2003

Conceptual roles of data in program: analyses and applications

Yunbo Deng
Iowa State University

Conceptual roles of data in program:

analyses and applications

by

Yunbo Deng

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Suraj Kothari, Major Professor
Daniel Berleant
Jim Davis
Shashi K. Gadia
Manimaran Govindrasu

Iowa State University

Ames, Iowa

2003

Graduate College
Iowa State University

This is to certify that the doctoral dissertation of

Yunbo Deng

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Major Professor

Signature was redacted for privacy.

For the Major Program

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

First I want to give my heartfelt thanks to my advisor Dr. Suraj Kothari for his persistent guidance and confidence in my research. Our program comprehension methodology is driven by his vision of domain-specific knowledge. Without this vision, we would have never been able to achieve what we have.

During the years, I worked closely with my teammates Yogy Namara, Jeremias Sauceda, Sastra Winarta, Nikhil Renade, Jaekyu Cho, etc. Yogy helped me to explore different tools that eventually aided our rapid development. His acute sense of technology always impressed me. Jeremias primarily took the lead on developing our program analysis frontend *EDG-to-XML* utility. His graphic package *Veras* also has been used for developing the forward/backward program slice browser in our comprehensive tool *SeeCORE*. Sastra and I worked together on prototyping SeeCORE in summer 2002. He was the lead developer on the visualization of multiple views of program. We cooperated closely and I am always proud that we accomplished the framework of the prototype in just one month. I learned a lot from Nikhil and Jaekyu both inside and outside of the academics. Working with them was a very pleasant experience and we developed close friendships. Undoubtedly, all these people are in remembrance with the open, friendly, and peaceful environment only available in Iowa State University.

I also want to thank my program committee for their time reviewing my progress and thesis. I took their valuable suggestions in the preliminary exam to the final dissertation.

Finally, how can I express my deep appreciation to my family who always stood beside me through these years? For their persistent understandings, encouragements and expectations, I motivated myself to do my best in all ways to make them proud of me.

ABSTRACT

Program comprehension is the prerequisite for many software evolution and maintenance tasks. Currently, the research falls short in addressing how to build tools that can use domain-specific knowledge to provide powerful capabilities for extracting valuable information for facilitating program comprehension. Such capabilities are critical for working with large and complex program where program comprehension often is not possible without the help of domain-specific knowledge.

Our research advances the state-of-art in program analysis techniques based on domain-specific knowledge. The program artifacts including variables and methods are carriers of domain concepts that provide the key to understand programs. Our program analysis is directed by domain knowledge stored as domain-specific rules. Our analysis is iterative and interactive. It is based on flexible inference rules and inter-exchangeable and extensible information storage. We designed and developed a comprehensive software environment SeeCORE based on our knowledge-centric analysis methodology. The SeeCORE tool provides multiple views and abstractions to assist in understanding complex programs. The case studies demonstrate the effectiveness of our method. We demonstrate the flexibility of our approach by analyzing two legacy programs in distinct domains.

Keywords: Program Comprehension, Domain-Specific Knowledge Centric, Conceptual Analysis, Software Reengineering

CHAPTER 1. INTRODUCTION

In the area of software engineering, there are two major fields - forward engineering and reverse engineering, also called reengineering. Research in forward engineering focuses on developing new software while reverse engineering focuses on reusing, improving, and leveraging existing software. In practice, these two seemingly reverse directions often coexist and need to work together. Legacy code is a valuable asset. Developing software from scratch is expensive, and not practical in many cases. In addition, rewriting software is actually not as effective as it seems to be. Previously experienced problems could reoccur when software is rewritten. This makes the idea of completely rewriting software risky. Indeed, reusing legacy code, improving performance, and adding new features are daily practices in software industries.

