providing the computing facilities Mr. Elias Khalaf for the excellent system management.

Special thank goes to Mr. Jim and Ms. Sally Davidson of Thermalscan Inc. for their love and affection. Members of the Software Engineering Group have created an atmosphere which made the work described here possible. In particular, I would cherish the friendship of Jigang Liu, Srinivas Lingineni, and Chenga Reddy.

Nothing would have been possible without the patient cooperation, constant encouragement, and unconditional sacrifice of my wonderful wife Neeraja. She helped me stay focused all the time. I am indebted to my parents Prof. E. Venkata Rao and Dr. Kameswari for teaching me everything in life. I thank my in-laws Mr. K.S.N. Murthy and Dr. Sita and the rest of my family members for their support, love, and affection during the entire duration of my doctoral program.

Finally, I thank God for giving me the strength to achieve my goals and objectives.

Table of Contents

List of Tables

List of Figures

Abstract

Migration of code from sequential environments to the parallel processing environments is often done in an ad hoc manner. The purpose of this research is to develop a reverse engineering methodology to facilitate systematic migration of code from sequential to the parallel processing environments. The research results include the development of a three-phase methodology and the design and development of a reverse engineering toolkit (abbreviated as RETK) which serves to establish a working model for the methodology.

The methodology consists of three phases: Analysis, Synthesis, and Transformation. The Analysis phase uses concepts from reverse engineering research to recover the sequential design description from programs using a new design recovery technique. The Synthesis phase is comprised of processes that compute the data and control dependences by using the design abstractions produced by the Analysis phase to construct the program dependence graph. The Transformation phase consists of processes that require knowledge-based analysis of the program and dependence information produced by the Analysis and Synthesis phases, respectively. Design recommendations for parallel environments are the key output of the Transformation phase.

The main components of RETK are an Information Extractor, a Dependence Analyzer, and a Design Assistant that implement the processes of the Analysis, Synthesis, and Transformation phases, respectively. The object-oriented design and

implementation of the Information Extractor and Dependence Analyzer are described. The design and implementation of the Design Assistant using C Language Interface Production System (CLIPS) are described. In addition, experimental results of applying the methodology to test programs by RETK are presented. The results include analysis of a Numerical Aerodynamic Simulation (NAS) benchmark program.

By uniquely combining research in reverse engineering, dependence analysis, and knowledge-based analysis, the methodology provides a systematic approach for code migration. The benefits of using the methodology are increased comprehensibility and improved efficiency in migrating sequential systems to parallel environments.

Chapter 1

Introduction

A problem that is faced by many companies and government organizations is that their legacy systems run on outdated platforms thereby inhibiting growth and change. Legacy systems are computer programs that are on an average 15 to 20 years old. They do not have the capacity to scale up to the changes and advances of the computing community [Ning 94]. These systems were developed primarily for uni-processor environments using programming languages and coding techniques that pre-date some of the expressive and powerful languages that are available today. As most legacy systems are *working* systems, it is difficult to retire them. However, considerable effort and money are being spent to maintain these systems.

In recent years much progress has been made in the area of parallel and distributed architectures and computing techniques. Efficient system interconnections for fast communications among multiple processors and shared memory, I/O, and peripheral devices are used in these architectures to meet the demands of parallel processing. It has been predicted that millions of lines of sequential code will migrate to parallel environments [Harr 93].

In this research we 1) present a methodology to facilitate migration of code from the uni-processor to the parallel processing environment, 2) define a new approach based on the object-oriented paradigm for the design recovery of FORTRAN code, 3) define new knowledge-based representation schemes to represent program

and dependence facts of sequential programs, and 4) define a graphical representation scheme to represent parallel design recommendations. A significant aspect of the methodology is its potential for automated support. By uniquely combining research in reverse engineering, dependence analysis, and knowledge-based analysis, the methodology provides a systematic approach for code migration. The remainder of this chapter presents an overview of the problem of code migration, the objectives of this research, and finally an outline describing the organization of this dissertation.

1.1 Overview

Over the past decade, hardware costs diminished and performance increased. Some of the desk-top computers that are available today are more powerful than the main-frame computers of the 1970s. New software design and production is at its peak taking full advantage of these machines. Unfortunately, legacy systems cannot be easily modified to fit into the realm of these advances. Although an existing system can be retired by re-developing a new system, the option is seldom exercised due to a number of reasons. First, it is cost prohibitive to develop software systems from scratch. Secondly, these systems have embedded in them important business rules that may not be documented elsewhere. In addition, most legacy systems, usually built from multi-vendor contracts, have few or no formal design documents. Moreover, years of "patching" has resulted in systems that are poorly structured, coded, and documented. This lack of documentation makes the redevelopment of the systems much more difficult [Osbo 90].

Although there is widespread research in the area of parallel programming environments, its acceptance remains fairly restricted to the academic world, mainly because parallel machines are expensive and many companies and government agencies are reluctant to port their applications fearing a lack of return of their investment. Major strides have occurred in the recent years in the field of high performance architectures and parallel algorithms; however, there is growing apprehension in the computing community that legacy systems will still continue to run on old platforms such as an IBM/3090 mainframe machine. Unfortunately there is no panacea for migration of legacy code to new platforms. The migration is heavily dependent on the problem the software system solves as well as on the architecture of the target parallel machine.

Several research initiatives exist to migrate code from the uni-processor to parallel environments. Much of the effort has been devoted to code-to-code transformations with the help of a parallelizing or vectorizing compiler. These compilers apply program transformations primarily to loops and partition large computations into sub-computations to take advantage of available vector hardware or multiplicity of processors. The main focus is an attempt to achieve high performance gains. However, code analysis with no accompanying design analysis does not provide complete insight into the overall structure and comprehensibility of the underlying system. Code-to-code transformations also suffer from the garbage-in garbage-out syndrome [Jarz 95] in that if the original system is unwieldy, then the results produced by code-to-code transformations may not produce meaningful results.

A major problem in migrating from an imperative paradigm to other paradigms, including the parallel/distributed paradigm, is understanding the original code. Reverse engineering techniques can be applied to provide support for the understanding of legacy code. Reverse engineering involves analyzing an existing system to "identify the system's components and their interrelationships" and to "create representation of the system in another form or at a higher level of abstraction." Reverse engineering is different from reengineering which involves actually modifying the system to restructure or meet new requirements [Chik 90]. One product of reverse engineering a systems is the design of a system. An effective design of a software system should not only attempt to satisfy the requirements but should also provide a blueprint for its implementation. Reverse engineering techniques aid in the extraction of such a design. The extraction of a design will in turn lead to the introduction of the much needed design phase of the software life-cycle. The benefits of providing a distinct design phase include: increased understanding of a system, reduction in implementation errors, reduction in testing time as more errors will be detected in the design phase, increased quality of documentation, and reduced cost of the overall system [Pres 92].

Another problem in the migration process is ascertaining whether or not a given program has high potential for parallelism. For example, the amount of actual parallelism that is available in a program at the code level is limited by its *dependences*; data, control, and resource. A dependence between two program segments is a conflict that prevents the segments from executing concurrently [Lilj 94]. Fortunately many legacy systems do have a potential for at least coarse grain parallelism in their

programs [Harr 93]. Therefore, the dependences must be understood before attempting to realize a parallel solution. Identifying dependences in a large program can be extremely tedious. Automated support helps reduce the burden.

The ability to simply recognize design decisions in programs is not sufficient [Ruga 90]. The organization of these decisions is also vital. The amount of information that could be potentially elicited from reverse engineering and dependency analysis could be substantial. Intelligent decisions need to be made to accept (or reject) information that is pertinent (or not pertinent) in the parallelization harness. In addition due to the inherent differences between parallel architectures (SIMD and MIMD computers), it is essential to take into account the characteristics of these machines to arrive at parallel design recommendations. Knowledge-based techniques aid in providing such intelligent support. Knowledge-based programs assist in solving problems in a particular domain using an inference procedure. Research in the area of knowledge-based techniques for program understanding uses programming plans and strategies to construct mappings. PAT (Program Analysis Tool) uses an object-oriented methodology of programming concepts and a heuristic-based concept-recognition mechanism to understand programs [Hara 90].

The combination of reverse-engineering followed by reengineering will serve to not only provide structure and accurate documentation to current systems but also to allow the systems to take advantage of parallel and distributed system advantages. A methodology that systematically approaches the problem of code migration, with emphasis on automation offers an economical choice for software managers who

constantly battle to reduce personnel costs. Such a methodology is defined in this research.

1.2 Objectives

In this dissertation, we

1) define a methodology that facilitates migration of code from the uni-processor environment to the parallel processing environment,
2) establish a working model for the methodology,
3) design and develop the components of the methodology to assess their automation potential,
4) present experimental results.

1.3 Outline of the Dissertation

The outline of the dissertation is as follows:

Chapter 1 has presented the problem statement and the objectives of this research. A brief description about the various techniques used in this dissertation has been described.

Chapter 2 presents related research in reverse engineering, dependence analysis, and knowledge-based analysis. A brief survey of parallel and distributed architectures is also presented. Next, tool sets and case studies targeted to provide solutions to code migration are presented. The chapter ends with a section which describes the relevance of the related research to the dissertation.

Chapter 3, which relates to the primary objectives of the research, presents a methodology for design parallelization. The chapter gives an overview of the methodology, brief description of the source language, and a comprehensive description of the different phases and processes that serve to collectively define the methodology.

Chapter 4 describes a reverse engineering toolkit (RETK) designed to demonstrate the automation potential of the methodology. Detailed design and implementation issues of the various components of the toolkit are presented.

Chapter 5 presents experimental results of actual code analyzed by RETK. The results include analysis of the NAS kernel benchmark programs [Bail 94]. The NAS kernel benchmark programs were developed at NASA Ames Research Center for the performance evaluation of highly parallel supercomputers. A brief description of the NAS programs is also presented in the chapter. The chapter ends with a section describing the effectiveness of RETK, and thereby the methodology to provide parallel design recommendations in a systematic and automated manner.

Finally, chapter 6 presents a summary of the dissertation, significance of the research, and ideas for future research.

Chapter 2

Related Research

A methodology for systematic migration of legacy systems from old platforms to newer platforms requires sound techniques and methods. Research in the areas of reverse engineering, dependence analysis, and knowledge-based analysis has provided many techniques to understand and renovate existing systems. However, there is a growing need to explore new methods and methodologies in these areas to tackle the diverse problems associated with migration of legacy systems.

In this chapter, we present related research in the areas of reverse engineering, dependence analysis, and knowledge-based analysis. Techniques that are directly applicable for legacy systems migration are discussed. In order to emphasize the capabilities of modern computing platforms, a brief introduction to parallel architectures is given. Next, existing methodologies that address legacy systems migration are described. The chapter ends with a section that describes the relevance of the related research to this dissertation.

2.1 Reverse Engineering

The term "reverse engineering" has its roots in the hardware world where the primary objective is to decipher how competitor products work. In software engineering, the term is used to describe the process of examining one's own system to aid maintenance, gain insight, and enhance overall understandability [Chik 90]. The cen-

tral theme of reverse engineering research involves the development of tools, techniques, and methodologies for the analysis, synthesis, and representation of information about existing software systems. Research in reverse engineering is motivated largely due to the need for 1) understanding the design of existing systems, 2) transforming old systems into modern computing environments, and 3) allowing for the reuse of existing models [Ingl 94]. Due to the many practical benefits that reverse engineering has to offer, it is recognized as one of the most important parts of software engineering [Wate 94]. The area of reverse engineering can be broadly classified into 1) design recovery, and 2) identification of components. In the following sections we examine research in each of these areas.

2.1.1 Design Recovery

One method that recovers the system design from a specific environment is RECAST (Reverse Engineering into CASE Technology) [Edwa 93]. RECAST transforms the source code of a COBOL system into a format suitable for structured systems analysis. RECAST offers support tools in the form of a command language interface, a user transparent DBMS, an analyzer, and a report generator. The design representations produced by RECAST include data flow diagrams (DFDs), logical data structure (E-R diagrams), structure diagrams using Jackson's structure chart notation [Jack 75], and relational data analysis.

Prototype tools that evaluate, assess, redesign, and reengineer COBOL code for the eventual purpose of transforming the code to a formal specification language such as Z are described in [Lano 93]. The reverse engineering process consists of

transforming the source code into an intermediate language called UNIFORM [Zuyl 93]. An automatic extraction of the design representation is then performed. The design representation produced by the tools include structure graphs, logical data structure, and objects.

A greedy approach to object identification in imperative code is described in [Ache 94]. The approach views a subroutine as a basic unit of functionality. Since the actual parameters are integral to the correct execution of the subroutine, the algorithm presented in this research effort obtains a strong cohesive unit with the minimal set of parameters. The methodology described in this research effort has a high potential for systematic development of an automated system.

[Choi 90] suggests that the structural, functional, dynamic, and behavioral properties of a system would be helpful for extracting and restructuring the design of large systems. The need to understand programs for conceptualization purposes is addressed in [Bigg 94]. A parsing process is described as one of the simplest operational models for concept recognition.

2.1.2 Identification of Components

According to [Ning 93; Ning 94], the problems being faced by many large companies with respect to the legacy systems can be combated only by providing a methodology that allows automated support. Their work describes a set of tools, called COBOL/SRE (COBOL System Renovation Environment), for identifying and extracting components from large legacy COBOL systems. COBOL/SRE tools use program segmentation to "focus" and "factor" out functionally related pieces of code and package them into a self contained module.

A methodology for reverse engineering the Department of Defense (DoD) legacy information systems is reported in [Aike 93; Aike 94]. An approach to extract business rules, domain information, functional requirements, and data architectures in the form of logical data models is presented. Due to the diverse nature of information systems in the defense industry, a pilot study has been conducted to assess the costs of reverse engineering legacy systems and the viability of reengineering such systems. Statistics to assess the economic impact of maintaining these systems is also provided. [Aike 93] observe that:

> *The Department of Defense spends more than $9 billion annually in non-combat information technology development at more than 1700 DoD Data Centers currently running hundreds of legacy systems.*

Most software practitioners and text books on software engineering [Ghez 91; Scha 96] estimate that maintenance costs around 60 percent of the total cost of a software system. Thus, maintenance claims a major portion of DoD spending on legacy systems. The urgent need to revamp these systems is never more crucial than now especially with the government downsizing the defense industry.

The development of a tool for automating and modularization of large COBOL programs using enabling technology for reengineering is described in [Newc 93; Mark 94]. The main features of this technology are: 1) Representation of the software contained in the COBOL system in the form of abstract syntax trees in an object-oriented database, and 2) using commercially available tools to operate on code captured in this form.

Lack of proper documentation is one of the main problems associated with legacy systems. Identification and extraction of "domain independent" components in

a large programs which lack in proper documentation is addressed in [Cuti 93]. Slicing is used as the main technique for extracting and grouping code segments that are interspersed among the various modules of a large program. The techniques presented have a high degree of automation potential.

A method for identifying abstract data types for reuse reengineering is presented in [Canf 93]. The main activities of this approach include assessing existing systems for the identification of candidate reuse components, modifying and packaging the components, and finally understanding the meaning of the components. The last step culminates in producing related specification of the candidate reuse components. Similar ideas have been investigated in [Ache 95]. An algorithm to identify and extract "candidate objects" in imperative code such as FORTRAN-77 is presented. The algorithm for the identification of such objects relies on the features of the language such as subroutine calls and variable definitions. Since data is given more importance in the object-oriented paradigm, data flow analysis is used in the definition and refinement of the candidate objects.

In summary, reverse engineering has become one of the most actively researched fields [WCRE 93] [WCRE 95] in software engineering partly because the research carries immense practical value. Apart from design recovery and identification and extraction of components, research is being focused on the analysis of non-code sources. Test case generation by recovering information from textual documents, such as manuals, is presented in [Luts 95]. Information recovered from textual documents provides valuable input to automated test systems. Since manuals usually contain both text and diagrams, [Butl 95] describe a method to recover information

from diagrammatic sources such as data flow diagrams. Manuals are scanned and processed to generate formal semantics of diagrams. [Newc 95] presents a method for automatic translation of procedural systems into non-procedural architectures using a knowledge-based tools framework.

2.2 Dependence Analysis

[Lilj 94] defines a dependence as follows:

> *A dependence between two program statements is a conflict that prevents the statements from executing concurrently.*

The use of dependence analysis originated in compiler design for the purposes of optimization [Aho 77]. The same principles are now being applied to programs for the purposes of testing and debugging. Dependences can be categorized into three types: resource, data and control [Kuck 78; Lilj 94].

Resource dependences

Resource dependences between two statements arise due to the limited availability of hardware resources such as multipliers in a computer system. It is possible to exclude most resource dependences by the addition of extra hardware.

Data dependences

Consider the following sequence of statements:

```
s1:   A = B + C;
s2:   D = A - E;
```

The value of the variable A is defined (computed) in s1 and used in s2. Clearly reversing the order of execution of s1 and s2 changes the semantic nature of the piece of

code. A data dependence exists between statements s1 and s2. This corresponds to a *read-after-write* conflict. Another type of situation is reflected by a *write-after-read* conflict as illustrated in the following sequence of statements:

```
s1:   D = A * 2
s2:   A = B - C
```

The value of variable A is used in s1 and defined in s2. Again, reversing the order of execution of s1 and s2 changes the semantic nature of the code fragment. A data dependence graph is a graphical representation of data dependences in a program. Nodes are used to represent statements and directed edges (represented as solid lines) between nodes represent data dependences. Table 2.1 shows a code fragment and the corresponding equivalent statement ordering with the loop unrolled. Figure 2.1 shows the data dependence graph for the code fragment shown in Table 2.1. A comprehensive expostulation of data dependences is given in [Ferr 87].

Control Dependences

An intuitive definition of control dependence is given in [Lilj 94]:

> *A control dependence from statement S_i to statement S_j exists when statement S_j should be executed only if statement S_i produces a certain value.*

For example, consider the following sequence of statements:

```
s1:   if (X.EQ.1) then
s2:           B = C * D
      end if
```

s2 depends on the truth of falsity of the predicate X. The value of X determines whether or not s2 is executed. A control dependence graph is a graphical representation of the control dependences in a program. Nodes are statements and directed edges

Table 2.1 Code fragment and the corresponding equivalent statement ordering to illustrate data dependences

Code Fragment:

```
        X(1) = C(1)
        DO 20  I = 2 ,4
                X(I) = C(I)
                DO 20  J=1, I-1
                        X(I) = A(I,J) * X(J) + X(I)
20      CONTINUE
```

Loop Unrolled Statement Ordering

```
s1:     X(1) = C(1)
s2:     X(2) = C(2)
s3:     X(2) = A(2,1) * X(1) +  X(2)
s4:     X(3) = C(3)
s5:     X(3) = A(3,1) * X(1) + A(3,2) * X(2) + X(3)
s6:     X(4) = C(4)
s7:     X(4) = A(4,1) * X(1) + A(4,2) * X(2) + A(4,3) * X(3) + X(4)
```

(represented as dotted lines) represent control dependences. Table 2.2 shows a code fragment consisting of a sequence of statements. Figure 2.2 shows the corresponding control dependence graph.

Due to the interrelationship between data and control dependences, they must be considered in unison. A technique that merges control and data dependencies into a single program dependence graph is described in [Ferr 87]. A post-dominator algorithm [Aho 77] is used for the representation of the PDG.

The problem of analyzing ordinary FORTRAN-like programs to determine the number of operations that could be performed simultaneously is explored in [Kuck 72 where 20 FORTRAN programs, consisting of nearly 1000 lines, were analyzed to produce the conclusion that 16 processors could be effectively used in a parallel fashion to

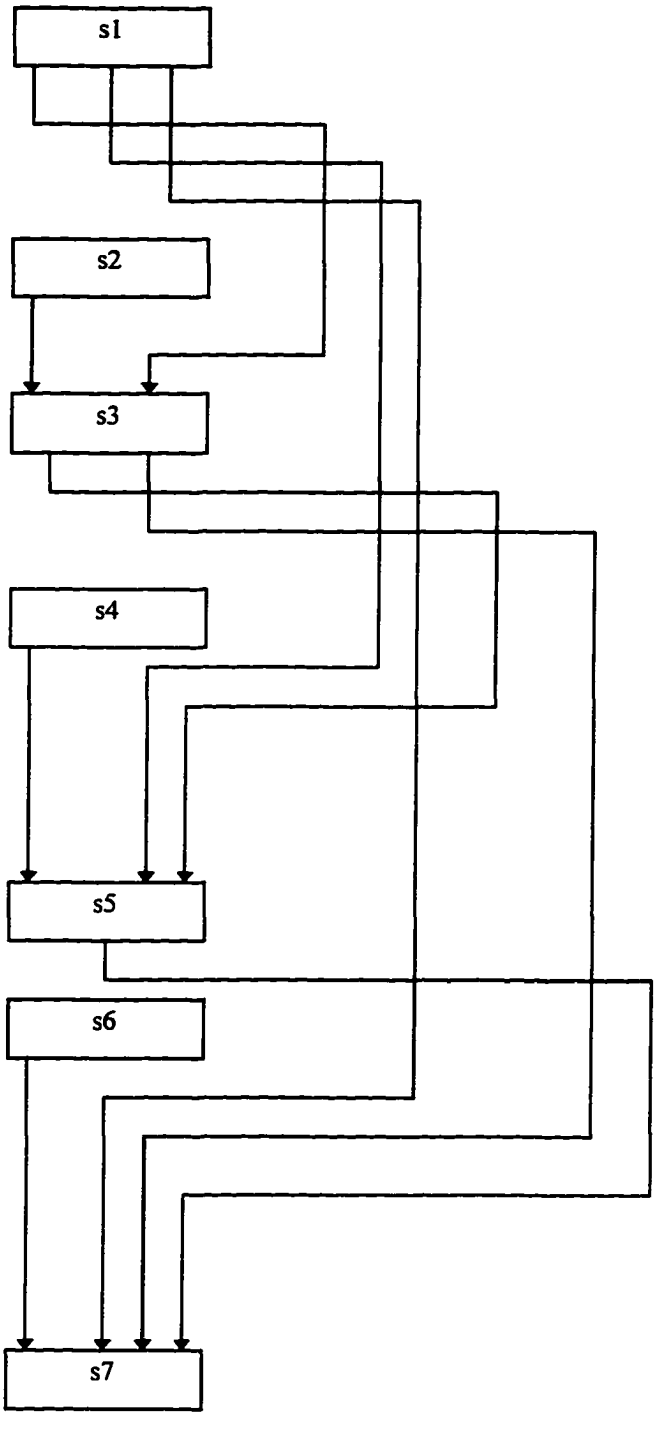


Figure 2.1 Data dependence graph of the code fragment listed in Table 2.1

Table 2.2 Code fragment to illustrate control dependences

```
s1:     A = B
s2:     X = 2
s3:     if ( A.GT.1) then
s4:             Y = X * 20
        endif
```

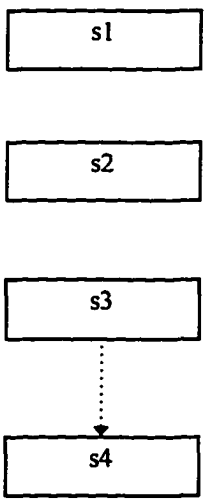


Figure 2.2 Control dependence graph of the code fragment listed in Table 2.2

obtain speedup. The problem of interprocedural slicing — generating a slice of an entire program, where the slice crosses the boundaries of procedure calls — using a "system dependence graph" is described in [Horw 90]. A survey of several architectures and compilation techniques based on the "critical dependence ratio" to exploit parallelism in loops is presented in [Lilj 94]. The critical dependence ratio gives an indication about the maximum speedup that can be achieved by unrolling loops.

The issues involved in generating a program dependence graph for reverse engineering research are emphasized in [Jack 94]. This model is particularly suited to reverse engineering since it assumes the procedures (subroutines) to be modular and

dependences to be fine-grained. Fine-grained dependences consider individual variables rather than single program statements. Fine-grained dependences provide a way to ensure that data structures are not neglected. In addition, almost all reverse engineering efforts are document intensive. Databases or knowledge bases are built that categorize information that is extracted during the reverse engineering process. Hence, it is essential that dependences be fine-grained for queries and reports to be comprehensive.

A new model that takes into account the granularity of dependences and extends the representation of the PDG is found in [Jack 94]. The model introduces the notion of a site which conceptually relates to a statement in code where a state change to a variable occurs. A site is a combination of the use and definition lists. Special sites named *entry* and *exit* have only definition and use lists, respectively. For all other sites, use list variable(s) are the variable(s) on the right-hand side and definition list variable(s) are the variable(s) on the left-hand side. As an illustrative example of the model a code fragment is listed in Table 2.3 and its corresponding PDG is shown in Figure 2.3.

2.3 Knowledge-Based Analysis

The Programmer's Apprentice project is a research effort to understand how expert programmers conduct the activities of writing programs [Ric 88b]. The main goal of the project is to apply techniques from the field of artificial intelligence to automate the process of programming. Although program generators that produce applications from specifications work well for narrow domains, fully automated

Table 2.3 A code fragment to illustrate fine-grained dependences

```
s1:    A = B + C
s2:    if (B.EQ.10) then
s3:            A = A + 20
s4:    end if
```

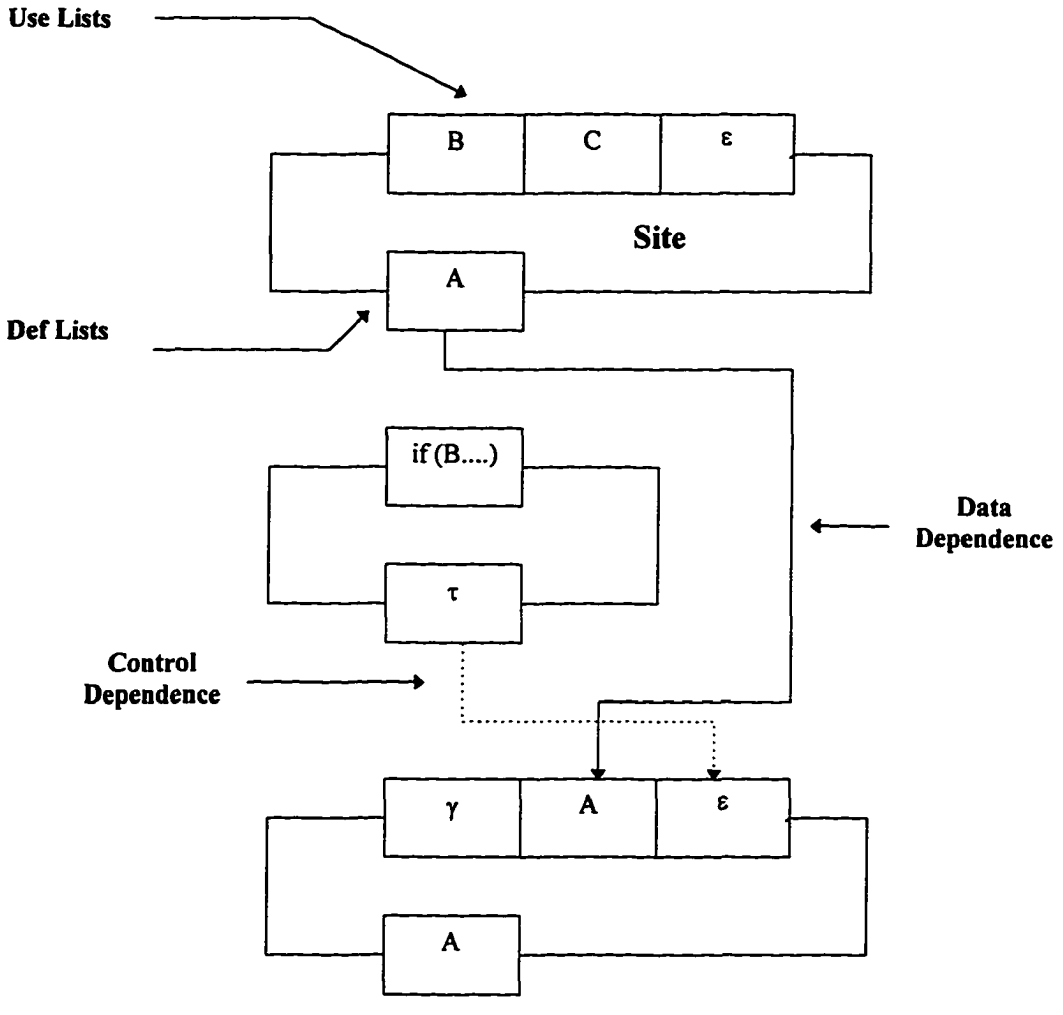


Figure 2.3 Program dependence graph of the code fragment listed in Table 2.3

programming is not perceived as an immediate realistic goal [Ric 88a]. Consequently the emphasis shifted from replacing programmers towards assisting programmers. In order to provide intelligent support, the Programmer's Apprentice project introduced the notion of a *cliché*. A cliché is defined as a commonly used programming structure for implementing higher level abstractions [Rich 90]. The cornerstone of efficient programming is the use of sophisticated data structures and algorithms. Therefore, both data structures and algorithms are represented as clichés. A formal graphical representation for programs and programming clichés is called the Plan Calculus [Ric 88b], which is a language-independent representation. The representation scheme of Plan Calculus is a combination of flowcharts, dataflow schemas, and abstract data types.

[Rich 90] describes a system, the *Recognizer*, that automatically identifies all occurrences of a given set of clichés in a program and constructs a hierarchical description of the program in terms of clichés. Recognizer identifies (recognizes) design decisions in programs by first translating a program into the Plan Calculus and then encoding the program as a flow graph. Finally, a design tree (the key output of the Recognizer) of the program is produced by parsing the flow graph with the help of clichés.

Jarzabeck and Keam point out that even the simplest reverse engineering task is a "knowledge-intensive" process. Accepting or rejecting decisions made during this task needs the involvement of a domain expert. They describe the design of a generic reverse engineering toolkit in [Jarz 95]. An approach to recognition of detailed programming plans (patterns) that combine top-down and bottom-up strategies are

examined in [Quil 94]. The application of template-matching techniques from knowledge-based systems research in extrapolating the intended program function are explored in [Hara 90].

Often problems associated with understanding a program can be attributed to functional "interleaving" where a piece of code is responsible for accomplishing more than one function [Ruga 95]. A formal definition of interleaving in terms of plans [Rich 90] and a method to delocalize the is found in [Ruga 95].

One of the most important aspects in the design of knowledge-based programs is the knowledge representation itself. The three most widely used knowledge representation schemes are rules, semantic nets, and frames [Wins 92].

2.4 Parallel Architectures

Conventional sequential computers are based on the von Neumann architecture. The sequential execution of programs on scalar data is an inherent characteristic of von Neumann architectures. Large-scale numerical applications typically require 10^{12} to 10^{15} Flops (floating point operations) to achieve accurate results [Ston 94]. The sequential execution of large problems on conventional machines places serious time limits. Researchers have developed techniques to improve the performance of sequential computers using lookahead, multiple functional units, and pipelining [Hwan 93]. However, when solving large problems, the intrinsic sequential nature of these computers results in a time-intensive solution.

An alternative to sequential computers is parallel computers. Parallel computers can be broadly classified as either SIMD (single instruction stream over multiple

data streams) or MIMD (multiple instruction stream over multiple data streams) [Flyn 72]. The processors in such systems communicate with each other via shared variables in a common memory or through explicit message passing through an interconnection network. The architectural structures of parallel computers are often biased towards solving particular classes of problems. The characteristics of SIMD and MIMD computers are briefly described in the ensuing sections.

2.4.1 SIMD Computers

Hwang specifies an operational model of a SIMD computer by a 5-tuple [Hwan 93]:

$$P = <N, C, I, M, R>$$

where

P	*is the SIMD model under consideration*
N	*is the number of processing elements (PEs)*
C	*is the set of instructions executed by the control unit (CU)*
M	*is the set of masking schemes to enable or disable subsets of PEs*
R	*is the set of data-routing functions for inter-PE communications*

The operational model of a SIMD computers is shown in Figure 2.4[1]. Each processing element (PE) has its own processor and memory units. All PEs in a SIMD configuration execute the same instruction at each clock cycle. The control unit (CU) broadcasts the instructions to the PEs which operate in lockstep. Only those PEs located in the active set, which can be user-defined by means of masking schemes, carry out the instructions received from CU. The effectiveness of SIMD computer lies in exploiting spatial parallelism in data parallel applications. The computational

[1] Figures 2.4 and 2.5 adapted from [Hwan 93].

parallelism in such applications ensues due to the physical parallel structure of the data expressed in terms of array variables.

MasPar Computer Corporation MP-1 Family is a commercially available SIMD computer. The number of PEs of MP-1 ranges from 1024 to 16,384 processors. Each PE in MP-1 is a RISC processor with 16Kbytes of local memory. The interconnection network used in MP-1 is an X-Net mesh where each PE has 4 neighbors and a mutistage crossbar connection [MasP 91]. Other representative SIMD computers include Thinking Machines Corporation CM-2 and Active Memory Technology DAP600 Family.

2.4.2 MIMD Computers

There are two major categories of MIMD computers, namely, shared-memory mutiprocessors and message-passing muticomputers. The operational model of a shared-memory MIMD computer is shown in Figure 2.5[1]. The processors in these computers communicate with each other via shared variables in a common memory. Both instructions and data are stored in the shared memory. Each processor is controlled by a separate control unit (CU) which issues the instruction stream. The data stream for the operations identified by the instruction stream is obtained from the shared-memory. In addition, each processor is responsible for its own I/O. An example of a shared-memory MIMD computer is Sequent Symmetry S-81 which consists of 30 processors connected by means of a bus.

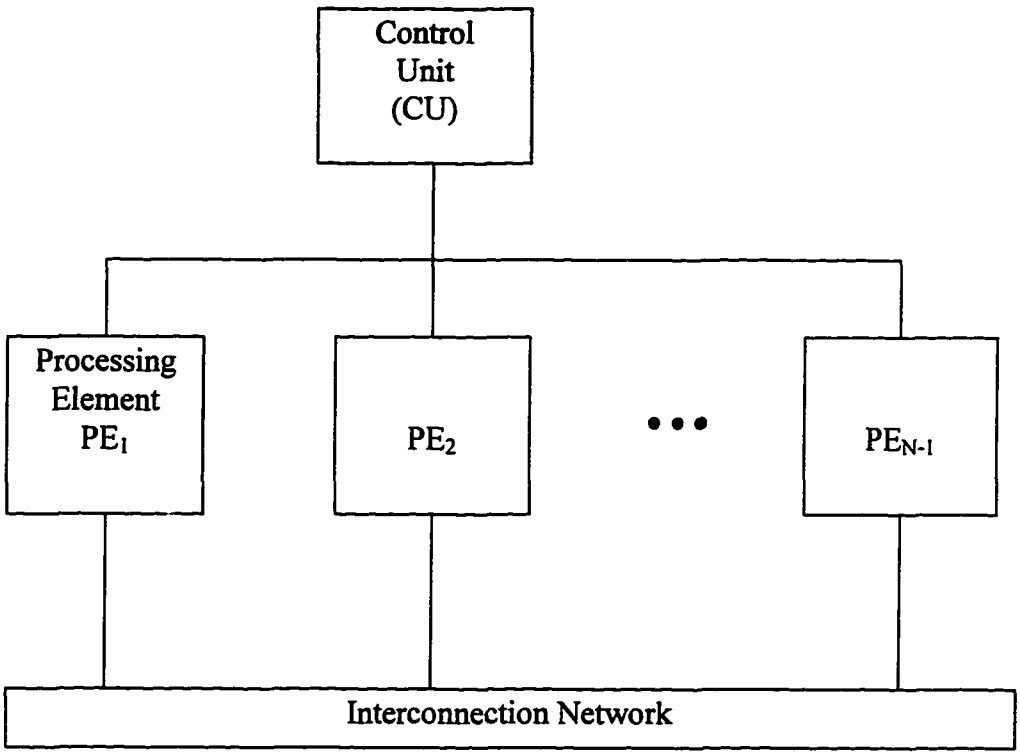
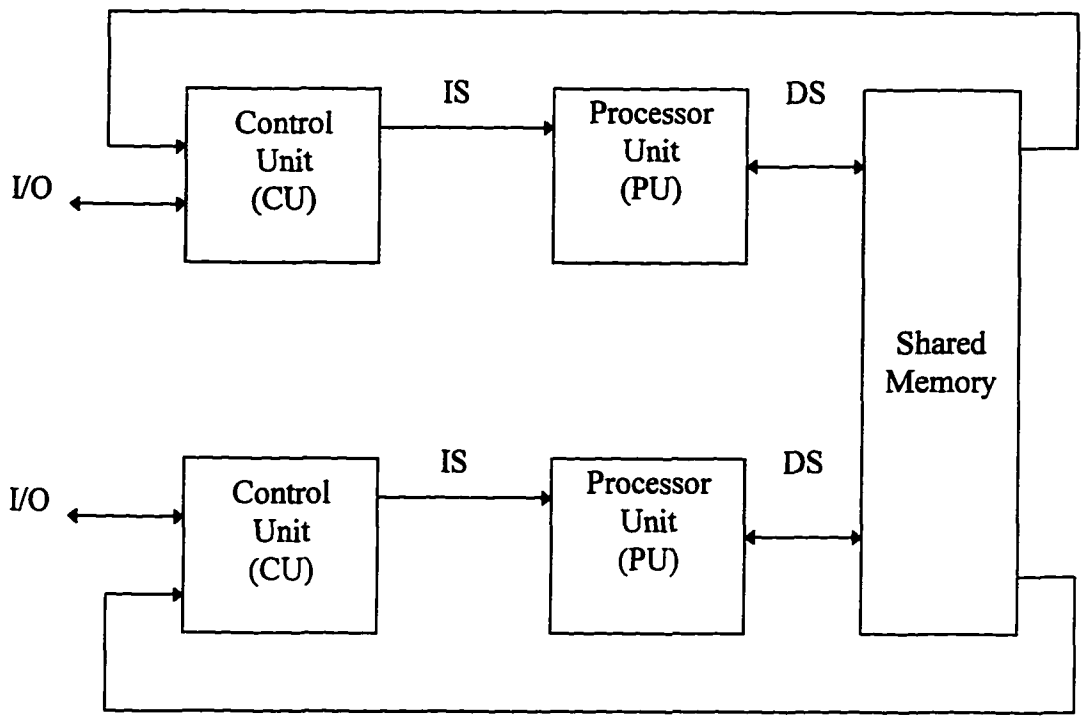


Figure 2.4 An operational model of a SIMD computer



Legend:

IS: Instruction Stream
DS: Data Stream
I/O: Input and Output

Figure 2.5 Operational model of a shared-memory MIMD computer

The operational model of a distributed memory MIMD computer is shown in Figure 2.6. A MIMD computer can be specified by a 2-tuple:

$$M = \langle N, I \rangle$$

where

M is the MIMD model under consideration

N is the number of autonomous computers (also referred to as nodes)

I is the interconnection network

Each node in a MIMD computer consists of a processor and local memory. In addition I/O equipment may be attached to each node. Communication between nodes, to exchange data, is carried out through explicit message-passing. An example of an interconnection network used in these computers is a hypercube in which each node occupies a vertex of multidimensional cube spanning along n dimensions, with two nodes per dimension.

An example of a distributed-memory MIMD computer is Intel iPSC/860. One configuration of iPSC/860 has $2^3 = 8$ nodes. The nodes are interconnected in a hypercube where each node has 3 neighbors. Other representative distributed-memory MIMD computers include nCUBE/2 6480 and Parsys Ltd. SuperNode1000.

2.5 Tool Sets and Case Studies

A set of tools called parallel Reverse Engineering ToolSet (pRETS) which supports semi-automated conversion of FORTRAN programs into Strand foreign language kernels is described in [Harr 93]. Strand is a concurrent programming language based on Prolog. Among the various components of pRETS is a FORTRAN analyzer

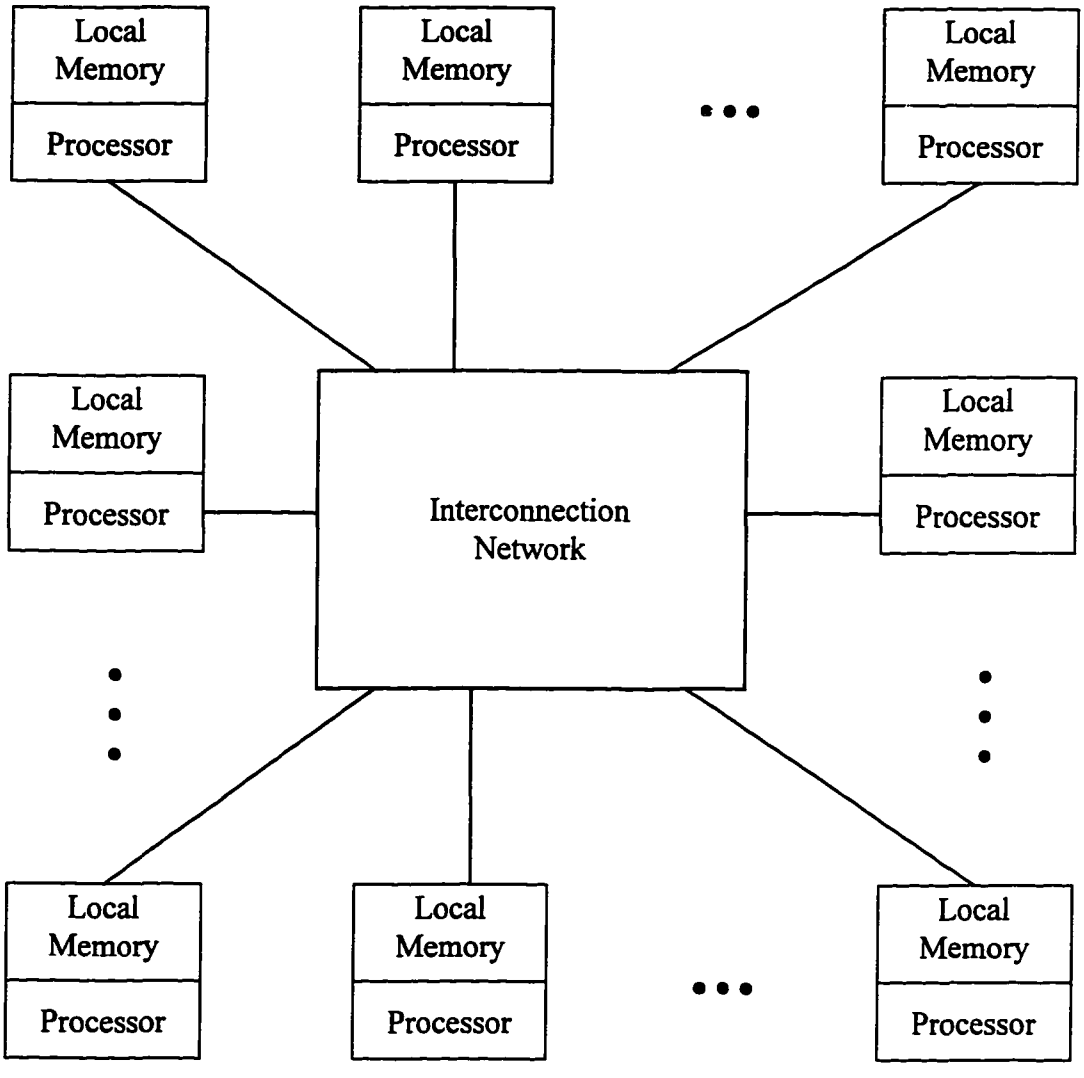


Figure 2.6 Operational model of a distributed-memory MIMD computer

"fa" which builds the general knowledge base of a program in terms of program facts, store facts, and label facts. Using these facts, another component "dataflow" converts the program into a special-purpose dataflow knowledge base. Finally, using the information produced by "dataflow", a set of Prolog rules transform the subroutines of the program into Strand foreign language kernels.