Maintaining software is a difficult and also costly task. With many years of developmental effort software becomes complex. The original design often gets buried in the complex code. Even experts who are familiar with the code may find difficulties managing the software due to its massive size and complexity. Thus, tools are necessary to assist the expert in capturing his or her interests or concerns. Tools are also needed to help the expert to manipulate the legacy code, including refactoring, partial rewriting etc.

An effective software tool should be a tool that emulates a domain expert's effective techniques of dealing with complex software problems. Domain experts, as in our paper, are programmers who are familiar with the application domain logic and related high-level techniques that are necessary to understand how the program works. The tool should be able to assist the domain expert in a way which is similar to the manual process so that he or she can use the tool naturally and efficiently.

Our research on software tools is motivated by our experiences of working with experts in

domains such as mechanical engineering, aerospace engineering, mathematics and physics and observing how their domain knowledge is applied in dealing with complex problems arising in legacy software in the scientific domain. In section 1.1, we use a parallelization of a massive weather prediction code MM5 as an example to demonstrate the manual processes used in reengineering legacy code.

1.1 MM5 - An Example of Our Observations

NCAR/Penn State MM5 is a mesoscale meteorology model, a climate simulation designed for studying weather prediction [Anthes and Warne, 1978]. To study problems such as global warming, long-term simulations ranging over hundred years are necessary. Climate modeling is highly computation and data intensive. Parallelizing MM5 can provide the capability necessary to perform long-term simulations and improve accuracy by allowing scientists to use finer grids in space.

Without proper domain knowledge of MM5 as well as mesoscale model, theoretically, a novice can also "parallelize" MM5. The methodology the novice could use is similar to many automated parallelization tools: Scanning through the program and identifying all compute-intensive parts, e.g., loops, and determining data partition as well as computation partition. The novice must be knowledgeable of the communications needed for data exchange between different grids since he or she will need to carefully measure the balance between computation benefits from parallelization and extra data communication costs.

Unfortunately, the novice's approach is not always effective, especially for massive actual code like MM5. A general parallelization approach like the novice's is inadequate. Global parallelization optimization has been proved to be a NP-complete problem, which implies that the automated parallelization may not be able to achieve optimized parallel code by using reasonable resources [Li and Chen, 1991]. The novice may manage to achieve certain speedups; however, the parallelization requires very outstanding performance in respects of both speedup and efficiency.

In addition to the inherent algorithmic problems, the complexity of the legacy code itself

is already a big challenge. Since the code has evolved for many years, the program patterns have become inconsistent and ambiguous. It is very difficult for general pattern recognizer to detect all kinds of computing patterns and then choose appropriate parallelization strategies. We categorize the novice's methodology as bottom-up methodology. The philosophy is that disregarding the domain knowledge or high-level knowledge of the code, a person or a tool basically recognizes local patterns of a code, and then applies domain-independent strategies.

Experienced experts have successfully accomplished the parallelization of MM5. The results are sufficient to meet the practical needs [Kothari *et al.*, 2002]. In contrast with the novice's methodology, the domain experts applied top-down strategy which benefits from the domain knowledge.

A domain expert possesses the domain knowledge of the underlying numerical method, the programming knowledge of MM5 code and the bindings between the domain knowledge the program entities. The domain expert understands the following facts [Mitra *et al.*, 2000]:

1. The underlying numerical technique is finite difference method using a regular grid for discretization of space. Arrays are used to store spatial data.

2. Index variables represent spatial coordinates.

3. Physical interactions only occur between neighboring grid cells and that is reflected by the code pattern where the indexes of array accesses are all in form of $i \pm \alpha$, where α is a constant less than 3.

4. Due to the physical properties of the modeling, the computation of the grid cells can be done in parallel along both or either of the two dimensions on the horizontal plane.

5. The computing model also reflects on the uses of array indexes. Generally indices i, j, and k are used in the first, second, and third dimension of an array represent dimension X , Y and Z in space respectively. Most time, the uses of the index variables are consistent throughout the program with few abnormalities.