A case study that documents the reverse engineering and reengineering of a twenty year old system is presented in [Kara 95]. The reverse engineering tools were used mainly to "filter" the source code to retrieve the control structure of the program. Restructuring and dependence analysis were done manually. Subsequently, the original system was reengineered into a PVM based parallel implementation.

In the same spectrum, parallelizing compilers migrate code from the uniprocessor to parallel processing environments by performing code-to-code transformations. For example, Parafrase-2 is a high-performance multilingual restructuring parallelizing compiler [Hagh 91]. The main thrust in parallelizing compilers is elaborate dependence analysis, program restructuring, and program transformations. Parallelizing compilers are architecture specific in that program restructuring is aimed at utilizing the available hardware resources on a particular architecture.

2.6 Relevance to the Dissertation

Legacy systems need to be revamped to adapt to current computing trends. Program comprehension and design recovery are primary issues involved in the migration of these systems to new platforms. Reverse engineering research reveals that there is high potential for program comprehension and design recovery. Although

there is widespread research to reverse engineer legacy systems for maintenance, component recovery, and redocumentation [Edwa 93] [Ning 94], very few initiatives exist to migrate legacy systems to parallel platforms using reverse engineering techniques. The research presented in this dissertation is motivated largely due to previous work in the migration of FORTRAN code by [Harr 93]. While case studies described in [Kara 95] are useful, they do not offer automated solutions to help analyze systems in a general way.

Research in the area of dependence analysis may be applied to understand dependences in a large programs. Models such as the one described in [Jack 94] are particularly suited for reverse engineering research. The potential to exploit parallel and distributed system advantages is another reason to migrate legacy systems. Since the reverse engineering process is "knowledge-intensive", research in the field of knowledge-based program analysis may be applied for meaningful and intelligent analysis of information present in current legacy systems.

An explicit design recovery step is absent in parallelizing compilers. Since legacy systems usually lack explicit documentation, the absence of a design recovery procedure is a serious limitation that hinders understandability and maintainability of the original system as well as the parallelized version.

Hence a methodology that provides benefits for the migration process and the maintenance process is essential. This dissertation presents a methodology which is aimed at providing systematic migration of sequential code using principles, methods, and techniques at the intersection of reverse engineering, dependence analysis, and knowledge-based analysis.

The relevance of the related research presented in this chapter to the dissertation is summarized in Table 2.4. The research presented in this dissertation is listed in the first column of Table 2.4. Other columns of Table 2.4 represent previous work [Kara 95] [Harr 93] [Hagh 91]. The rows represent the various characteristics that are considered.

Table 2.4 Related reverse engineering research

	RETK	BNDPKG [Kara 95]	pRETS [Harr 93]	Parafrase-2 [Hagh 91]
Design Recovery	Comprehensive and Automated	Manual	Semi-Automated	No explicit design recovery procedure
Program Dependency Analysis	Elaborate Automated	Manual	Manual	Rigorous Elaborate Automated
Knowledge-Based Approach	Elaborate Automated	None	Elaborate Automated	None
Restructuring	None	Manual	Manual	Rigorous Elaborate Automatic
Potential for Reengineering	Yes	Yes	Yes	Yes

Chapter 3

A Reverse Engineering Methodology for Design Parallelization

This chapter describes a methodology for the migration of code from the uni-processor to the parallel processing environments. The methodology defines a set of phases that serve to systematically approach the problem of code migration. A set of processes are defined for each phase. The processes within the each phase address the issues of design recovery, dependence analysis, and knowledge-based program comprehension as put forth in Chapters 1 and 2. The processes of each phase represent a different approach to the problem of code migration to those employed by parallelizing compilers. Benefits of the methodology are increased overall comprehensibility and improved maintainability of the existing system.

In this chapter, we first present an overview of the three-phase approach of the methodology. Central to the conceptualization of the methodology is the source language used to write existing systems. We therefore present issues related to the selection of a source language for the purposes of establishing a working model for the methodology. Next, we describe each phase and the processes of each phase in detail. One of the unique features of the methodology is the incorporation of automated intelligent support in the migration process. Issues pertaining to the selection of a knowledge-based tool are also discussed.

3.1 Overview of the Methodology

The three main phases of the migration methodology are: Analysis, Synthesis, and Transformation. The three phases represented as a cascaded architecture are shown in Figure 3.1.

The primary objective of the Analysis phase is to extract the original design description of the existing system. The purpose of this phase is to help cope with the complexity of existing systems. Representation of the sequential design description is also one of the important processes of this phase. Large programs, comprised of several modules, typically run over many thousands of lines of program code. According to [Chik 90] and [Ning 94], legacy systems are both voluminous and complex and can only be combated with enough automated support. Therefore, the Analysis phase must be supported by tools with a high degree of automated support to combat complexity.

The Synthesis phase combines the design description (produced by the Analysis phase) associated with the various modules to arrive at a holistic view of the design. Representation of the program dependences in the form of a Program Dependence Graph (PDG) is one of the main processes of this phase. Since a large program potentially has several program dependences, the Synthesis phase must also be supported by tools for automated support.

Finally, the Transformation phase is comprised of processes that require knowledge-based analysis of the information produced by the Analysis and Synthesis phases. Design recommendations for parallel environments are the key output of the Transformation phase. Tools for this phase must be equipped with support for knowledge-based representation, search, and analysis.

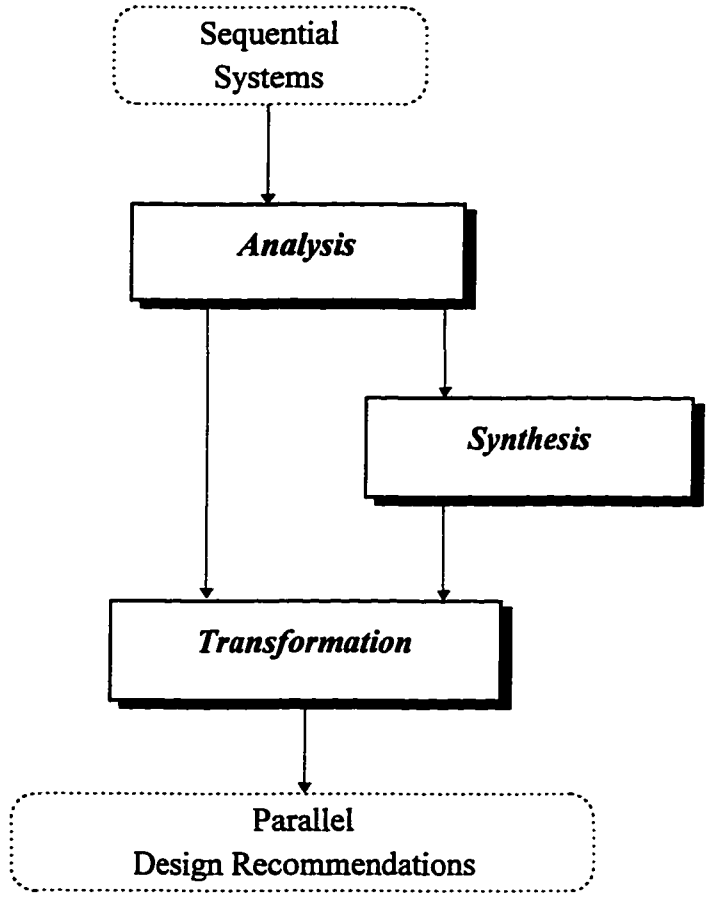


Figure 3.1 Cascaded architecture of the 3-phase migration methodology

3.2 Source Language

FORTRAN, which stands for FORmula TRANslation, is a high-level language that is used to solve problems in science and engineering. The language is now about 40 years old. FORTRAN was developed for the IBM 704 computer and the first compiler was released in April 1957. As computer hardware improved, the FORTRAN language continued to evolve with new refinements and extensions. FORTRAN IV was the fourth version developed between 1960-19962 and was the standard version until 1978 [Seba 93]. Because of the proliferation of FORTRAN, portability of programs across different machines became a problem. In order to achieve uniformity, the American National Standards Institute (ANSI) published FORTRAN standards. As extensions for the standard version continued to be developed, it became apparent that these extensions should be incorporated into a new standard. The updated ANSI FORTRAN standard was called FORTRAN-77 [ANSI 78]. Several versions of FORTRAN have been released since FORTRAN-77, with the most recent being FORTRAN-90.

FORTRAN-77 is a high-level language which provides sequential, selective, and iterative structures allowing operations on data types of integer, real, complex, character, and logical. Features of FORTRAN-77 include implicit data typing, the COMMON statement, and the EQUIVALENCE statement. Implicit data-typing was the only mechanism available in the early versions of FORTRAN. Variables whose names begin with I, J, K, L, M, and N are implicitly integer type, and all others are implicitly real. Later versions of FORTRAN included the IMPLICIT statement which identifies a group of variables whose names begin with a certain letter to be of a

particular type. The COMMON statement establishes a specified portion of memory in which data can be stored and retrieved directly by each of the program modules. The COMMON statement has two forms: unlabeled and labeled. While unlabeled COMMON statement establishes an *unnamed* reserved block of memory, labeled COMMON statements establish a *named* reserved block of memory. Labeled COMMON statements are used in situations where it may be desirable to share one set of variables among some program modules and to share another set among other program modules. Consequently, only one unlabeled COMMON statement is allowed per program, whereas multiple labeled COMMON statements are allowed. The EQUIVALENCE statement provides for the association of variables and arrays, in the same program unit, to share the same memory locations. The EQUIVALENCE statement was once of immense use in reducing the memory requirements of programs, as it allows large arrays to share the same memory locations. However in recent years EQUIVALENCE statements are used infrequently because memories have become less expensive.

Although the use of FORTRAN-77 is widespread, it has several undesirable features: (i) with the absence of constructs for pre-test and post-test, GOTOs are used, potentially introducing severe readability problems, (ii) identifiers are restricted to 6 characters which is also a source for poor readability, (iii) type checking between actual parameters in a subroutine call and the formal parameters in subroutine definition is not done, (iv) recursion is not allowed, (v) separate compilation is not allowed, and (vi) aliasing is caused by the use of COMMON and EQUIVALENCE statements. Since each of the above mentioned features is undesirable for programming in the

large and because equivalent efficiency can be obtained by using other programming languages, the benefits gained by the continuation of use of FORTRAN-77 are questionable. Specifically, the introduction of the object-oriented paradigm and languages such as C++ pose challenges to FORTRAN-77.

FORTRAN-90 which is the "modernized" version of FORTRAN-77, borrowed many programming language concepts from contemporary block structured languages while retaining its original structure. Some of the noteworthy features are recursive procedures, a built-in collection of functions for array operations, dynamic allocation and deallocation of arrays, pointers, control statements like CASE for multiple selection, and abstract data types similar to those of Ada and Modula-2 [Seba 93].

The High Performance Fortran Forum was founded in 1992 to "improve the performance and usability of FORTRAN-90 for computationally intensive applications on a wide variety of machines including massively parallel SIMD and MIMD systems and vector processors." Among the various extensions considered were data distribution, parallel statements, extended intrinsic functions and standard library, extrinsic procedures, parallel I/O statements, and changes in sequence and storage association. One goal of the forum is to provide a High Performance Fortran (HPF) programming model for software developers to write parallel programs for distributed-memory systems. Programs can be written in a single-program, multiple-data (SPMD) style. Information about desired locality or distribution can be provided by annotating the code with HPF data-mapping *directives*. The resultant code can then be compiled using an architecture-specific compiler [Love 93].

The reasoning behind the selection of FORTRAN-77 as the source language for the migration methodology is twofold. Firstly, because of its domain of application and the length of existence, FORTRAN represents a large portion of software currently in use. Secondly, FORTRAN-77 was chosen because it represents the original version of FORTRAN for which standards exist [Holo 83].

3.3 Analysis Phase - Abstraction of the Original Design Description

In this section, we describe the Analysis phase shown in Figure 3.1 in detail. The processes of this phase are: (i) Code Assessment, (ii) Code Re-structuring, (iii) Code Segmentation, (iv) Code Parsing, and (v) Design Aggregation. Figure 3.2 shows a graphical layout of the processes mentioned above.

3.3.1 Code Assessment

Before attempting to realize the sequential design description of the existing system, it is essential to assess the code associated with the existing system. Metrics information enables objective assessment of the software for reliability and maintainability. This process provides metric information like number of lines of code (executable, commented, and blank) and complexity information like McCabe's cyclomatic complexity. The metric information helps software managers to get an initial "feel" for the existing system. In addition, the metric information is useful in the reengineering process by giving software engineers access to the in-code-documentation, if any, of the existing system. One of the reengineering tasks is the

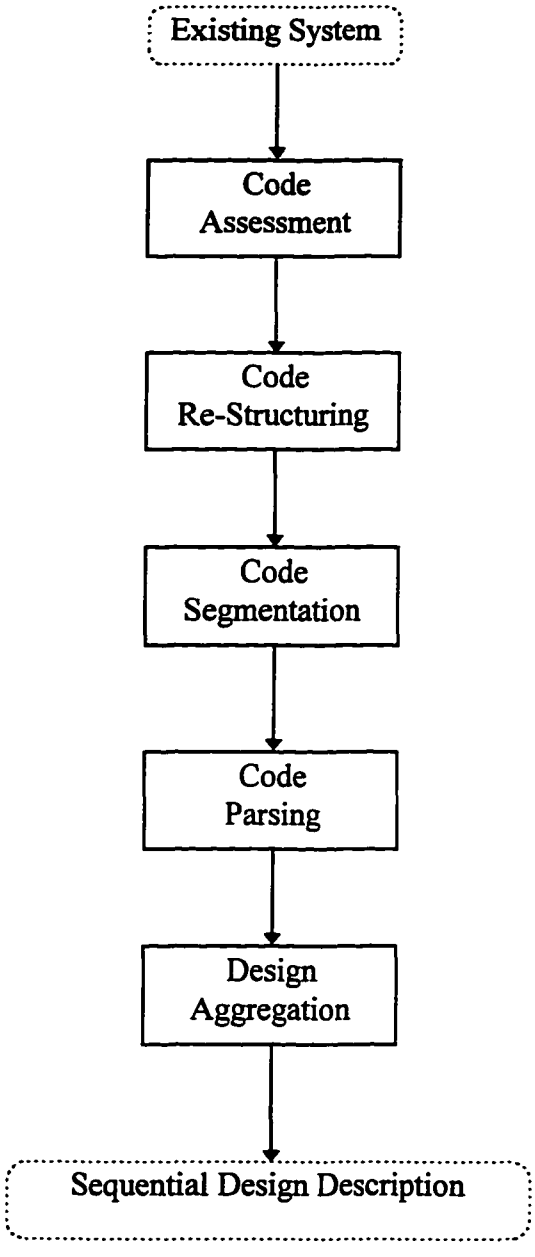


Figure 3.2 Processes of the analysis phase

modification of existing systems to add new functions or capabilities. In such situations, one important issue to be resolved is the impact of side-effects. The availability of in-code-documentation helps to assess specific changes to variables that have a potential to introduce a side-effect.

3.3.2 Code Re-Structuring

A major problem with some old programs is unstructured code, which the computing community refers to as "spaghetti code." The common element which causes code to become unstructured is unconditional branching using GOTO statements. In some languages, especially FORTRAN-77, GOTO statements are the only way to provide control in certain situations. However, frequent use of GOTO statements in large programs affects readability, understandability, and maintainability.

To successfully recover the design information form existing systems, the associated code must be structured. A set of axioms exist in the literature for replacing GOTOs with appropriate loop or alternation constructs [Dijk 76] [Marc 86]. Dijkstra's D-structures can be used to restructure code.

A D-structure is defined as one of the following:

(i) a basic action — an assignment statement, a procedure call, or an input-output statement,

(ii) a sequence of D-structures,

(iii) a conditional structure of the form

if <condition> then
    <d-struct$_1$> else
    <d-struct$_2$> else
endif
where, d-struct$_1$and d-struct$_2$ are D-structures, and

(iv) an iterative structure of the form

```
while <condition> loop
        <d-struct>
end loop
where d-struct is a D-structure.
```

Boehm and Jacopini have proved that any program can be written using only D-structures [Marc 86]. We therefore assume that any program can be written completely using D-structures.

3.3.3 Code Segmentation

In order to analyze the source code associated with a large program, the program must be split into segments. Meaningful segmentation of the source code is based on the syntax and semantics of the underlying programming language. Also, the segmentation activity is designed in way that essentially culminates in the design description of the source code. For example, a FORTRAN-77 program consists of a main program, zero or more subroutines, and zero or more user-defined function modules. These modules collectively define the design of the program. Therefore, the code segmentation process of the Analysis phase employs a segmentation scheme that separates the source code of a FORTRAN-77 program into individual modules. Additional features of the segmentation process includes processes for removing *visual sugar* like tabs, indents, and leading spaces associated with code.

The methodology uses the object-oriented paradigm for the design of the code segmentation process. Each module in the program, the main program, subroutines, and user-defined functions, is considered to be an object. Attributes of each object includes the source code pertinent to the module. The keywords "PROGRAM",

"SUBROUTINE", and "FUNCTION" in the source code identify the beginning of a module and the keyword "END" identifies the end of a module. For syntactically correct programs, all lines of code after the keyword "END" of one module, say *Module-i*, and the beginning of the next module, say *Module-j*, are non-executable lines. These non-executable lines serve as comments or in-code documentation. After observing several programs, we determined that the non-executable lines of code after *Module-i* ends and before *Module-j* begins are usually associated with *Module-j*. Examples of associations include names of author(s), meaning of variable name, and description of algorithms used to implement a module. Hence the code segmentation process associates these non-executable lines of code with *Module-j*. An algorithm for the code segmentation process is presented in Table 3.1

A detailed description of the object-oriented design of the various processes of the Analysis phase are described in Chapter 4.

3.3.4 Code Parsing

The code parsing process analyzes the code associated with individual modules identified by the code segmentation process. An issue related to the code parsing activity is to determine whether or not the programming language allows reserved words in the language design. If the language allows reserved words, then these words may be utilized for the parsing activity. In languages such as FORTRAN-77, where there are no reserved words, all keywords can be assumed to be reserved words.

Table 3.1 Algorithm for code segmentation

```
// Algorithm for Code Segmentation

Main_Program program;                    // Declare Main Program object
Subroutine subs[M];                      // Declare M Subroutine objects
Function funs[N];                        // Declare N Function objects
int sub_no = 0;                          // Index of next subroutine object
int fun_no = 0;                          // Index of next function object
char **lines;                            // Physical lines in source code
int line_no = 0;                         // Index of next line to be read
int k = 0;                               // Index of the next line to be processed
char *token;                             // Declare token to be of type string
int type;                                // Type =1 for Main Program, 2 for
                                         // Subroutine and 3 for Function
boolean flag;                            // Flag is either TRUE or FALSE

do while NOT EOF {
        Read lines[line_no];             // Read line from file and increment
        line_no++;                       // the index
}

do while ( k <= line_no ) {
  Remove_Visual_Sugar ( lines[k] );      // This function removes indents, tabs, and
extra
                                         // spaces in lines[k]
  token = Extract_First_Token ( lines[k] );  // This function returns the first token of
                                         // lines[k]
  if ( token == "PROGRAM" ) {
    Copy lines[k] into program object;
    type = 1;
  }
  else if ( token == "SUBROUTINE" ) {
    Copy lines[k] into subs[sub_no] object;
    type = 2;
  }
  else if ( token == "FUNCTION" ) {
    Copy lines[k] into funs[fun_no] object;
    type = 3;
  }
  k++;

  flag = TRUE;

  do while ( flag == TRUE ) {
    Remove_Visual_Sugar ( lines[k] );
    token = Extract_First_Token ( lines[k] );
```

(table con'd.)

```
switch ( type ) {

case 1:  Copy lines[k] into program object;
             break;

case 2:  Copy lines[k] into subs[sub_no] object;
             break;

case 3:  Copy lines[k] into funs[fun_no] object;
             break;
}

if ( token == "END" ) {
      switch (type) {
        case 2:  sub_no++;
                   break;
        case 3:  fun_no++;
                   break;
      }

      flag = FALSE;
}

k++;
  }
}
```

The code parsing process is the main activity in the realization of an abstraction of the original design description. The code parsing process relies on the syntax and semantics of FORTRAN-77. Keywords play a major role in the abstraction of the design description. Keywords in conjunction with the grammar of FORTRAN-77 define the syntactic constructs of a program. In addition, keywords are useful in extracting semantically related pieces of code. For instance, in FORTRAN-77,

subroutine calls are preceded by the keyword "CALL" and explicit declaration of variables are preceded by data-type keywords like "INTEGER" and "REAL". Therefore the parsing process uses keywords to define the process and data models by using techniques like program slicing [Weis 84]. Slicing is performed at various levels: statement, construct, and block. The parsing process provides the design representation of the original program and includes call graphs, structure charts, hierarchical diagrams, local and non-local variable description, and state change information. The synopsis of the information recovered at the end of the parsing activity is as shown in Figures 3.3, 3.4 and Tables 3.2 and 3.3. An algorithm for the code parsing process is shown in Table 3.4.

3.3.5 Design Aggregation

Once the design associated with the individual modules is obtained, the overall design of the program is obtained by combining information extracted from individual modules. A global call-graph that shows the interactions between the various modules is constructed. Consequently, tools designed for this phase should provide scope to generate graphical representations. These representations will aid in the overall comprehensibility and improved maintainability of the source system. An algorithm for the design aggregation process is shown in Table 3.5.

3.4 Synthesis of the Sequential Design Description

In this section, we describe the Synthesis phase shown in Figure 3.1. The processes of this phase are (i) Module Dependence Analysis and (ii) Synthesis of the Program Dependence Graph (PDG). The design information recovered in the Analysis

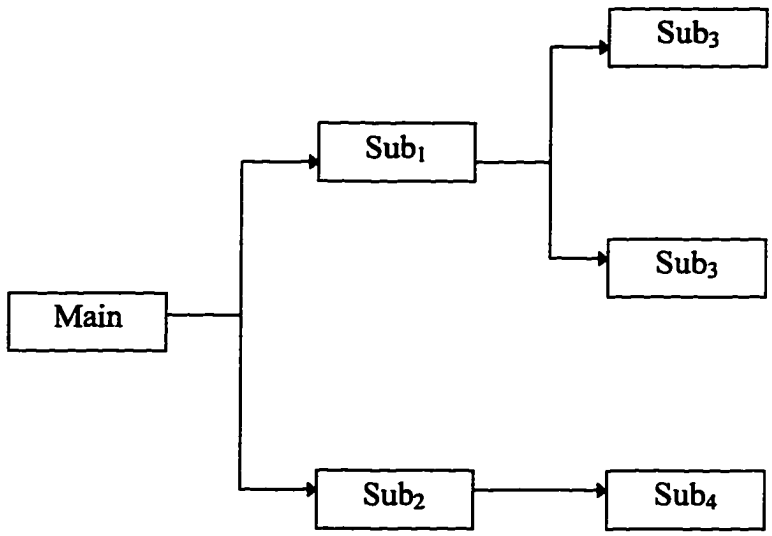


Figure 3.3 Representation of a call graph

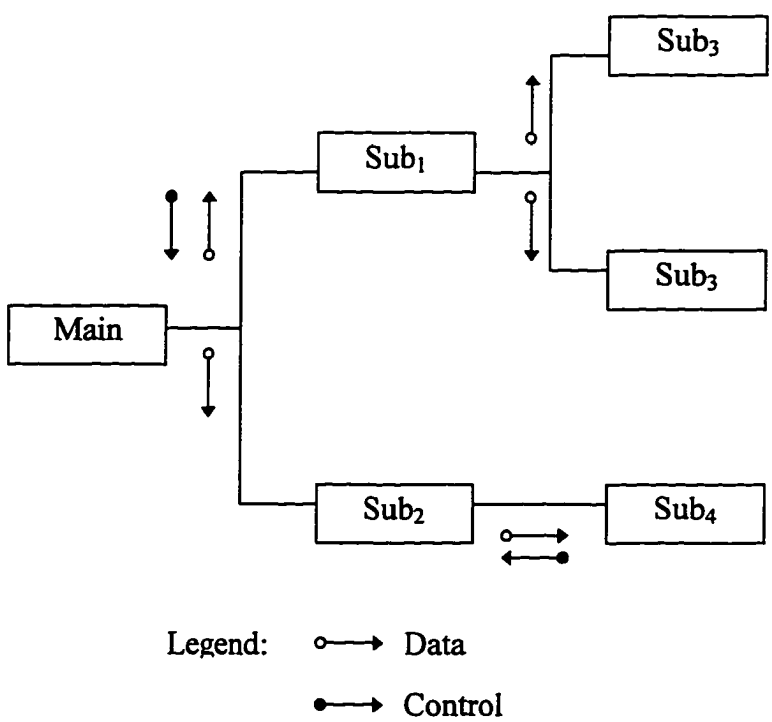


Figure 3.4 Representation of a structure chart

Table 3.2 Representation of local and non-local variable description

Module Name	Variable Name	Data Type	Is Variable an Array	Dimensions if Array	Scope of the Variable
		• INTEGER • REAL • DOUBLE PRECISION • CHARACTER • LOGICAL	• Yes • No		• Local • Global • Formal

Table 3.3 Representation of state change information

Module Name	State Change Variable Name	Variable Names involved in the State Change	Type	Semantic Nature
			• Simple • Iterative • Conditional	• Direct Assign-ment Statement • Assignment via a subroutine call

Table 3.4 Algorithm for the code parsing process

```
// Algorithm for code parsing.
// This algorithm is used to parse the source associated with each component.

Main ( )
{
        Component.Extract_Calls ( );
        Component.Extract_Variable_Description ( );
        Component.Extract_State_Changes ( );
}

Component::Extract_Calls ( )
{
        int     k = 0;                  // Index of next line to be parsed
        char    *token;                 // String that holds first token of a line
        char    *rest;                  // String that holds the rest of the line

        do while ( k < MAX_LINES_IN_COMPONENT ) {
                token = Extract_First_Token ( Component.lines[k] );
                rest = Rest ( Component.lines[k] );

                if ( token == "CALL" )
                        Extract_Actual_Parameters ( rest );

                Save_Subroutine_Call_Information;
        }
}

Component::Extract_Variable_Description ( )
{
        int     k = 0;                  // Index of next line to be parsed
        char    *token;                 // String that holds first token of a line
        char    *rest;                  // String that holds the rest of the line

        do while ( k < MAX_LINES_IN_COMPONENT ) {
                token = Extract_First_Token ( Component.lines[k] );
                rest = Rest ( Component.lines[k] );

                if ( token == "REAL" or token == "INTEGER" or
                   token == "DOUBLE" or  token == "PARAMETER" or
                   token == "CHARACTER" )
                        Extract_Variables ( rest );
                else if ( token == "COMMON" ) {
                        if ( labeled_COMMON == True ) {
                                Extract_Label_Name ( rest );
                        }
```

(table con'd.)

```
                    Extract_Common_Variables ( rest );
                }
                Save_Subroutine_Call_Information;
        }
}

Component::Extract_State_Changes ( )
{
        int     k = 0;                  // Index of next line to be parsed
        char    *token;                 // String that holds first token of a line

        do while ( k < MAX_LINES_IN_COMPONENT ) {
                token = Extract_First_Token ( Component.lines[k] );

                if ( token != Keyword )
                        Process_Assignment ( Component.lines[k] );
                else if ( token == "READ")
                        Process_READ ( Component.lines[k] );
                else if ( token == "IF" ) {
                        Process_IF ( Component.lines[k] );
                        Build_IF_List ( Component.lines[k] );
                }
                else if ( token == "DO" )
                        Build_IF_List ( Component.lines[k] );
        }
}

Component::Process_Assignment ( char *Passed_Line )
{
        char    *L_Value;
        char    *R_Value;

        L_Value = Extract_L_Value ( Passed_Line );
        R_Value = Extract_R_Value ( Passed_Line );
        Extract_State_Change_Variable ( L_Value );
        Extract_Function_Calls ( R_Value )
        Extract_RHS_Variables ( R_Value )

        Save_State_Change_Attributes ( );
}
```

phase is used in the Synthesis phase. Research in the area of dependence analysis, as outlined in Chapter 2, has concentrated on determining dependences directly from code. Parallelizing compilers also determine dependences directly from code. In contrast, the processes defined in the Synthesis phase focus on the analysis using design representations recovered in the Analysis phase. The output of the Synthesis phase is the Program Dependence Graph (PDG) which makes explicit both data and control dependences. Figure 3.5 shows a graphical layout of the processes of the Synthesis phase.

One major issue in the design of processes of the Synthesis phase is the selection of an appropriate PDG representation. The choice of the PDG representation depends on the desired granularity of the dependences. For reverse engineering research, the model proposed by [Jack 94] is appropriate because of its clean representation scheme. The model in [Jack 94] views procedures as modular and dependences as fine-grained.

3.4.1 Module Dependence Analysis

The objective of the module dependence analysis is to determine the data and control dependences of each module identified by the Analysis phase. The state change information recovered in the Analysis phase is used as a starting point for this process. Using the notation defined in [Jack 94], sites are built for each state change. By definition, a site conceptually relates to a state change to a variable in the state change information table (Table 3.3). A site is a combination of use lists and definition lists. Special sites named *entry* and *exit* have only definition and use lists,

Table 3.5 Algorithm for design aggregation

```
// Algorithm for Design Aggregation


// Definition of link
struct link
        //      one element of a list
{
        char    *component_name;        // Name of the module
        short   num_children;           // Number of children
        char    **names_of_children;    // Names of children
        link    *next;                  // Pointer to next link
};

//      Definition of class LinkList
class LinkList                          // A List of Links
{
        private:
                link    *first;         // Pointer to first link

        public:
                LinkList()              // No-argument constructor
                {
                        first = NULL;   // Initialize first link to NULL
                }

                Build_Global_Call_Graph();      // Member funcion which builds
                                                // the Global Call Graph

                Display_Global_Call_Graph();    // Member funcion which
                                                // displays the Global Call Graph
};

//      Main Algorithm

main ()
{
        LinkList ll;        // Create An Instance "ll" of the  Class LinkList

        ll.Build_Global_Call_Graph();     // Call to member function

        ll.Display_Global_Call_Graph();   // Call to member function

}
```

(table con'd.)

```
//      Memeber Function Build_Global_Call_Graph
LinkList::Build_Global_Call_Graph()
{
        char    *name_of_curr_comp;     // Name of the Current Component
        char    *comps;                 // Names of the Components called by
                                        // Component 'name_of_curr_comp'
        int     num;                    // Number of Components called by
                                        // Component 'name_of_curr_comp'

        For All Components in the Design  {

                name_of_curr_comp = Get Name of the Component;
                comps = Get Names of all components called by 'name_of_curr_comp';
                num = Get Number of all components called by 'name_of_curr_comp';

                //      Create New Link

                link    *newlink = new link;

                // Add Information to Link

                newlink->component_name = name_of_curr_comp;
                newlink->names_of_children = comps;
                newlink->num_children = num;

                newlink->next = first;

                first = newlink;
        }
}

//      Memeber Function Display_Global_Call_Graph
LinkList::Display_Global_Call_Graph()
{
        link    *current = first;

        while ( current != NULL ) {
                Print   current->component_name;
                Print   current->num_children;
                Print All current->names_of_children;

                current = current->next;
        }
}
```

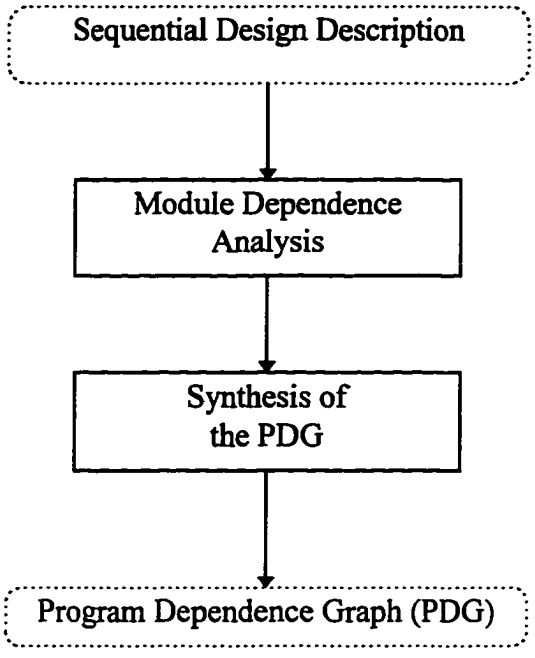


Figure 3.5 Processes of the synthesis phase

respectively. For all other sites, use list variable(s) are the variable name(s) involved in the state change and the definition list variable(s) are the variable name(s) to which the state change is occurring. In other words, use list variable name(s) correspond to the third column and definition list variable name(s) correspond to the second column of Table 3.3 respectively. Some programming language issues are in order here. Since FORTRAN allows only one variable to be defined in an assignment statement, only one definition list variable for every site is defined. However, we modified this definition slightly to include the loop variable in the definition list in the site for the last state change in the loop scope. The meaning of the special symbols, defined in

[Jack 94], are retained in the notation. The special symbols are "τ, a temporary that holds the result of a conditional test, γ, which represents a constant, and ε, which stands for 'execution'" [Jack 94]. In order to obtain a comprehensive summary of the control dependences, we have extended the meaning of τ to include τT (true part of the conditional test) and τF (false part of the conditional test). The original notation defined in [Jack 94] with modifications and extensions as applied to the state change information is described below:

StCh = State Change = {LV, RVs, Type}
LV = Left Hand Side Variable
RVs = Right Hand Side Variable(s)
Type = {Simple, Iterative, Conditional}
UL = Use Lists = StCh.RVs ∪ {γ, ε}
DL = Definition Lists = {StChs.LVs ∪ LoopVariable ∪ {τ, τT, τF}}
Site = {UL × DL} ∪ {entry, exit}
$\tau_x = \{\tau, \tau T, \tau F\}$

Three relations which represent the dependences are *fd* for dataflow dependences (forward flow), *ld* for loop dependences (backward flow), and *cd* for control dependences. Using the notation described above, the three relations are defined as:

$fd \subseteq \{((x, i), (y, j)) \mid x \in DL \wedge y \in UL \wedge x = y \wedge i, j \in Site \wedge i < j\}$
$ld \subseteq \{((x, i), (y, j)) \mid x \in DL \wedge y \in UL \wedge x = y \wedge i, j \in Site \wedge i > j\}$
$cd \subseteq \{((\varepsilon, i), (\tau_x, j)) \mid \varepsilon \in UL \wedge \tau_x \in DL \wedge i, j \in Site \wedge i < j\}$

Since in FORTRAN, it is possible to declare variables implicitly, all variables defined at the *entry* are only the variables that are explicitly declared. All the variables used at the *exit* are both variables that explicitly and implicitly defined.

We define the term *abstract site* for every module except the main program. An abstract site's use list variable(s) are the non-local variables of the module which

include formal parameters (if any) and COMMON variables. The definition list variables of the abstract site are the same non-local variables to which a state change occurs within the module. Abstract sites are useful in determining whether or not two modules can be parallelized. An algorithm for the creation of sites is listed in Table 3.6. An algorithm for computing the data dependences from site information is presented in Table 3.7. An algorithm for computing the control dependences from site information is given in Table 3.8.

3.4.2 Construction of the PDG

After the dependence analysis is carried out on each module, the results of the analysis is combined to form the PDG. The call graph is used as the starting point to build the PDG. Since the call graph is a tree representation, the PDG is built starting at the leaves and moving to the root. The outcome is that the PDG is essentially the call graph but with explicit dependence analysis. Calls made to subroutines (or modules) from a given subroutine are replaced by the called subroutine's abstract site.

Since a large program might consist of several dependences, tools designed for this activity should be capable of generating and displaying graphical representations. Appropriate color coding schemes should be employed to differentiate data and control dependences. Thus, the processes of the Synthesis phase combine the dependence analysis of all the modules in the design to construct an overall view of the existing system.

Table 3.6 Algorithm for creating sites and their use and definition lists

```
// Algorithm for Creating Sites and their Use and Def Lists
Main
   Refers Sts[i]
   for every Sts[i] do
      if Sts[i].Category is NULL
         call Insert_Into_Def_List(i);
         call Separate_RHS_and_Insert_Into_Use_List(i);
      else     // Conditional detected either DO or IF
         int return_code;   // return code is 1 if Assignment
                            // Statement to be processed and is 0 if NOT

         return_code = call Ascertain_Type_of_Conditional_and_Build_Tau_Sites(i);
         if return_code is 1
            call Insert_Into_Def_List(i);
            call Separate_RHS_and_Insert_Into_Use_List(i);
         end if
      end if
   end do
End Main

proc int Ascertain_Type_of_Conditional_and_Build_Tau_Sites(int i)
   int   j = 0;    // To keep track of Sts[i].Category char by char
   int   number;
   char  temp[80];
   int   t = 0;

   while isdigit(Sts[i].Category[j])
         j++;
   while (Sts[i].Category[j] != '-')
         temp[t++]= Sts[i].Category[j++];
         temp[t]='\0';

         number = atoi(temp);
         j++;      // skip *

         char      rest[80];
         int       r=0;

         while (Sts[i].Category[j] != '\0') {
            rest[r++] = Sts[i].Category[j++];

         rest[r] = '\0';

         call Build_Tau_Sites(i, number, rest));

         if(rest[0] == 'T')
            return 1;
         else
            return 0;
end proc
```

(table con'd.)

```
proc Build_Tau_Sites(int i, int number, char *rest)
  char  ch;
  int   next_number;

  ch = rest[0];

  switch(ch) {
        case 'D': next_number = Iteratives[number].Prev_Id;
                  break;
        case 'T', 'F':      next_number = Conditionals[number].Prev_Id;
                            break;
        default:  break;
  }

  char  *p;
  char  *new_rest;
  char  copy_of_rest[80];
  char  str[3];

  sprintf(str,"%c\0", ch);
  strcpy(copy_of_rest, rest);
  p = strtok(copy_of_rest, str);
  new_rest = strtok(NULL, "\0");
  if (new_rest)
        call Build_Tau_Sites(i, next_number, new_rest);
  else {
        switch(ch) {
              case 'D': if (number >= do_lists_built_so_far) {
                          call Build_DO_Tau(i, number);
                          do_lists_built_so_far++;
                        }
                        break;
              case 'T': if (number >= if_lists_built_so_far) {
                          call Build_IF_Tau(i, number);
                          if_lists_built_so_far++;
                        }
                        break;
              default:  break;
        }
  }

end proc
```

(table con'd.)

```
proc Build_DO_Tau(int i, int number)

   call Insert_Into_Def_List(i);
   if (atoi(Iteratives[number].Intial_Value) != 0)
         strcpy(Sites[num_sites].uls[num_uls++].name, Iteratives[number].Intial_Value)

   strcpy(Sites[num_sites].uls[num_uls++].name, "*E*")
   num_sites++;

   char  ul[8];
   sprintf(ul, "DO [%d]\0", number);
   strcpy(Sites[num_sites].dls[num_dls++].name, "*Tau*")
   strcpy(Sites[num_sites].uls[num_uls++].name, ul)
   num_sites++;

end proc

proc Build_IF_Tau(int i, int number)

   char  ul[8];
   sprintf(ul, "IF [%d]\0", number);
   strcpy(Sites[num_sites].dls[num_dls++].name, "*Tau T*")
   strcpy(Sites[num_sites].dls[num_dls++].name, "*Tau F*")
   strcpy(Sites[num_sites].uls[num_uls++].name, ul)
   num_sites++;

end proc
```

Table 3.7 Algorithm for computing data dependences

```
// Algorithm for computing Data Dependences
Main
for all sites identified by s starting from 1 {
     for each Use List variable identified by u starting from 0 to < (num_uls-1) {
          int       found = 0;
          if uls[u].name is 0 or *Gamma*
                    do nothing
          else if uls[u].name is DO[x]
                    Process_DO_DataDep(s)
          else if uls[u].name is IF[y]
                    Process_IF_DataDep(s)
          Check(u, name, s);

     }
  }
End Main
```

(table con'd.)

```
proc Check_Name(int u, char *name, int s)
{
   for (s'=s - 1; s'=0; s'--) {
      for(d=0;d<Sites[s'].num_dls;d++) {
          if name == Sites[s'].dls[d].name {
             found = 1;
             ud_site = s';
             break;
          }
      }
      if (found)
          break;
   }

   Sites[s].uls[u].ud[num_du].name = name;
   Sites[s].uls[u].ud[num_du++].site_number = ud_site;
}
end proc

proc Process_DO_DataDep(int s)

   index = Get_Index(s);
   if (Iteratives[index].CV is not 0 or number)
          Check_Name(0, Iteratives[index].CV, s);
   if (Iteratives[index].IV is not 0 or number)
          Check_Name(0, Iteratives[index].IV, s);
   if (Iteratives[index].LIM is not 0 or number)
          Check_Name(0, Iteratives[index].LIM, s);
   if (Iteratives[index].STEP is not 0 or number or NULL)
          Check_Name(0, Iteratives[index].STEP, s);
end proc


proc Process_IF_DataDep(s)

   char   **vars;
   int    var_count = 0;

   index = Get_Index(s);
   Collect_Varaibles(Conditionals[index].Logical Expression, vars);

   while(vars[var_count][0] != 0)
          Check_Name(0, vars[var_count], s);

end proc
```

Table 3.8 Algorithm for computing control dependences

```
// Algorithm for computing Control Dependences.

// This member function will be invoked after the sites are built.

Main
if no do loops or if conditions
        no control dependeces
        return;
else
  Set j=0
  while j < num_sites
    search for DO[x] or IF[y].
    case DO[x] is found :
        Extract x.
        Index into the xth's Iterative structure.
        Get Iteratives x's Last Line.
        Get site number (k) (from j) whose site_id is equal to (Iteratives x's (Last Line -1)
        For sites s from (j+1) to k add control dependence to site s where Edge components
          have name=*Tau* and site_number = j
    case IF[y] is found :
        Extract y.
        Index into the yth's Conditional structure.
        Get Conditional y's True Part Last Line.
        Get Conditional y's False Part Last Line.
        Get site number (k) (from j) whose site_id is equal to
          (Conditional y's (True Part Last Line) || (True Part Last Line -1)
        if Conditional y's site_id equals True Part Last Line then {
          For sites s from (j+1) to k-1 add control dependence to site s where Edge
            components have name=*Tau T* and site_number = j
        }
        else if Conditional y's site_id equals (True Part Last Line - 1) then {
          For sites s from (j+1) to k add control dependence to site s where Edge
            components have name=*Tau T* and site_number = j
        }
        Get site number (m) (from k) whose site_id is equal to
          (Conditional y's (False Part Last Line -1))
        For sites s' from (k) to m add control dependence to site s' where Edge
          components have name=*Tau F* and site_number = j
      end case
  Increment j;
  continue while loop.
End if
end Main
```

3.5 Design Recommendations for Parallel Environments

In this section, we describe the Transformation phase of Figure 3.1. The Transformation phase uses a knowledge-based approach to arrive at the parallel design recommendations. The processes of this phase are (i) knowledge acquisition and knowledge representation, (ii) selection of an inference procedure, and (iii) representation of the parallel design recommendations. Figure 3.6 shows a graphical layout of these processes.

3.5.1 Knowledge Acquisition and Knowledge Representation

Knowledge is the key element in the knowledge-based approach. Two problems specific to the knowledge-based approach are knowledge acquisition and knowledge representation. Knowledge acquisition is the process of building the knowledge base of the system. Knowledge representation is the process of encoding and storing the knowledge base of the system.

3.5.1.1 Knowledge Acquisition

To arrive at the parallel design recommendations, the migration methodology knowledge of the design of the existing system forms the knowledge base of the problem domain. This knowledge base design falls under the class of an embedded application design. Embedded knowledge-based systems are systems that used as part of some larger system [Sell 85]. Since the Transformation phase is one part of the migration methodology, the design of the knowledge base falls under the class of embedded knowledge-based systems. One issue related to the knowledge acquisition for an embedded system is that most of the knowledge should be available *a priori*.

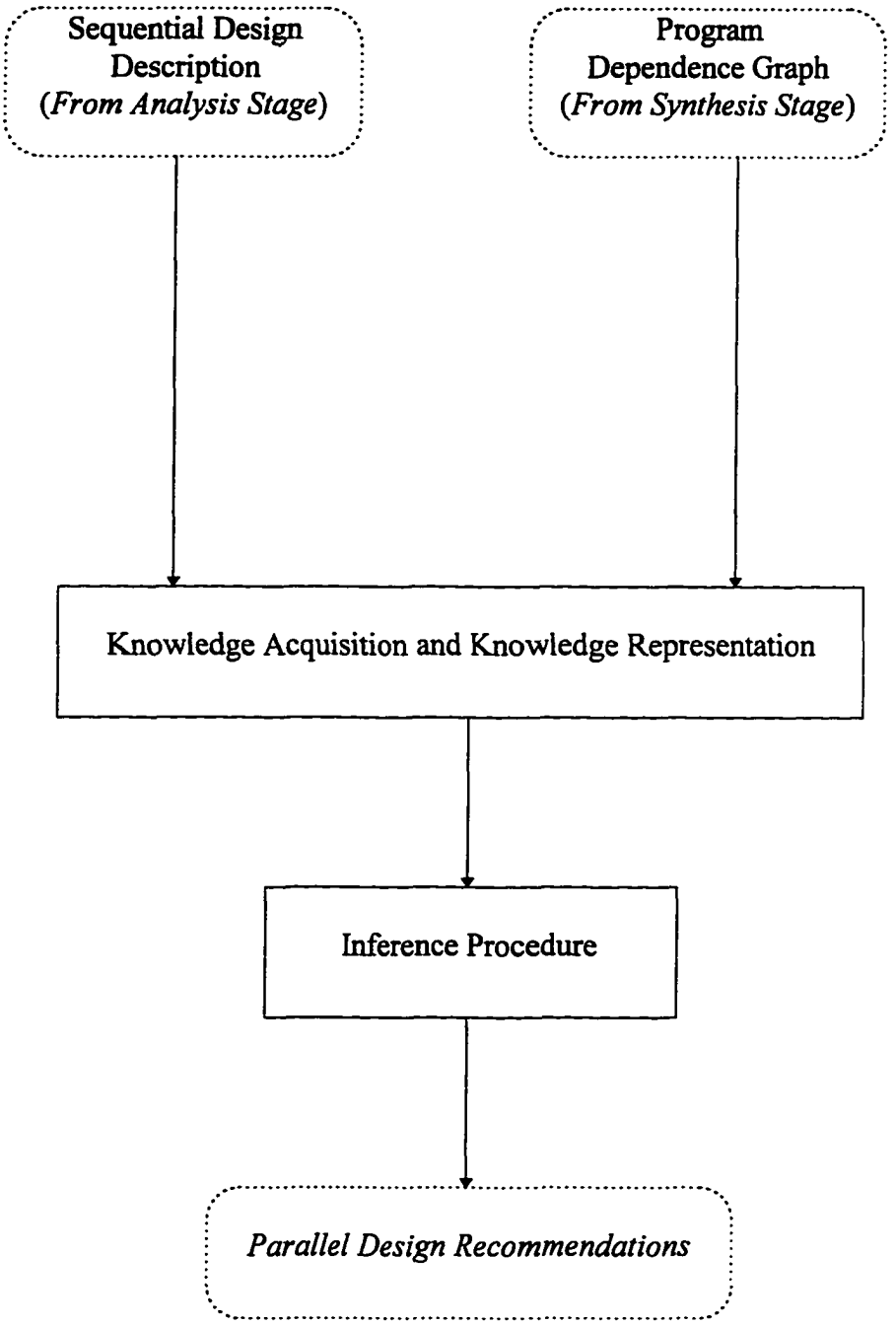


Figure 3.6 Processes of the transformation phase

The Analysis phase and Synthesis phase of the migration methodology provide known facts about the design and dependence analysis *a priori* to the Transformation phase.

3.5.1.2 Knowledge Representation

Knowledge representation deals with the issue of encoding the knowledge acquired through the knowledge acquisition process. Although several techniques exist in literature for the representation of structured knowledge, these techniques can be coarsely divided into two types: declarative and procedural. In practice, most representations make use of a combination of both the types. In addition, the C Language Interface Production System (CLIPS), a knowledge-based programming environment, allows knowledge to be represented in the form of objects using the object-oriented paradigm.

Declarative knowledge representation, which uses predicate logic, is composed of a static collection of facts and a small set of procedures for manipulating the facts. In contrast, procedural knowledge representation uses procedures as the major representation mechanism. Object-oriented knowledge representation uses objects supported by classes, message-handlers, abstraction, encapsulation, inheritance, and polymorphism.

The choice of an appropriate knowledge representation scheme for the Transformation phase is an object-oriented knowledge representation. For effective representation of structured knowledge, four properties need to be satisfied: representational adequacy, inferential adequacy, inferential efficiency, and acquisitional efficiency [Rich 83]. The object-oriented knowledge representation meets all the above mentioned properties. Another reason for choosing the object-oriented

knowledge representation is because the processes of the Analysis phase and Synthesis phase follow an approach that closely mimics the object-oriented approach.

Using an object-oriented knowledge representation scheme, the sequential design description and the program dependence graph are represented as objects. An object-oriented analysis of the problem domain is performed to identify objects, associations, attributes, and methods. Classes are formed to serve as templates to the identified objects. Inheritance relationship are defined to form appropriate class hierarchies. Details of the knowledge representation schemes are provided in Chapter 4.

3.5.2 Inference Procedure

In addition to the facts about the problem domain, the knowledge-base consists of a collection of rules to pattern match on existing facts, procedures, and objects to deduce new facts. A rule is a combination of zero or more *if* patterns and zero or more *then* patterns. A rule can be defined as:

$Rule_i$: < *if* [Pattern]>* ; left-hand Side
=>
< *then* [Pattern]>* ; right-hand Side

Conventionally, the left-hand side is referred to as the antecedent and the right-hand side is referred to as the consequent. Figure 3.7 shows a graphical notation for a rule. If all of the antecedents of a rule are true, then all of the consequents are true. In deduction systems, the consequents usually specify new facts that could be derived from existing facts.

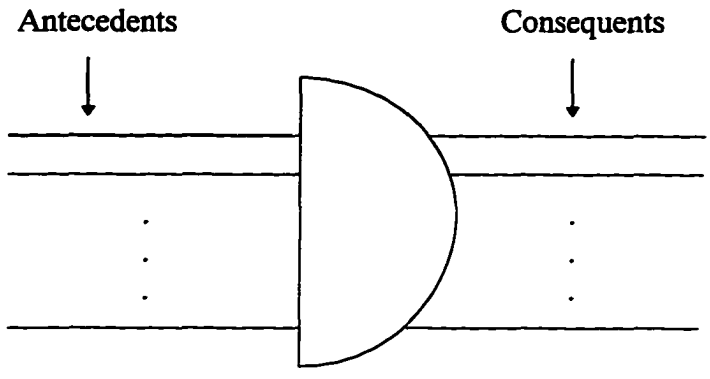


Figure 3.7 Graphical notation of a rule

In the context of the methodology, a typical rule in the knowledge-base that would assist in the parallelization process is to pattern match the formal parameters and COMMON parameters of two modules to determine whether or not the modules can be parallelized. If the design facts of the two modules are available, the rule shown in Figure 3.8 may be used to ascertain their parallelization potential.

Rules like the rule shown in Figure 3.8 are formulated to cover different scenarios by which the parallelization potential between modules and the parallelization potential within a module can be ascertained. These rules are given in Chapter 4 after the scenarios are defined. Once the rules are in place, an appropriate deduction procedure should be selected. There are two possible directions at this juncture: forward chaining and backward chaining.

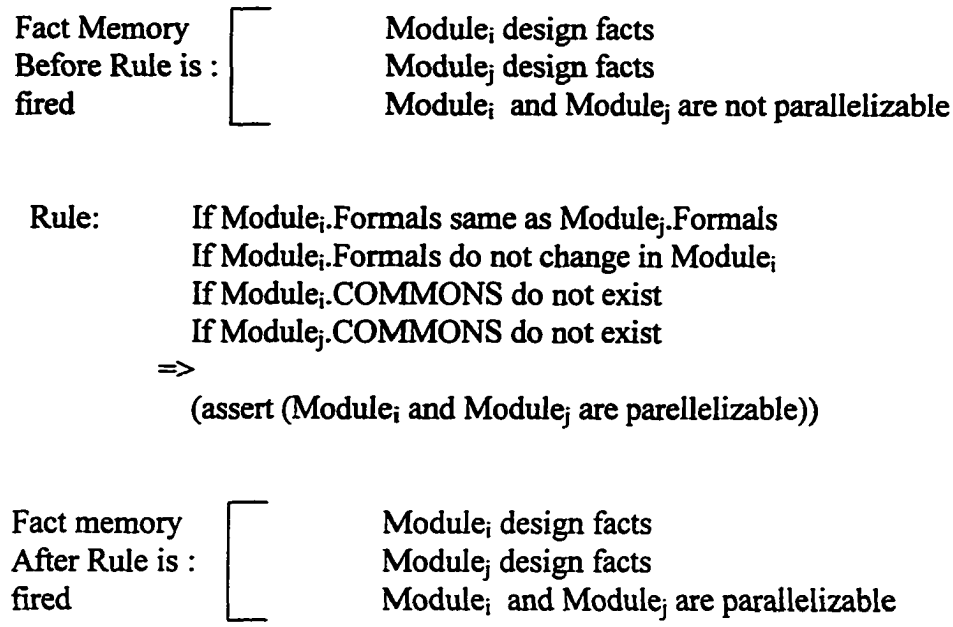


Figure 3.8 A typical rule in the knowledge-base of the migration methodology

Forward chaining is the process of moving from the antecedents to the consequents. The antecedents identify appropriate situations for the deduction of a new assertion. It is like working forward from the given to the conclusion. The mechanism of forward chaining is as follows:

> *Whenever all the antecedents of a rule are satisfied, the rule is triggered. Whenever a triggered rule leads to a new assertion, it is fired. When no more rules can fire the procedure terminates.*

It is possible that more than one rule is triggered at the same time. In such cases a conflict-resolution strategy is needed to decide which rule should fire. Most knowledge-

based programming environments have some sort of conflict-resolution strategy in place. After the forward chaining procedure terminates, the fact base consists of the conclusions.

Backward chaining is the process of moving backward from the goal state to the initial state. The mechanics of backward chaining, as described in [Wins 92] is as follows :

> *Form a hypothesis. Whenever a rule's consequent matches the current hypothesis, try to support each of the rule's antecedents by pattern matching assertions in the fact base or by backchaining through another rule, thus creating a new hypothesis. If all the rule's antecedents are satisfied, conclude that the hypothesis is true.*

For the design of the inference engine of the Transformation phase of the methodology, an important issue to resolve is whether to chain forward or backward. [Sell 85] and [Wins 92] describe several rules of thumb that may be used to decide the direction of chaining. One such rule of thumb is related to the availability of the facts. Forward chaining is recommended if all the facts ever going to be needed are available *a priori*. Backward chaining is recommended for conversational systems where the user answers questions and builds the fact base. Since all known facts are available *a priori*, the inference engine uses forward chaining.

3.5.3 Representation of Parallel Design Recommendations

The purpose of a representation is to present a true interpretation. The parallel design recommendations need to be represented in a form that enhances the overall understandability. Although textual description of the recommendations would be useful, for example, for generating pseudo-code for a parallel implementation, a graphical layout of the parallelizable segments of the original design is more useful for

comprehension purposes. Therefore we chose to define a graphical notation called Parallel Design Recommendations (PDR). PDR makes explicit parallelizable segments in the original design using the classical fork and join representation. PDR uses ovals to represent subroutines with the name of the subroutine appearing inside the oval. Subscripts to subroutine names indicate that they may be executed in parallel. Rectangles represent processes that are indicative of partitioning large computations into sub-computations. Rectangles with rounded edges represent any additional processes that need to be incorporated into the design to allow parallelization such as initializing variables, fork, and join. As an illustrative example, Figure 3.9 shows the call-graph of a sequential design. In this example, the main program calls two subroutines SETBV and SETIV. SETBV in turn calls EXACT six times. The processes of the migration methodology deduce that the six calls to subroutine EXACT from subroutine SETBV are independent of one another and hence can be parallelized. The corresponding PDR is shown in Figure 3.10

3.6 Summary

This chapter presented an overview of a methodology for systematic migration of sequential code to parallel processing environments. The distinct phases of the methodology — Analysis, Synthesis, and Transformation — combine current techniques and new techniques and principles in the areas of reverse engineering, dependence analysis, and knowledge-based analysis. The processes for each phase in the methodology are automatable. Chapter 4 describes an automated reverse engineering toolkit (RETK) that serves to establish a working model for the methodology.

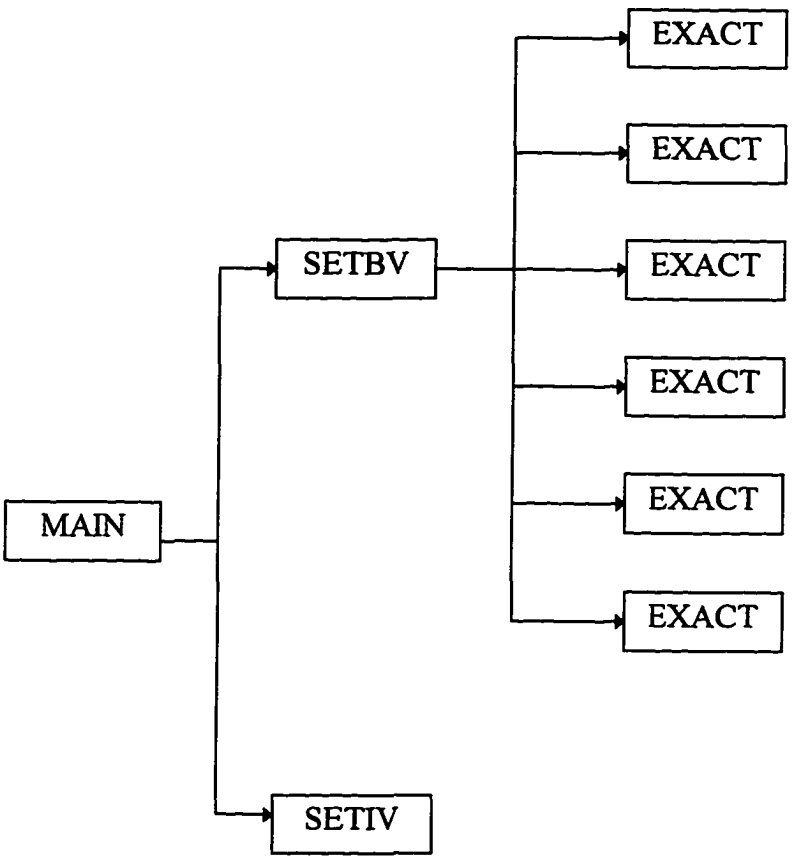


Figure 3.9 Example call graph of a sequential design

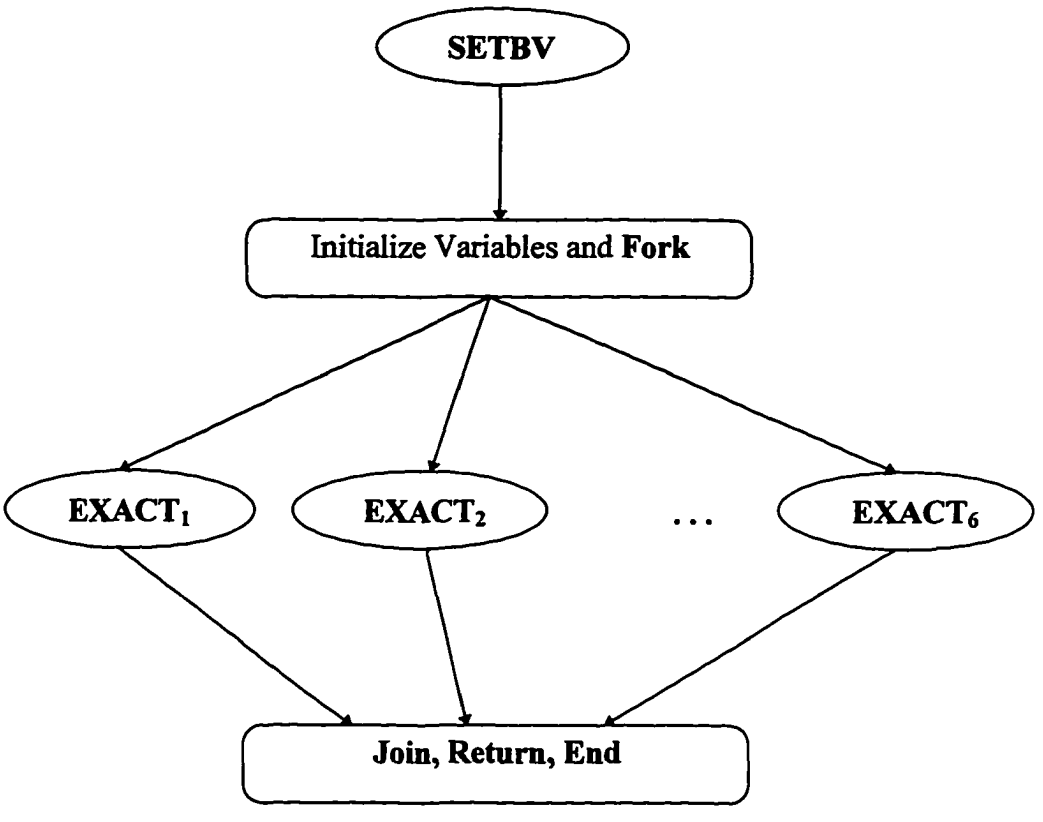


Figure 3.10 PDR representation of subroutine SETBV of Figure 3.9

Chapter 4

RETK: A Reverse Engineering Toolkit for Design Parallelization

This chapter describes the reverse engineering toolkit (RETK) that was developed to establish a working model for the migration methodology described in Chapter 3. Tools were designed and developed for the Analysis and Synthesis phases of the methodology. NASA's C Language Interface Production system (CLIPS), a knowledge-based programming environment, was used to design and implement the Transformation phase of the methodology. Information provided by the toolkit aids in increased understanding of the existing system. RETK provides a comprehensive toolkit spanning all the three phases of the migration methodology.

In this chapter, we first present a system overview of RETK. Next, we describe the design and implementation of the various components of RETK that serve to systematically migrate sequential code to parallel architectures. We briefly describe CLIPS, the knowledge-based tool used in RETK.

4.1 System Overview

Figure 4.1 shows the system overview of RETK. The main components of RETK are 1) an Information Extractor, 2) a Dependence Analyzer, and 3) an intelligent Design Assistant. The Information Extractor (IE) uses reverse engineering

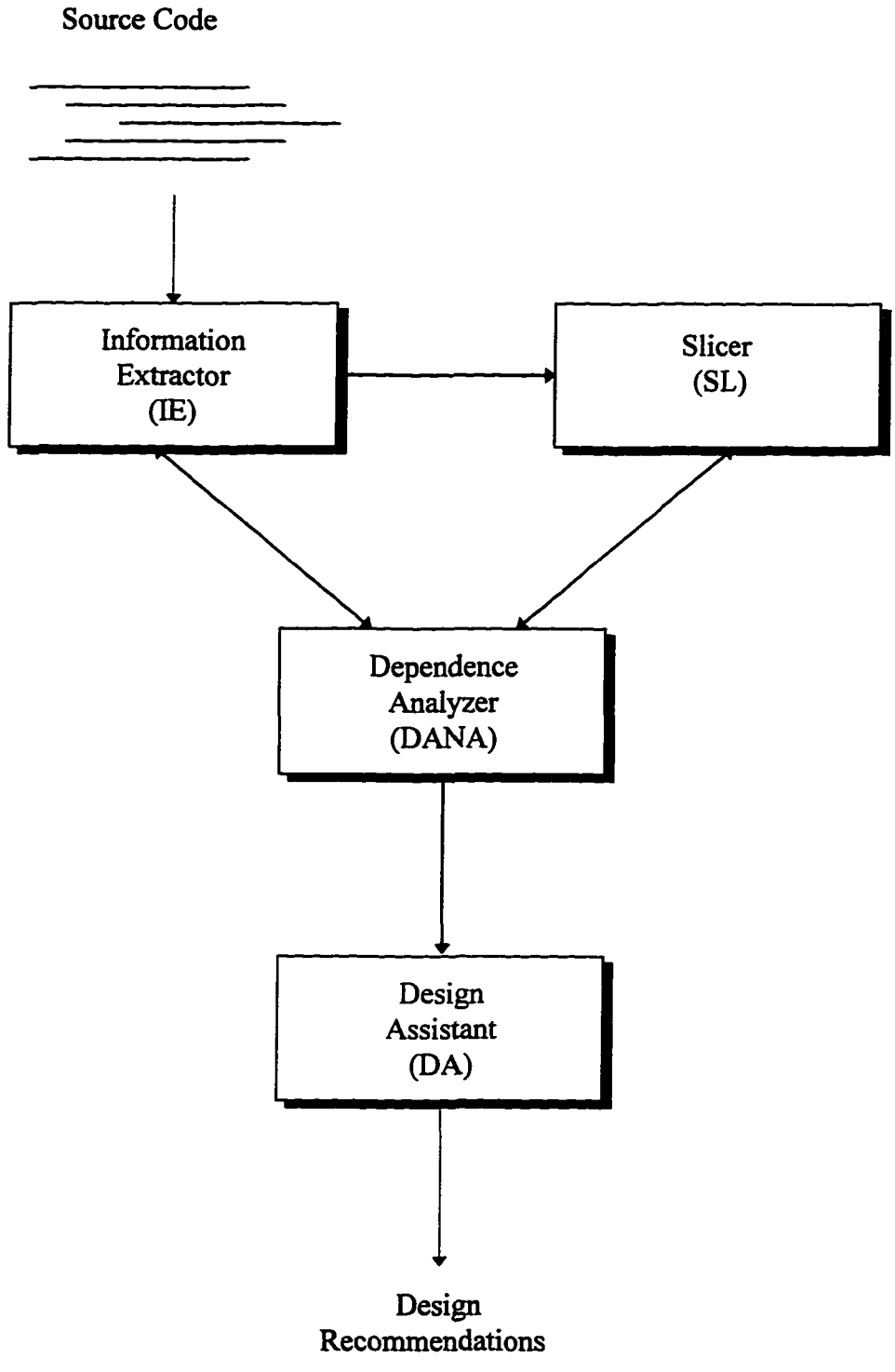


Figure 4.1 System overview of RETK

techniques to recover the sequential design of a FORTRAN-77 program. The Dependence Analyzer (DANA) uses the model described in [Jack 94] to analyze the data and control dependences in the design and builds a program dependence graph (PDG) in terms of the design elements. The Slicer (SL) extracts specific information from the sequential design description produced by the Information Extractor to facilitate the construction of the program dependence graph. Finally, the Design Assistant (DA) uses knowledge-based techniques to extract potential parallelism in the sequential design and provide parallel design recommendations.

4.2 Information Extractor

This section presents the design issues and implementation details of the Information Extractor (IE) of the reverse engineering toolkit (RETK) [Erra 96]. IE is the automated component of RETK that corresponds to the Analysis phase of the migration methodology presented in Chapter 3.

4.2.1 Design

IE is based on an object-oriented design to achieve software design concepts like abstraction, information hiding, and modularity. We chose Rumbaugh's OMT notation [Rumb 91] for the representation of the design. The problem statement for the design of IE can be stated as follows:

> *"Design the software to support an information extractor component of a reverse engineering toolkit. A FORTRAN program will serve as input to the information extractor. Interaction between the user and IE will be via a graphical user interface. FORTRAN code will serve as input to the information extractor. Every complete FORTRAN program has a Main Program, zero or more Subroutines, and zero or more user-defined Functions. The IE*

> *should analyze the FORTRAN code and recover the underlying sequential design description. The design information includes Metric Information, Call Graphs, Structure Charts, Variable Description, and State Changes. This design information should be provided for the Main Program, every subroutine and function in the original FORTRAN program."*

An object-oriented analysis of the above mentioned problem led to the identification of the objects, their associations, attributes, and operations. Using the OMT methodology, the object classes were identified by extracting nouns from the problem statement. The identified classes formed a set of temporary classes. Object class refinement was performed by eliminating redundant classes, irrelevant classes, and implementation constructs. Also classes which represent attributes and operations were also eliminated. Figure 4.2 shows the relevant classes for the design of the Information Extractor system.

The next step in the OMT methodology is the identification of associations. [Rumb 91] define an association as "a reference from one class to another." Associations were identified by extracting verb phrases from the problem description. As with object classes, a refinement is made to just retain meaningful associations. The object diagram with the object classes and associations is shown in Figure 4.3.

Attributes for each object class were identified by extracting general properties of each class. For example, physical lines of code in the FORTRAN program is an attribute of the *FORTRAN Source File* object class. Attributes for the *Main Program*, *Subroutine*, and *Function* classes include data structures for storing design information like Metric Information, Call Graphs, Structure Charts, Local and Global Variable descriptions, and State Changes. The object model with typical attributes for each class is shown in Figure 4.4.

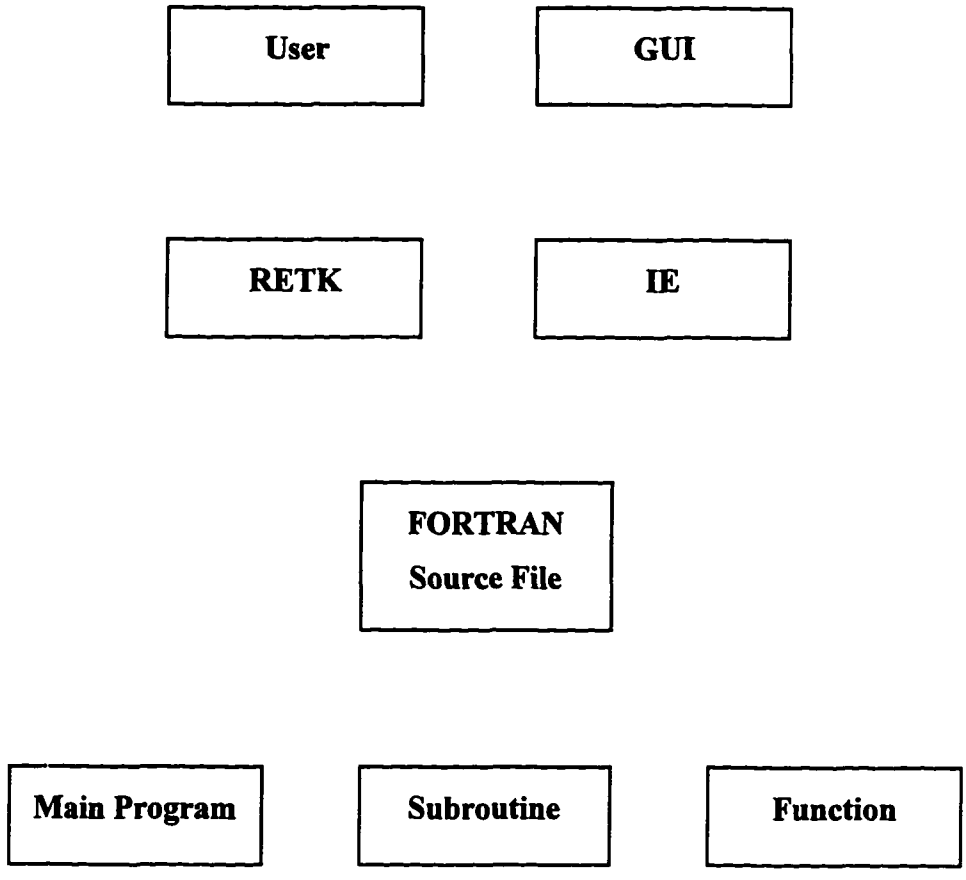


Figure 4.2 Relevant object classes for the Information Extractor

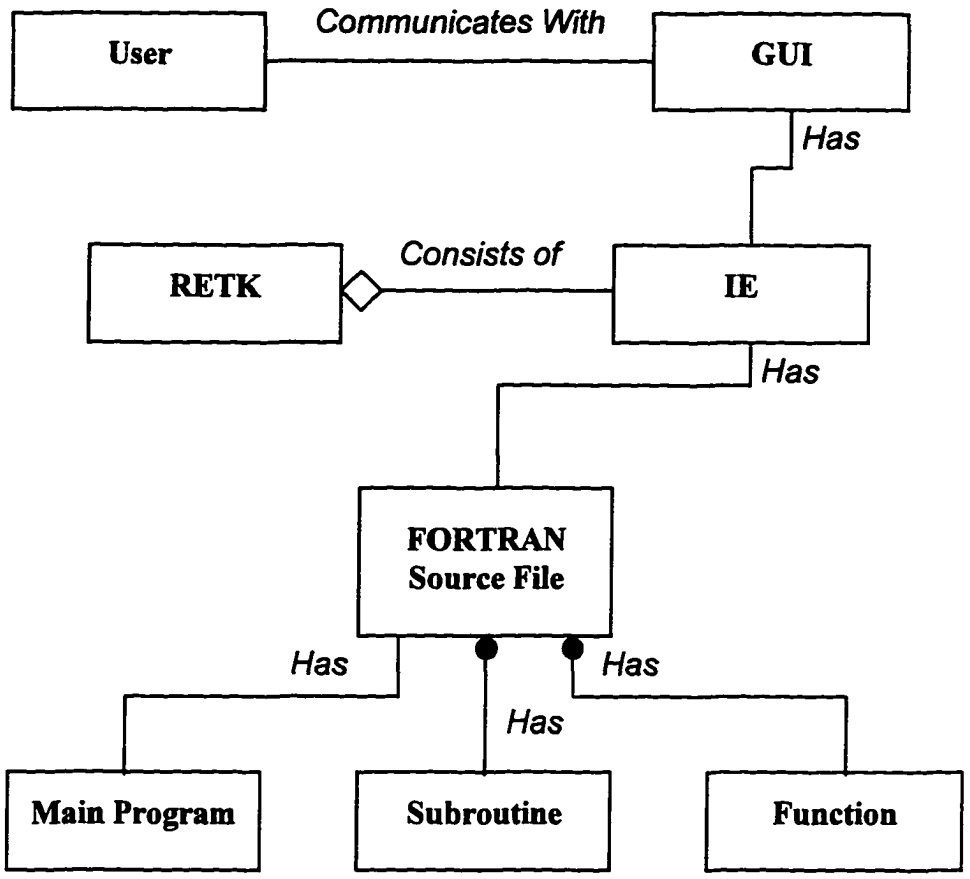


Figure 4.3 Object classes and their associations for the Information Extractor

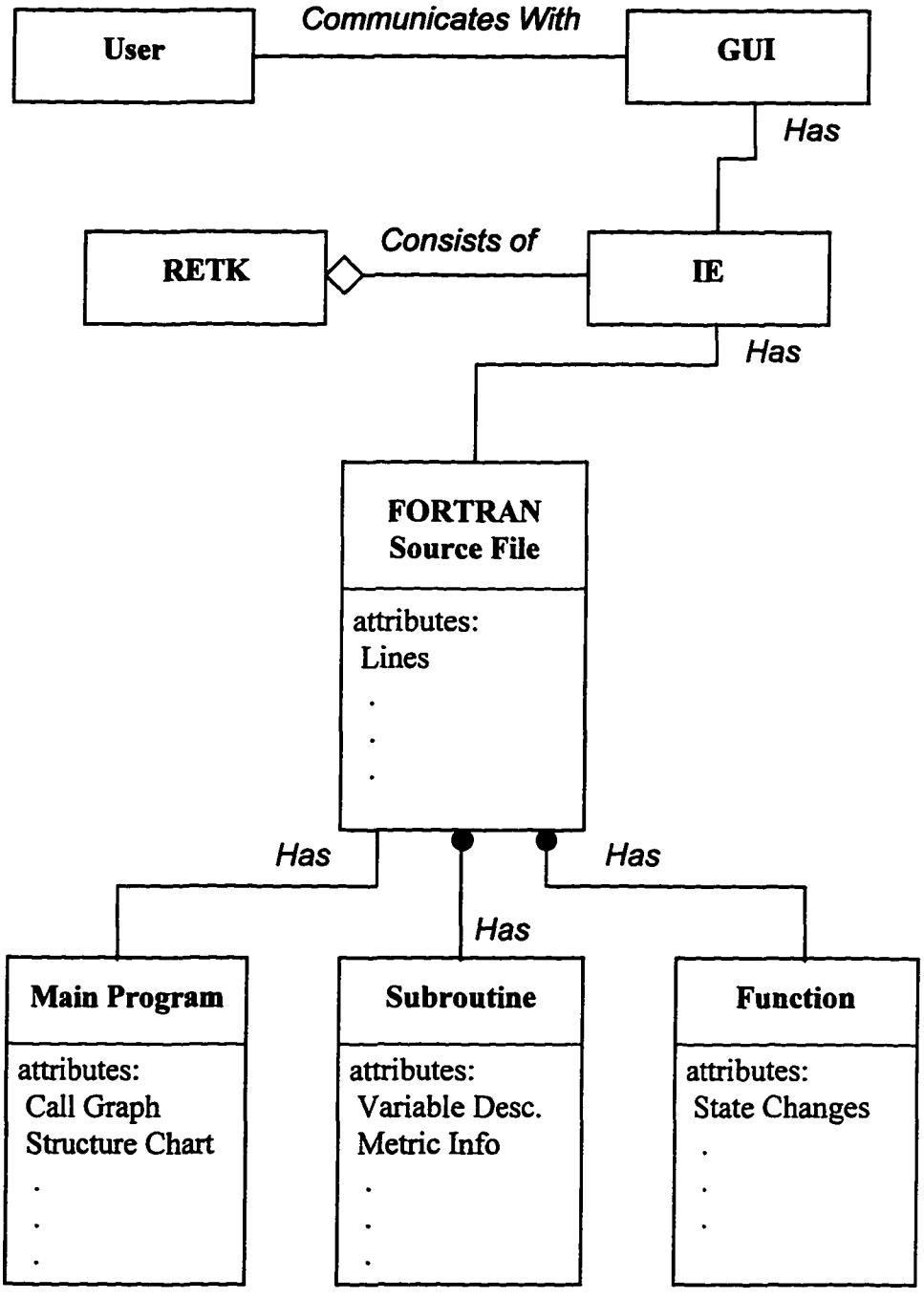


Figure 4.4 Object model for IE with typical attributes

The next step in the OMT methodology is to arrange the classes to define an inheritance relationship. By searching classes with similar attributes, we identified that *Main Program*, *Subroutine*, and *Function* classes share commonalties. A new class, called *Component*, which generalizes the *Main Program*, *Subroutine*, and *Function* classes was introduced into the object model. In other words, the *Component* class serves as a base class for the derived classes. The class *Component* is composed of the common attributes of the derived classes. Some of the common attributes of the derived classes such as data structures for Call Graphs and Metric Information were moved to the *Component* class. The derived class in the object-oriented paradigm inherits the description of the base class and can be further developed by adding or deleting attributes. An example of an attribute for the derived class, say *Subroutine*, is a data structure for formal parameters of a given subroutine. The class hierarchy is shown in Figure 4.5.

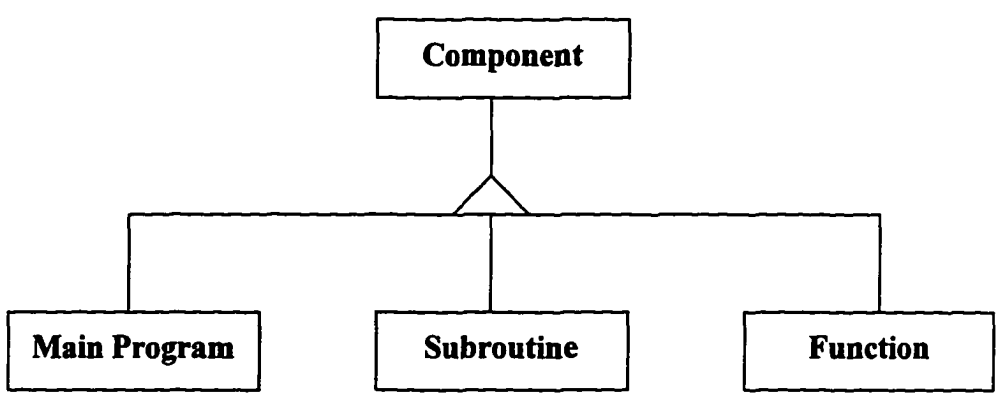


Figure 4.5 Inheritance relationship between object classes for IE

Finally operations are specified for each class. Common operations that were included in the base class include methods to extract the call structure and COMMON variables. An example of an operation that is part of the derived class Subroutine is a method that extracts the formal parameters. With the introduction of a new class, new associations between the new class and the remaining classes in the original object model have to be defined. Figure 4.6 shows the completed object model for IE with inheritance, associations, attributes, and operations. The complete model serves as a blueprint for an object-oriented implementation.

4.2.2 Implementation

The toolkit was developed on an IBM RS/6000 machine running under the AIX operating system. The object-oriented programming language C++ was used to code the toolkit and compiled using a GNU Project C++ compiler (Version 2.6).

The common operations of the derived classes are specified as virtual functions in the base class. A virtual function is used to *defer* the implementation decision of the function. Examples of the common operations of the derived classes were the identification of the subroutine calls, the extraction of the subroutine calls, the identification of the local variables, and the extraction of the local variables. The class definitions of Component, Main Program, Subroutine, and Function are shown in Tables 4.1 - 4.4. The class definitions shown in Tables 4.1 - 4.4 are defined as header files in C++. From the class definitions it can be noted that the class Component is an abstract base class because the keyword *virtual* appears in the class definition and also

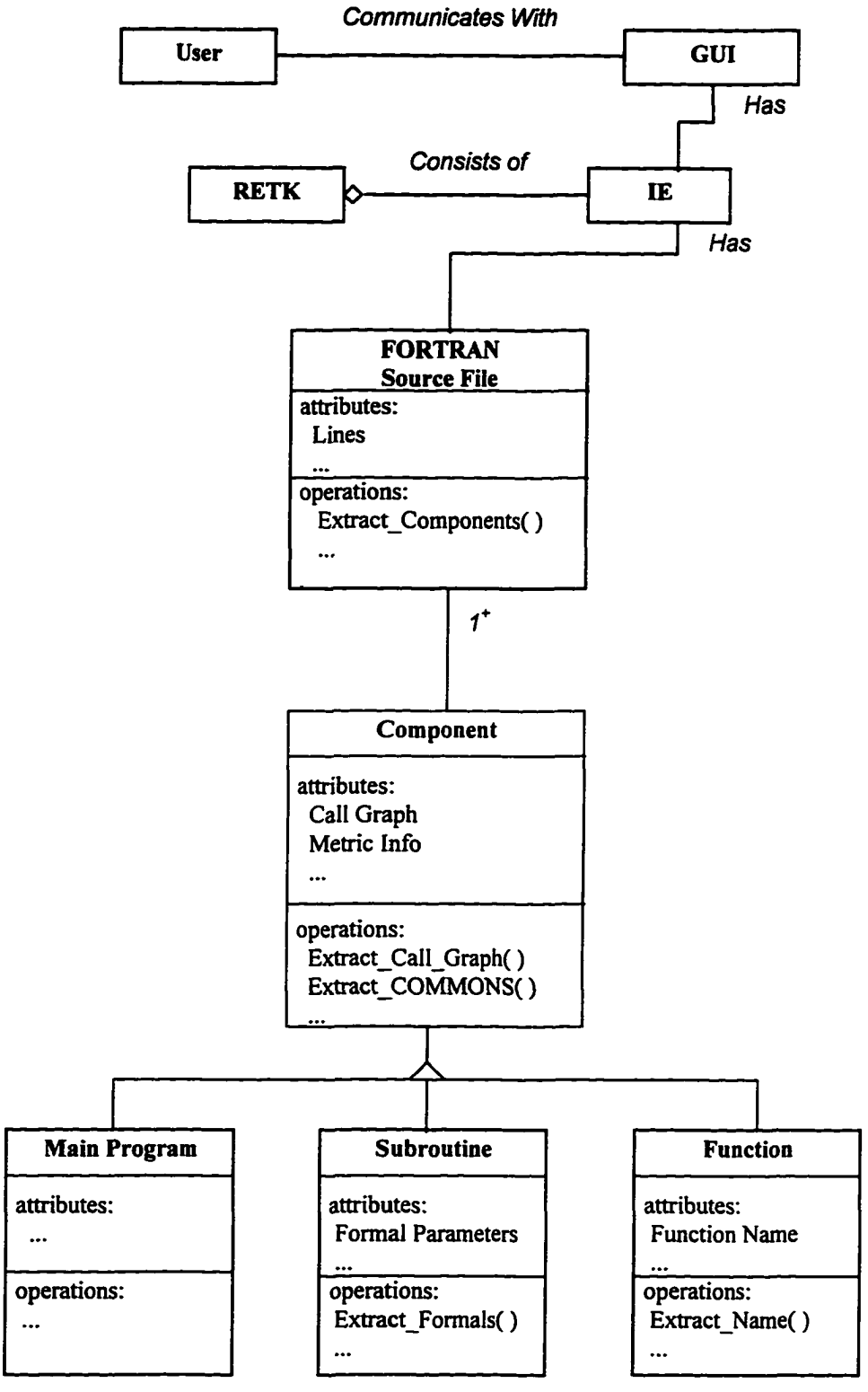


Figure 4.6 Completed object model for the Information Extractor

Table 4.1 C++ definition of class Component

```
class Component {

    protected:

    struct DOs {
            int     line_number;
            ...
    } *Iteratives;

    struct IFs {
            int     line_number;
            ...
    } *Conditionals;

    struct Calls {
            char    name[12];
            ...
    } *Acts, *Funcs;

    struct Variables {
            char    var_name[13];
            ...
    } *Vars;

    struct States {
            int     Line_Num;
            ...
    } *Sts;

    struct Labeled_Commons {
            char    Common_Block_Name[10];
            ...
    } *Commons;

    struct UnLabeled_Commons {
            char    **commons;
            int     com_count;
    } ULC;

    ...

    virtual char ** Get_Formals()
            { return NULL; }
    virtual int
            Get_Number_of_Formals()
            { return 0; }

    ...

    public:

    Component ();
    virtual ~Component ();
    void    Extract_Metric_Information();
    void    Print_Metric_Information();
    void    Extract_Calls();
    void    Print_Calls();
    void    Extract_Variables();
    void    Print_Variables();
    void    Extract_State_Changes();
    void    Print_State_Changes();
    void    Print_Function_Calls();
    int     Write_FORTRAN_File(int s);
    void    Print_Component();
};
```

Table 4.2 C++ definition of class Main Program

```
#include    "component.h"

class Main_Program : public Component {

    private:
            char     *program_name;


    public:
                     // Constructor
                     Main_Program ();
                     // Destructor
                     ~Main_Program();
                     // Member function which extracts the name of the main program
            void     Extract_Sub_Name();
                     // Member function which prints complete information of
                     // the main program
            void     Print_Information();
                     // Member function which prints the Common Variables
                     //  of the main program
            void     Print_Commons();
                     // Member function which writes the attributes of  the main program
                     // in textual form
            void     Write_to_Disk();
                     // Member function which writes the attributes of  the main program
                     // in flat file format
            void     Write_Object();
                     // Member function which reads the attributes of  the main program
                     // from a flat file passed as an argument
                     void     Read_Object(char * file_name);
                     // Member function which prints the call graph of the main program
            void     Write_Call_Graph();
};
```

Table 4.3 C++ Definition of class Subroutine

```
#include "component.h"

class Subroutine : public Component  {

    private:
            char    *sub_name;

            int     num_formals;

            char    **formals;

    public:
                    Subroutine ();
                    ~Subroutine();
            int     Get_Number_of_Formals()
                    {
                            return num_formals;
                    }
            char    ** Get_Formals()
                    {
                            return formals;
                    }
            void    Extract_Sub_Name();
            void    Extract_Formal_Parameters();
            void    Print_Info();
            void    Print_Formals_and_Commons();
            char    * get_name()
                    {
                            return sub_name;
                    }
            void Write_to_Disk();
            void Write_Object();
            void Read_Object(char * file_name);
            void Write_Call_Graph();

};
```

Table 4.4 C++ Definition of class Function

```
#include "component.h"

class Function : public Component  {

    private:
            char    *sub_name;

            int     num_formals;

            char    **formals;

    public:
                    Function ();
                    ~Function();
            int     Get_Number_of_Formals()
                    {
                            return num_formals;
                    }
            char    ** Get_Formals()
                    {
                            return formals;
                    }
            void    Extract_Sub_Name();
            void    Extract_Formal_Parameters();
            void    Print_Info();
            void    Print_Formals_and_Commons();
            char    * get_name()
                    {
                            return sub_name;
                    }
            void Write_to_Disk();
            void Write_Object();
            void Read_Object(char * file_name);
            void Write_Call_Graph();

};
```

because the class Component serves as a base class for the derived classes Main Program, Subroutine, and Function.