Once the conceptual model of MM5 is understood, the difficulty of the optimized parallelization algorithms is reduced to discovering the index variables that represent the dimensions in space and the data exchange points after data and computing partition. An automated parallelization tool that incorporates the above knowledge has been built [Mitra *et al.*, 2000]. ParAgent, a domain-specific parallel compiler specifically targeted at finite difference methods, is able to parallelize the MM5 code and achieve satisfactory results [Kothari *et al.*, 2002].

The successes of the domain experts and the domain-knowledge aided tool result from the following key elements:

1. Conceptual model of a program

Domain experts know what problem the application is designed to solve, what elements are necessary in the problem solving (e.g., in MM5, the mathematical concepts like *nodes*, *grids*, *coordinates*, etc. have to be represented some way in the code), what strategies could be used to solve the problem, and what results the application is expected to produce. The conceptual model also includes the relationships between domain concepts. For example, the domain expert knows that an *element* in *finite element method* model must have coordinate attributes and the coordinates are related to some *nodes*.

2. Program concepts and program entity bindings

Domain concepts in a program have to be represented in a program by entities such as variables, objects, files, methods, controls, messages or events. A novice is not able to understand the underlying conceptual meanings of a program entity before he or she receives necessary training. An expert can bind the concepts to the program entities. The extent to which the binding can be done depends on the complexity of the program. Ideally, an expert should always be able to build such bindings, however, the process can be very tedious and time consuming and appropriately designed tools are necessary when the program is complex.

3. Conceptual strategies employed in the program

The strategies we mention here refer to algorithmic methods designed to solve the problem. For example, to solve multi-variable linear equations, programmers can use either the direct method of Gaussian elimination or an iterative method such as Jacobian solver. The domain

expert is able to identify the specific patterns associated with the methods. In a more general definition, we could also consider a strategy employed is a concept in the domain. We differentiate strategy and concept here to emphasize the differences between the dynamic properties of a strategy and the static properties of a concept in a conceptual model.

4. Conceptual strategy and program algorithm bindings

The domain expert also knows how a strategy is typically implemented in a program. For example, it is reasonable to expect a linear equation solver to be implemented in a 3-level nested loop and value reductions are expected in the computing.

5. Binding concept-level solution and transformation

Usually, it is easier for a domain expert to identify a solution at concept-level. In MM5, the conceptual solution is partitioning data and computing along spatial dimensions and discovering all communication points where data exchanges occur between neighboring grids. To implement concept-level solution, the programmer applies dataflow analysis across the loops where grids are computed and detecting specific index access patterns. Unlike other bottom-up general flow analyzer, the analyzer empowered by the domain knowledge already knows the optimal data partition. Minor variations in the code cannot change the analyzer's decision while the variations usually have great impact on the bottom-up methods.

A domain-specific tool such as ParAgent can accomplish this since the binding between concepts and program entities and the binding between strategies and algorithms are known. The tool just needs to translate the solution at concept-level into the transformation (parallelization) of the program.

Figure 1.1 shows the process that a domain expert uses to solve a domain-specific problem.

1.2 Program Analysis Background

There are many aspects of program analysis. In this paper, we only focus on the methods used for extracting relevant information from a program to facilitate understanding and transformation of existing programs. We categorize program analysis by the levels at which the code information is used, i.e., textual analysis, syntactic analysis, semantic analysis, and