A diagram of the major processing steps of the Information Extractor is given in Figure 4.7. The FORTRAN source program served as the input to the program. The toolkit handles only syntactically correct programs. The source program is assumed to be structured. A restructuring algorithm needs to be applied a priori if the code is not structured. After the source file is read, the various components like the *main program*, *subroutines*, and *functions* are separated and objects are created. Each object is then analyzed by removing *visual sugar* like tabs, indents, and leading spaces. Special attention is given to save the documentation, if any, associated with the modules. The saved documentation helps in the process of re-engineering.

The next step is to analyze the source code associated with each object. The main program is analyzed first. Although FORTRAN does not have reserved words, we assume that all keywords are reserved words. The various subroutine calls are identified and the information extracted and placed in a data structure for future use. The name of the subroutine being called, the actual parameters in the call, the calling sequence, and the type of call are identified. The types of calls that are identified include 1) a simple call, 2) an iterative call, and 3) a conditional call. The complete format of the information extracted in this step is as shown in Table 4.5.

The local and non-local variable description is then extracted and stored in an appropriate data structure. Variables declared as COMMON variables are given particular importance. Labeled as well as unlabeled COMMON variables are stored separately as private attributes of the object. In addition, for the *Subroutine* and

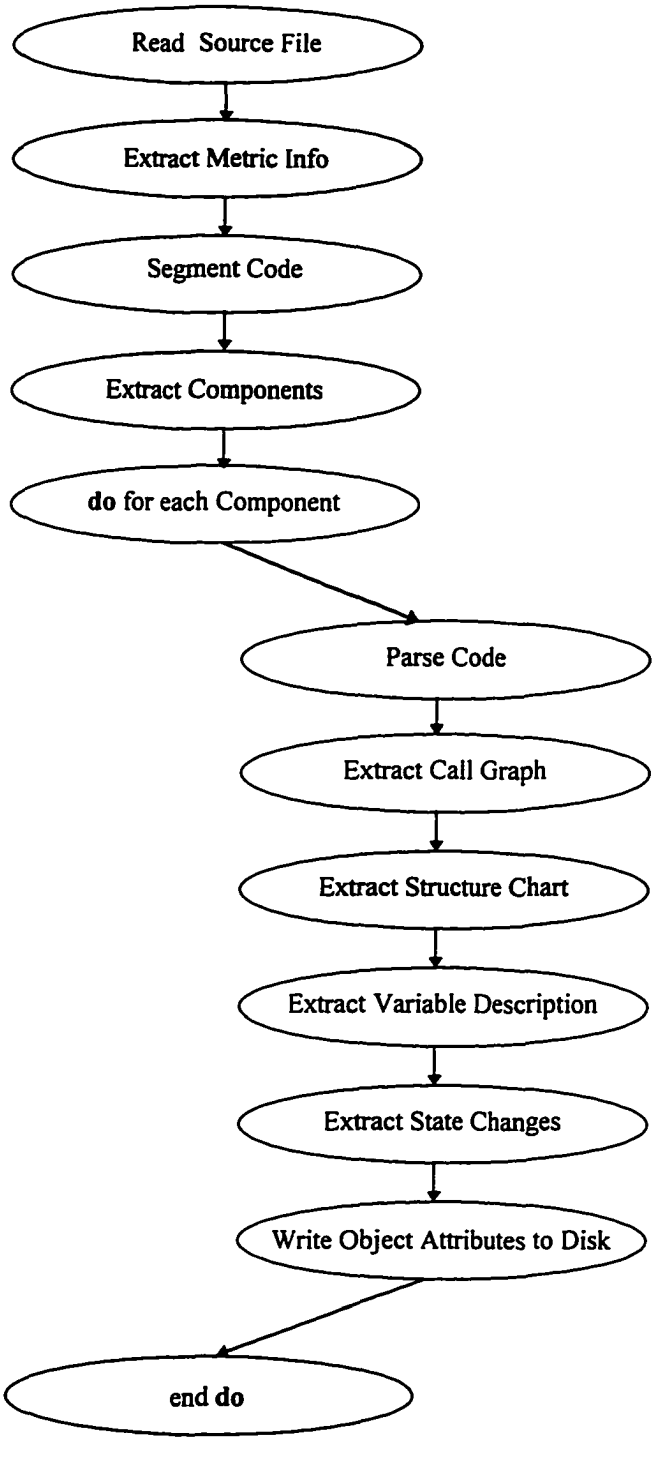


Figure 4.7 Major processing steps of the Information Extractor

Function objects, formal parameters are also stored as private attributes. Since FORTRAN uses pass by reference as the parameter passing method between modules, the formal parameters are treated as non-local to a given module. COMMON variables are clearly non-local to the module. The format of the information extracted in this step is as shown in Table 4.6. After the calls and variable description are identified, the structure chart of the main program is extracted. The structure chart corresponds to the call graph with the flow of data and the type of the data between the subroutines explicit.

Table 4.5 Summary of subroutine calls

Name of the subroutine in the CALL	Actual parameters in the CALL	Calling Sequence Number	Type of CALL
			• Simple • Iterative • Conditional

Table 4.6 Summary of variable description

Variable Name	Type	Is Declared	Is Array	Dimensions	Scope
					• Local • Common • Formal

The state changes in the program are then identified and stored in an appropriate data structure. The format of the information extracted in this step is as shown in Table 4.7. The "type" column in Table 4.7 refers to whether the state change variable is a simple assignment statement, an iterative assignment statement, or a conditional assignment statement. The "loop scope" column in Table 4.7 refers to the nesting of the DO loops. The "semantic nature" column in Table 4.7 documents how the state change is occurring; whether it is due to an assignment statement or due to a subroutine call. Table 4.8 lists additional metric information that is collected. The metrics include total number of lines, total number of commented lines, total number of blank lines, total number of discriminations, and McCabe's cyclomatic complexity.

Following the analysis of the main program, the same process is carried out on each subroutine and function object. Object attributes are saved for future use by the Dependency Analyzer and the Design Assistant.

Table 4.7 Summary of state changes

Line	State Change Variable Name	Rhs Variable(s)	Type	Loop Scope	Semantic Nature

Table 4.8 Metric information

Total Number of Lines	Total Number of Commented Lines	Total Number of Blank Lines	Total Number of Discrimina-tions	McCabe's Cyclomatic Complexity

4.3 Dependence Analyzer

This section presents the design issues and implementation details of the Dependency Analyzer (DANA) of the reverse engineering toolkit (RETK). DANA is the automated component of RETK that corresponds to the Synthesis phase of the migration methodology presented in Chapter 3.

4.3.1 Design

DANA is also based on an object-oriented design. An object-oriented analysis of the problem led to the identification of objects, associations, attributes, inheritance, and operations. The completed object model for DANA is shown in Figure 4.8. It can be observed that the object model of DANA is similar to that of IE (Figure 4.6). The basis for such similarity is because DANA analyzes the data and control dependences of each component by reading the object files produced by IE. The object files contain the design information of the components of the original FORTRAN program. DANA uses the notation defined in [Jack 94] to build data and control dependences. While all previous work in the area of dependence analysis concentrates on code, this work analyzes dependences from the sequential design representations produced by IE. Since a component may have many dependences associated with it, the design of DANA is aimed at providing graphical representations of the dependences. Graphical representations aid in overall comprehensibility.

The design of DANA utilizes the services of Slicer (SL) to build the data and control dependences of a component. SL is a program that was developed to read the

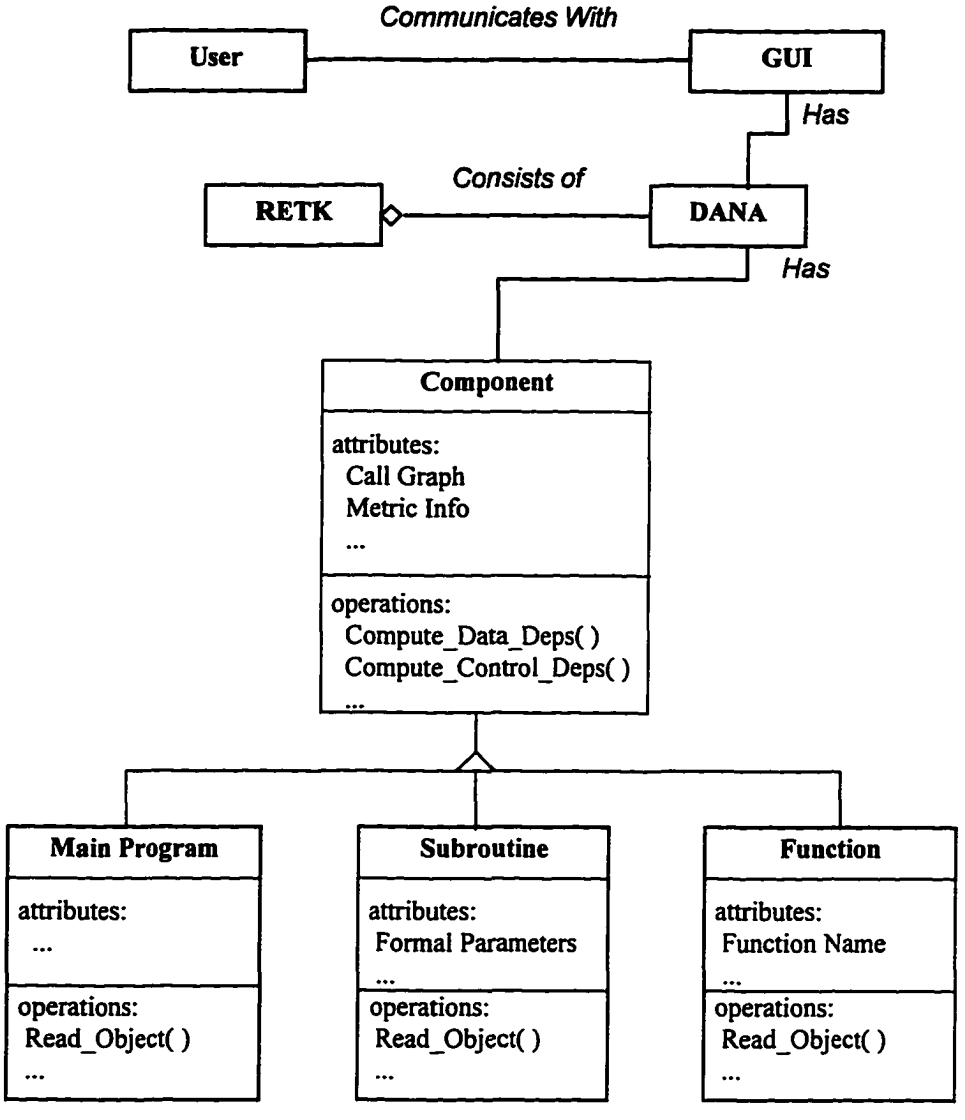


Figure 4.8 Object model for the Dependence Analyzer

object file associated with a component and return specific information from the sequential design description. SL provides information on all variables, local variables, non-local variables, variables declared as arrays, data-type information about a particular variable, and state changes of a particular variable. An algorithm of the tasks handled by SL is provided in Table 4.9.

4.3.2 Implementation

DANA was developed on an IBM RS/6000 machine running under the AIX operating system. C++ was used to code DANA and compiled using a GNU Project C++ compiler (Version 2.6). A diagram of the major processing steps is given in Figure 4.9. The object files containing the individual design information of each component served as the input to the programs. Since a component can be one of three types, namely Main program, Subroutine, and Function, an appropriate method is executed to read the design information about a component. Using the Call Graph the dependence graph of each component is synthesized until the Main program is reached. The synthesized dependence graph of the Main program constitutes the output of DANA — the program dependence graph (PDG).

The output format of DANA is both textual and graphical. The textual output documents the use lists, definition lists of each site and the data dependences and control dependences of the design. The graphical output is a file automatically generated by DANA which is compatible with the UNIX program *xfig*. The *xfig* representation uses boxes to represent sites. The dependences are represented as inter-site directed edges. Different colors are used to represent the flow dependences (forward data

Table 4.9 Algorithm for the slicer

```
// Algorithm for tasks handled by Slicer

Open Attribute File associated with a component
Define List_of_All_Variables 1
Define List_of_Local_Variables 2
Define List_of_Non_Local_Variables 3
Define More_Information 4

Read Request_Type

switch ( Request_Type ) {

        case 1: Read_Variable_Description_From_Attribute_File ( );
                for All Variables {
                        return Varaible_Description;
                }
                break;

        case 2: Read_Variable_Description_From_Attribute_File ( );
                for All Variables {
                        if (Variable.NonLocal == False ) {
                                return Variable_Description
                        }
                }
                break;

        case 3: Read_Variable_Description_From_Attribute_File ( );
                for All Variables {
                        if (Variable.NonLocal == True {
                                return Variable_Description
                        }
                }
                break;
        case 4: Read_Variable_Description_From_Attribute_File ( );
                for All Variables {
                        return Variable_Description.Name
                        return Variable_Description.Type
                        return Variable_Description.Is_Array
                        return Variable_Description.Dimensions
                        return Variable_Description.NonLocal
                }
                break;
```

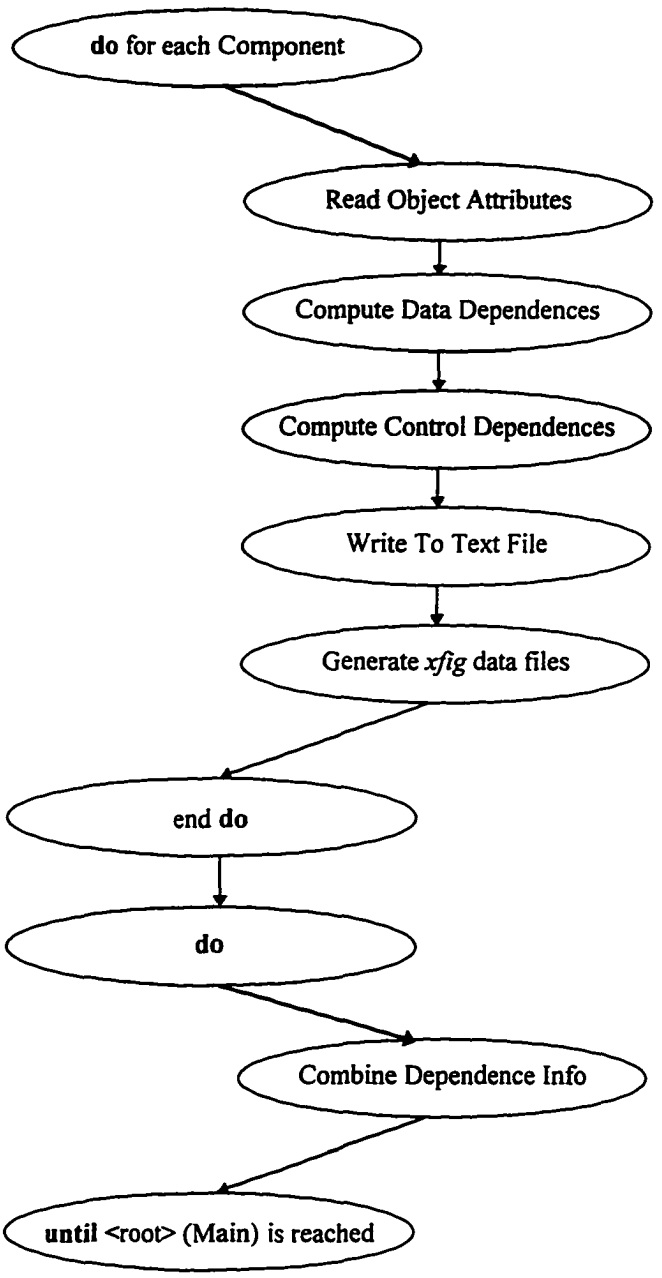


Figure 4.9 Major processing steps of the Dependence Analyzer

flow), loop dependences (backward data flow), and control dependences so that the design is clearly apparent to the user. Calls made to other subroutines and functions from a given subroutine are represented as one single site, reflecting changes to formal and common parameters. Table 4.10 shows the format of the textual dependence information generated of a typical site.

An example of a code fragment for the scenario where a state change to array variable NFACE occurs in a loop controlled by the loop variable I is listed in Table 4.11. The state changes for the code fragment shown in Table 4.11 are listed in Table 4.12. From Table 4.12 it can be observed that there are two state changes to variables. The dependence analysis of the state changes is listed in Table 4.13. The first two sites correspond to the loop variable assignment and conditional testing for loop termination. The third site corresponds to the state change to variable NFACE. The definition lists and use lists of each site are also listed in Table 4.13. The first site (row) of Table 4.13 corresponds to the assignment of the initial value to the loop variable I. The definition list variable for this site is I and the use list variables are γ and ε. All constants are represented by the symbol γ. The last use list variable of every site is represented by the symbol ε which stands for execution. If an edge exists between the use list variable ε of one site and the definition list variable of another site, then a control dependence exists between the sites. The second site of Table 4.13 corresponds to the conditional testing of loop termination. The definition list variable for this site is τ which holds the results of the conditional test. The use list variables are the loop scope index DO[0] and ε. The third site of Table 4.13 corresponds to the assignment statement to variable NFACE. The definition list variables of this site are NFACE and

Table 4.10 Format of the dependence information of a typical site

Site-Number : S_i
Site-Id : S_{id}
Number of Use-List Variables: j
Number of Def-List Variables: k
Use-List Variables:

ULV_1
Number of flow dependences: l
$fd_1, fd_2, \ldots, fd_l$
where $fd_x = \langle S_y, DLV_z \rangle \cap y < i \cap S_y.DLV_z = ULV_1$

ULV_2
Number of flow dependences: l
$fd_1, fd_2, \ldots, fd_l$

...

ULV_{j-1}
Number of flow dependences: l
$fd_1, fd_2, \ldots, fd_l$

$ULV_j = \varepsilon$
Number of control dependences: m
$cd_1, cd_2, \ldots, cd_m$
where $cd_x = \langle S_y, DLV_z \rangle \cap y < i \cap S_y.DLV_z = \tau_x$

Def-List Variables:

DLV_1
Number of loop dependences: n
$ld_1, ld_2, \ldots, ld_n$
where $ld_x = \langle S_y, ULV_z \rangle \cap y > i \cap S_y.ULV_z = DLV_1$

DLV_2
Number of loop dependences: n
$ld_1, ld_2, \ldots, ld_n$

...

DLV_k
Number of loop dependences: n
$ld_1, ld_2, \ldots, ld_n$

Table 4.11 A code fragment to illustrate the analysis of dependences

```
Line n:          DO I = 1, 6
Line n+1:            NFACE(I) = I
Line n+2:        END DO
```

Table 4.12 State changes of the code fragment listed in Table 4.11

Line	State Change Variable Name	Rhs Variable(s)	Type	Loop Scope	Semantic Nature
n	I	1, 6	Iterative	DO[0]	Loop Variable Assignment
n+1	NFACE	I	Iterative	DO[0]	Assignment Statement

Table 4.13 Textual description of dependences of the state changes listed in Table 4.12

Site Number: 1
Site Id: n
Number of Use-List Variables: 2
Number of Def-List Variables: 1
Use-List Variables:
γ
ε
Def-List Variables:
I

Site Number: 2
Site Id: n
Number of Use-List Variables: 2
Number of Def-List Variables: 1
Use-List Variables:
DO[0]
Number of flow dependences : 1
Flow Dependences:
<1, I>
ε
Def-List Variables:
τ

Site Number: 3
Site Id: n+1
Number of Use-List Variables: 2
Number of Def-List Variables: 2
Use-List Variables:
DO[0]
Number of flow dependences : 1
Flow Dependences:
<1, I>
ε
Number of control dependences: 1
Control Dependences:
<2, τ>
Def-List Variables:
NFACE
I
Number of loop dependences: 2
Loop Dependences:
< 2, DO[0]>, < 3, I>

I. The variable I needs to be included in the definition list of site 3 because site 3 corresponds to the last state change within the loop scope. The use list variables of site 3 are I and ε.

After all the sites are constructed, DANA computes the data and control dependences between the sites using the algorithms defined in Chapter 3. It can be observed that there exists a flow data dependence between the definition list variable I of site 1 and the use list variable DO[0] of site 2. Also, there is a flow data dependence between the definition list variable I of site 1 and the use list variable I of site 3. There exists a control dependence between the definition list variable τ of site 2 and the use list variable ε of site 3 indicating that site 3 is executed only of the loop is not terminated. Loop data dependences exist from the definition list variable I of site 3 to the use list variable DO[0] of site 2 and the use list variable I of site 3. Figure 4.10 shows an *xfig* graphical of the dependence analysis of the code fragment. Figure 4.10 depicts dependences in different shades of gray to simulate the different colors used in the *xfig* representation. In addition to producing textual and graphical representations, DANA generates dependence facts to be used by the Design Assistant (described in Section 4.4). The dependence facts are generated for each component in the sequential design.

4.4 Design Assistant

This section describes the design and implementation details of the design assistant. Before the design and implementation issues are discussed, a brief

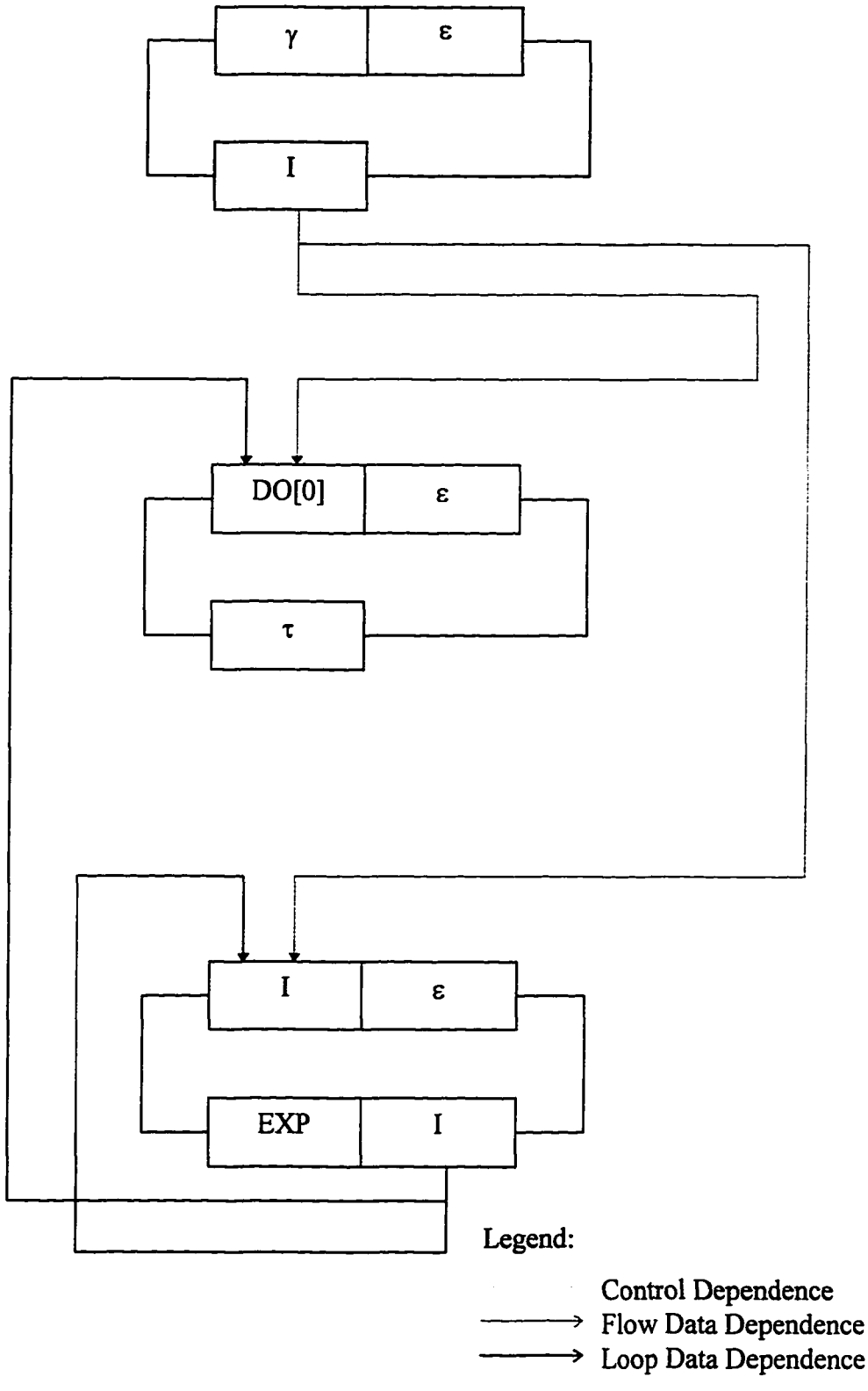


Figure 4.10 An example *xfig* graphical representation of dependences

introduction to NASA's CLIPS [Clp 93a] [Clp 93b] [Clp 93c] is presented. Since CLIPS is a comprehensive knowledge-based programming environment, only the information relevant to this dissertation will be covered in the introduction to CLIPS.

4.4.1 CLIPS

The C-Language Interface Production System (CLIPS) is an expert system tool developed at NASA's Johnson Space Center. The first release of CLIPS was released in 1985. Before CLIPS, LISP was used nearly in all expert system software tools at NASA. However, LISP posed certain problems that impeded its use in expert systems within NASA. The low availability of LISP on a broad range of conventional computers, the high cost of LISP tools and hardware, and the poor integration of LISP with other languages are some of the reasons why NASA chose to build CLIPS. The first couple of versions of CLIPS were developed for internal use (within NASA). The first release of CLIPS to the outside world was Version 3.0 in 1986. Since then CLIPS has undergone many refinements and improvements. The most recent version of CLIPS is Version 6.0 available both for UNIX-based systems and MS-DOS/MS-Windows based systems. CLIPS has received widespread acceptance throughout the government, industry, and academia. CLIPS can be integrated with external functions or applications. A user can define external functions to CLIPS at any place a function can normally be called. Once compiled and linked, all external functions can be used like built-in functions within the CLIPS environment. In addition to being both portable and extensible, CLIPS allows embedded applications to be built easily. It is possible to embed CLIPS system calls in C, Ada, and FORTRAN programs.

4.4.1.1 Knowledge Abstraction

Information in CLIPS can be represented using facts, objects, and global variables. Facts indicate the truth or falsity of a certain piece of information. CLIPS maintains a *fact-list* which consists of all the facts that are asserted. Facts in the *fact-list* can be manipulated using the **assert** command to add facts, the **retract** command to remove facts, the **modify** command to modify facts, and the **duplicate** command to duplicate facts. Any number of facts may be added to the fact-list, which is limited only by the memory capacity of the computer. CLIPS uses a fact-index to keep track of the facts in the fact-list. Fact-index is an unique integer that is given to each new or changed fact. Facts in CLIPS can either be ordered or non-ordered. Ordered facts consists of a symbol followed by a sequence of zero or more fields. The entire fact is enclosed in parenthesis[1]. The first field of an ordered fact specifies a "relation" that is applied to the remaining fields. An example of a set of ordered fact is given below:

(FORTRAN-Program has Main-Program)
(Main-Program name is "STATS")
(FORTRAN-Program has 2 Subroutines)
(FORTRAN-PROGRAM has 1 Functions)

Since ordered facts encode information positionally, a user must know the structure of the data. On the other hand, non-ordered facts allows the user to assign names to each field in the fact thereby providing an abstraction of the structure of the facts. A **deftemplate** construct is used to create a template of a fact. Fields in the template are

[1] In fact, all commands and constructs in CLIPS are delimited by an opening parenthesis on the left and a closing parenthesis on the right.

called **slots**. Unlike fields in an ordered fact, default values can be specified for slots in a non-ordered fact. An example of a non-ordered fact is given below:

(FORTRAN-Program
(name "STATS")
(number-of-subroutines 2)
(number-of-functions 1))

The advantages of non-ordered facts are clarity and slot order independence. The **deffacts** construct allows initial knowledge to be specified as a collection of facts. When the **reset** command is issued, all facts specified within a deffacts construct are added the fact-list.

One of the unique features of CLIPS is that information can be represented using objects. In CLIPS, an object is defined to be a symbol, a string, a floating-point number or integer number, a multi-field value, an external-address, or an instance of a user-defined class. The general notion of class and object of the object-oriented paradigm apply to classes and objects within CLIPS. A class is a template for common attributes and methods of objects which are instances of that class. Objects in CLIPS can be instances of primitive classes (such as SYMBOL and USER) and user-defined classes. User-defined classes are defined using the **defclass** construct. Roles for each class can be defined to be either abstract or concrete. Object instances of classes with abstract roles cannot be created. Attributes of a class are specified using slots or multislots. Methods or operations on attributes of a class are specified using **defmessage-handler** construct. The difference between objects and non-ordered facts is that inheritance allows the attributes and methods of a class to be described in terms of other classes. CLIPS supports both single and multiple inheritance and is specified using is-a link. An example of a user-defined class is shown below:

```
(defclass Component (is-a USER) (role abstract)
(slot num-components))

(defmessage-handler Component increment-num-components ()
        (+ ?self:num-components 1))
```

Objects are created using the **make-instance** construct. CLIPS maintains an *instance-list* of instances of all classes that are created. Just like facts, objects can also be manipulated by adding using the **make-instance** construct and deleting using the **unmake-instance** construct. The **definstances** construct allows initial knowledge to be specified as a collection of objects. Whenever the reset command is issued, all objects specified within a definstances construct are added the instance-list.

4.4.1.2 Knowledge Representation

Knowledge in CLIPS can be represented using the heuristic paradigm or the procedural paradigm. A combination of both paradigms can also be used to represent knowledge. Rules specify a set of "actions" to be performed whenever a set of conditions are satisfied. Rules are specified using the **defrule** construct. An example of a rule is given below:

```
(defrule find-modularity
(and (eq (find-instance ?ins Subroutine) FALSE)
     (eq (find-instance ?ins Function) FALSE))
=>
(printout t "No modules in program.
            Inter-Module parallelism not possible!" crlf))
```

CLIPS places rules as they defined in an *agenda-list.* CLIPS uses pattern matching to determine which facts or objects satisfy the conditions specified by a particular rule. If more than one rule is satisfied, a conflict resolution strategy is used by CLIPS to select

a rule from the agenda-list CLIPS uses a forward-chaining inference procedure to execute rules in the agenda-list. One concern with a forward-chaining procedure is that there might be a search-explosion problem. This problem can be combated by using the rules and procedural knowledge representation.

Procedural knowledge representation allows users to manipulate the knowledge in the CLIPS knowledge-base using **deffunction** construct. The deffunction allow users to define procedural code in terms of CLIPS constructs. A combination of procedural knowledge representation with rules controls the search process.

4.4.1.3 Inference Procedure

After a user builds the knowledge-base using facts, objects, rules, and functions, CLIPS is ready to execute. Rules are automatically executed by CLIPS whereas user-defined functions (specified by deffunctions) have to be explicitly executed by the user. However, if an user-defined function is specified as consequent of a rule, then execution of such an user-defined function is controlled by the firing of the rule.

CLIPS provides seven conflict resolution strategies: depth, breadth, simplicity, complexity, lex, mea, and random. The default strategy is depth which places newly activated rules *above* all rules of the same salience. For example, given that an object of class A activates rule-1 and rule-2 and an object of class B activates rule-3 and rule-4, then if an instance of class A is asserted before an instance of class B, rule-3 and rule-4 will be placed above rule-1 and rule-2 in the agenda-list. Positioning of rule-1 relative to rule-2 and rule-3 relative to rule-4 is arbitrary. For most cases the default depth conflict resolution strategy works satisfactorily. A comprehensive discussion of all the conflict resolution strategies can be found in [Clp 93a].

4.4.2 Knowledge Representation of the Design Assistant

The information produced/analyzed by the Information Extractor and Dependence Analyzer forms the base for program facts and dependence facts, respectively. Since human intervention is inevitable in any knowledge-based application, user information forms the base for user facts. All facts are encoded using CLIPS constructs and CLIPS object-oriented language (COOL). Rules on how to react to different facts are encoded into the knowledge repository. In addition to rules, procedural knowledge is encoded into the knowledge repository using deffunction constructs. Figure 4.11 shows the overall design of the design assistant.

Table 4.14 shows the knowledge representation of program facts. The topmost class is Component which is a sub-class of USER which is a primitive class in CLIPS. All user-defined classes in CLIPS must be sub-classes of USER. Main-Program, Subroutine, Function are some of the sub-classes of Component. The roles of all classes except that of Component are defined to be concrete, thus allowing instances to be created. Other classes include Subroutine-Calls, Function-Calls, DO-Loop-Information, IF-Conditions, Variable-Description, State-Changes, Labeled-Commons, and Un-Labeled Commons. All classes are sub-classes of Component. The attributes of each class are specified using slots and mutislots. The nomenclature for each attribute of a given class should be clearly inferred in the context of the class definition. For example, the class Subroutine has three attributes: 1) Name, which corresponds to the name of the subroutine, 2) Sub-Formals, which corresponds to the list of formal parameters of the subroutine, and 3) Num-formals, which corresponds to the number of formal parameters.

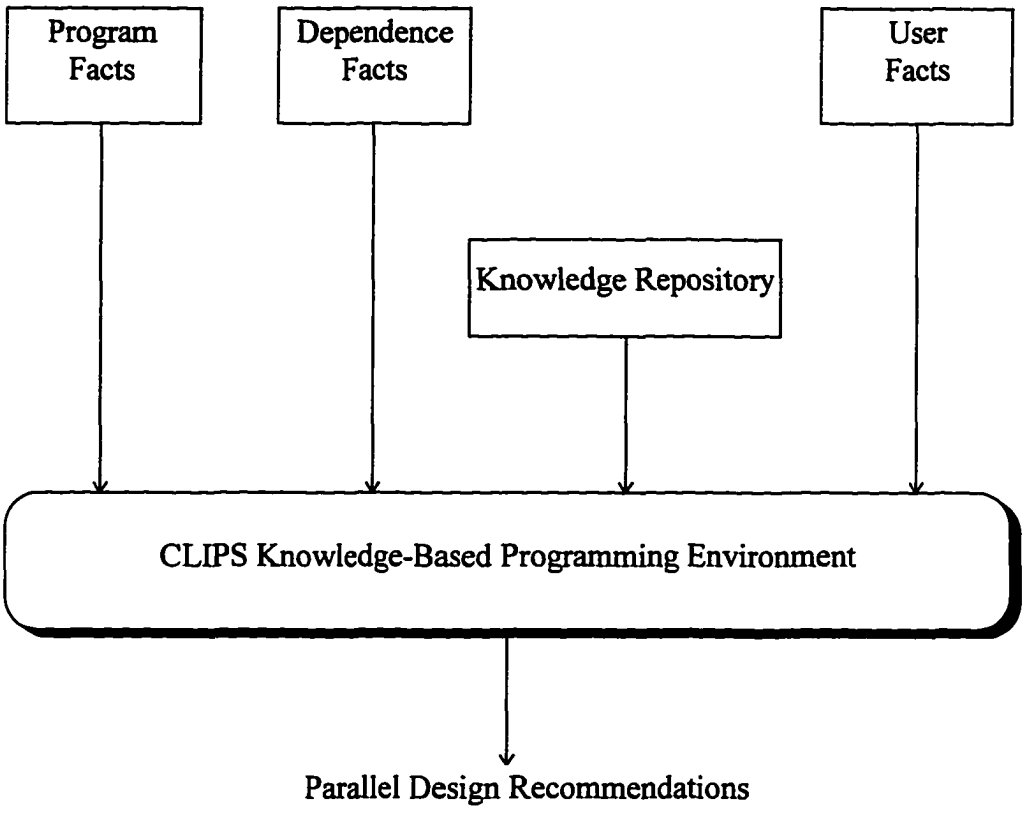


Figure 4.11 Overall design of the Design Assistant

Table 4.14 Representation of program facts in CLIPS

```
(defclass Component (is-a USER)                          ; Definition of class component
(role abstract)                                          ; Role Abstract
(slot comp-name (create-accessor read-write))            ; Name of the component
(slot num-calls (create-accessor read-write))            ; Number of subroutine calls
(slot num-funcs (create-accessor read-write))            ; Number of Function calls
(slot num-locals (create-accessor read-write))           ; Number of variables for component
(slot num-states (create-accessor read-write))           ; Number of state changes
(slot num-labeled-commons (create-accessor read-write))    ; Number of labeled Commons
(slot num-un-labeled-commons (create-accessor read-write)) ; Number of unlabeled Comms
(slot num-do-loops (create-accessor read-write))         ; Number of DO Loops
(slot num-if-conds (create-accessor read-write)))        ; Number of IF Conditions

(defclass Main-Program (is-a Component)
(role concrete)
(slot Name (create-accessor read-write)))

(defclass Subroutine (is-a Component)
(role concrete)
(slot Name (create-accessor read-write))
(multislot Sub-Formals (create-accessor read-write))
(slot num-formals (create-accessor read-write)))

(defclass Function (is-a Component)
(role concrete)
(slot Name (create-accessor read-write))
(multislot Fun-Formals (create-accessor read-write))
(slot num-formals (create-accessor read-write)))

(defclass Subroutine-Calls (is-a Component)
(role concrete)
(slot sub-name (create-accessor read-write))
(slot sub-line (create-accessor read-write))
(slot sub-call-seq-no (create-accessor read-write))
(multislot sub-actuals (create-accessor read-write))
(slot act-count (create-accessor read-write)))

(defclass Function-Calls (is-a Component)
(role concrete)
(slot fun-name (create-accessor read-write))
(slot fun-line (create-accessor read-write))
(slot fun-call-seq-no (create-accessor read-write))
(multislot fun-actuals (create-accessor read-write))
(slot act-count (create-accessor read-write)))

(defclass DO-Loop-Information (is-a Component)
(role concrete)
(slot DO-Start-Line (create-accessor read-write))
(slot DO-Last-Line (create-accessor read-write))
(slot DO-Label (create-accessor read-write))
```

(table con'd.)

```
(slot DO-Counting-Variable (create-accessor read-write))
(slot DO-Initial-Value (create-accessor read-write))
(slot DO-Limit (create-accessor read-write))
(slot DO-Step (create-accessor read-write))
(slot DO-Nesting-Level (create-accessor read-write))
(slot DO-Prev-Type (create-accessor read-write))
(slot DO-Prev-Id (create-accessor read-write)))

(defclass IF-Conditions (is-a Component)
(role concrete)
(slot IF-Start-Line (create-accessor read-write))
(slot IF-True-Part-Last-Line (create-accessor read-write))
(slot IF-False-Part-Last-Line (create-accessor read-write))
(slot IF-Logical_Expression (create-accessor read-write))
(slot IF-Nesting-Level (create-accessor read-write))
(slot IF-Prev-Type (create-accessor read-write))
(slot IF-Prev-Id (create-accessor read-write)))

(defclass Variable-Description (is-a Component)
(role concrete)
(slot Var-Name (create-accessor read-write))
(slot Type (create-accessor read-write))
(slot Declared (create-accessor read-write))
(slot Array (create-accessor read-write))
(multislot Dimensions (create-accessor read-write))
(slot Non-Local (create-accessor read-write)))

(defclass State-Changes (is-a Component)
(role concrete)
(slot Line-Num (create-accessor read-write))
(slot Var-Name (create-accessor read-write))
(multislot Rhs-Variables (create-accessor read-write))
(slot Category (create-accessor read-write)))

(defclass Labeled-Commons (is-a Component)
(role concrete)
(slot Common-Name (create-accessor read-write))
(multislot Commons (create-accessor read-write))
(slot Commons-Count (create-accessor read-write)))

(defclass Un-Labeled-Commons (is-a Component)
(role concrete)
(multislot Commons (create-accessor read-write))
(slot Commons-Count (create-accessor read-write)))
```

The class definitions shown in Table 4.14 serve as templates for object instances. All the slots include a create-accessor facet which instructs CLIPS to automatically create explicit message-handlers for reading and writing data to a slot. Explicit message-handlers are created for each slot for each class. The general format of the message-handler that is created by CLIPS for the create-accessor facet of a slot with read access privilege is shown below:

(defmessage-handler <class> get-<slot-name> primary ()
?self:<slot-name>)

The general format of the message-handler that is created by CLIPS for the create-accessor facet of a slot with write access privilege is shown below:

(defmessage-handler <class> put-<slot-name> primary () (?value)
(bind ?self:<slot-name> ?value))

The design of each component in the original sequential design description will be encoded as program facts in CLIPS by creating object-instances of the classes.

Table 4.15 shows the knowledge representation of dependence facts. The top-most class is Component which is a sub-class of USER and is defined in Table 4.14. Six new classes are defined for representing dependence information: 1) Site is a sub-class of Component and corresponds to site information, 2) Use-Lists is a sub-class of Site and corresponds to the use list of a site, 3) Def-Lists is a sub-class of Site and corresponds to the definition list of a site, 4) Du-Edges is a sub-class of Use-Lists and corresponds to flow dependences between variables, 5) Ud-Edges is a sub-class of Def-Lists and corresponds to loop dependences between variables, and 6) Cd-Edges is a sub-class of Use-Lists and corresponds to control dependences between sites. CLIPS

Table 4.15 Representation of dependence facts in CLIPS

```
(defclass Site (is-a Component)
(role concrete)
(slot Site-Id (create-accessor read-write))
(slot Site-Number (create-accessor read-write))
(slot Number-of-Uls (create-accessor read-write))
(slot Number-of-Dls (create-accessor read-write))
(slot Number-of-Cds (create-accessor read-write)))

(defclass Use-Lists (is-a Site)
(role concrete)
(slot Use-List-Name (create-accessor read-write))
(slot Number-of-Dus (create-accessor read-write)))

(defclass Def-Lists (is-a Site)
(role concrete)
(slot Def-List-Name (create-accessor read-write))
(slot Number-of-Uds (create-accessor read-write)))

(defclass Du-Edges (is-a Use-Lists)
(role concrete)
(slot Du-Edges-Name (create-accessor read-write))
(slot Du-Dest-Site-Number (create-accessor read-write)))

(defclass Ud-Edges (is-a Def-Lists)
(role concrete)
(slot Ud-Edges-Name (create-accessor read-write))
(slot Ud-Dest-Site-Number (create-accessor read-write)))

(defclass Cd-Edges (is-a Use-Lists)
(role concrete)
(slot Cd-Edges-Name (create-accessor read-write))
(slot Cd-Dest-Site-Number (create-accessor read-write)))
```

creates explicit message-handlers for each slot in the class definition *via* the specification of the create-accessor facet. Dependence information of each component, which includes the Program Dependence Graph, is encoded as dependence facts by creating object-instances of the classes.

4.4.3 Inference Mechanism of the Design Assistant

The inference mechanism for ascertaining the parallelization potential in the original program involves the specification of rules (for heuristic knowledge) and functions (for procedural knowledge) into the knowledge repository to react to the program and dependence facts. We define a set of scenarios which help assess the parallelization potential and then formulate the defined set of scenarios into the knowledge repository.

We have identified a set of six scenarios by which parallelization potential is identified. Of the six scenarios, five cover intra-component parallelism and one covers inter-component parallelism. Description of the scenarios and the formal definitions are given below. If there is potential for intra-component parallelism, then the parallel design recommendations are coarse-grained. If there is potential for inter-component parallelism, then the parallel design recommendations are fine-grained.

Scenario 1 (Intra-Component Parallelism):

Consider two simple calls to subroutines Sub-1 and Sub-2 from Component-X (by definition a simple call to a subroutine is one which is not invoked in a loop). If Sub-1 and Sub-2 have different actual parameters and Sub-1 and Sub-2 do not modify Labeled or Un-labeled Common Variables, then the calls to Sub-1 and Sub-2 can be parallelized. The formal definition of Scenario 1 is:

Let Call-1 and Call-2 be two subroutine calls from Component-X.
Let Sub-1 and Sub-2 be the called subroutines in Call-1 and Call-2 respectively.
Let $ap_1^{(1)}, ap_2^{(1)}, \ldots, ap_r^{(1)}$ be the actual parameters of subroutine Sub-1 in call Call-1.
Let $ap_1^{(2)}, ap_2^{(2)}, \ldots, ap_s^{(2)}$ be the actual parameters of subroutine Sub-2 in call Call-2.
Let $cv_1^{(1)}, cv_2^{(1)}, \ldots, cv_t^{(1)}$ be the common variables of subroutine Sub-1.
Let $cv_1^{(2)}, cv_2^{(2)}, \ldots, cv_u^{(2)}$ be the common variables of subroutine Sub-2.
Let $st\text{-}cv_x^{(y)}$ be a boolean variable which indicates whether or not a state change occurs to common variable cv_x in subroutine Sub-y.
for all (i = 1, r and j = 1, s)
    if ($ap_i^{(1)} \neq ap_j^{(2)}$) then
        for all (k = 1, t and l = 1, u)
            if ($st\text{-}cv_k^{(1)}$ = False and $st\text{-}cv_l^{(2)}$ = False) then
                parallel (Call-1, Call-2) = True
            end if
        end for
    end if
end for

Scenario 2 (Intra-Component Parallelism):

Consider two simple calls to subroutines Sub-1 and Sub-2 from Component-X. If Sub-1 and Sub-2 have common actual parameters, and Sub-1 and Sub-2 do not modify Labeled or Un-labeled Common Variables, and the common actual parameters themselves are not modified then the calls to Sub-1 and Sub-2 can be parallelized. The formal definition of this scenario is:

Let Call-1 and Call-2 be two subroutine calls from Component-X.
Let Sub-1 and Sub-2 be the called subroutines in Call-1 and Call-2 respectively.
Let $ap_1^{(1)}, ap_2^{(1)}, \ldots, ap_r^{(1)}$ be the actual parameters of subroutine Sub-1 in call Call-1.
Let $ap_1^{(2)}, ap_2^{(2)}, \ldots, ap_s^{(2)}$ be the actual parameters of subroutine Sub-2 in call Call-2.
Let $cv_1^{(1)}, cv_2^{(1)}, \ldots, cv_t^{(1)}$ be the common variables of subroutine Sub-1.
Let $cv_1^{(2)}, cv_2^{(2)}, \ldots, cv_u^{(2)}$ be the common variables of subroutine Sub-2.
Let $st\text{-}cv_x^{(y)}$ be a boolean variable which indicates whether or not a state change occurs to common variable cv_x in subroutine Sub-y.
Let $st\text{-}ap_w^{(z)}$ be a boolean variable which indicates whether or not a state change occurs to formal parameter aliased to the actual parameter ap_w in subroutine Sub-z.

for all (i = 1, r and j = 1, s)
    if (exists m and n such that $1 \leq m \leq r$, $1 \leq n \leq s$ and $ap_m^{(1)} = ap_n^{(2)}$) then
        if (st-$ap_m^{(1)}$ = False and st-$ap_n^{(2)}$ = False) then
            for all (k = 1, t and l = 1, u)
                if (st-$cv_k^{(1)}$ = False and st-$cv_l^{(2)}$ = False) then
                    parallel (Call-1, Call-2) = True
                end if
            end for
        end if
    end if
end for

Scenario 3 (Intra-Component Parallelism):

Consider two simple calls to subroutines Sub-1 and Sub-2 from Component-X. If Sub-1 and Sub-2 have different actual parameters, and Sub-1 and Sub-2 access (read/write) Labeled and/or Un-labeled Common Variables at different locations, then the calls to Sub-1 and Sub-2 can be parallelized. The formal definition of Scenario 3 is:

Let Call-1 and Call-2 be two subroutine calls from Component-X.
Let Sub-1 and Sub-2 be the called subroutines in Call-1 and Call-2 respectively.
Let $ap_1^{(1)}, ap_2^{(1)}, \ldots, ap_r^{(1)}$ be the actual parameters of subroutine Sub-1 in call Call-1.
Let $ap_1^{(2)}, ap_2^{(2)}, \ldots, ap_s^{(2)}$ be the actual parameters of subroutine Sub-2 in call Call-2.
Let $cv_1^{(1)}, cv_2^{(1)}, \ldots, cv_t^{(1)}$ be the common variables of subroutine Sub-1.
Let $cv_1^{(2)}, cv_2^{(2)}, \ldots, cv_u^{(2)}$ be the common variables of subroutine Sub-2.
Let st-$cv_x^{(y)}$ be a boolean variable which indicates whether or not a state change occurs to common variable cv_x in subroutine Sub-y.
for all (i = 1, r and j = 1, s)
    if ($ap_i^{(1)} \neq ap_j^{(2)}$) then
        for all (k = 1, t and l = 1, u)
            if (exists g and h such that $1 \leq g \leq t$, $1 \leq h \leq u$ and
                st- $cv_g^{(1)}$ = True and st- $cv_h^{(2)}$ = True) then
                if ($cv_g^{(1)} \neq cv_h^{(2)}$) then
                    parallel (Call-1, Call-2) = True
                end if
            end if
        end for
    end if
end for

Scenario 4 (Intra-Component Parallelism):

Consider two simple calls to subroutines Sub-1 and Sub-2 from Component-X. If Sub-1 and Sub-2 have common actual parameters, and Sub-1 and Sub-2 access (read/write) Labeled and/or Un-labeled Common Variables at different locations, and the common actual parameters themselves are not modified, then the calls to Sub-1 and Sub-2 may be parallelized. Scenario 4 is formally defined as:

Let Call-1 and Call-2 be two subroutine calls from Component-X.
Let Sub-1 and Sub-2 be the called subroutines in Call-1 and Call-2 respectively.
Let $ap_1^{(1)}, ap_2^{(1)}, \ldots, ap_r^{(1)}$ be the actual parameters of subroutine Sub-1 in call Call-1.
Let $ap_1^{(2)}, ap_2^{(2)}, \ldots, ap_s^{(2)}$ be the actual parameters of subroutine Sub-2 in call Call-2.
Let $cv_1^{(1)}, cv_2^{(1)}, \ldots, cv_t^{(1)}$ be the common variables of subroutine Sub-1.
Let $cv_1^{(2)}, cv_2^{(2)}, \ldots, cv_u^{(2)}$ be the common variables of subroutine Sub-2.
Let st-$cv_x^{(y)}$ be a boolean variable which indicates whether or not a state change occurs to common variable cv_x in subroutine Sub-y.
Let st-$ap_w^{(z)}$ be a boolean variable which indicates whether or not a state change occurs to formal parameter aliased to the actual parameter ap_w in subroutine Sub-z.

```
for all ( i = 1, r and j = 1, s)
    if ( exists m and n such that 1≤ m ≤ r, 1 ≤ n ≤ s and ap_m^(1) = ap_n^(2) ) then
        if ( st-ap_m^(1) = False and st-ap_n^(2) = False ) then
            for all ( k = 1, t and l = 1, u)
                if ( exists g and h such that 1 ≤ g ≤ t, 1 ≤ h ≤ u and
                    st- cv_g^(1) = True and st- cv_h^(2) = True ) then
                    if (cv_g^(1) ≠ cv_h^(2) ) then
                        parallel (Call-1, Call-2) = True
                    end if
                end if
            end for
        end if
    end if
end for
```

Scenario 5 (Intra-Component Parallelism):

Consider a call to subroutine Sub from Component-X within a loop. Let $Sub^{(i)}$ be the call in the i th iteration of the loop and $Sub^{(i+1)}$ be the call in the i+1 th iteration of the loop. If the actual parameters of $Sub^{(i)}$ and $Sub^{(i+1)}$ are different and Common vari-

ables are not modified, all the iterations of the loop may be parallelized. The formal definition of Scenario 5 is:

Let $Call^{[I]}$ and $Call^{[I+1]}$ be two calls to the same subroutine Sub from Component-X within a loop in iteration I and iteration I+1 respectively.
Let $Sub^{[I]}$ and $Sub^{[I+1]}$ be the calls to subroutine Sub iteration I and iteration I+1 respectively.
Let $ap_1^{[I]}, ap_2^{[I]}, \ldots, ap_r^{[I]}$ be the actual parameters of subroutine Sub in iteration I.
Let $ap_1^{[I+1]}, ap_2^{[I+1]}, \ldots, ap_s^{[I+1]}$ be the actual parameters of subroutine Sub in iteration I+1.
Let $cv_1^{[I]}, cv_2^{[I]}, \ldots, cv_t^{[I]}$ be the common variables of subroutine Sub in iteration I.
Let $cv_1^{[I+1]}, cv_2^{[I+1]}, \ldots, cv_u^{[I+1]}$ be the common variables of subroutine Sub in iteration I+1.
Let st-cv_x be a boolean variable which indicates whether or not a state change occurs to common variable cv_x in subroutine Sub-y.
Let st-ap_w be a boolean variable which indicates whether or not a state change occurs to formal parameter aliased to the actual parameter ap_w in subroutine Sub-z.

```
for all ( i = 1, r and j = 1, s)
        if ( ap_i^[I] ≠ ap_j^[I+1] ) then
                for all ( k = 1, t and l = 1, u)
                        if ( st-cv_k^[I] = False and st-cv_l^[I+1] = False ) then
                                parallel (Call^[I], Call^[I+1]) in loop = True
                        end if
                end for
        end if
end for
```

Scenario-6 (Inter-Component Parallelism):

This scenario applies only to loops within a given component. Let $\{St_1, St_2, \ldots, St_m\}$ be a set of state changes within a loop. Let $\{St_1^{(1)}, St_2^{(1)}, \ldots, St_m^{(1)}\}$, $\{St_1^{(2)}, St_2^{(2)}, \ldots, St_m^{(2)}\}, \ldots, \{St_1^{(n)}, St_2^{(n)}, \ldots, St_m^{(n)}\}$ be the set of state changes with the loop unrolled. If for all j = 1, 2, . . ., m, and k = 1, 2, . . ., n, $St_j^{(k)}$ are independent, then set of computations described by $\{St_1, St_2, \ldots, St_m\}$ may be partitioned into parallel computations. Scenario 6 is formally defined as:

Let $St_1, St_2, \ldots, St_m$ be a set of state changes within a loop in Component-X.
Let $St_1^{[1]}, St_2^{[1]}, \ldots, St_m^{[1]}$ be the state changes in iteration 1.
Let $St_1^{[N]}, St_2^{[N]}, \ldots, St_m^{[N]}$ be the state changes in iteration N.
Let $Site_1, Site_2, \ldots, Site_m$ be the list of sites for the corresponding state changes.
Let $Site_s\text{-}ulv_1, Site_s\text{-}ulv_2, \ldots, Site_s\text{-}ulv_p$ be the use list variables of site s.
Let $Site_s\text{-}dlv_1, Site_s\text{-}dlv_2, \ldots, Site_s\text{-}dlv_q$ be the definition list variables of site s.
for all (j = 1, m and k = 1, N)
    for all (l = 1, p and r = 1, q)
        if (not exists t and u $1 \leq t, u \leq m$ such that
            $Site_t\text{-}dlv_r = Site_u\text{-}ulv_l$) then
                partition ($St_1, St_2, \ldots, St_m$) = *True*
        end if
    end for
end for

Table 4.16 shows the encoding of the general purpose functions (rules) and the encoding of the scenarios using CLIPS deffunction construct. Explanation of the functions is provided in Table 4.16 using CLIPS commenting style. All characters after a semicolon are ignored by CLIPS.

4.5 Execution

The program facts, dependence facts, and the knowledge repository are loaded into the CLIPS environment using the load command. Each component has two knowledge-base files: 1) program fact file, and 2) dependence fact file. Once all the facts are in place, a reset command is issued. The reset command creates object instances of appropriate classes for each component. Subsequently, the run command is issued and the parallel design recommendations are output to the user. Chapter 5 presents experimental results from RETK and the parallelization methodology.

Table 4.16 Representation of the knowledge repository

```
(deffunction extcalls (?parent)      ;      This function extracts the subroutine calls of a
(do-for-all-instances ((?a Calls))   ;      component identified by the variable ?parent
        TRUE
        (if (eq (send ?a get-comp-name) ?parent)
        then
        (printout t ?a:sub-name crlf)))
        (return)
        else
        (printout t "No Subroutine Calls!" crlf))

(deffunction extfuns (?parent)       ;      This function extracts the function calls of a
(do-for-all-instances ((?a Func-Calls)) ;   component identified by the variable ?parent
        TRUE
        (if (eq (send ?a get-comp-name) ?parent)
        then
        (printout t ?a:fun-name crlf)))
        (return)
        else
        (printout t "No Function Calls!" crlf))

(deffunction get-number-of-calls (?ins-name ?ins-type) ; Return number of subroutine calls
(if (class-existp ?ins-type)                           ; of a class instance ?ins-name
then                                                   ; and class type ?ins-type
(do-for-all-instances ((?x ?ins-type))
        TRUE
        (if (and (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine)
                     (eq ?ins-type Function))
          (eq ?x:Name ?ins-name))
        then
        (return ?x:num-calls)))
        FALSE
        (if (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine)
              (eq ?ins-type Function))
        then
        (printout t ?ins-name ": No such instance of type " ?ins-type crlf)
       (return))
        FALSE
        (printout t "Query type not allowed for Class " ?ins-type crlf)
        (return))
        FALSE
        (if (not (class-existp ?ins-type))
        then
        (printout t ?ins-type ": No such class" crlf)))
```

(table con'd.)

```
(deffunction get-number-of-funcs (?ins-name ?ins-type) ; Return number of subroutine calls
(if (class-existp ?ins-type)                            ; of a class instance ?ins-name
then                                                    ; and class type ?ins-type
(do-for-all-instances ((?x ?ins-type))
        TRUE
        (if (and (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine)
                     (eq ?ins-type Function))
          (eq ?x:Name ?ins-name))
        then
        (return ?x:num-funcs)))
        FALSE
        (if (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine)
              (eq ?ins-type Function))
        then
        (printout t ?ins-name ": No such instance of type " ?ins-type crlf)
    (return))
        FALSE
    (printout t "Query type not allowed for Class " ?ins-type crlf)
    (return))
        FALSE
    (if (not (class-existp ?ins-type))
    then
    (printout t ?ins-type ": No such class" crlf)))

(deffunction get-number-of-locals (?ins-name ?ins-type) ; Return number of local variables
(if (class-existp ?ins-type)
then
(do-for-all-instances ((?x ?ins-type))
        TRUE
        (if (and (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine)
                     (eq ?ins-type Function))
          (eq ?x:Name ?ins-name))
        then
        (return ?x:num-locals)))
        FALSE
        (if (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine)
              (eq ?ins-type Function))
        then
        (printout t ?ins-name ": No such instance of type " ?ins-type crlf)
    (return))
        FALSE
    (printout t "Query type not allowed for Class " ?ins-type crlf)
    (return))
        FALSE
    (if (not (class-existp ?ins-type))
    then
    (printout t ?ins-type ": No such class" crlf)))
```

(table con'd.)

```
(deffunction get-number-of-states (?ins-name ?ins-type) ; Return number of states
(if (class-existp ?ins-type)
then
(do-for-all-instances ((?x ?ins-type))
        TRUE
        (if (and (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine) (eq ?ins-type
Function))
          (eq ?x:Name ?ins-name))
        then
        (return ?x:num-states)))
        FALSE
        (if (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine)
            (eq ?ins-type Function))
        then
        (printout t ?ins-name ": No such instance of type " ?ins-type crlf)
    (return))
        FALSE
    (printout t "Query type not allowed for Class " ?ins-type crlf)
    (return))
        FALSE
    (if (not (class-existp ?ins-type))
    then
    (printout t ?ins-type ": No such class" crlf)))

(deffunction get-number-of-labeled-commons (?ins-name ?ins-type)
(if (class-existp ?ins-type)                    ; Return number of labeled commons
then
(do-for-all-instances ((?x ?ins-type))
        TRUE
        (if (and (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine)
                (eq ?ins-type Function))
          (eq ?x:Name ?ins-name))
        then
        (return ?x:num-labeled-commons)))
        FALSE
        (if (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine)
            (eq ?ins-type Function))
        then
        (printout t ?ins-name ": No such instance of type " ?ins-type crlf)
    (return))
        FALSE
    (printout t "Query type not allowed for Class " ?ins-type crlf)
    (return))
        FALSE
    (if (not (class-existp ?ins-type))
    then
    (printout t ?ins-type ": No such class" crlf)))
```

(table con'd.)

```
(deffunction get-number-of-un-labeled-commons (?ins-name ?ins-type)
(if (class-existp ?ins-type)                    ; Return number of un-labeled commons
then
(do-for-all-instances ((?x ?ins-type))
        TRUE
        (if (and (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine) (eq ?ins-type
Function))
          (eq ?x:Name ?ins-name))
        then
        (return ?x:num-un-labeled-commons)))
        FALSE
        (if (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine)
              (eq ?ins-type Function))
        then
        (printout t ?ins-name ": No such instance of type " ?ins-type crlf)
    (return))
        FALSE
    (printout t "Query type not allowed for Class " ?ins-type crlf)
    (return))
        FALSE
    (if (not (class-existp ?ins-type))
    then
    (printout t ?ins-type ": No such class" crlf)))

(deffunction get-number-of-do-loops (?ins-name ?ins-type)
(if (class-existp ?ins-type)                    ; Return number of do loops
then
(do-for-all-instances ((?x ?ins-type))
        TRUE
        (if (and (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine)
                  (eq ?ins-type Function))
          (eq ?x:Name ?ins-name))
        then
        (return ?x:num-do-loops)))
        FALSE
        (if (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine)
             (eq ?ins-type Function))
        then
        (printout t ?ins-name ": No such instance of type " ?ins-type crlf)
    (return))
        FALSE
    (printout t "Query type not allowed for Class " ?ins-type crlf)
    (return))
        FALSE
    (if (not (class-existp ?ins-type))
    then
    (printout t ?ins-type ": No such class" crlf)))
```

(table con'd.)

```
(deffunction get-number-of-if-conditions (?ins-name ?ins-type)
(if (class-existp ?ins-type)                    ; Return number of if conditions
then
(do-for-all-instances ((?x ?ins-type))
        TRUE
        (if (and (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine)
                 (eq ?ins-type Function))
          (eq ?x:Name ?ins-name))
        then
        (return ?x:num-if-conds)))
        FALSE
        (if (or (eq ?ins-type Main-Program) (eq ?ins-type Subroutine)
              (eq ?ins-type Function))
        then
        (printout t ?ins-name ": No such instance of type " ?ins-type crlf)
    (return))
        FALSE
    (printout t "Query type not allowed for Class " ?ins-type crlf)
    (return))
        FALSE
    (if (not (class-existp ?ins-type))
    then
    (printout t ?ins-type ": No such class" crlf)))



(deffunction scenario-1-1 (?comp)
(do-for-instance ((?f1 Func-Calls) (?f2 Func-Calls))
        (and    (eq ?f1:fun-name ?f2:fun-name)
                (eq ?f1:comp-name ?f2:comp-name)
                (eq ?f1:comp-name ?comp)
                (neq ?f1:fun-actuals ?f2:fun-actuals)
                (find-instance ((?F1 Function))
                        (and  (eq ?F1:Name ?f1:fun-name)
                              (eq ?F1:num-labeled-commons nil)
                              (eq ?F1:num-un-labeled-commons nil)))
                (find-instance ((?F2 Function))
                        (and  (eq ?F2:Name ?f2:fun-name)
                              (eq ?F2:num-labeled-commons nil)
                              (eq ?F2:num-un-labeled-commons nil))))
        (printout t "Scenario 1.1 " crlf
                  "---------- " crlf crlf
                  "Function Call " ?f1:fun-name " of "?f1:comp-name
                  " at line number " ?f1:fun-line crlf
                  " <-- Can be Parallelized with --> " crlf
                  "Function Call" ?f2:fun-name " of "?f2:comp-name
                  " at line number " ?f2:fun-line crlf crlf))
```

(table con'd.)

```
        (return)
        FALSE
        (printout t "Calls cannot be parallelized  " ?comp crlf))


(deffunction scenario-1-2 (?comp)
(do-for-instance ((?s1 Calls) (?s2 Calls))
        (and    (eq ?s1:sub-name ?s2:sub-name)
                (eq ?s1:comp-name ?s2:comp-name)
                (eq ?s1:comp-name ?comp)
                (neq ?s1:sub-actuals ?s2:sub-actuals)
                (find-instance ((?S1 Subroutine))
                        (and    (eq ?S1:Name ?s1:sub-name)
                                (eq ?S1:num-labeled-commons nil)
                                (eq ?S1:num-un-labeled-commons nil)))
                (find-instance ((?S2 Subroutine))
                        (and    (eq ?S2:Name ?s2:sub-name)
                                (eq ?S2:num-labeled-commons nil)
                                (eq ?S2:num-un-labeled-commons nil))))
        (printout t "Scenario 1.2 " crlf
                  "---------- " crlf crlf
                  "Subroutine Call " ?s1:sub-name " of "?s1:comp-name
                  " at line number " ?s1:sub-line crlf
                  " <-- Can be Parallelized with --> " crlf
                  "Subroutine Call " ?s2:sub-name " of "?s2:comp-name
                  " at line number " ?s2:sub-line  crlf crlf))
        (return)
        FALSE
        (printout t "Calls cannot be parallelized  " ?comp crlf))


(deffunction scenario-1-3 (?comp)
(do-for-instance ((?s1 Calls) (?f2 Func-Calls))
        (and    (eq ?s1:sub-name ?f2:fun-name)
                (eq ?s1:comp-name ?f2:comp-name)
                (eq ?s1:comp-name ?comp)
                (neq ?s1:sub-actuals ?f2:fun-actuals)
                (find-instance ((?S1 Subroutine))
                        (and    (eq ?S1:Name ?s1:sub-name)
                                (eq ?S1:num-labeled-commons nil)
                                (eq ?S1:num-un-labeled-commons nil)))
                (find-instance ((?F2 Function))
                        (and    (eq ?F2:Name ?f2:fun-name)
                                (eq ?F2:num-labeled-commons nil)
                                (eq ?F2:num-un-labeled-commons nil))))
```

(table con'd.)

```
        (printout t "Scenario 1.3 " crlf
                  "---------- " crlf crlf
                  "Subroutine Call " ?s1:sub-name " of "?s1:comp-name
                  " at line number " ?s1:sub-line crlf
                  " <-- Can be Parallelized with --> " crlf
                  "Function Call " ?f2:fun-name " of "?f2:comp-name
                  " at line number " ?f2:fun-line  crlf crlf))
        (return)
        FALSE
        (printout t "Calls cannot be parallelized  " ?comp crlf))

(deffunction scenario-2-1 (?comp)
(do-for-instance ((?f1 Func-Calls) (?f2 Func-Calls))
        (and    (eq ?f1:fun-name ?f2:fun-name)
                (eq ?f1:comp-name ?f2:comp-name)
                (eq ?f1:comp-name ?comp)
                (eq ?f1:fun-actuals ?f2:fun-actuals)
                (find-instance ((?s1 State-Changes))
                        (eq ?s1:comp-name ?f1:fun-name))
                (find-instance ((?s2 State-Changes))
                        (eq ?s2:comp-name ?f2:fun-name))
                        (or (and (eq ?s1:Var-Name ?f1:fun-actuals)
                                 (neq ?s2:Var-Name ?f2:fun-actulas))
                            (and (neq ?s1:Var-Name ?f1:fun-actuals)
                                 (eq ?s2:Var-Name ?f2:fun-actuals)))
                (find-instance ((?F1 Function))
                        (and  (eq ?F1:Name ?f1:fun-name)
                              (eq ?F1:num-labeled-commons nil)
                              (eq ?F1:num-un-labeled-commons nil)))
                (find-instance ((?F2 Function))
                        (and  (eq ?F2:Name ?f2:fun-name)
                              (eq ?F2:num-labeled-commons nil)
                              (eq ?F2:num-un-labeled-commons nil))))
        (printout t "Scenario 2.1 " crlf
                  "---------- " crlf crlf
                  "Function Call " ?f1:fun-name " of "?f1:comp-name
                  " at line number " ?f1:fun-line crlf
                  " <-- Can be Parallelized with --> " crlf
                  "Function Call" ?f2:fun-name " of "?f2:comp-name
                  " at line number " ?f2:fun-line  crlf crlf))
        (return)
        FALSE
        (printout t "Calls cannot be parallelized  " ?comp crlf))
```

(table con'd.)

```
(deffunction scenario-2-2 (?comp)
(do-for-instance ((?s1 Calls) (?s2 Calls))
        (and    (eq ?s1:sub-name ?s2:sub-name)
                (eq ?s1:comp-name ?s2:comp-name)
                (eq ?s1:comp-name ?comp)
                (eq ?s1:sub-actuals ?s2:sub-actuals)
                (find-instance ((?t1 State-Changes))
                        (eq ?t1:comp-name ?s1:sub-name))
                (find-instance ((?t2 State-Changes))
                        (eq ?t2:comp-name ?s2:sub-name))
                        (or (and (eq ?t1:Var-Name ?s1:sub-actuals)
                                (neq ?t2:Var-Name ?s2:sub-actulas))
                          (and (neq ?t1:Var-Name ?s1:sub-actuals)
                                (eq ?t2:Var-Name ?s2:sub-actuals)))
                (find-instance ((?S1 Subroutine))
                        (and  (eq ?S1:Name ?s1:fun-name)
                                (eq ?S1:num-labeled-commons nil)
                                (eq ?S1:num-un-labeled-commons nil)))
                (find-instance ((?S2 Subroutine))
                        (and  (eq ?S2:Name ?s2:fun-name)
                                (eq ?S2:num-labeled-commons nil)
                                (eq ?S2:num-un-labeled-commons nil))))
        (printout t "Scenario 2.2 " crlf
                  "---------- " crlf crlf
                  "Subroutine Call " ?s1:sub-name " of "?s1:comp-name
                  " at line number " ?s1:sub-line crlf
                  " <-- Can be Parallelized with --> " crlf
                  "Subroutine Call " ?s2:sub-name " of "?s2:comp-name
                  " at line number " ?s2:sub-line  crlf crlf))
        (return)
        FALSE
        (printout t "Calls cannot be parallelized  " ?comp crlf))

(deffunction scenario-2-3 (?comp)
(do-for-instance ((?s1 Calls) (?f2 Func-Calls))
        (and    (eq ?s1:sub-name ?f2:fun-name)
                (eq ?s1:comp-name ?f2:comp-name)
                (eq ?s1:comp-name ?comp)
                (eq ?s1:sub-actuals ?f2:fun-actuals)
                (find-instance ((?t1 State-Changes))
                        (eq ?t1:comp-name ?s1:sub-name))
                (find-instance ((?t2 State-Changes))
                        (eq ?t2:comp-name ?f2:fun-name))
                        (or (and (eq ?t1:Var-Name ?s1:sub-actuals)
                                (neq ?t2:Var-Name ?f2:fun-actulas))
                          (and (neq ?t1:Var-Name ?s1:sub-actuals)
                                (eq ?t2:Var-Name ?f2:fun-actuals)))
```

(table con'd.)

```
                (find-instance ((?S1 Subroutine))
                        (and (eq ?S1:Name ?s1:sub-name)
                        (eq ?S1:num-labeled-commons nil)
                        (eq ?S1:num-un-labeled-commons nil)))
                (find-instance ((?F2 Function))
                        (and (eq ?F2:Name ?f2:fun-name)
                        (eq ?F2:num-labeled-commons nil)
                        (eq ?F2:num-un-labeled-commons nil))))
        (printout t "Scenario 2.3 " crlf
                  "---------- " crlf crlf
                  "Subroutine Call " ?s1:sub-name " of "?s1:comp-name
                  " at line number " ?s1:sub-line crlf
                  " <-- Can be Parallelized with --> " crlf
                  "Function Call " ?f2:fun-name " of "?f2:comp-name
                  " at line number " ?f2:fun-line  crlf crlf))
        (return)
        FALSE
        (printout t "Calls cannot be parallelized  " ?comp crlf))

(deffunction scenario-3-1 (?comp)
(do-for-instance ((?f1 Func-Calls) (?f2 Func-Calls))
        (and    (eq ?f1:fun-name ?f2:fun-name)
                (eq ?f1:comp-name ?f2:comp-name)
                (eq ?f1:comp-name ?comp)
                (neq ?f1:fun-actuals ?f2:fun-actuals)
                (find-instance ((?F1 Function))
                        (and  (eq ?F1:Name ?f1:fun-name)
                        (or (neq ?F1:num-labeled-commons nil)
                          (neq ?F1:num-un-labeled-commons nil))))
                (find-instance ((?F2 Function))
                        (and  (eq ?F2:Name ?f2:fun-name)
                        (or (eq ?F2:num-labeled-commons nil)
                          (eq ?F2:num-un-labeled-commons nil))))
                (find-instance ((?LCom1 Labeled-Commons))
                        (eq ?LCom1:comp-name ?f1:fun-name))
                (find-instance ((?LCom2 Labeled-Commons))
                        (eq ?LCom2:comp-name ?f2:fun-name))
                (find-instance ((?t1 State-Changes))
                        (eq ?t1:comp-name ?f1:fun-name))
                (find-instance ((?t2 State-Changes))
                        (eq ?t2:comp-name ?f2:fun-name))
                        (or (and (eq ?t1:Var-Name ?LCom1:Commons)
                        (neq ?t2:Var-Name ?LCom2:Commons))
                        (and (neq ?t1:Var-Name ?LCom1:Commons)
                        (eq ?t2:Var-Name ?LCom1:Commons))))
```

(table con'd.)

```
        (printout t "Scenario 3.1 " crlf
                    "---------- " crlf crlf
                    "Function Call " ?f1:fun-name " of "?f1:comp-name
                    " at line number " ?f1:fun-line crlf
                    " <-- Can be Parallelized with --> " crlf
                    "Function Call" ?f2:fun-name " of "?f2:comp-name
                    " at line number " ?f2:fun-line  crlf crlf))
        (return)
        FALSE
        (printout t "Calls cannot be parallelized  " ?comp crlf))


(deffunction scenario-3-2 (?comp)
(do-for-instance ((?s1 Calls) (?s2 Calls))
        (and    (eq ?s1:sub-name ?s2:sub-name)
                (eq ?s1:comp-name ?s2:comp-name)
                (eq ?s1:comp-name ?comp)
                (neq ?s1:sub-actuals ?s2:sub-actuals)
                (find-instance ((?S1 Subroutine))
                        (and  (eq ?S1:Name ?s1:sub-name)
                        (or (neq ?S1:num-labeled-commons nil)
                        (neq ?S1:num-un-labeled-commons nil))))
                (find-instance ((?S2 Subroutine))
                        (and  (eq ?S2:Name ?s2:sub-name)
                        (or (eq ?S2:num-labeled-commons nil)
                        (eq ?S2:num-un-labeled-commons nil))))
                (find-instance ((?LCom1 Labeled-Commons))
                        (eq ?LCom1:comp-name ?s1:sub-name))
                (find-instance ((?LCom2 Labeled-Commons))
                        (eq ?LCom2:comp-name ?s2:sub-name))
                (find-instance ((?t1 State-Changes))
                        (eq ?t1:comp-name ?s1:sub-name))
                (find-instance ((?t2 State-Changes))
                        (eq ?t2:comp-name ?s2:sub-name))
                        (or (and (eq ?t1:Var-Name ?LCom1:Commons)
                        (neq ?t2:Var-Name ?LCom2:Commons))
                        (and (neq ?t1:Var-Name ?LCom1:Commons)
                        (eq ?t2:Var-Name ?LCom1:Commons))))

        (printout t "Scenario 3.2 " crlf
                    "---------- " crlf crlf
                    "Subroutine Call " ?s1:sub-name " of "?s1:comp-name
                    " at line number " ?s1:sub-line crlf
                    " <-- Can be Parallelized with --> " crlf
                    "Subroutine Call " ?s2:sub-name " of "?s2:comp-name
                    " at line number " ?s2:sub-line  crlf crlf))
        (return)
        FALSE
        (printout t "Calls cannot be parallelized  " ?comp crlf))
```

(table con'd.)

```
(deffunction scenario-3-3 (?comp)
(do-for-instance ((?s1 Calls) (?f2 Func-Calls))
        (and    (eq ?s1:sub-name ?f2:fun-name)
                (eq ?s1:comp-name ?f2:comp-name)
                (eq ?s1:comp-name ?comp)
                (neq ?s1:sub-actuals ?f2:fun-actuals)
                (find-instance ((?S1 Subroutine))
                        (and  (eq ?S1:Name ?s1:sub-name)
                        (or (neq ?S1:num-labeled-commons nil)
                          (neq ?S1:num-un-labeled-commons nil))))
                (find-instance ((?F2 Function))
                        (and  (eq ?F2:Name ?f2:fun-name)
                        (or (eq ?F2:num-labeled-commons nil)
                          (eq ?F2:num-un-labeled-commons nil))))
                (find-instance ((?LCom1 Labeled-Commons))
                        (eq ?LCom1:comp-name ?s1:sub-name))
                (find-instance ((?LCom2 Labeled-Commons))
                        (eq ?LCom2:comp-name ?f2:fun-name))
                (find-instance ((?t1 State-Changes))
                        (eq ?t1:comp-name ?s1:sub-name))
                (find-instance ((?t2 State-Changes))
                        (eq ?t2:comp-name ?f2:fun-name))
                        (or (and (eq ?t1:Var-Name ?LCom1:Commons)
                        (neq ?t2:Var-Name ?LCom2:Commons))
                        (and (neq ?t1:Var-Name ?LCom1:Commons)
                        (eq ?t2:Var-Name ?LCom1:Commons))))
        (printout t "Scenario 3.3 " crlf
                  "---------- " crlf crlf
                  "Subroutine Call " ?s1:sub-name " of "?s1:comp-name
                  " at line number " ?s1:sub-line crlf
                  " <-- Can be Parallelized with --> " crlf
                  "Function Call " ?f2:fun-name " of "?f2:comp-name
                  " at line number " ?f2:fun-line  crlf crlf))
        (return)
        FALSE
        (printout t "Calls cannot be parallelized  " ?comp crlf))


(deffunction scenario-4-1 (?comp)
(do-for-instance ((?f1 Func-Calls) (?f2 Func-Calls))
        (and    (eq ?f1:fun-name ?f2:fun-name)
                (eq ?f1:comp-name ?f2:comp-name)
                (eq ?f1:comp-name ?comp)
                (eq ?f1:fun-actuals ?f2:fun-actuals)
        (find-instance ((?F1 Function))
                (and  (eq ?F1:Name ?f1:fun-name)
                (or (neq ?F1:num-labeled-commons nil)
                  (neq ?F1:num-un-labeled-commons nil))))
```

(table con'd.)

```
        (find-instance ((?F2 Function))
                (and  (eq ?F2:Name ?f2:fun-name)
                        (or (eq ?F2:num-labeled-commons nil)
                          (eq ?F2:num-un-labeled-commons nil))))
        (find-instance ((?LCom1 Labeled-Commons))
                (eq ?LCom1:comp-name ?f1:fun-name))
        (find-instance ((?LCom2 Labeled-Commons))
                (eq ?LCom2:comp-name ?f2:fun-name))
        (find-instance ((?t1 State-Changes))
                (eq ?t1:comp-name ?f1:fun-name))
        (find-instance ((?t2 State-Changes))
                (eq ?t2:comp-name ?f2:fun-name))
                (or (and (eq ?t1:Var-Name ?LCom1:Commons)
                (neq ?t2:Var-Name ?LCom2:Commons))
                (and (neq ?t1:Var-Name ?LCom1:Commons)
                        (eq ?t2:Var-Name ?LCom1:Commons)))
        (find-instance ((?t3 State-Changes))
                (eq ?t3:comp-name ?f1:fun-name))
        (find-instance ((?t4 State-Changes))
                (eq ?t4:comp-name ?f2:fun-name))
                (or (and (eq ?t3:Var-Name ?f1:fun-actuals)
                  (neq ?t4:Var-Name ?f2:fun-actulas))
                  (and (neq ?t3:Var-Name ?f1:fun-actuals)
                  (eq ?t4:Var-Name ?f2:fun-actuals))))
        (printout t "Scenario 4.1 " crlf
                  "---------- " crlf crlf
                  "Function Call " ?f1:fun-name " of "?f1:comp-name
                  " at line number " ?f1:fun-line crlf
                  " <-- Can be Parallelized with --> " crlf
                  "Function Call" ?f2:fun-name " of "?f2:comp-name
                  " at line number " ?f2:fun-line  crlf crlf))
        (return)
        FALSE
        (printout t "Calls cannot be parallelized  " ?comp crlf))


(deffunction scenario-4-2 (?comp)
(do-for-instance ((?s1 Calls) (?s2 Calls))
        (and    (eq ?s1:sub-name ?s2:sub-name)
                (eq ?s1:comp-name ?s2:comp-name)
                (eq ?s1:comp-name ?comp)
                (eq ?s1:sub-actuals ?s2:sub-actuals)
                (find-instance ((?S1 Subroutine))
                        (and  (eq ?S1:Name ?s1:sub-name)
                        (or (neq ?S1:num-labeled-commons nil)
                          (neq ?S1:num-un-labeled-commons nil))))
                (find-instance ((?S2 Subroutine))
                        (and  (eq ?S2:Name ?s2:sub-name)
                        (or (eq ?S2:num-labeled-commons nil)
                          (eq ?S2:num-un-labeled-commons nil))))
```

(table con'd.)

```
                (find-instance ((?LCom1 Labeled-Commons))
                        (eq ?LCom1:comp-name ?s1:sub-name))
                (find-instance ((?LCom2 Labeled-Commons))
                        (eq ?LCom2:comp-name ?s2:sub-name))
                (find-instance ((?t1 State-Changes))
                        (eq ?t1:comp-name ?s1:sub-name))
                (find-instance ((?t2 State-Changes))
                        (eq ?t2:comp-name ?s2:sub-name))
                        (or (and (eq ?t1:Var-Name ?LCom1:Commons)
                        (neq ?t2:Var-Name ?LCom2:Commons))
                        (and (neq ?t1:Var-Name ?LCom1:Commons)
                        (eq ?t2:Var-Name ?LCom1:Commons)))
                (find-instance ((?t3 State-Changes))
                        (eq ?t3:comp-name ?s1:sub-name))
                (find-instance ((?t4 State-Changes))
                        (eq ?t4:comp-name ?s2:sub-name))
                        (or (and (eq ?t3:Var-Name ?s1:sub-actuals)
                                (neq ?t4:Var-Name ?s2:sub-actulas))
                        (and (neq ?t3:Var-Name ?s1:sub-actuals)
                                (eq ?t4:Var-Name ?s2:sub-actuals))))
        (printout t "Scenario 4.2 " crlf
                  "---------- " crlf crlf
                  "Subroutine Call " ?s1:sub-name " of "?s1:comp-name
                  " at line number " ?s1:sub-line crlf
                  " <-- Can be Parallelized with --> " crlf
                  "Subroutine Call " ?s2:sub-name " of "?s2:comp-name
                  " at line number " ?s2:sub-line  crlf crlf))
        (return)
        FALSE
        (printout t "Calls cannot be parallelized  " ?comp crlf))


(deffunction scenario-4-3 (?comp)
(do-for-instance ((?s1 Calls) (?f2 Func-Calls))
        (and    (eq ?s1:sub-name ?f2:fun-name)
                (eq ?s1:comp-name ?f2:comp-name)
                (eq ?s1:comp-name ?comp)
                (eq ?s1:sub-actuals ?f2:fun-actuals)
                (find-instance ((?S1 Subroutine))
                        (and  (eq ?S1:Name ?s1:sub-name)
                        (or (neq ?S1:num-labeled-commons nil)
                          (neq ?S1:num-un-labeled-commons nil))))
                (find-instance ((?F2 Function))
                        (and  (eq ?F2:Name ?f2:fun-name)
                        (or (eq ?F2:num-labeled-commons nil)
                          (eq ?F2:num-un-labeled-commons nil))))
                (find-instance ((?LCom1 Labeled-Commons))
                        (eq ?LCom1:comp-name ?s1:sub-name))
                (find-instance ((?LCom2 Labeled-Commons))
                        (eq ?LCom2:comp-name ?f2:fun-name))
```

(table con'd.)

```
                (find-instance ((?t1 State-Changes))
                        (eq ?t1:comp-name ?s1:sub-name))
                (find-instance ((?t2 State-Changes))
                        (eq ?t2:comp-name ?f2:fun-name))
                        (or (and (eq ?t1:Var-Name ?LCom1:Commons)
                                (neq ?t2:Var-Name ?LCom2:Commons))
                          (and (neq ?t1:Var-Name ?LCom1:Commons)
                                (eq ?t2:Var-Name ?LCom1:Commons)))
                (find-instance ((?t3 State-Changes))
                        (eq ?t3:comp-name ?s1:sub-name))
                (find-instance ((?t4 State-Changes))
                        (eq ?t4:comp-name ?f2:fun-name))
                        (or (and (eq ?t3:Var-Name ?s1:sub-actuals)
                        (neq ?t4:Var-Name ?f2:fun-actulas))
                          (and (neq ?t3:Var-Name ?s1:sub-actuals)
                        (eq ?t4:Var-Name ?f2:fun-actuals))))
        (printout t "Scenario 4.3 " crlf
                  "---------- " crlf crlf
                  "Subroutine Call " ?s1:sub-name " of "?s1:comp-name
                  " at line number " ?s1:sub-line crlf
                  " <-- Can be Parallelized with --> " crlf
                  "Function Call " ?f2:fun-name " of "?f2:comp-name
                 " at line number " ?f2:fun-line  crlf crlf))
        (return)
        FALSE
        (printout t "Calls cannot be parallelized  " ?comp crlf))


(deffunction scenario-5 (?comp)
(do-for-instance ((?s1 Calls))
        (and    (eq ?s1:comp-name ?comp)
                (find-instance ((?I1 DO-Loop-Information))
                        (eq ?I1:comp-name ?s1:comp-name))
                (> ?s1:sub-line ?I1:DO-Start-Line)
                (<= ?s1:sub-line ?I1:DO-Last-Line)
                (neq ?s1:sub-actuals ?s2:sub-actuals)
                (find-instance ((?S1 Subroutine))
                        (and  (eq ?S1:Name ?s1:sub-name)
                                (eq ?S1:num-labeled-commons nil)
                                (eq ?S1:num-un-labeled-commons nil)))
        (printout t "Scenario 5 " crlf
                  "---------- " crlf crlf
                  "Subroutine Call " ?s1:sub-name " of "?s1:comp-name
                  " at line number " ?s1:sub-line crlf
                  " <-- Can be Parallelized within the loop --> " crlf
                  " Identified by " ?I1:Label " at Relative Line " crlf
                  " Line Number " ?I1:DO-Start-Line crlf crlf))
```

(table con'd.)

```
        (return)
        FALSE
        (printout t "Calls cannot be parallelized  " ?comp crlf))


(deffunction scenario-6 (?comp)
(do-for-instances ((?s1 State-Changes))
        (and    (eq ?s1:comp-name ?comp)
                (find-instance ((?I1 DO-Loop-Information))
                        (eq ?I1:comp-name ?s1:comp-name))
                (> ?s1:sub-line ?I1:DO-Start-Line)
                (<= ?s1:sub-line ?I1:DO-Last-Line)
                (not (find-instance ((?E1 Site))
                        (eq ?E1:comp-name ?s1:comp-name))
                      (find-instance ((?DLV Def-Lists))
                        (eq ?DLV:comp-name ?E1:comp-name))
                      (find-instance ((?ULV Use-Lists))
                        (eq ?ULV:comp-name ?E1:comp-name))
                        (eq ?DLV:Def-List-Name ?ULV:Use-List-Name)))
        (printout t "Scenario 6 " crlf
                  "---------- " crlf crlf
                  "State Changes " ?s1:Var-name " of "?s1:comp-name
                  " at line number " ?s1:Line-Num crlf
                  " <-- Can be Parallelized within the loop --> " crlf
                  " Identified by " ?I1:Label " at Relative Line " crlf
                  " Line Number " ?I1:DO-Start-Line crlf crlf))
        (return)
        FALSE
        (printout t "State Changes cannot be parallelized  " ?comp crlf))
```

Chapter 5

Experimental Results

In this chapter, we present experimental results of applying the methodology to test programs by the reverse engineering toolkit (RETK). To demonstrate the full capabilities of the methodology and of RETK, comprehensive results for a simple sample program are described. Summary results for larger programs are then provided.

5.1 A Sample Program

A sample program which reads two sets of data values for a statistics experiment and computes the standard deviation of the each set of experimental values is shown in Table 5.1. The analysis of the sample program by RETK involves the extraction of the sequential design description by the Information Extractor (IE), dependence analysis by the Dependence Analyzer (DANA), and derivation of parallel design recommendations by the CLIPS-based Design Assistant (DA). The output of each component is given.

5.1.1 Information Extraction

The output of the Information Extractor is the sequential design of the original program. Snapshots of the session with IE are shown in Figures 5.1 - 5.3. Metric information for an original assessment of the program is shown in Figure 5.1. Figure 5.1

Table 5.1 A sample FORTRAN program

```
C
C     MAIN PROGRAM TO CALCULATE THE STANDARD DEVIATION
C
      PROGRAM STATISTICS
      INTEGER M, N
C     EXPA - OBSERVATIONS FROM EXPERIMENT A
C     EXPB - OBSERVATIONS FROM EXPERIMENT B
      REAL EXPA(14), EXPB(14), STDA, STDB
      CALL INPUT(M, EXPA, N, EXPB)
C     STDA - STANDARD DEVIATION OF OBSERVATIONS OF EXPERIMENT A
      STDA = STD(EXPA,M)
C     STDB - STANDARD DEVIATION OF OBSERVATIONS OF EXPERIMENT B
      STDB = STD(EXPB, N)
      CALL PRINT(EXPA, M, STDA, EXPB, N, STDB)
      END
C     *** SUBROUTINE INPUT ***
      SUBROUTINE INPUT(M,EXPA,N,EXPB)
      INTEGER M, N
      REAL EXPA(14), EXPB(14)
      READ(5,*) M
      DO 10 I=1,14
         READ(5,15) EXPA(I)
15       FORMAT (F4.1)
10    CONTINUE
      READ(5,*) N
      DO 20 J=1, 14
         READ(5,30) EXPB(J)
30       FORMAT (F4.1)
20    CONTINUE
      END
C     *** FUNCTION STD ***
      FUNCTION STD(EXPX,X)
      INTEGER X
      REAL MEAN, EXPX(14), IND(14), TOT
      TOT = 0
      SUM = 0
      DO 60 I = 1,X
         TOT = TOT + EXPX(I)
60    CONTINUE
      MEAN = TOT / X
      DO 70 J = 1,X
         IND(J) = MEAN - EXPX(J)
         SUM = SUM + IND(J)**2
70    CONTINUE
      STD = SQRT(SUM/(X-1))
      END
C     *** SUBROUTINE PRINT ***
      SUBROUTINE PRINT(EXPA,M,STDA,EXPB,N,STDB)
      REAL STDA, STDB, EXPA(14), EXPB(14)
      WRITE(6,80) 'EXPERIMENT A ----', 'MEASUREMENTS',
     *            ((EXPA(I)),I=1,M)
80    FORMAT (T2,A20/T5,A14/(T10,F4.1))
      WRITE(6,90) 'STANDARD DEVIATION', STDA
90    FORMAT ('0'//T5,A22,F5.2)
      WRITE(6,100) 'EXPERIMENT B ----', 'MEASUREMENTS',
     *             ((EXPB(J)),J=1,N)
100   FORMAT ('0'////T2,A20/T5,A14/(T10,F4.1))
      WRITE(6,90) 'STANDARD DEVIATION', STDB
      RETURN
      END
```

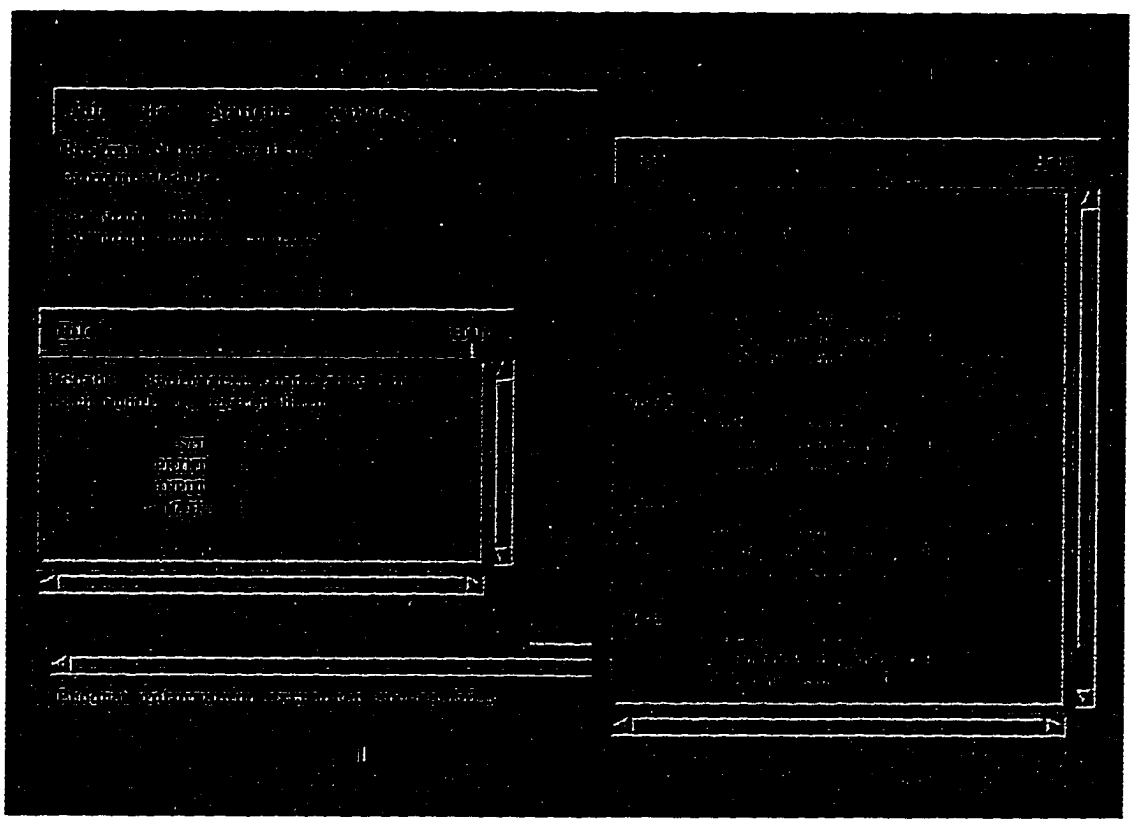


Figure 5.1 Metric Information of the sample program

reveals that there are 4 components in the program: the MAIN program (called STATS), 2 subroutines, INPUT and PRINT, and 1 user-defined function STD. The call graph of the program is shown in Figure 5.2. Figure 5.2 shows that the MAIN program calls subroutine INPUT once, function STD twice, and subroutine PRINT once. Neither INPUT nor PRINT calls any other component. Function STD does not call any other component but makes use of the FORTRAN built-in square-root function SQRT*. The asterisks at the end of SQRT* is an indication to the user that it is a built-in function. Each component of the original program is represented as an X/Motif-based button in the call graph representation. An user may click any of the components to obtain complete information about the component. The information includes the name of the component, metric information, subroutine call information, function call information, formal parameters, if any, of the component, variable description and state change information. Figure 5.3 shows an X/Motif-based scrollable text window for click event of the MAIN program component. A complete description of the design of the MAIN program is given in Tables 5.2-5.7.

Metric information like total number of lines in the module, number of commented lines, number of blank lines, number of discriminations in the module, and McCabe's cyclomatic complexity are listed in Table 5.2. LOC (lines of code), another widely accepted metric, can be calculated directly from the information listed in Table 5.2 by subtracting the sum of commented and blank lines from the total number of lines in the module. Tables 5.3 and 5.4 list the details of subroutines and functions called from MAIN. The *type* column in Tables 5.3 and 5.4 describes the manner in which a particular subroutine or function is called.

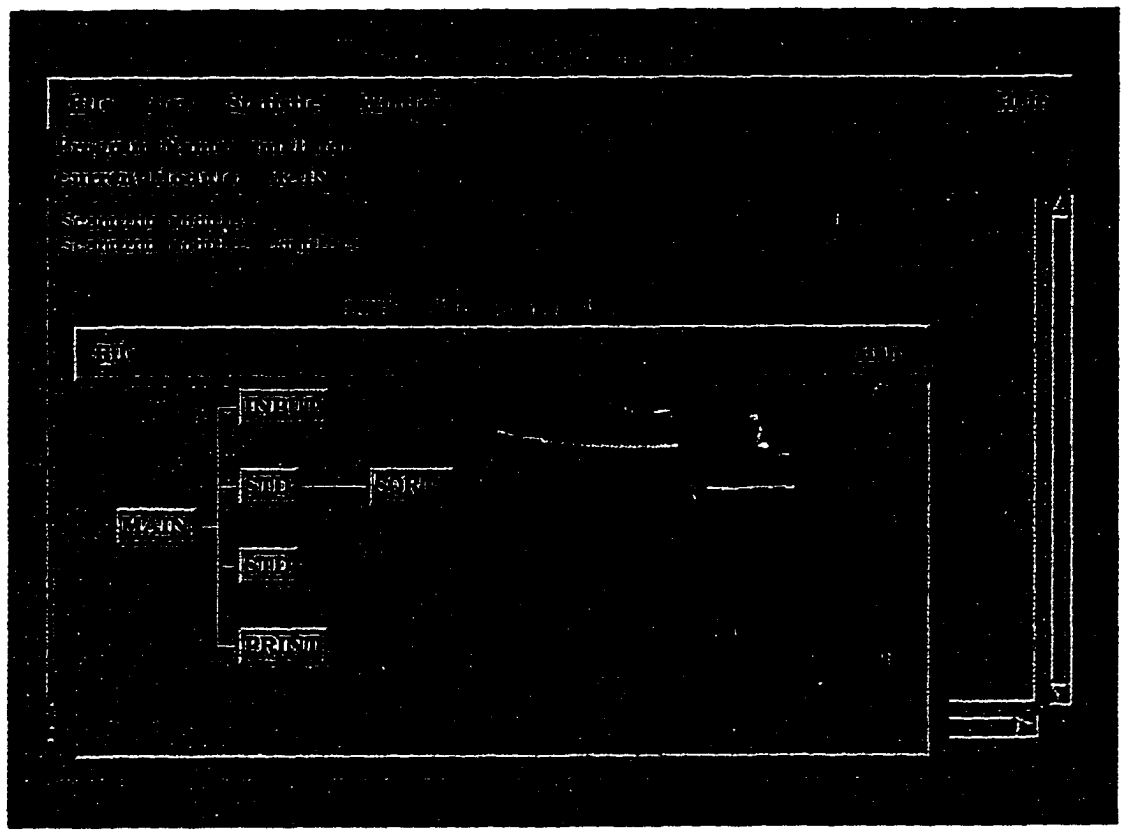


Figure 5.2 Call graph of the sample program

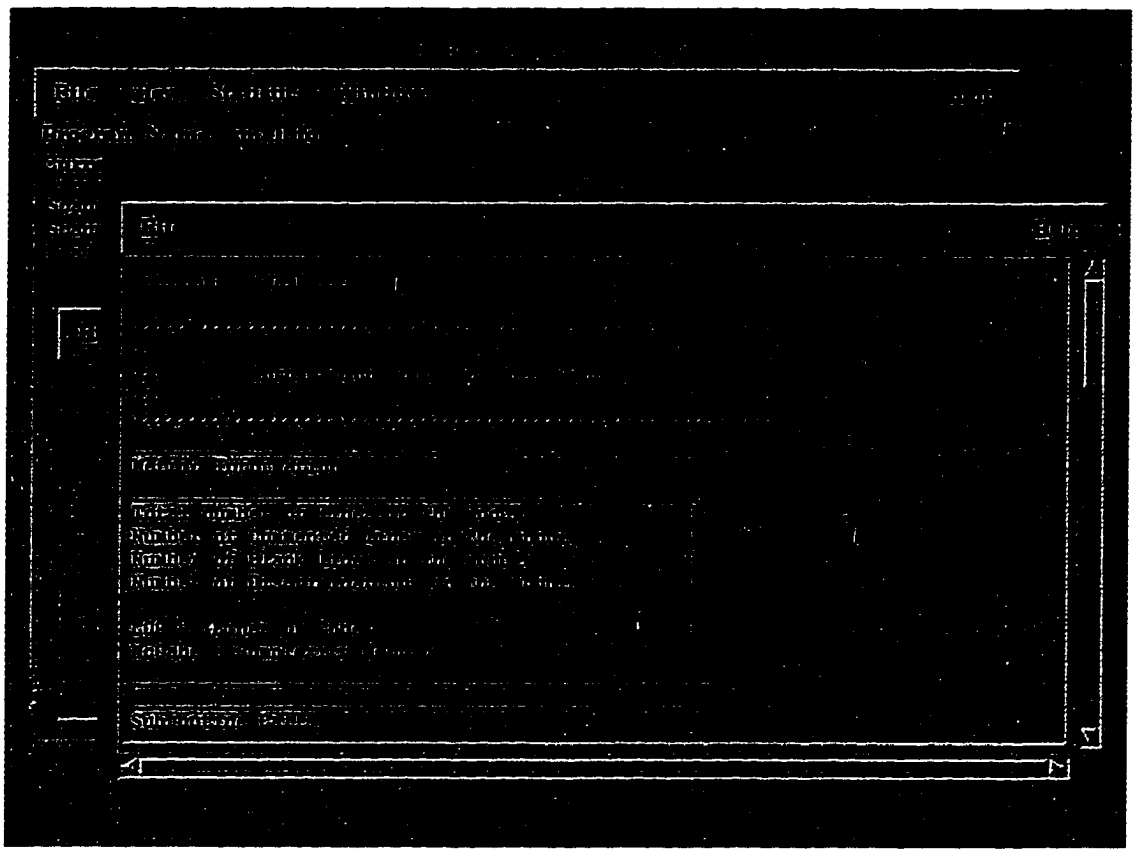


Figure 5.3 Information about MAIN program of sample program

Table 5.2 Metric Information of MAIN

Total Number of Lines	Total Number of Commented Lines	Total Number of Blank Lines	Total Number of Discriminations	McCabe's Cyclomatic Complexity
15	7	0	0	1

Table 5.3 Summary of subroutine calls of MAIN

Name of the subroutine in the CALL	Actual parameters in the CALL	Calling Sequence Number	Type of CALL
INPUT	M EXPA N EXPB	1	Simple
PRINT	EXPA M STDA EXPB N STDB	2	Simple

Table 5.4 Summary of function calls of MAIN

Name of the subroutine in the CALL	Actual parameters in the CALL	Calling Sequence Number	Type of CALL
STD	EXPA M	1	Simple
STD	EXPB N	2	Simple

Local and non-local variable descriptions are listed in Table 5.5. The *type* column of Table 5.5 details the type of variable. For those variables that are explicitly declared, the Information Extractor associates the explicit type declaration to a variable. However, for those variables that are *not* explicitly declared, the Information Extractor infers the type using the FORTRAN implicit variable naming convention. The state change information for MAIN is shown in Table 5.6. The column *loop scope* is relevant only when the state change for a particular variable under consideration occurs within a loop. The type column of Table 5.6 describes the type of state change. Additional information about MAIN is listed in Table 5.7. Typical additional information about a Main Program component includes labeled and unlabeled COMMON variable description.

The design information for the rest of the components of the original program is listed in Tables 5.8 - 5.25. Information about subroutine INPUT is listed in Tables 5.8 - 5.13. Information about subroutine PRINT is listed in Tables 5.14 - 5.19. Information about function STD is listed in Tables 5.20 - 5.25. The word "<None>" appears in the first column of a table for which information is absent. For example, subroutines INPUT and PRINT do not call any other subroutines or functions. Therefore, the word "<None>" appears in the first column of Tables 5.9, 5.10, 5.15, and 5.16. For all subroutines and user-defined functions, tables which list additional information about a particular component include a formal variable description in addition to a COMMON variable description.

Table 5.5 Summary of variable description of MAIN

Variable Name	Type	Is Declared	Is Array	Dimensions	Scope
M	INT	yes	no		Local
N	INT	yes	no		Local
EXPA	REAL	yes	yes	14	Local
EXPB	REAL	yes	yes	14	Local
STDA	REAL	yes	no		Local
STDB	REAL	yes	no		Local
STD	REAL	no	no		Function Return

Table 5.6 Summary of state changes of MAIN

Line	State Change Variable Name	Rhs Variable(s)	Type	Loop Scope	Semantic Nature
11	STDA	STD, EXPA, M	Simple		Assignment statement
13	STDB	STD, EXPB, N	Simple		Assignment statement

Table 5.7 Additional information about MAIN

NO Labeled Common Variables. NO Unlabeled Common Variables.

Table 5.8 Metric information of subroutine INPUT

Total Number of Lines	Total Number of Commented Lines	Total Number of Blank Lines	Total Number of Discriminations	McCabe's Cyclomatic Complexity
15	1	0	2	3

Table 5.9 Summary of subroutine calls of subroutine INPUT

Name of the subroutine in the CALL	Actual parameters in the CALL	Calling Sequence Number	Type of CALL
<None>			

Table 5.10 Summary of function calls of subroutine INPUT

Name of the subroutine in the CALL	Actual parameters in the CALL	Calling Sequence Number	Type of CALL
<None>			

Table 5.11 Summary of variable description of subroutine INPUT

Variable Name	Type	Is Declared	Is Array	Dimensions	Scope
M	INT	yes	no		Formal
N	INT	yes	no		Formal
EXPA	REAL	yes	yes	14	Formal
EXPB	REAL	yes	yes	14	Formal
I	INT	no	no		Local
J	INT	yes	no		Local

Table 5.12 Summary of state changes of subroutine INPUT

Line	State Change Variable Name	Rhs Variable(s)	Type	Loop Scope	Semantic Nature
5	M		Simple		READ statement
6	I	1, 14	Iterative	DO[0]	Loop Variable Assignment
7	EXPA	I	Iterative	DO[0]	READ Statement
10	N		Simple		READ statement
11	J	1, 14	Iterative	DO[1]	Loop Variable Assignment
12	EXPB	J	Iterative	DO[1]	READ Statement

Table 5.13 Additional information of subroutine INPUT

Formal Parameters: M, EXPA, N, EXPB
NO Labeled Common Variables.
NO Unlabeled Common Variables.

Table 5.14 Metric information of subroutine PRINT

Total Number of Lines	Total Number of Commented Lines	Total Number of Blank Lines	Total Number of Discriminations	McCabe's Cyclomatic Complexity
12	1	0	0	1

Table 5.15 Summary of subroutine calls of subroutine PRINT

Name of the subroutine in the CALL	Actual parameters in the CALL	Calling Sequence Number	Type of CALL
<None>			

Table 5.16 Summary of function calls of subroutine PRINT

Name of the subroutine in the CALL	Actual parameters in the CALL	Calling Sequence Number	Type of CALL
<None>			

Table 5.17 Summary of variable description of subroutine PRINT

Variable Name	Type	Is Declared	Is Array	Dimensions	Scope
M	INT	yes	no		Formal
N	INT	yes	no		Formal
EXPA	REAL	yes	yes	14	Formal
EXPB	REAL	yes	yes	14	Formal
I	INT	no	no		Local
J	INT	yes	no		Local

Table 5.18 Summary of state changes of subroutine PRINT

Line	State Change Variable Name	Rhs Variable(s)	Type	Loop Scope	Semantic Nature
<None>					

Table 5.19 Additional information of subroutine PRINT

Formal Parameters: EXPA, M, STDA, EXPB, N, STDA
NO Labeled Common Variables.
NO Unlabeled Common Variables.

Table 5.20 Metric information of function STD

Total Number of Lines	Total Number of Commented Lines	Total Number of Blank Lines	Total Number of Discriminations	McCabe's Cyclomatic Complexity
16	1	0	2	3

Table 5.21 Summary of subroutine calls of function STD

Name of the subroutine in the CALL	Actual parameters in the CALL	Calling Sequence Number	Type of CALL
<None>			

Table 5.22 Summary of function calls of function STD

Name of the subroutine in the CALL	Actual parameters in the CALL	Calling Sequence Number	Type of CALL
SQRT*	SUM/(X-1)	1	Simple

Table 5.23 Summary of variable description of function STD

Variable Name	Type	Is Declared	Is Array	Dimensions	Scope
X	INT	yes	no		Formal
MEAN	REAL	yes	no		Local
EXPX	REAL	yes	yes	14	Formal
IND	REAL	yes	yes	14	Local
TOT	REAL	no	no		Local
SUM	REAL	no	no		Local
I	INT	no			Local
J	INT	no			Local
STD	REAL	Implicit			Function Return

Table 5.24 Summary of state changes of function STD

Line	State Change Variable Name	Rhs Variable(s)	Type	Loop Scope	Semantic Nature
5	TOT	0	Simple		Assignment Statement
6	SUM	0	Simple		Assignment Statement
7	I	1, X	Iterative	DO[0]	Loop Variable Assignment
8	TOT	TOT, EXPX, I	Iterative	DO[0]	Assignment Statement
10	MEAN	TOT, X	Simple		Assignment Statement
11	J	1, X	Iterative	DO[1]	Loop Variable Assignment
12	IND	J, MEAN, EXPX	Iterative	DO[1]	Assignment Statement
13	SUM	SUM, IND, J, 2	Iterative	DO[1]	Assignment Statement
15	STD	SUM, X, 1	Simple		Assignment Statement

Table 5.25 Additional information of function STD

Formal Parameters: EXPX, X
NO Labeled Common Variables.
NO Unlabeled Common Variables.

5.1.2 Dependence Analysis

The sequential design representations produced by the Information Extractor were presented in Section 5.1.2. Using these design representations, the Dependence Analyzer computes the data and control dependences of each component. Subsequently, the program dependence graph (PDG) of each component is constructed. As mentioned in Chapter 4, the dependences are represented both textually and graphically. The dependences are computed for each component in the design and later synthesized to form the PDG. The call graph is used as the basis for the construction of the PDG that proceeds in a bottom-up manner.

The textual description of the dependences of subroutine INPUT is listed in Table 5.26. The format described in Table 4.10 of Chapter 4 is used to represent information regarding dependences. The first row of Table 5.26 indicates the total number of sites for subroutine INPUT. Subsequent rows of Table 5.26 list the particulars of each site: site number, site id, number of use list variables, number of definition list variables, use list variables, number of flow dependences (if any), flow dependences (if any), number of control dependences (if any), control dependences (if any), definition list variables, number of loop dependences (if any), and loop dependences (if any). It has to be noted that the first site (Site 0) consists of only definition list variables and the last site (Site 9 in this case) consists of only use list variables. The symbols "γ", "τ", and "ε" in the notation described in [Jack 94] are represented as "*Gamma*", "*Tau*", and "*E*" respectively. The asterisks are used to differentiate these special variables from possible use as variable names in a program.

Table 5.26 Textual description of dependences of subroutine INPUT

```
Total Number of Sites: 10

Site Number: 0
Site Id:   1
        Number of Use-List Variables: 0
        Number of Def-List Variables: 4

                Def-List Variables:
                        M
                        N
                        EXPA
                        EXPB

Site Number: 1
Site Id:   5
        Number of Use-List Variables: 2
        Number of Def-List Variables: 1
                Use-List Variables:
                        *File*
                        *E*
                Def-List Variables:
                        M

Site Number: 2
Site Id:   6
        Number of Use-List Variables: 2
        Number of Def-List Variables: 1
                Use-List Variables:
                        *Gamma*
                        *E*
                Def-List Variables:
                        I

Site Number: 3
Site Id:   6
        Number of Use-List Variables: 2
        Number of Def-List Variables: 1
                Use-List Variables:
                        DO[0]
                           Number of flow dependences: 1
                               Flow Dependences:
                               <2, I>
                        *E*
                Def-List Variables:
                        *Tau*
```

(table con'd.)

```
Site Number: 4
Site Id:   7
        Number of Use-List Variables: 2
        Number of Def-List Variables: 2
                Use-List Variables:
                        I
                           Number of flow dependences: 1
                                Flow Dependences:
                                <2, I>
                        *E*
                           Number of control dependences: 1
                                Control Dependences:
                                 <3, *Tau*>
                Def-List Variables:
                        EXPA
                        I
                           Number of loop dependences: 2
                              Loop Dependences:
                              <4, I>, <3, I>
```

```
Site Number: 5
Site Id:   10
        Number of Use-List Variables: 2
        Number of Def-List Variables: 1
                Use-List Variables:
                        *File*
                        *E*
                Def-List Variables:
                        N
```

```
Site Number: 6
Site Id:   11
        Number of Use-List Variables: 2
        Number of Def-List Variables: 1
                Use-List Variables:
                        *Gamma*
                        *E*
                Def-List Variables:
                        J
```

```
Site Number: 7
Site Id:   11
        Number of Use-List Variables: 2
        Number of Def-List Variables: 1
                Use-List Variables:
                        DO[1]
                           Number of flow dependences: 1
                                Flow Dependences:
                                <6, J>
                        *E*
                Def-List Variables:
                        *Tau*
```

(table con'd.)

```
Site Number: 8
Site Id:   12
        Number of Use-List Variables: 2
        Number of Def-List Variables: 2
                Use-List Variables:
                        J
                            Number of flow dependences: 1
                                Flow Dependences:
                                <6, J>
                        *E*
                            Number of control dependences: 1
                                Control Dependences:
                                 <7, *Tau*>
                Def-List Variables:
                        EXPB
                        J
                            Number of loop dependences: 2
                               Loop Dependences:
                               <8, J>, <7, J>
```

```
Site Number: 9
Site Id:   12
        Number of Use-List Variables: 6
        Number of Def-List Variables: 0
                Use-List Variables:
                        M
                                Number of flow dependences: 1
                                Flow Dependences:
                               <1, M>
                        N
                                Number of flow dependences: 1
                                Flow Dependences:
                               <5, N>
                        EXPA
                                Number of flow dependences: 1
                                Flow Dependences:
                               <4, EXPA>
                        EXPB
                                Number of flow dependences: 1
                                Flow Dependences:
                               <8, EXPB>
                        I
                                Number of flow dependences: 1
                                Flow Dependences:
                               <4, I>
                        J
                                Number of flow dependences: 1
                                Flow Dependences:
                               <8, J>
```

In addition to the textual description of dependences described in Table 5.26, a *xfig* based graphical representation of the dependences is automatically generated by the Dependence Analyzer for each component. The *xfig* depiction of the dependence graph of subroutine INPUT is shown in Figure 5.4. In order to show all the sites in the dependence graph, the original picture was zoomed in at the ratio of 10:1 . A blowup of sites 3 and 4 is shown in Figure 5.5. The dependences are represented as edges between the sites. The actual *xfig* representation uses different color encoding for easy identification of the dependences. Some of the color encoding is apparent in Figure 5.5. In particular, edges which represent loop dependences are shown thick when compared to flow or control dependences edges which are shown thin.

The textual description of dependences of subroutine PRINT and the user-defined function STD are listed in Tables 5.27 and 5.28 respectively. As defined in Chapter 4, an abstract site is built for all subroutines and user-defined functions. A component's abstract site's use list variable(s) are the non-local variables of the module which include formal parameters and COMMON variables. The definition list variables of the abstract site are the same non-local variables to which a state change occurs within the module. The textual description of the abstract site of subroutine INPUT is given in Table 5.29. The abstract site description of subroutine PRINT and user-defined function STD are listed in Tables 5.30 and 5.31 respectively.

The textual description of the dependences of the MAIN program is shown in Table 5.32. Sites which represent calls to subroutines and user defined-functions from MAIN are replaced by the abstract sites.

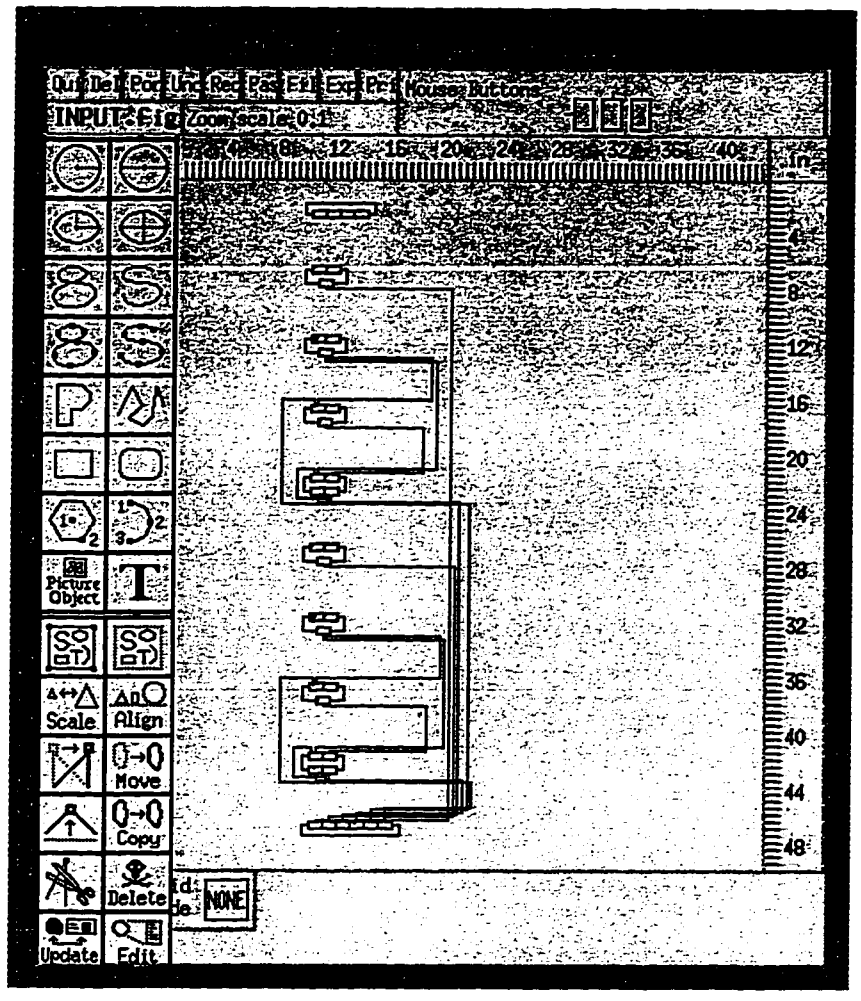


Figure 5.4 Dependence graph of subroutine INPUT

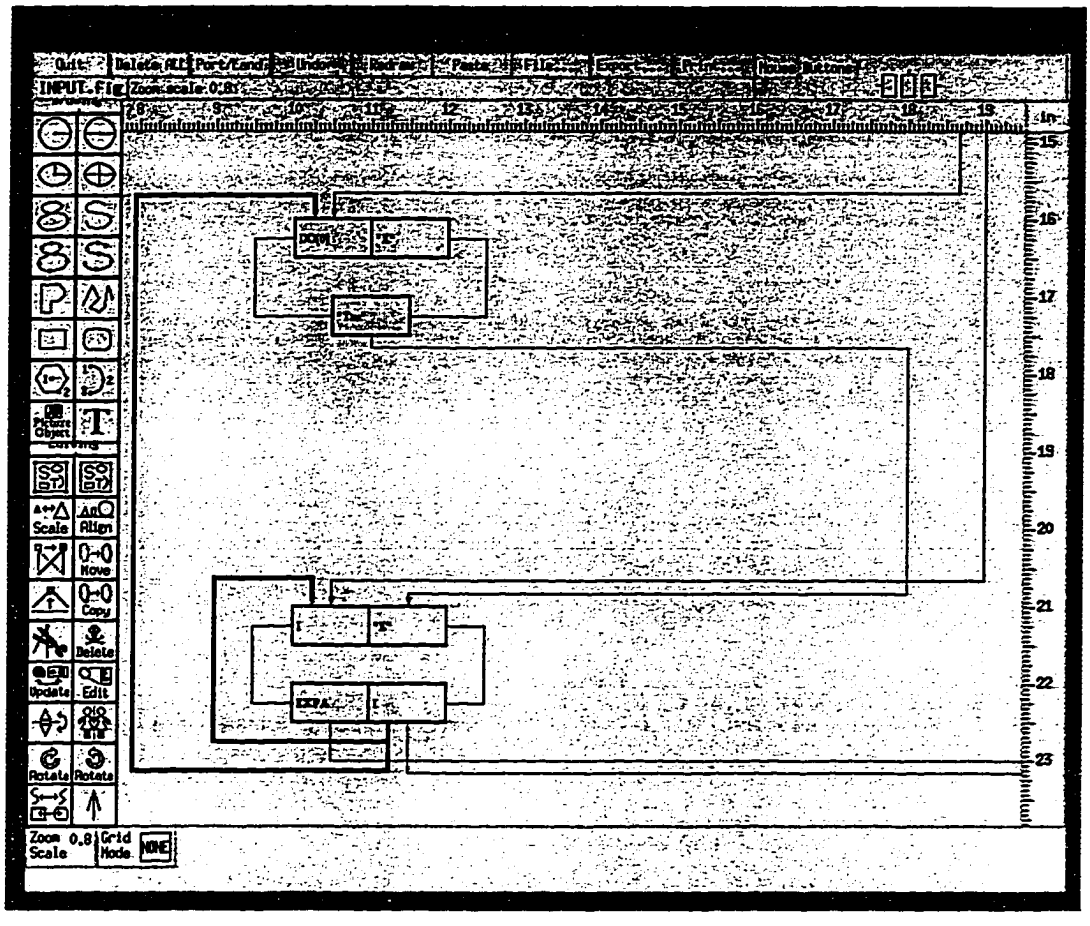


Figure 5.5 Blowup of two sites of the dependence graph of subroutine INPUT

Table 5.27 Textual description of dependences of subroutine PRINT

```
Total Number of Sites: 2

Site Number: 0
Site Id:   1
        Number of Use-List Variables: 0
        Number of Def-List Variables: 4

                Def-List Variables:
                        STDA
                        STDB
                        EXPA
                        EXPB

Site Number: 1
Site Id:   11
        Number of Use-List Variables: 4
        Number of Def-List Variables: 0
                Use-List Variables:
                        STDA
                                Number of flow dependences: 1
                                Flow Dependences:
                                <0, STDA>
                        STDB
                                Number of flow dependences: 1
                                Flow Dependences:
                                <0, STDB>
                        EXPA
                                Number of flow dependences: 1
                                Flow Dependences:
                                <0, EXPA>
                        EXPB
                                Number of flow dependences: 1
                                Flow Dependences:
                                <0, EXPB>
```

Table 5.28 Textual description of dependences of function STD

Total Number of Sites: 13

Site Number: 0
Site Id: 1
Number of Use-List Variables: 0
Number of Def-List Variables: 5

Def-List Variables:
X
MEAN
EXPX
IND
TOT

Site Number: 1
Site Id: 5
Number of Use-List Variables: 2
Number of Def-List Variables: 1
Use-List Variables:
Gamma
E
Def-List Variables:
TOT

Site Number: 2
Site Id: 6
Number of Use-List Variables: 2
Number of Def-List Variables: 1
Use-List Variables:
Gamma
E
Def-List Variables:
SUM

Site Number: 3
Site Id: 7
Number of Use-List Variables: 2
Number of Def-List Variables: 1
Use-List Variables:
Gamma
E
Def-List Variables:
I

(table con'd.)

```
Site Number: 4
Site Id:    7
        Number of Use-List Variables: 2
        Number of Def-List Variables: 1
                Use-List Variables:
                        DO[0]
                           Number of flow dependences: 2
                               Flow Dependences:
                               <3, I>, <0, X>
                        *E*
                Def-List Variables:
                        *Tau*
```

```
Site Number: 5
Site Id:    8
        Number of Use-List Variables: 4
        Number of Def-List Variables: 2
                Use-List Variables:
                        TOT
                           Number of flow dependences: 1
                               Flow Dependences:
                               <1, TOT>
                         EXPX
                           Number of flow dependences: 1
                               Flow Dependences:
                               <0, EXPX>
                         I
                           Number of flow dependences: 1
                               Flow Dependences:
                               <3, I>
                        *E*
                           Number of control dependences: 1
                               Control Dependences:
                                <4, *Tau*>
                Def-List Variables:
                        TOT
                        I
                          Number of loop dependences: 2
                             Loop Dependences:
                             <5, I>, <4, I>
```

(table con'd.)

```
Site Number: 6
Site Id:   10
        Number of Use-List Variables: 3
        Number of Def-List Variables: 1
                Use-List Variables:
                        TOT
                           Number of flow dependences: 1
                                Flow Dependences:
                                <5, TOT>
                        X
                           Number of flow dependences: 1
                                Flow Dependences:
                                <0, X>
                        *E*
                Def-List Variables:
                        MEAN
```

```
Site Number: 7
Site Id:   11
        Number of Use-List Variables: 2
        Number of Def-List Variables: 1
                Use-List Variables:
                        *Gamma*
                        *E*
                Def-List Variables:
                        J
```

```
Site Number: 8
Site Id:   11
        Number of Use-List Variables: 2
        Number of Def-List Variables: 1
                Use-List Variables:
                        DO[1]
                           Number of flow dependences: 2
                                Flow Dependences:
                                <7, J>, <0, X>
                        *E*
                Def-List Variables:
                        *Tau*
```

(table con'd.)

```
Site Number: 9
Site Id:   12
        Number of Use-List Variables: 4
        Number of Def-List Variables: 1
                Use-List Variables:
                        J
                           Number of flow dependences: 1
                                Flow Dependences:
                                <7, J>
                        MEAN
                           Number of flow dependences: 1
                                Flow Dependences:
                                <6, MEAN>
                        EXPX
                           Number of flow dependences: 1
                                Flow Dependences:
                                <0, EXPX>
                        *E*
                           Number of control dependences: 1
                                Control Dependences:
                                 <8, *Tau*>
                Def-List Variables:
                        IND
```

```
Site Number: 10
Site Id:   13
        Number of Use-List Variables: 5
        Number of Def-List Variables: 2
                Use-List Variables:
                        SUM
                            Number of flow dependences: 1
                                Flow Dependences:
                                <2, SUM>
                         IND
                            Number of flow dependences: 1
                                Flow Dependences:
                                <9, IND>
                         J
                            Number of flow dependences: 1
                                Flow Dependences:
                                <7, J>
                         *Gamma*
                        *E*
                            Number of control dependences: 1
                                Control Dependences:
                                 <8, *Tau*>
                Def-List Variables:
                        SUM
                        J
                           Number of loop dependences: 3
                              Loop Dependences:
                              <8, J>, <9, J>, <10, J>
```

(table con'd.)

```
Site Number: 11
Site Id:   15
        Number of Use-List Variables: 4
        Number of Def-List Variables: 1
                Use-List Variables:
                        SUM
                            Number of flow dependences: 1
                                Flow Dependences:
                                <10, SUM>
                         X
                            Number of flow dependences: 1
                                Flow Dependences:
                                <0, X>
                         *Gamma*
                        *E*
                Def-List Variables:
                        STD
```

```
Site Number: 12
Site Id:   15
        Number of Use-List Variables: 7
        Number of Def-List Variables: 0
                Use-List Variables:
                        X
                                Number of flow dependences: 1
                                Flow Dependences:
                                <0, X>
                        MEAN
                                Number of flow dependences: 1
                                Flow Dependences:
                                <6, MEAN>
                        EXPX
                                Number of flow dependences: 1
                                Flow Dependences:
                                <0, EXPX>
                        IND
                                Number of flow dependences: 1
                                Flow Dependences:
                                <9, IND>
                        TOT
                                Number of flow dependences: 1
                                Flow Dependences:
                                <5, TOT>
                        SUM
                                Number of flow dependences: 1
                                Flow Dependences:
                                <10, SUM>
                        STD
                                Number of flow dependences: 1
                                Flow Dependences:
                                <11, STD>
```

Table 5.29 Abstract site of subroutine INPUT

```
Site Number: Abstract
Site Id:    nil
        Number of Use-List Variables: 5
        Number of Def-List Variables: 4
                Use-List Variables:
                        M
                        N
                        EXPA
                        EXPB
                        *E*
                Def-List Variables:
                        M
                        N
                        EXPA
                        EXPB
```

Table 5.30 Abstract site of subroutine PRINT

```
Site Number: Abstract
Site Id:    nil
        Number of Use-List Variables: 5
        Number of Def-List Variables: 0
                Use-List Variables:
                        STDA
                        STDB
                        EXPA
                        EXPB
                        *E*
```

Table 5.31 Abstract site of function STD

```
Site Number: Abstract
Site Id:    nil
        Number of Use-List Variables: 2
        Number of Def-List Variables: 0
                Use-List Variables:
                        X
                        EXPX
```

Table 5.32 Textual description of dependences of MAIN

```
Total Number of Sites: 6

Site Number: 0
Site Id:   8
        Number of Use-List Variables: 0
        Number of Def-List Variables: 6

                Def-List Variables:
                        M
                        N
                        EXPA
                        EXPB
                        STDA
                        STDB

Site Number: 1
Site Id:   9
        Number of Use-List Variables: 5
        Number of Def-List Variables: 4
                Use-List Variables:
                        M
                        N
                        EXPA
                        EXPB
                        *E*
                Def-List Variables:
                        M
                        N
                        EXPA
                        EXPB

Site Number: 2
Site Id:   11
        Number of Use-List Variables: 4
        Number of Def-List Variables: 1
                Use-List Variables:
                        STD
                        EXPA
                           Number of flow dependences: 1
                               Flow Dependences:
                              <1, EXPA>
                         M
                           Number of flow dependences: 1
                               Flow Dependences:
                              <1, M>
                        *E*
                Def-List Variables:
                        STDA
```

(table con'd.)

```
Site Number: 3
Site Id:    13
        Number of Use-List Variables: 4
        Number of Def-List Variables: 1
                Use-List Variables:
                        STD
                        EXPB
                           Number of flow dependences: 1
                              Flow Dependences:
                              <1, EXPB>
                         N
                           Number of flow dependences: 1
                              Flow Dependences:
                              <1, N>
                       *E*
                Def-List Variables:
                        STDB
```

```
Site Number: 4
Site Id:    14
        Number of Use-List Variables: 5
        Number of Def-List Variables: 0
                Use-List Variables:
                        STDA
                        STDB
                        EXPA
                        EXPB
                        *E*
```

(table con'd.)

```
Site Number: 5
Site Id:   14
        Number of Use-List Variables: 6
        Number of Def-List Variables: 0
                Use-List Variables:
                        M
                            Number of flow dependences: 1
                                Flow Dependences:
                                <1, M>
                        N
                            Number of flow dependences: 1
                                Flow Dependences:
                                <1, N>
                        EXPA
                            Number of flow dependences: 1
                                Flow Dependences:
                                <1, EXPA>
                        EXPB
                            Number of flow dependences: 1
                                Flow Dependences:
                                <1, EXPB>
                        STDA
                            Number of flow dependences: 1
                                Flow Dependences:
                                <2, STDA>
                         STDB
                            Number of flow dependences: 1
                                Flow Dependences:
                                <3, STDB>
```

5.1.3 Parallel Design Recommendations

The Information Extractor (IE) and Dependence Analyzer (DANA) automatically produce program facts and dependence facts for each component in the design. The program facts and dependence facts, which are based on CLIPS syntax, make use of the class definitions outlined in Chapter 4 to create object instances.

The program facts of subroutine INPUT are listed in Table 5.33. The program facts are defined using the *deffunction* construct of CLIPS. The function "read-objects-INPUT" is an user-defined function which consists of all the program facts for subroutine INPUT. Documentation of the facts listed in Table 5.33 is done using CLIPS commenting style. A line which starts with a semicolon is treated as a comment by CLIPS. The program facts include information about subroutine calls, function calls, COMMON variables, complete variable description, state changes, DO-loops, IF-conditions. A rule is defined at the end which invokes the function "read-objects-INPUT". The program facts similar to that of subroutine INPUT are generated for all other components.

The dependence facts of the main program, MAIN are listed in Table 5.34. The dependence facts are defined using the deffunction construct of CLIPS. The function "read-Dependences-MAIN" is an user-defined function based on the *deffunction* construct of CLIPS. The function "read-Dependences-MAIN" consists of all the dependence facts for MAIN. The dependence facts include site information, flow data dependences, loop data dependences and control dependences. A rule is defined at the end which invokes the function "read-Dependences-MAIN" function. The program facts similar to that of MAIN are generated for all other components.

Table 5.33 Program facts of subroutine INPUT

```
(deffunction read-objects-INPUT ()
 ; Program Facts for Subroutine INPUT
 ; Subroutine Calls Information

 ; Function Calls Information

 ; Labeled Commons Information

 (bind ?x (make-instance (gensym) of Labeled-Commons))
 (send ?x put-comp-name INPUT)

 ; Un Labeled Commons Information

 (bind ?x (make-instance (gensym) of Un-Labeled-Commons))
 (send ?x put-comp-name INPUT)

 ; Variable(s) Description

 (bind ?x (make-instance (gensym) of Locals
 (Var-Name M)
 (Type INTEGER)
 (Declared y)
 (Array n)
 (Non-Local y)))
 (send ?x put-comp-name INPUT)

 (bind ?x (make-instance (gensym) of Locals
 (Var-Name N)
 (Type INTEGER)
 (Declared y)
 (Array n)
 (Non-Local y)))
 (send ?x put-comp-name INPUT)

 (bind ?x (make-instance (gensym) of Locals
 (Var-Name EXPA)
 (Type REAL)
 (Declared y)
 (Array y)
 (Dimensions 14)
 (Non-Local y)))
 (send ?x put-comp-name INPUT)

 (bind ?x (make-instance (gensym) of Locals
 (Var-Name EXPB)
 (Type REAL)
 (Declared y)
 (Array y)
 (Dimensions 14)
 (Non-Local y)))
(send ?x put-comp-name INPUT)
```

(table con'd.)

```
(bind ?x (make-instance (gensym) of Locals
(Var-Name I)
(Type INTEGER)
(Declared n)
(Array n)
(Non-Local n)))
(send ?x put-comp-name INPUT)

(bind ?x (make-instance (gensym) of Locals
(Var-Name J)
(Type INTEGER)
(Declared n)
(Array n)
(Non-Local n)))
(send ?x put-comp-name INPUT)

; State Change Information

(bind ?x (make-instance (gensym) of State-Changes
(Line-Num 5)
(Var-Name M)
(Category *)))
(send ?x put-comp-name INPUT)

(bind ?x (make-instance (gensym) of State-Changes
(Line-Num 6)
(Var-Name I)
(Rhs-Variables 1 14)
(Category "->0-*D")))
(send ?x put-comp-name INPUT)

(bind ?x (make-instance (gensym) of State-Changes
(Line-Num 7)
(Var-Name EXPA)
(Rhs-Variables I)
(Category "->0-*D")))
(send ?x put-comp-name INPUT)

(bind ?x (make-instance (gensym) of State-Changes
(Line-Num 10)
(Var-Name N)
(Category *)))
(send ?x put-comp-name INPUT)

(bind ?x (make-instance (gensym) of State-Changes
(Line-Num 11)
(Var-Name J)
(Rhs-Variables 1 14)
(Category "->1-*D")))
(send ?x put-comp-name INPUT)
```

(table con'd.)

```
(bind ?x (make-instance (gensym) of State-Changes
 (Line-Num 12)
 (Var-Name EXPB)
 (Rhs-Variables J)
 (Category "->1-*D")))
 (send ?x put-comp-name INPUT)

 ; DO Loops Information

 (bind ?x (make-instance (gensym) of DO-Loops
 (DO-Start-Line 6)
 (DO-Last-Line 8)
 (DO-Label 15)
 (DO-Counting-Variable I)
 (DO-Initial-Value 1)
 (DO-Limit 14)
 (DO-Step nil)
 (DO-Nesting-Level 0)
 (DO-Prev-Type X)
 (DO-Prev-Id 0)))
 (send ?x put-comp-name INPUT)

 (bind ?x (make-instance (gensym) of DO-Loops
 (DO-Start-Line 11)
 (DO-Last-Line 13)
 (DO-Label 30)
 (DO-Counting-Variable J)
 (DO-Initial-Value 1)
 (DO-Limit 14)
 (DO-Step nil)
 (DO-Nesting-Level 0)
 (DO-Prev-Type X)
 (DO-Prev-Id 0)))
 (send ?x put-comp-name INPUT)

 ; IF Conditionals Information

 ; Subroutine Information
 (bind ?x (make-instance (gensym) of Subroutine
 (Name INPUT)
 (Sub-Formals M EXPA N EXPB)
 (num-formals 4)))
 (send ?x put-num-locals 6)
 (send ?x put-num-states 6)
 (send ?x put-num-do-loops 2)
 (send ?x put-comp-name INPUT))
(defrule Sub-INPUT
=>
(read-objects-INPUT))
```

Table 5.34 Dependence facts of MAIN

```
; Dependence Knowledge Representation for Main Program MAIN
(deffunction read-Dependences-MAIN ()

; Main Program Information

 (bind ?x (make-instance (gensym) of Main-Program
          (Name MAIN)))
          (send ?x put-num-sites 6)
          (send ?x put-comp-name MAIN)

;------ Site Number 0 ------

 (bind ?x (make-instance (gensym) of Site
          (Site-Number 0)
          (Site-Id 1)
          (Number-of-Uls 0)
          (Number-of-Dls 6)
          (Number-of-Cds 0)))
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Def-Lists
          (Def-List-Name M)))
 (send ?x put-Site-Number 0)
 (send ?x put-Site-Id 1)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Def-Lists
          (Def-List-Name N)))
 (send ?x put-Site-Number 0)
 (send ?x put-Site-Id 1)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Def-Lists
          (Def-List-Name EXPA)))
 (send ?x put-Site-Number 0)
 (send ?x put-Site-Id 1)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Def-Lists
          (Def-List-Name EXPB)))
 (send ?x put-Site-Number 0)
 (send ?x put-Site-Id 1)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Def-Lists
          (Def-List-Name STDA)))
 (send ?x put-Site-Number 0)
 (send ?x put-Site-Id 1)
 (send ?x put-comp-name MAIN)
```

(table con'd.)

```
(bind ?x (make-instance (gensym) of Def-Lists
        (Def-List-Name STDB)))
 (send ?x put-Site-Number 0)
 (send ?x put-Site-Id 1)
 (send ?x put-comp-name MAIN)

; ----- Site Number 1 -----

 (bind ?x (make-instance (gensym) of Site
        (Site-Number 1)
        (Site-Id 9)
        (Number-of-Uls 5)
        (Number-of-Dls 4)
        (Number-of-Cds 0)))
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name M)))
 (send ?x put-Site-Number 1)
 (send ?x put-Site-Id 9)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name N)))
 (send ?x put-Site-Number 1)
 (send ?x put-Site-Id 9)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name EXPA)))
 (send ?x put-Site-Number 1)
 (send ?x put-Site-Id 9)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name EXPB)))
 (send ?x put-Site-Number 1)
 (send ?x put-Site-Id 9)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name *E*)))
 (send ?x put-Site-Number 1)
 (send ?x put-Site-Id 9)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Def-Lists
        (Def-List-Name M)))
 (send ?x put-Site-Number 1)
 (send ?x put-Site-Id 9)
 (send ?x put-comp-name MAIN)
```

(table con'd.)

```
(bind ?x (make-instance (gensym) of Def-Lists
          (Def-List-Name N)))
 (send ?x put-Site-Number 1)
 (send ?x put-Site-Id 9)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Def-Lists
          (Def-List-Name EXPA)))
 (send ?x put-Site-Number 1)
 (send ?x put-Site-Id 9)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Def-Lists
          (Def-List-Name EXPB)))
 (send ?x put-Site-Number 1)
 (send ?x put-Site-Id 9)
 (send ?x put-comp-name MAIN)

; ----- Site Number 2 -----

 (bind ?x (make-instance (gensym) of Site
          (Site-Number 2)
          (Site-Id 11)
          (Number-of-Uls 4)
          (Number-of-Dls 1)
          (Number-of-Cds 0)))
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
          (Use-List-Name STD)))
 (send ?x put-Site-Number 2)
 (send ?x put-Site-Id 11)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
          (Use-List-Name EXPA)
          (Number-of-Dus 1)))
 (send ?x put-Site-Number 2)
 (send ?x put-Site-Id 11)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Du-Edges
          (Du-Edges-Name EXPA)
          (Du-Dest-Site-Number 1)))
 (send ?x put-Use-List-Name EXPA)
 (send ?x put-Site-Number 2)
 (send ?x put-Site-Id 11)
 (send ?x put-comp-name MAIN)
```

(table con'd.)

```
(bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name M)
        (Number-of-Dus 1)))
 (send ?x put-Site-Number 2)
 (send ?x put-Site-Id 11)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Du-Edges
        (Du-Edges-Name M)
        (Du-Dest-Site-Number 1)))
 (send ?x put-Use-List-Name M)
 (send ?x put-Site-Number 2)
 (send ?x put-Site-Id 11)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name *E*)))
 (send ?x put-Site-Number 2)
 (send ?x put-Site-Id 11)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Def-Lists
        (Def-List-Name STDA)))
 (send ?x put-Site-Number 2)
 (send ?x put-Site-Id 11)
 (send ?x put-comp-name MAIN)

; ------ Site Number 3 ------

 (bind ?x (make-instance (gensym) of Site
        (Site-Number 3)
        (Site-Id 13)
        (Number-of-Uls 4)
        (Number-of-Dls 1)
        (Number-of-Cds 0)))
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name STD)))
 (send ?x put-Site-Number 3)
 (send ?x put-Site-Id 13)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name EXPB)
        (Number-of-Dus 1)))
 (send ?x put-Site-Number 3)
 (send ?x put-Site-Id 13)
 (send ?x put-comp-name MAIN)
```

(table con'd.)

```
(bind ?x (make-instance (gensym) of Du-Edges
        (Du-Edges-Name EXPB)
        (Du-Dest-Site-Number 1)))
(send ?x put-Use-List-Name EXPB)
(send ?x put-Site-Number 3)
(send ?x put-Site-Id 13)
(send ?x put-comp-name MAIN)

(bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name N)
        (Number-of-Dus 1)))
(send ?x put-Site-Number 3)
(send ?x put-Site-Id 13)
(send ?x put-comp-name MAIN)

(bind ?x (make-instance (gensym) of Du-Edges
        (Du-Edges-Name N)
        (Du-Dest-Site-Number 1)))
(send ?x put-Use-List-Name N)
(send ?x put-Site-Number 3)
(send ?x put-Site-Id 13)
(send ?x put-comp-name MAIN)

(bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name *E*)))
(send ?x put-Site-Number 3)
(send ?x put-Site-Id 13)
(send ?x put-comp-name MAIN)

(bind ?x (make-instance (gensym) of Def-Lists
        (Def-List-Name STDB)))
(send ?x put-Site-Number 3)
(send ?x put-Site-Id 13)
(send ?x put-comp-name MAIN)

; ------ Site Number 4 ------

(bind ?x (make-instance (gensym) of Site
        (Site-Number 4)
        (Site-Id 14)
        (Number-of-Uls 5)
        (Number-of-Dls 0)
        (Number-of-Cds 0)))
(send ?x put-comp-name MAIN)

(bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name STDA)))
(send ?x put-Site-Number 4)
(send ?x put-Site-Id 14)
(send ?x put-comp-name MAIN)
```

(table con'd.)

```
(bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name STDB)))
 (send ?x put-Site-Number 4)
 (send ?x put-Site-Id 14)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name EXPA)))
 (send ?x put-Site-Number 4)
 (send ?x put-Site-Id 14)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name EXPB)))
 (send ?x put-Site-Number 4)
 (send ?x put-Site-Id 14)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name *E*)))
 (send ?x put-Site-Number 4)
 (send ?x put-Site-Id 14)
 (send ?x put-comp-name MAIN)


; ------ Site Number 5 ------

 (bind ?x (make-instance (gensym) of Site
        (Site-Number54)
        (Site-Id 14)
        (Number-of-Uls 6)
        (Number-of-Dls 0)
        (Number-of-Cds 0)))
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name M)
        (Number-of-Dus 1)))
 (send ?x put-Site-Number 5)
 (send ?x put-Site-Id 14)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Du-Edges
        (Du-Edges-Name M)
        (Du-Dest-Site-Number 1)))
 (send ?x put-Use-List-Name M)
 (send ?x put-Site-Number 5)
 (send ?x put-Site-Id 14)
 (send ?x put-comp-name MAIN)
```

(table con'd.)

```
(bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name N)
        (Number-of-Dus 1)))
 (send ?x put-Site-Number 5)
 (send ?x put-Site-Id 14)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Du-Edges
        (Du-Edges-Name N)
        (Du-Dest-Site-Number 1)))
 (send ?x put-Use-List-Name N)
 (send ?x put-Site-Number 5)
 (send ?x put-Site-Id 14)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name EXPA)
        (Number-of-Dus 1)))
 (send ?x put-Site-Number 5)
 (send ?x put-Site-Id 14)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Du-Edges
        (Du-Edges-Name EXPA)
        (Du-Dest-Site-Number 1)))
 (send ?x put-Use-List-Name EXPA)
 (send ?x put-Site-Number 5)
 (send ?x put-Site-Id 14)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name EXPB)
        (Number-of-Dus 1)))
 (send ?x put-Site-Number 5)
 (send ?x put-Site-Id 14)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Du-Edges
        (Du-Edges-Name EXPB)
        (Du-Dest-Site-Number 1)))
 (send ?x put-Use-List-Name EXPB)
 (send ?x put-Site-Number 5)
 (send ?x put-Site-Id 14)
 (send ?x put-comp-name MAIN)

 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name STDA)
        (Number-of-Dus 1)))
 (send ?x put-Site-Number 5)
 (send ?x put-Site-Id 14)
 (send ?x put-comp-name MAIN)
```

(table con'd.)

```
(bind ?x (make-instance (gensym) of Du-Edges
        (Du-Edges-Name STDA)
        (Du-Dest-Site-Number 2)))

 (send ?x put-Use-List-Name STDA)

 (send ?x put-Site-Number 5)

 (send ?x put-Site-Id 14)

 (send ?x put-comp-name MAIN)


 (bind ?x (make-instance (gensym) of Use-Lists
        (Use-List-Name STDB)
        (Number-of-Dus 1)))

 (send ?x put-Site-Number 5)

 (send ?x put-Site-Id 14)

 (send ?x put-comp-name MAIN)


 (bind ?x (make-instance (gensym) of Du-Edges
        (Du-Edges-Name STDB)
        (Du-Dest-Site-Number 3)))

 (send ?x put-Use-List-Name STDB)

 (send ?x put-Site-Number 5)

 (send ?x put-Site-Id 14)

 (send ?x put-comp-name MAIN)
        )


(defrule Dependences-MAIN
=>
(read-Dependences-MAIN))
```

Using the program facts and dependence facts, the CLIPS forward-chaining inference procedure analyzed the dependences and produced the parallel design recommendations for the sample program. A snapshot of the CLIPS based design assistant is shown in Figure 5.6. A snapshot of the parallel design recommendations derived by the CLIPS-based Design Assistant is shown in Figure 5.7. From Figure 5.7 it can be observed that the Design Assistant derived four parallel design recommendations for the sample program. One of the unique features of the Design Assistant is the explanation mechanism that provides the reasons for each parallel design recommendation. Henceforth we denote the parallel design recommendations as PDR-<Num>, where <Num> is a number. For the sample program, <Num> ranges from 1 to 4. From Figure 5.7 it can be observed that PDR-1, PDR-2, and PDR-3 correspond to inter-component parallelization for subroutine INPUT. PDR-4 corresponds to intra-component parallelization between the main program, MAIN and the user-defined function STD.

PDR-1 states that the DO-loop identified by label '10' in subroutine INPUT may be parallelized. The reasons for arriving at PDR-1 are based on scenario 6 which identified that the state change to variable EXPA defined within the loop depends only on the loop index. PDR-2 states that the DO-loop identified by label '20' in subroutine INPUT may be parallelized. The reasons for arriving at PDR-2 are based on scenario 6 which identified that the state change to variable EXPB defined within the loop depends only on the loop index. PDR-3 states that the computation of the DO-loops identified by labels '10' and '20' in subroutine INPUT may be partitioned into

parallel computations. The reasons for arriving at PDR-3 are based on scenario 6 which identified that there is no data flow between the two loops.

PDR-4 states that the function calls to STD from MAIN at relative line numbers 11 and 13 may be parallelized. PDR-4 is based on scenario 1 which identified that the actual parameters EXPA, M in function call to STD in line 11 are different from the actual parameters EXPB, N in the function call to STD in line 13. The formal parameters EXPX, X are not modified in STD. Further, STD does not have COMMON variables.

We introduced a graphical representation termed as PDR to represent parallel design recommendations in Chapter 3. The PDR representation of subroutine INPUT is shown in Figure 5.8. The PDR representation of the MAIN program is shown in Figure 5.9. The textual description of the parallel design recommendations listed in Figure 5.7 is tied to the graphical elements of PDR shown in Figures 5.8 and 5.9 using a callout whose text is PDR-<Num>, where <Num> is a number.

Figures 5.8 and 5.9 along with textual parallel design recommendations listed in Figure 5.7 collectively serve as a parallel design. The set of parallel design recommendations provided by the Design Assistant are useful in the reengineering the original program to a parallel implementation. At this point, a suitable architecture needs to be investigated. Most recommendations work well for shared memory architectures. COMMON variable accessing is best resolved in shared memory architectures.

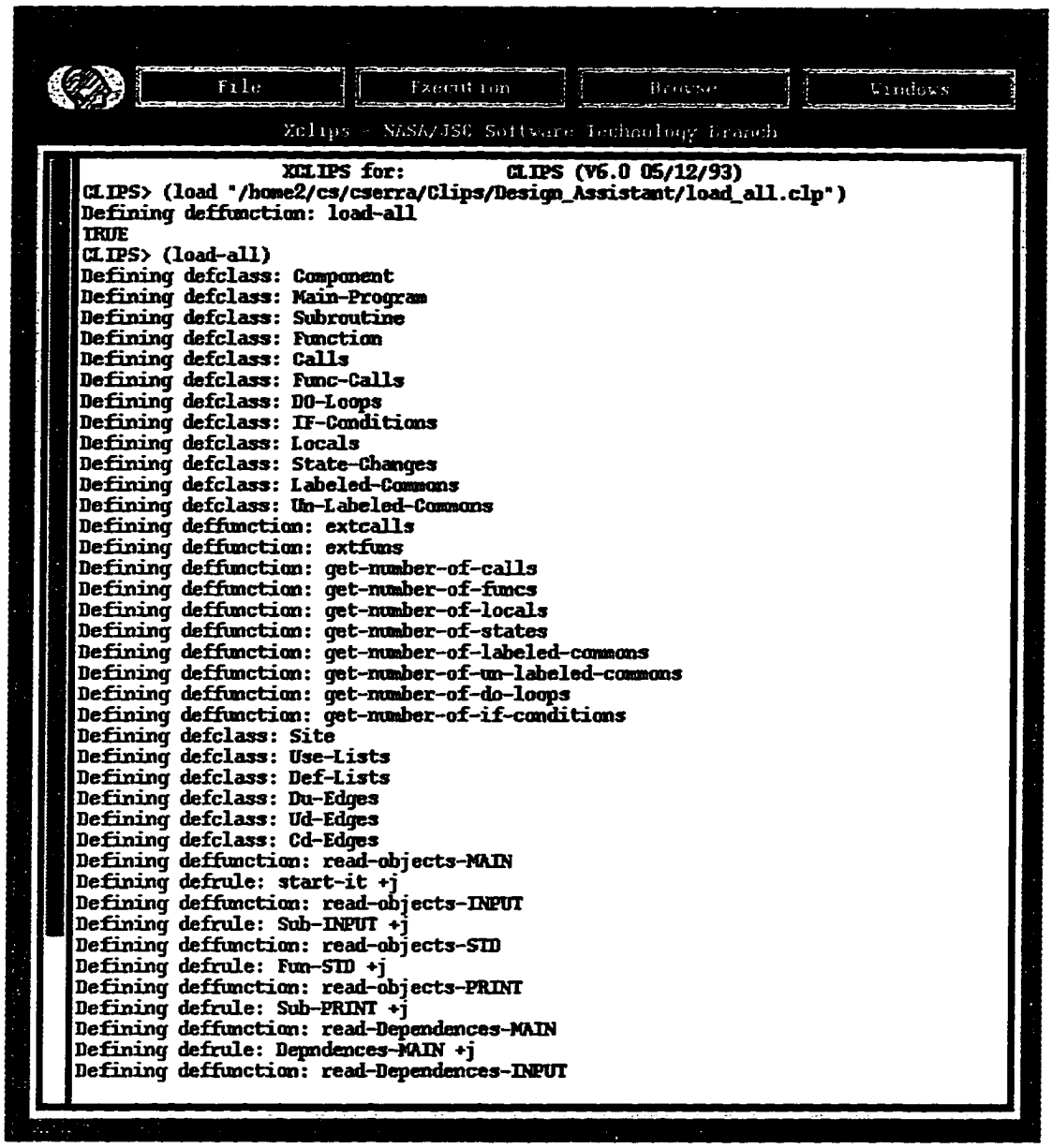


Figure 5.6 Snapshot of the CLIPS-based Design Assistant environment

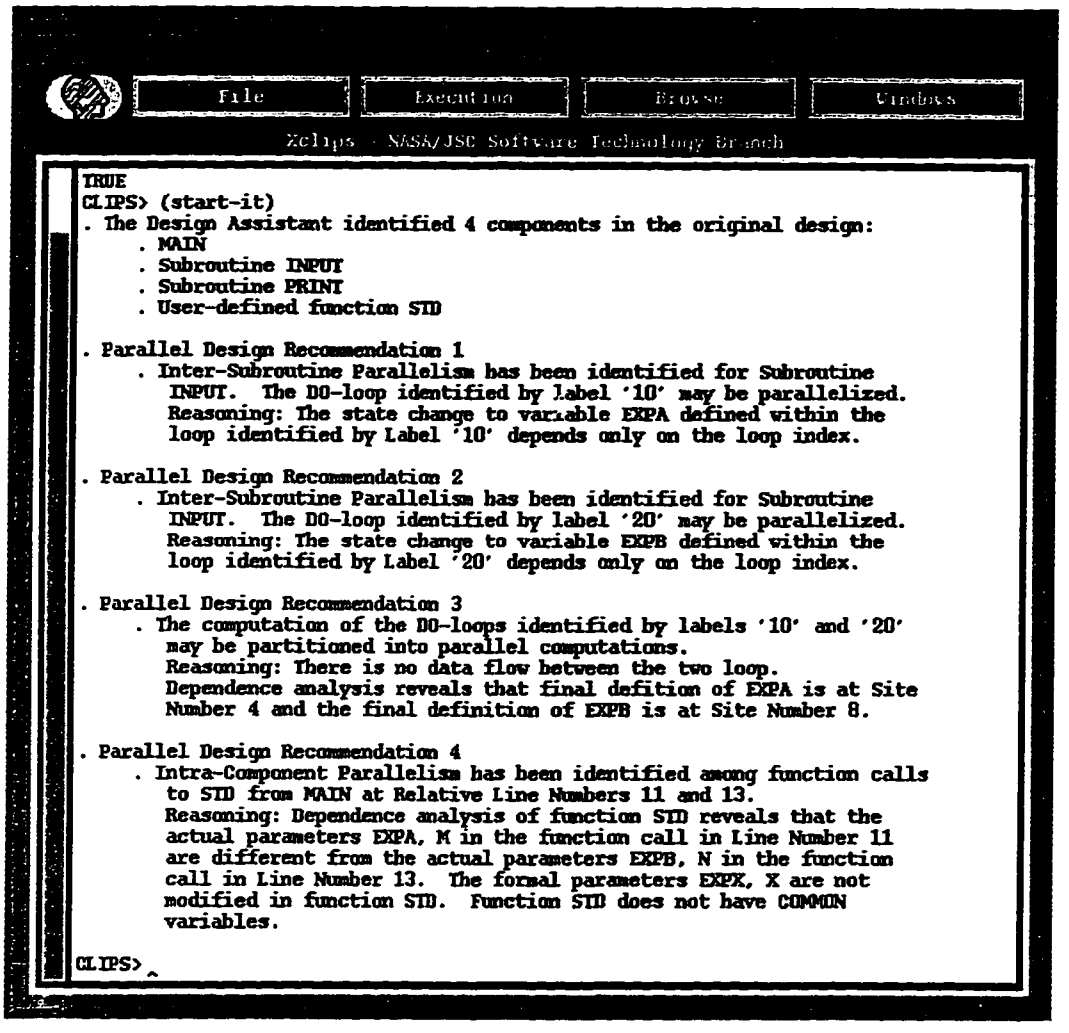


Figure 5.7 Snapshot of the parallel design recommendations in CLIPS

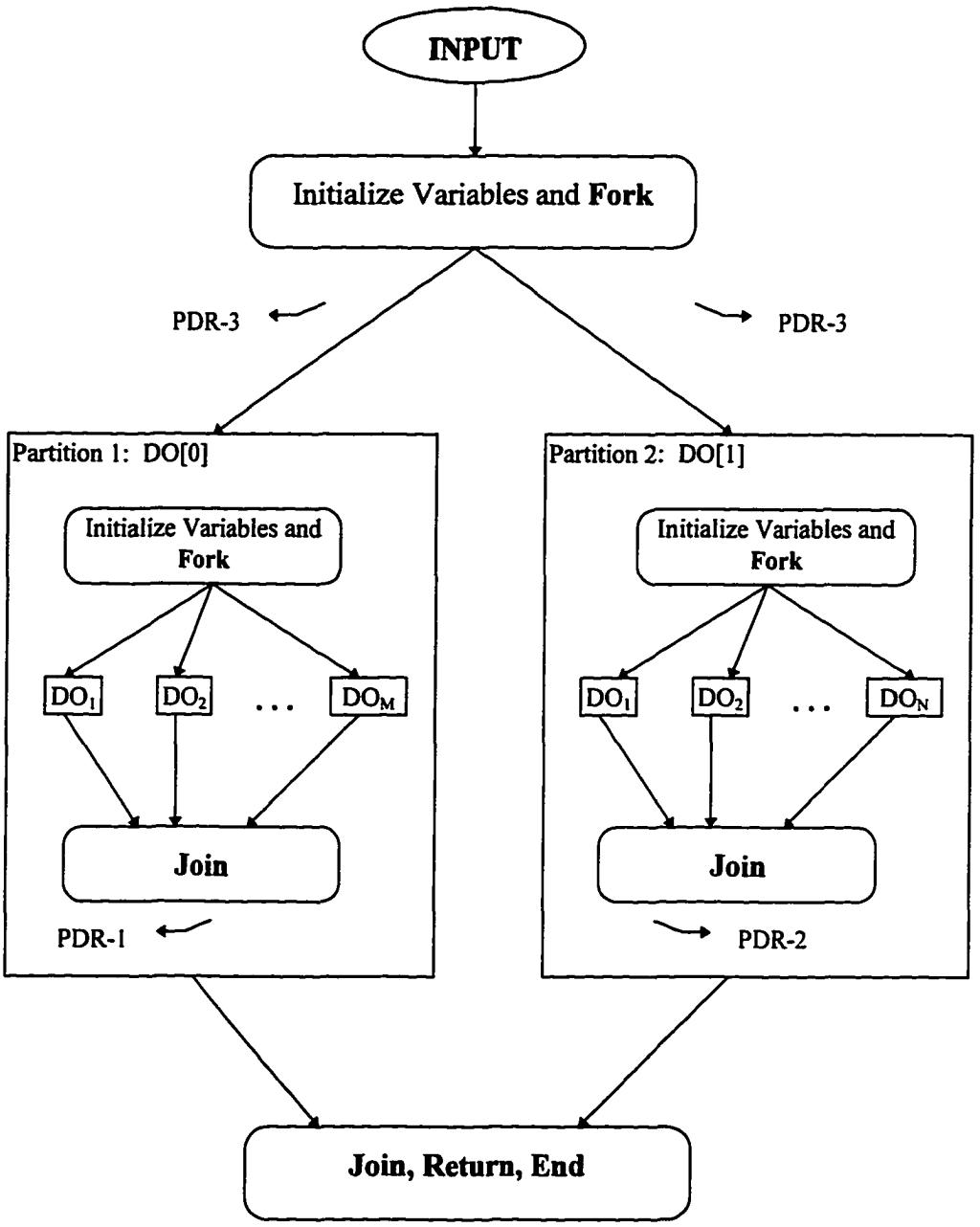


Figure 5.8 PDR representation of subroutine INPUT

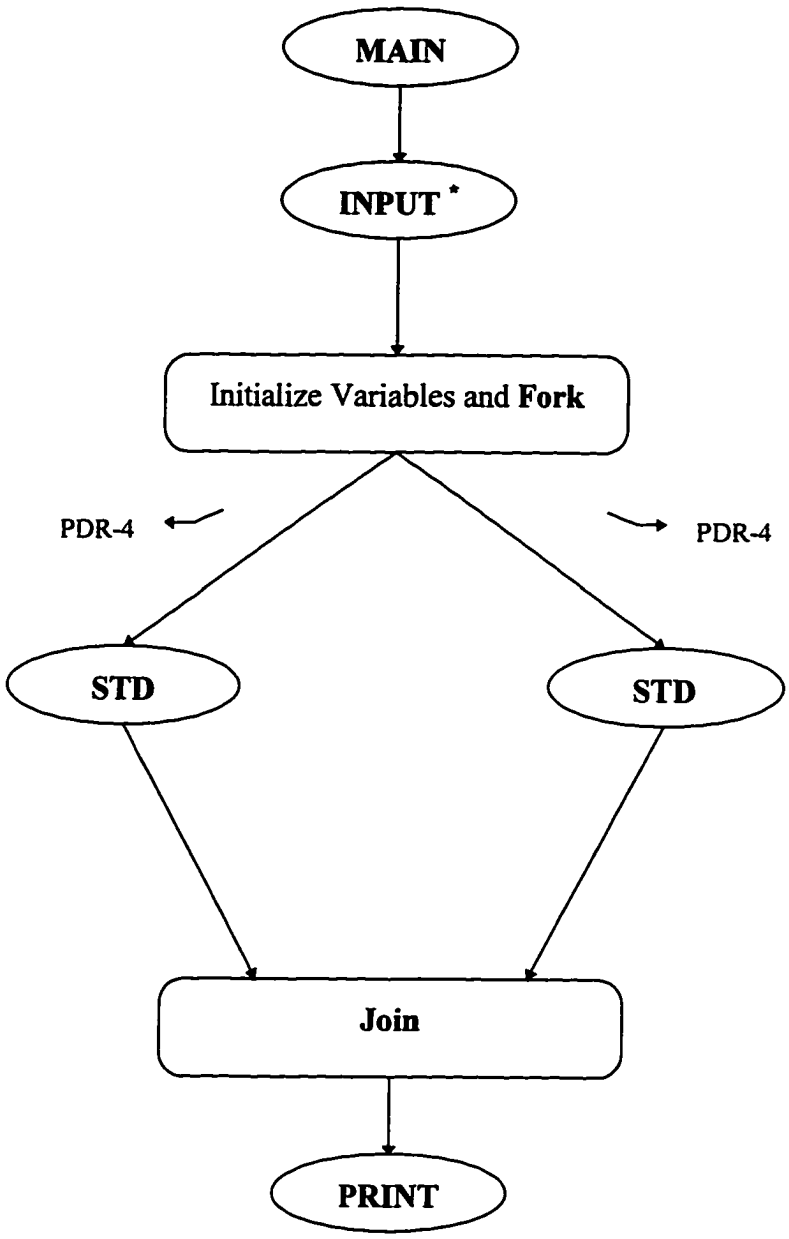


Figure 5.9 PDR representation of the sample program

5.2 Analysis of NAS Kernels

In order to assess scalability of the reverse engineering toolkit (RETK), programs of different sizes were analyzed. The information extractor (IE) was able to handle programs of varying sizes ranging from 60-10000 lines. The maximum size of a program analyzed by IE was CENTRM96 (a nuclear engineering based program) which consisted of more than 10000 lines of code spanning over 100 subroutines and functions.

Numerical Aerodynamic Simulation (NAS) kernel benchmark programs were developed at NASA Ames Research Center for evaluating the performance of highly parallel supercomputers [Bail 94]. The kernel programs, being benchmarks for parallel architectures, have a high potential for parallelization. The source code associated with the kernel programs was used to elicit potential parallelism from the programs. Source files of the kernel programs which were written in FORTRAN-77 include the block tridiagonal solver benchmark (APPBT.F), the lower/upper triangular solver benchmark (APPLU.F), the pentadiagonal solver benchmark (APPSP.F). We have arbitrarily selected the block tridiagonal solver benchmark (APPBT.F) for analysis.

5.2.1 Analysis of APPBT

The source code associated with the kernel program, APPBT.F, was input to IE and the sequential design description of the program was extracted. Figures 5.10-5.14 show several snapshots of the design of APPBT. Figures 5.10-5.12 show metric information which include number of lines in each module, number of commented lines, and number of blank lines. McCabe's cyclomatic complexity for each module is shown in Figure 5.13. From Figure 5.13 it can be observed that APPBT consists of 18

components of which one is the main program and the rest are subroutines. Figure 5.14 shows a partial call graph of APPBT. Table 5.35 lists a textual description of the complete call graph of the APPBT. Other design information including local and non-local variable description, state change information, DO-loop information, IF-conditions information were extracted for each component. Since FORTRAN-77 allows the DO ... END DO construct explicit label names were generated for loops. The label names are of the format RETK<Num> where <Num> is a unique number within a module. The design attributes of each component were saved on disk for use by the Dependence Analyzer (DANA). In addition to design information, program facts for later use by the Design Assistant were generated for each component. The dependence graph consisting of data and control dependences was generated for each component in the design. DANA produced dependence facts of each component in the design by reading the design attributes associated with the component. The dependence graph of each component is quite involved and is best viewed using *xfig*. However, the partial dependence graph for subroutine SETBV is shown in Figure 5.15.

The parallel design recommendations for subroutine SETBV and subroutine EXACT are listed in Tables 5.36. From Table 5.36 it can be observed that 3 parallel design recommendations are derived for subroutine SETBV and 1 for subroutine EXACT. PDR-1 states that the nested DO-loops identified by 'RETK1' and 'RETK2' may be parallelized.

The reasons for arriving at PDR-1 are based on scenario 5 which identified that the two calls to subroutine EXACT have different actual parameters in each call.

Further, each call modifies COMMON variables at different locations. PDR-2 states that the nested DO-loops identified by 'RETK3' and 'RETK4' may be parallelized.

The reasons for arriving at PDR-2 are based on scenario 5 which identified that the two calls to subroutine EXACT have different actual parameters in each call. Further, each call modifies COMMON variables at different locations. PDR-3 states that the nested DO-loops identified by 'RETK5' and 'RETK6' may be parallelized.

The reasons for arriving at PDR-3 are based on scenario 5 which identified that the two calls to subroutine EXACT have different actual parameters in each call. Further, each call modifies COMMON variables at different locations. PDR-4 states that the state changes defined in the DO-loop identified by RETK1 in subroutine EXACT may be partitioned.

The reasons for arriving at PDR-4 are based on scenario 6 which identified that there is no flow data dependence among the state changes in the DO-loop. The PDR representation of subroutine SETBV is shown in Figure 5.16. The PDR representation of subroutine EXACT is shown in Figure 5.17.

The textual description of the parallel design recommendations listed in Table 5.36 is tied to the graphical elements of PDR shown in Figures 5.16 and 5.17 using a callout whose text is PDR-<Num>, where <Num> is a number.

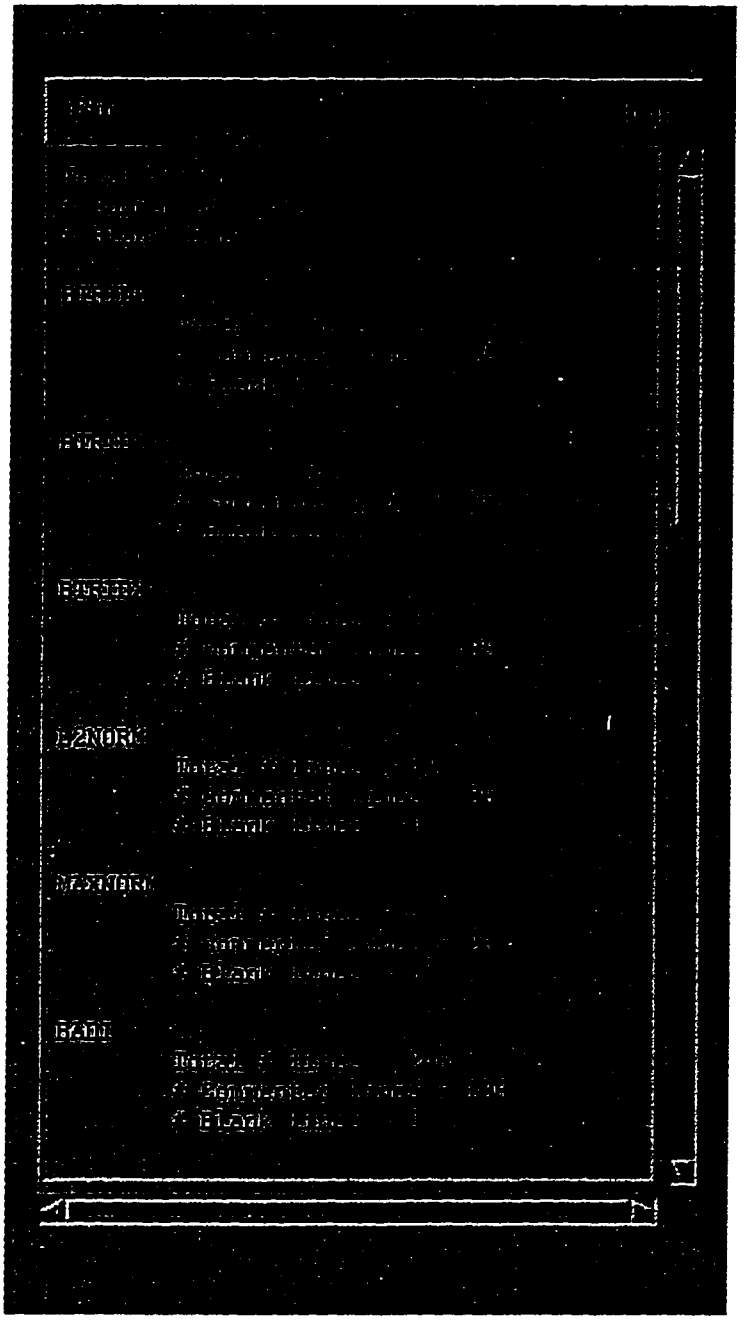


Figure 5.10 Metric information of NAS kernel program APPBT

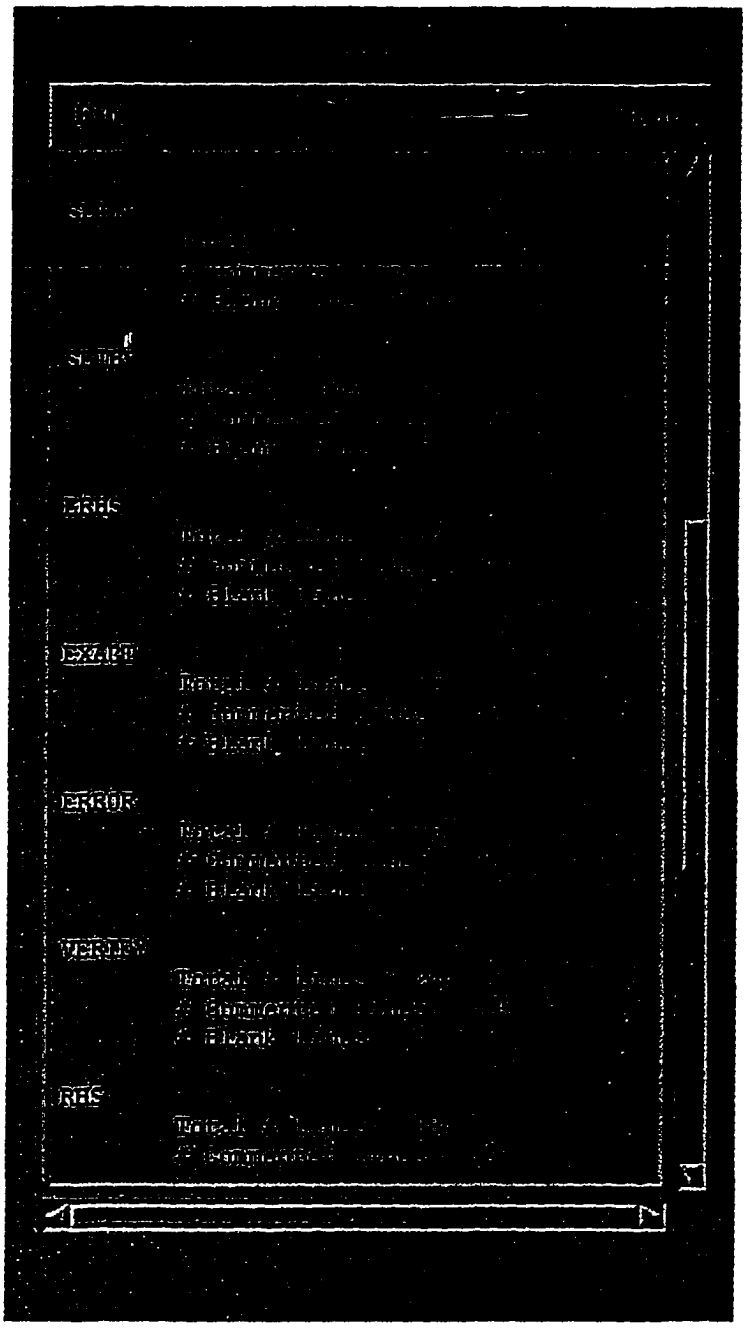


Figure 5.11 Metric information of NAS kernel program APPBT

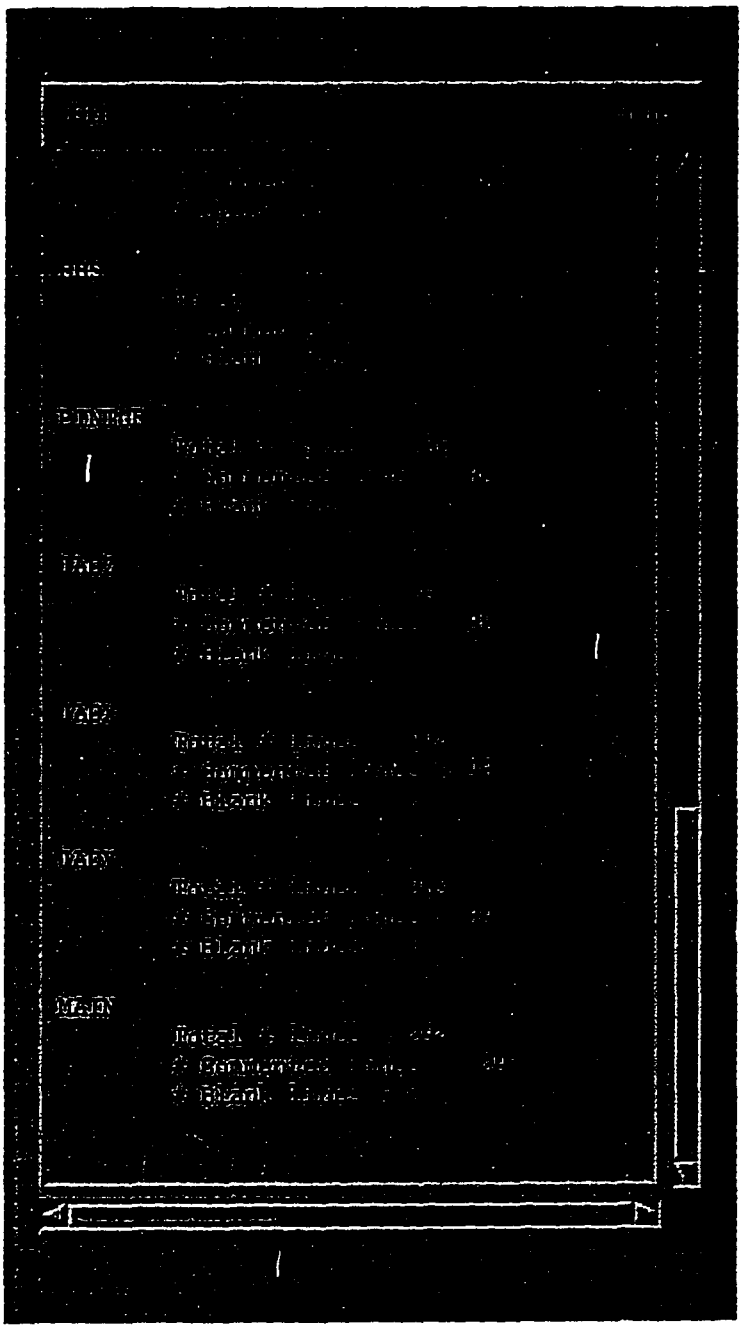


Figure 5.12 Metric information of NAS kernel program APPBT

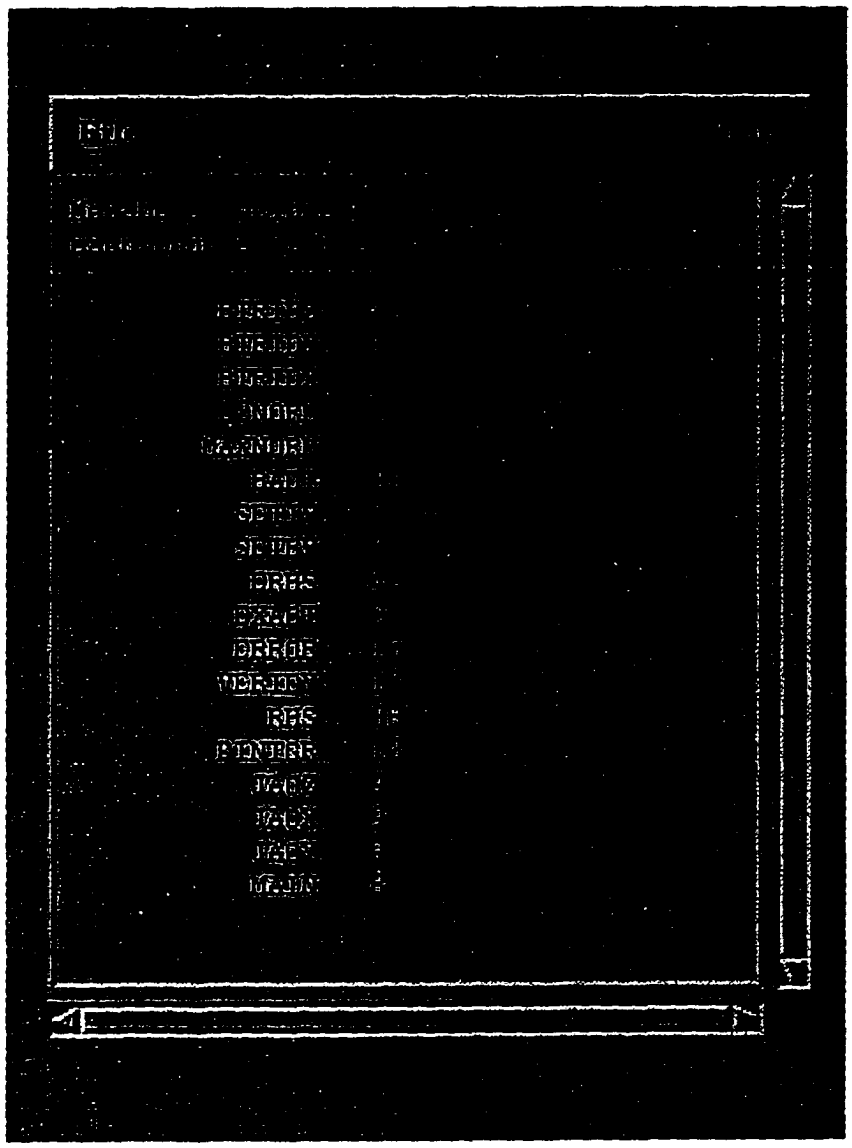


Figure 5.13 Cyclomatic complexity of the various components of the NAS kernel program APPBT

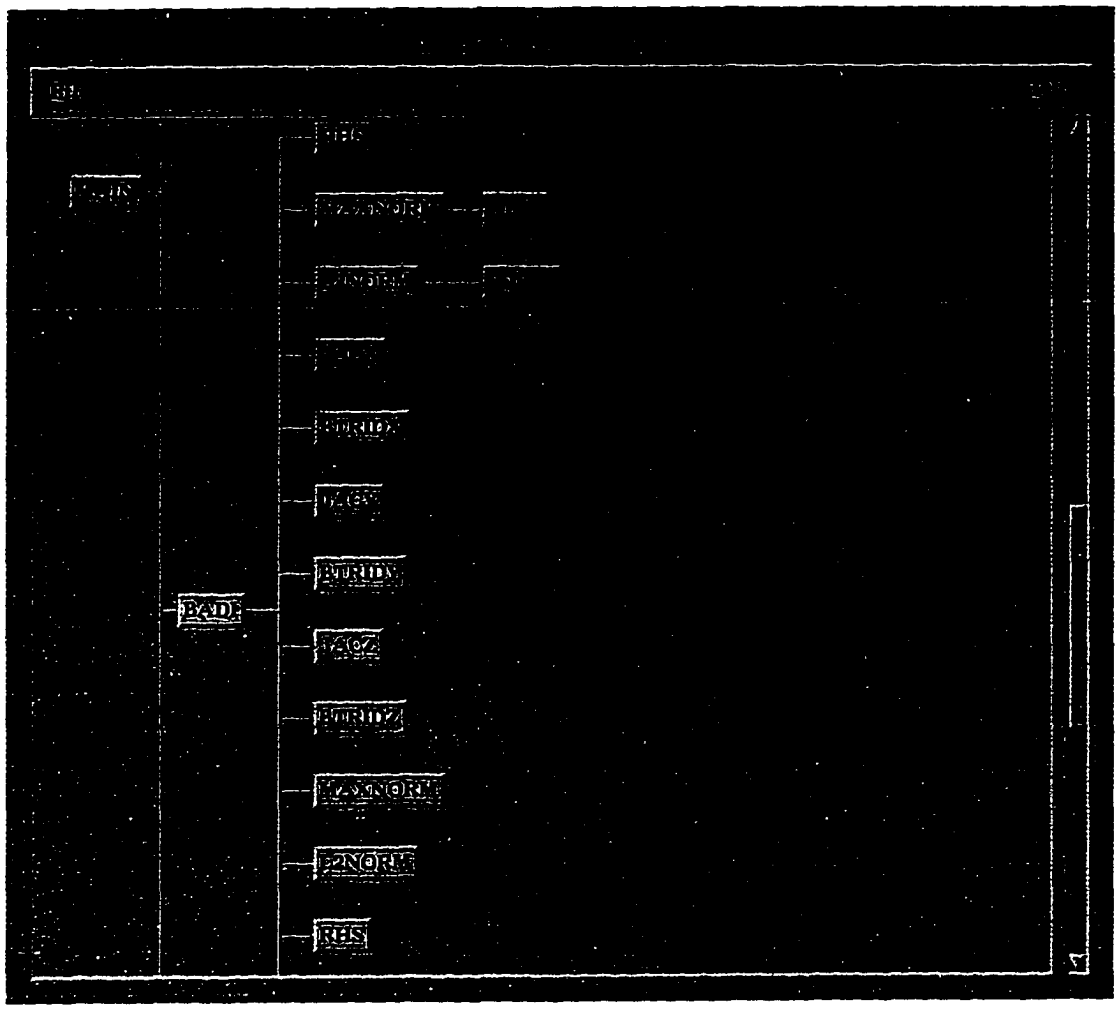


Figure 5.14 Partial call graph of NAS kernel program APPBT

Table 5.35 Complete call graph of the NAS kernel program APPBT

```
MAIN
        --->SETBV
                --->EXACT
                --->EXACT
                --->EXACT
                --->EXACT
                --->EXACT
                --->EXACT
        --->SETIV
        --->ERHS
        --->BADI
                --->RHS
                --->MAXNORM
                --->L2NORM
                --->JACX
                --->BTRIDX
                --->JACY
                --->BTRIDY
                --->JACZ
                --->BTRIDZ
                --->MAXNORM
                --->L2NORM
                --->RHS
                --->MAXNORM
                --->L2NORM
        --->ERROR
                --->EXACT
                --->EXACT
        --->PINTGR
        --->VERIFY
```

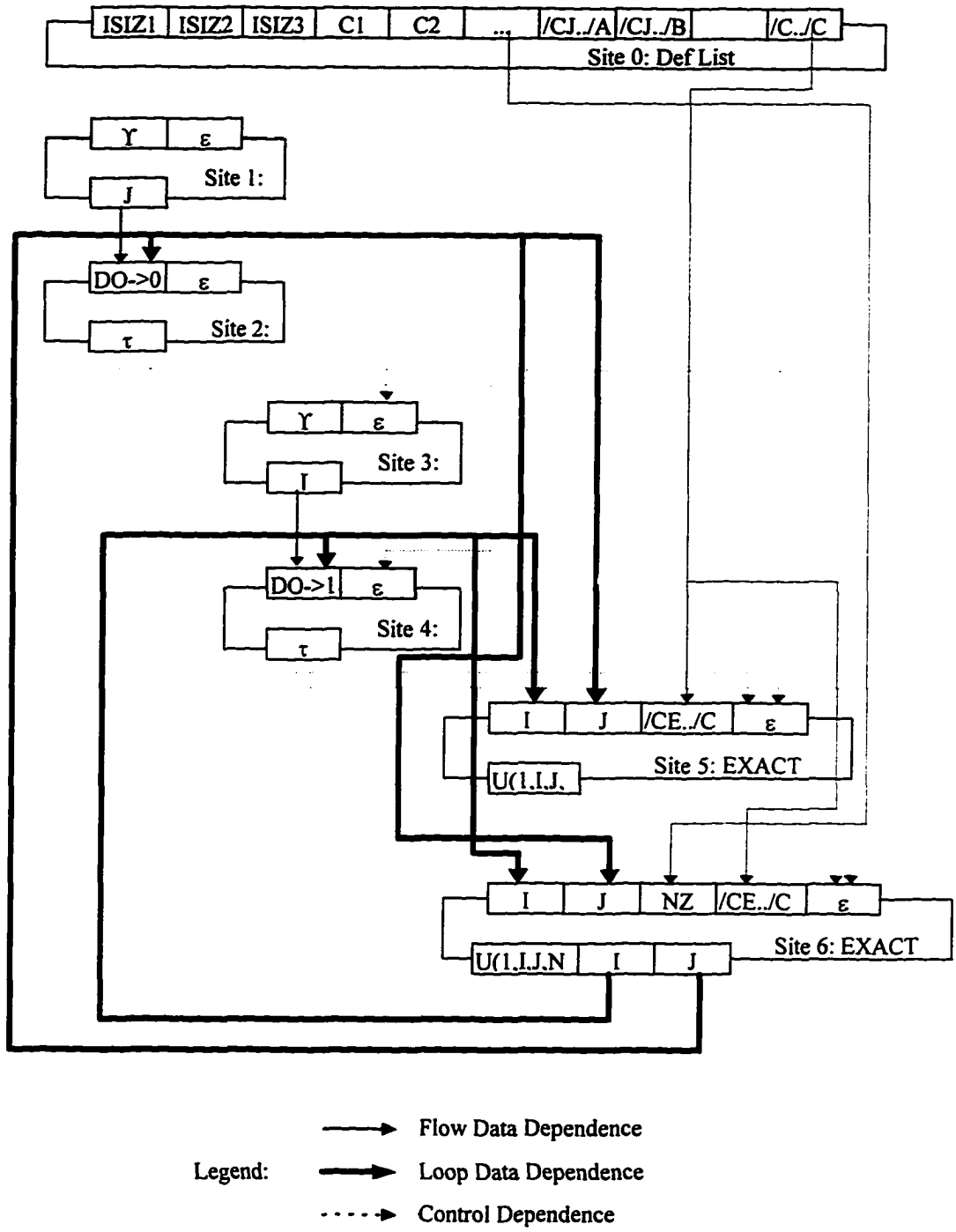


Figure 5.15 Partial dependence graph of subroutine SETBV

Table 5.36 Parallel design recommendations for NAS program APPBT

- The Design Assistant identified 18 components in the original design
- Parallel Design Recommendation 1
  - Intra-Subroutine Parallelism has been identified for Subroutine SETBV. The calls to subroutine EXACT within the nested DO-loop identified by RETK1 and RETK2 may be parallelized.
    Reasoning: Calls to subroutine EXACT have different actual parameters and each call modifies COMMON variables at different locations.
- Parallel Design Recommendation 2
  - Intra-Subroutine Parallelism has been identified for Subroutine SETBV. The calls to subroutine EXACT within the nested DO-loop identified by RETK3 and RETK4 may be parallelized.
    Reasoning: Calls to subroutine EXACT have different actual parameters and each call modifies COMMON variables at different locations.
- Parallel Design Recommendation 3
  - Intra-Subroutine Parallelism has been identified for Subroutine SETBV. The calls to subroutine EXACT within the nested DO-loop identified by RETK5 and RETK6 may be parallelized.
    Reasoning: Calls to subroutine EXACT have different actual parameters and each call modifies COMMON variables at different locations.
- Parallel Design Recommendation 4
  - Inter-Subroutine Parallelism has been identified for Subroutine EXACT. The state changes within the nested DO-loop identified by RETK1 may be parallelized.
    Reasoning: There is no data flow among the state changes. Dependence analysis reveals that the state changes are independent of each other.

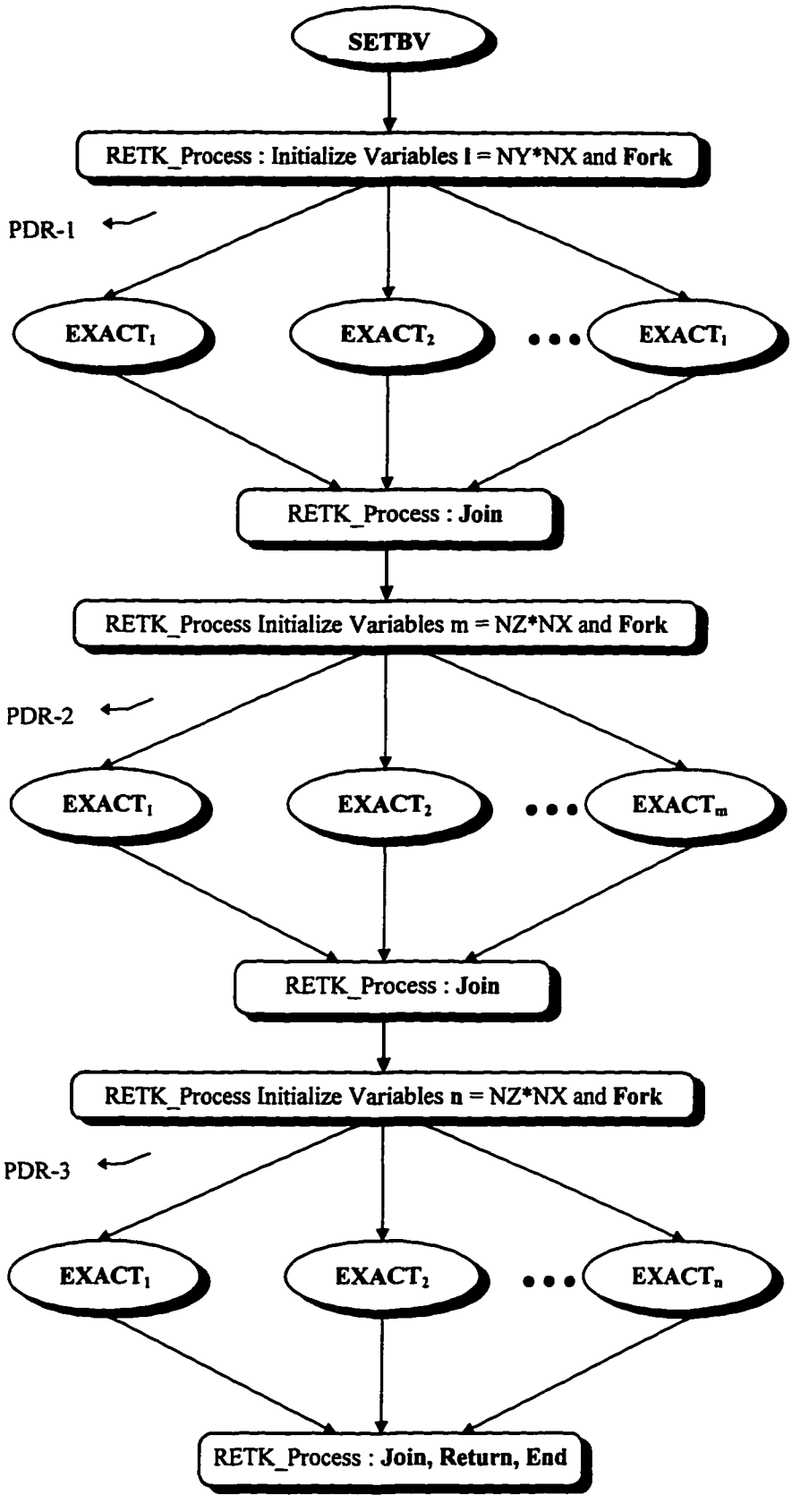


Figure 5.16 PDR representation of subroutine SETBV

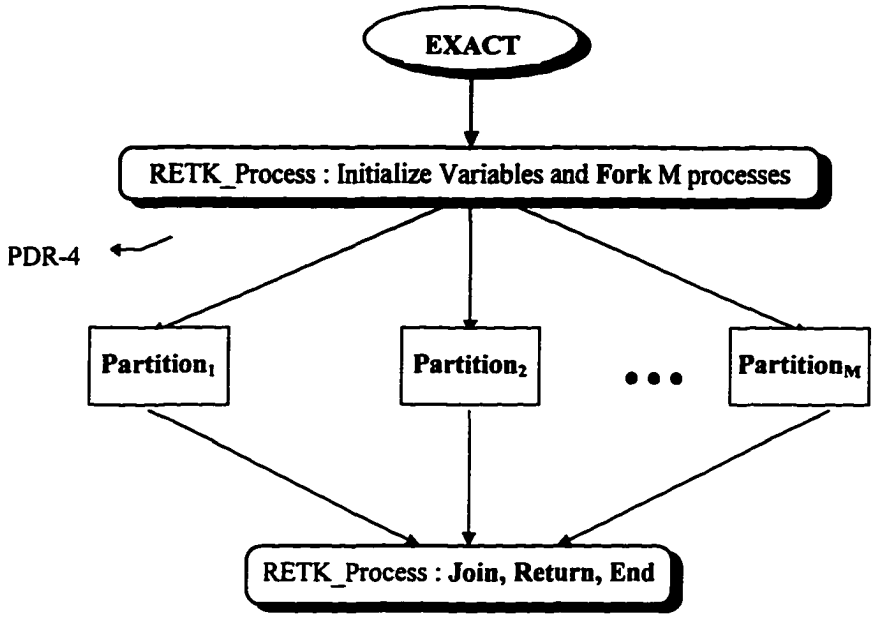


Figure 5.17 PDR representation of subroutine EXACT

5.3 Summary

This chapter presented results of programs to test the methodology presented in Chapter 3. The vehicle used to test-drive the methodology was RETK. Comprehensive results of a short sample program were presented. Results for the sample program include extraction of a sequential design description, analysis and synthesis of the program dependence graph and derivation of parallel design recommendations. Summary results of NAS kernel program APPBT were outlined. PDR notation was used to represent parallel design recommendations for the programs.

Chapter 6

Conclusions

This research was undertaken to 1) seek systematic solutions to the problem of code migration from the sequential to the parallel processing environment, and 2) to investigate and use reverse engineering techniques, dependence analysis, and knowledge-based techniques to provide automated support for the migration process. A summary of the research is presented in Section 6.1. The contributions of this research are summarized in Section 6.2. The extensions and future work possible in this research are explored in Section 6.3.

6.1 Summary

Migration of code from sequential environments to other environments, including parallel processing environments, is often done in an ad hoc manner. This research presents a methodology that facilitates migration of code from the uniprocessor to the parallel processing environments. By combining research in reverse engineering, dependence analysis, and knowledge-based analysis, the methodology provides a systematic approach for code migration.

This research encompasses three major areas; reverse engineering, dependence analysis, and knowledge-based analysis. Chapter 2 presented a literature review of the three major areas. Design recovery and identification of components are some of the main issues in reverse engineering research. The notion of program dependence graph

(PDG) [Ferr 87] and extensions to PDG by [Horw 90] and [Jack 94] were discussed. The PDG model defined in [Jack 94] is particularly suited for reverse engineering research as it views procedures to be modular and dependences to be fine-grained. Even the most trivial reverse engineering task is a "knowledge-intensive" process [Jarz 95]. Knowledge-based techniques discussed in Chapter 2 serve to provide intelligent support in accepting or rejecting decisions.

In Chapter 3, we defined a methodology for design parallelization. The methodology consists of three main phases: Analysis, Synthesis, and Transformation. The processes defined for the Analysis phase are to assess, segment, and extract the sequential design description of a FORTRAN program in the form of call graphs, variable description, and state change information. The processes defined for the Synthesis phase perform dependence analysis for each component and combine the dependence analysis of each component to construct the PDG. The processes defined for the Transformation phase use knowledge-based search and analysis to arrive at the parallel design recommendations. PDR, a graphical representation scheme to represent parallel design recommendations is defined.

Chapter 4 described the design and implementation of the reverse engineering toolkit (RETK). RETK serves to establish a working model for the migration methodology. Object-oriented design and implementation of the Information Extractor (IE) which incorporates the processes of the Analysis phase were presented in detail. Object-oriented design and implementation of the Dependence Analyzer (DANA) which incorporates the processes of the Synthesis phase were also presented. The design and implementation of the Design Assistant (DA) to realize the processes of the Transfor-

mation phase were described. NASA's C-Language Interface Production System (CLIPS) and CLIPS Object-Oriented Language (COOL) were used to design the knowledge base and inference procedure of DA. CLIPS-based knowledge representation schemes were defined for representing program and dependence facts. We provided formal definitions for a set of six scenarios which use program and dependence facts of the original design to identify the parallelization potential in the original design. The scenarios were encoded in the knowledge-base of DA using COOL.

Finally, in Chapter 5, we presented experimental results of actual code analyzed by RETK. A comprehensive description of the results of the analysis was presented for a short sample program. The results are indicative of the document-intensive nature of the output of the various phases of RETK. The results also include analysis of a large NAS kernel benchmark program, APPBT. The effectiveness of RETK, and thereby the migration methodology, to provide parallel design recommendations in a systematic and automated manner was also described.

6.2 Contributions

The undeniable advantages of parallel architectures and computing techniques have precipitated the migration of code from sequential environments. However, the migration process is often done in an ad hoc manner. This research provides systematic approaches to the problem of code migration. The contributions of this research are summarized below:

- A methodology that facilitates systematic migration of code from the uni-processor to parallel processing environments is a significant and original contribution of this

research because the methodology combines concepts from reverse engineering, dependence analysis and knowledge-based analysis whereas most other research efforts are based only on elaborate dependence analysis. In addition, the methodology is scalable unlike most other labor intensive efforts.

- The definition of the processes and the associated algorithms for each phase in the methodology advance the state of reverse engineering research. The processes of each phase operate at the design level of the software system life cycle whereas the processes in most other research efforts operate at the code level. The design level of the software system life cycle provides a higher abstraction of the structure of an existing system thereby facilitating information hiding, a key principle of software engineering.
- A unique knowledge representation scheme for the representation of program facts and dependence facts is an original contribution of this research. The knowledge representation scheme defined is useful not only for parallelization efforts but also for the maintenance of existing systems. The object-oriented knowledge representation scheme serves as a template for the instantiation of comprehensive program and dependence information recovered and analyzed by the Analysis and Synthesis phases respectively. For the purposes of maintenance, it is possible to retrieve specific information from the knowledge base with the help of meaningful queries.
- Since most legacy systems lack in proper documentation, the method that encodes the software design into a knowledge-base is extremely useful. This research defines a unique method to encode the software design into a knowledge-base.

- The scenarios in conjunction with general purpose functions that manipulate the knowledge base constitute a unique rule base for the derivation of parallel design recommendations.
- We defined a graphical notation to represent parallel design recommendations called PDR in order to elucidate the parallelization potential of the original program. PDR makes explicit parallelizable segments in the original design using the classical fork and join representation.
- Parallelization is a time-intensive process. To reduce the personnel costs involved in the parallelization process, high degree of automated support is desirable. Automated support reduces human input and thereby potentially reduces cost. A valuable contribution of this research is the demonstration of the complete automation of the migration methodology through RETK.
- This research uniquely incorporates an explanation mechanism to provide the reasons for arriving at a particular design recommendation thereby introducing traceability for the recommendations.
- The migration methodology and RETK can be used to derive parallel design recommendations from real-world code. Experimental results indicate that very large programs can be handled effectively by RETK.
- Finally, the research contributes to the field of reverse engineering by the integration of knowledge-base techniques. Irrespective of the design recovery procedure used, the knowledge representation scheme defined in this research can be employed to encode information into a knowledge-base for the eventual purpose of parallelization.

The process of re-engineering is most productive when considered as the complement to reverse engineering. Successful re-engineering of a system requires an in depth understanding of the concepts and functionality of the system. The reverse engineering methodology defined in this dissertation serves as a prelude to re-engineering.

6.3 Future Research

The extensions for this work include:

- RETK can be modified to support other source languages like C. Since C allows recursion and user-defined data types, the processes of the three phases of the methodology should take into account the subtleties of the source language during design and implementation..
- The derivation of architecture-specific and vendor-specific parallel design recommendations is a possible area of future research. A suitable knowledge representation scheme needs to be explored to represent the topologies of different parallel architectures in the knowledge-base of the Design Assistant. Specific topological considerations include the number of autonomous processors, the interconnection network for communication among the processors, and the choice of shared memory or distributed memory. For intra-component parallelization, the processing associated with different components may be assigned to different processors. Although all the processors in the configuration could be potentially employed, the interconnection network places a limitation on the communication of large data sets such as arrays. Shared memory configurations are suitable when

the communication overhead is high while distributed memory configurations are suitable for subroutines and functions that do not pass large arrays. Therefore, input data sets play a major role in the derivation of architecture-specific parallel design recommendations. Also, when deriving design recommendations for vendor-specific parallel architectures, it may be necessary to incorporate domain and application specific knowledge into the knowledge-base. Grouping of related domains and clustering of similar applications to arrive at generic knowledge-representation templates would be very useful.

- In recent years, the client-server paradigm has been very well received by the industry. Migration of legacy systems to client-server platforms with the help of the methodology presented in this research is an area worthy of investigation.
- The application of the migration methodology to extract reusable frameworks from legacy code is a possible area of future research.
- Reengineering tasks such as automatic code generation and automatic documentation from parallel design recommendations and existing sequential code would be an invaluable extension of this work.

Bibliography

[Ache 94] Achee, B.L., Carver, D.L., "A Greedy Approach to Object Identification in Imperative Code," In Proceedings of 3rd Workshop on Program Comprehension, 1994.

[Ache 95] Achee, B.L., Carver, D.L., "Identification and Extraction of Objects from Legacy Code," In Proceedings of the IEEE Aerospace Applications Conference, Snowmass, CO, (February 1995), pp. 181-190.

[Aho 77] Aho, A.V., Ullman, J.D., *Principles of Compiler Design*, Addison-Wesley, 1977.

[Aike 93] Aiken, P., Muntz, A., Richards, R., "A Methodology for Reverse Engineering DoD Legacy Information Systems," In Proceedings of the Working Conference on Reverse Engineering, Baltimore, MD, (May 1993), pp. 180-191.

[Aike 94] Aiken, P., Muntz, A., Richards, R., "DoD Legacy Systems: Reverse Engineering Data Requirements," Communications of the ACM, Vol. 37, No.5 (May 1994), pp. 26-41.

[ANSI 78] American National Standards Institute, American National Standard Programming Language FORTRAN, ANSI X3.9, 1978, ANSI, NY.

[Bail 94] Bailey, D., Barszcz, E., Barton, J., Browning, D., et. al. "The NAS Parallel Benchmarks," NASA RNR Technical Report RNR-94-007, March 1994.

[Beck 93] Beck, J., Eichmann, D., "Program and Interface Slicing for Reverse Engineering," 1993 IEEE 15th International Conference on Software Engineering, pp. 509-518.

[Bigg 94] Biggerstaff, T.J., Mitbander, B.G., Webster, D.E., "Program Understanding and the Concept Assignment Problem," Communications of the ACM, Vol. 37, No.5 (May 1994), pp. 72-83.

[Butl 95] Butler, G., Grogono, P., Shingal, R., Tjandra, I., "Retrieving Information from Data Flow Diagrams," In Proceedings of the Second Working Conference on Reverse Engineering, Toronto, Canada (July 1995), pp. 22-29.

[Canf 93] Canfora, G., Cimitile, A., Munro, M., "A Reverse Engineering Method for Identifying Reusable Abstract Data Types," In Proceedings of the Working Conference on Reverse Engineering, Baltimore, MD (May 1993), pp. 73-82.

[Chik 90] Chikofsky, E.J., Cross II, J.H., "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, Vol. 13, No. 1 (January 1990), pp.13-17.

[Choi 90] Choi, S.C., Scacchi, W., "Extracting and Restructuring the Design of Large Systems," IEEE Software, Vol. 13, No. 1 (January 1990), pp. 66-71.

[Clp 93a] CLIPS Reference Manual, , Volume I, "Basic Programming Guide," JSC-25012, CLIPS Version 6.0, Software Technology Branch, NASA Johnson Space Center.

[Clp 93b] CLIPS Reference Manual, , Volume II, "Advanced Programming Guide," JSC-25012, CLIPS Version 6.0, Software Technology Branch, NASA Johnson Space Center.

[Clp 93c] CLIPS Reference Manual, , Volume III, "Interfaces Guide," JSC-25012, CLIPS Version 6.0, Software Technology Branch, NASA Johnson Space Center.

[Cuti 93] Cutillo, F., Fiore, P., Visaggio, G., "Identification and Extraction of "Domain Independent" Components in Large Programs," In Proceedings of the Working Conference on Reverse Engineering, Baltimore, MD (May 1993), pp. 83-92.

[Dijk 76] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, 1976.

[Edwa 93] Edwards, H.M., Munro, M., "RECAST: Reverse Engineering from COBOL to SSADM Specification," In Proceedings of the Working Conference on Reverse Engineering, Baltimore, MD (May 1993), pp. 44-45.

[Erra 96] Erraguntla, R.C., Carver, D.L., "A Reverse Engineering Toolkit for Design Recovery," In Proceedings of the 2nd IEEE Aerospace Applications Conference, Snowmass, CO (February 1996), pp. 243-252.

[Ferr 87] Ferrante, J., Ottenstein, K.J., Warren, J.D., "The Program Dependence Graph and its use in Optimization," ACM Transactions on Programming Languages and Systems, Vol. 9, No. 3 (July 1987), pp. 319-350.

[Flyn 72] Flynn, M.J., "Some Computer Organizations and Their Effectiveness," IEEE Transactions on Computers, Vol. 21, No. 9 (September 1972), pp. 948-960.

[Ghez 91] Ghezzi, C., Jazayeri, M., Mandrioli, D., *Fundamentals of Software Engineering*, Prentice-Hall, 1991.

[Hagh 91] Haghighat, M., Pluchronopoulos, C., "Symbolic Dependence Analysis for High-Performance Parallelizing Compilers," in Advances in Languages and Compilers for Parallel Processing (Edited by Nicoloau, A., et. al.), MIT Press, 1991.

[Hara 90] Harandi, M.T., Ning, J.Q., "Knowledge-Based Program Analysis," IEEE Software, Vol. 13, No. 1 (January 1990), pp.74-81.

[Harr 93] Harrison, W., Gens, C., Gifford, B., "pRETS: a parallel Reverse-engineering ToolSet for FORTRAN," Software Maintenance: Research and Practice, Vol. 5 (1993), pp. 37-57.

[Holo 83] Holoien, M.O., Behforooz, A., *Problem Solving and Structured Programming with Fortran-77*, Brooks/Cole, 1983.

[Horw 90] Horwitz, S., Reps, T., Binkley, D., "Interprocedural Slicing Using Dependence Graphs," ACM Transactions on Programming Languages and Systems, Vol. 12, No. 1 (January 1990), pp. 26-60.

[Hwan 93] Hwang, K., *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw Hill, Inc., 1993.

[Ingl 94] Ingle, K.A., *Reverse Engineering*, McGraw-Hill, Inc., 1994.

[Jack 75] Jackson, M.A., *Principles of System Design*, Academic Press, London, 1975.

[Jack 94] Jackson, D., Rollins, E.J., "A New Model of Program Dependences for Reverse Engineering," Software Engineering Notes, SIGSOFT '94, ACM Press, Vol. 19, No. 5 (December 1994), pp. 2-10.

[Jarz 95] Jarzabek, S., Keam, T.P., "Design of a Generic Reverse Engineering Assistant Tool," In Proceedings of the Second Working Conference on Reverse Engineering, Toronto, Canada (July 1995), pp. 61-70.

[Kara 95] Karadimitriou, K., Tyler, J.M., Brener, N.E., "Reverse Engineering and Reengineering of a Large Serial System into a Distributed-Parallel Version," In Proceedings of the 1995 ACM Symposium on Applied Computing, Nashville, TN (February 1995), pp. 191-197.

[Kuck 72] Kuck, D.J., Muraoka, Y., Chen, S-C., "On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup," IEEE Transactions on Computers, Vol. C-21, No.12 (December 1972), pp.1293-1322.

[Kuck 78] Kuck, D.J., *The Structure of Computers and Computations, Volume 1.*, John Wiley & Sons, 1978.

[Lano 93] Lano, K., Haughton, H., "Integrating Formal and Structured Methods in Reverse-Engineering," In Proceedings of the Working Conference on Reverse Engineering, Baltimore, MD (May 1993), pp. 17-26.

[Lilj 94] Lilja, D. J., "Exploiting Parallelism Available in Loops," IEEE Computer, February 1994, pp. 13-26.

[Love 93] Loveman, D.B., "High Performance Fortran," IEEE parallel and distributed technology, February 1993, pp. 25-42.

[Luts 95] Lutsky, P., "Automating Testing by Reverse Engineering of Software Documentation," In Proceedings of the Second Working Conference on Reverse Engineering, Toronto, Canada (July 1995), pp. 8-12.

[Marc 86] Marcotty, M., Ledgard, H., *Programming Language Landscape*, Science Research Associates, Inc., 1986.

[Mark 94] Markosian, L., Newcomb, P., Brand, R., Burson, S., Kitzmiller, T., "Using an Enabling Technology to Reengineer Legacy Systems," Communications of the ACM, Vol. 37, No. 5 (May 1994), pp. 58-90.

[MasP 91] MasPar, "The MasPar Family Data-Parallel Computer," Technical summary, MasPar Computer Corporation, Sunnyvale, CA, 1991.

[Newc 93] Newcomb, P., Markosian, L., "Application of an Enabling Technology for Reengineering," In Proceedings of the Working Conference on Reverse Engineering, Baltimore, MD (May 1993), pp. 222-230.

[Newc 95] Newcomb, P., "Legacy System Cataloging Facility," In Proceedings of the Second Working Conference on Reverse Engineering, Toronto, Canada (July 1995), pp. 52-60.

[Ning 93] Ning, J. Q., Engberts, A., Kozaczynski, W., "Recovering Reusable Components from Legacy Systems by Program Segmentation," In Proceedings of the Working Conference on Reverse Engineering, Baltimore, MD (May 1993), pp. 64-72.

[Ning 94] Ning, J. Q., Engberts, A., Kozaczynski, W., "Automated Support for Legacy Code Understanding," Communications of the ACM, Vol. 37, No. 5 (May 1994), pp. 50-57.

[Osbo 90] Osborne, W.M., Chikofsky, E.J., "Fitting Pieces to the Maintenance Puzzle," IEEE Software, Vol. 13, No. 1 (January 1990), pp. 11-12.

[Pres 92] Pressman, R.S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill , Inc., Third Edition, 1992.

[Quil 94] Quilici, A., "A Memory-Based Approach," Communications of the ACM, Vol. 37, No. 5 (May 1994), pp. 84-93.

[Ric 88a] Rich, C., Waters, R.C., "Automatic Programming: Myths and Prospects," Computer, Vol. 21, No. 8 (August 1988), pp. 40-51.

[Ric 88b] Rich, C., Waters, R.C., "The Programmer's Apprentice: A Research Overview," Computer, Vol. 21, No. 11 (November 1988), pp. 10-25.

[Rich 83] Rich, E., *Artificial Intelligence*, McGraw-Hill, Inc. 1993.

[Rich 90] Rich, C., Wills, L.M., "Recognizing a Program's Design: A Graph-Parsing Approach," IEEE Software, Vol. 13, No. 1 (January 1990), pp. 82-89.

[Ruga 90] Rugaber, S., Ornburn, S.B., LeBlanc, Jr., R.J., "Recognizing Design Decisions in Programs," IEEE Software, Vol. 13, No. 1 (January 1990), pp. 46-54.

[Rumb 91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy., F., Lorensen, W., *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[Scha 96] Schach, S.R., *Classical and Object-Oriented Software Engineering*, 3rd Edition, Irwin, 1996.

[Seba 93] Sebesta, R.W., *Concepts of Programming Languages*, Benjamin/Cummings, Second. Edition, 1993.

[Sell 85] Sell, P.E., *Expert Systems: A Practical Introduction*, Halsted Press, London, 1985.

[Ston 94] Stone, H.S., High-Performance Computer Architecture, Addison-Wesley Publishing Company, 1994.

[Wate 94] Waters, R.C., Chikofsky, E., "Reverse Engineering: Progress Along Many Dimensions," Communications of the ACM, Vol. 37, No. 5 (May 1994), pp. 22-25.

[WCRE 93] Proceedings of the Working Conference on Reverse Engineering Baltimore, MD (May 1993), IEEE Computer Society Press, 1993.

[WCRE 95] Proceedings of the 2nd Working Conference on Reverse Engineering Toronto, Canada (July 1995), IEEE Computer Society Press, 1993.

[Weis 84] Weiser, M., "Program Slicing," IEEE Transactions on Software Engineering," Vol. SE-10 No. 4 (July 1984), pp. 352-357.

[Wins 92] Winston, P. H., *Artificial Intelligence*, Addison-Wesley Publishing Company, Third Edition, 1992.

[Zuyl 93] Zuylen H., (Ed.), The REDO Compendium of Reverse-Engineering for Software Maintenance, Wiley, 1993.

Vita

Ravi Erraguntla was born on October 5, 1965 in Visakhapatnam, India. Ravi received the B.E. degree in Electronics and Communications Engineering from Andhra University, India and the M.E. degree in Electrical Engineering from Bharatiar University, India. He worked as a Lecturer in Andhra University from 1989 - 1990. Ravi came to the U.S. in Fall 90 and enrolled as a graduate student in Louisiana State University. At LSU, Ravi received the M.S. degree in Nuclear Engineering and the Ph.D. degree in Computer Science. During his stint at LSU he worked as a Graduate Research Assistant in the Nuclear Science Center and the Department of Computer Science. As part of his research assistantship he was involved as a Research Consultant for Thermalscan Inc., Baton Rouge and Medical Thermal Diagnostics, Baton Rouge. He is presently working as a Consultant for Sabre Decision Technologies and is located in the Dallas area.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Ravi Chandra Erraguntla

Major Field: Computer Science

Title of Dissertation: A Reverse Engineering Methodology For Extracting Parallelism From Design Abstractions

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

August 29, 1996