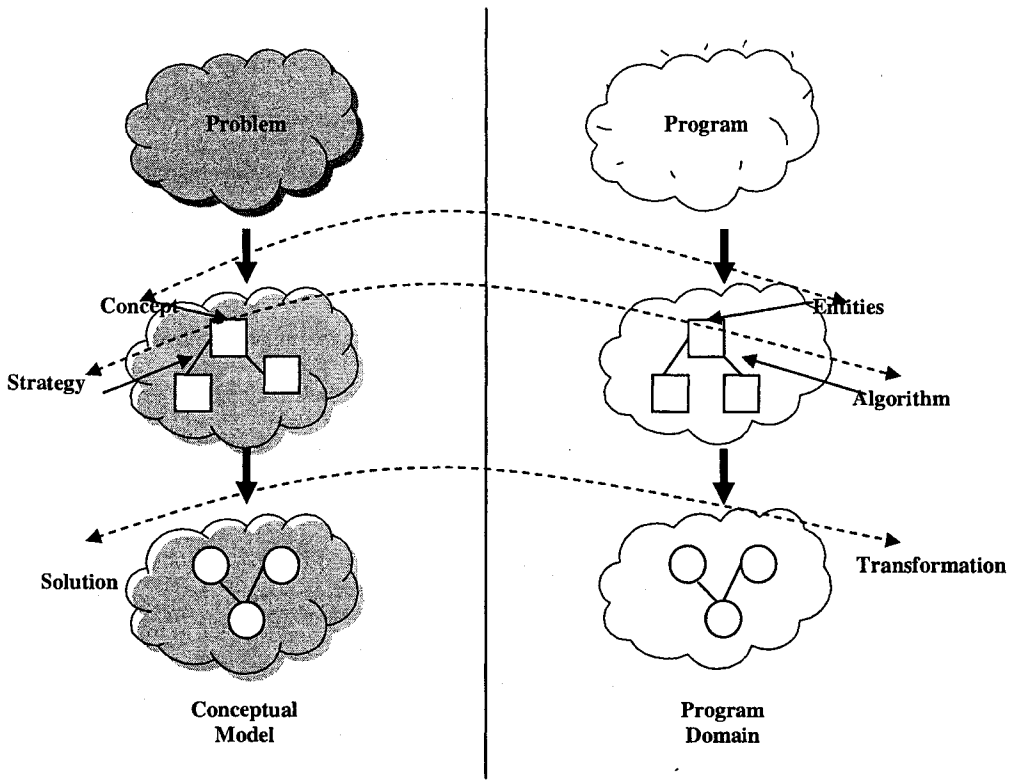

Figure 1.1 Conceptual model and program domain

conceptual analysis.

1.2.1 Textual Analysis

Textual analyses are the methods that extract information from source code using only textual information, such as string patterns. *grep* under UNIX and LINUX is an example of a tool based on textual analysis.

The textual analysis mechanism uses *regular expressions*. The advantage of textual analysis is that matching the patterns defined by regular expressions is simple. However, the disadvantages of textual analysis are obvious. The same string pattern could have different meanings under different contexts. String-pattern matching does not take structural information. Also, the textual analysis only collects pieces of information at local level. Unless some high-level or global methods are supplemented, textual analysis alone cannot extract a global view of a program.

In program analysis, pure textual analysis is rarely used alone. However, given certain preconditions, textual analysis can also be very successful. For example, if programmers were required to following certain naming conventions, the names of methods or variables (fields) themselves contain valuable semantic information. In industry, programming styles have received increasing attention, such as friendly naming conventions, source indentions, simple control flows, etc. Textual analysis tools can be quite useful if good naming conventions are strictly followed. For example in, jFTPd used *grep* to separate user's concerns such as debugging, GUI, logging, client feedback, etc. [Robillard and Murphy, 2000]

1.2.2 Syntactic Analysis

Syntactic analyses use syntactic properties of programs to reveal structural information. Abstract Syntax Tree (AST) and module dependence graphs are two main structures that syntactic analysis reveals. Popular tools *SCRUPLE* and *Rigi* are based on syntactic analysis.

1.2.2.1 SCRUPLE

SCRUPLE[Paul and Prakash, 1994], is a finite state machine-based searching tool. In SCRUPLE, the program patterns defined by regular expressions. A SCRUPLE user sends search requests in the form of a query written in a pattern language. SCRUPLE scans through the source code and find the piece of code that matches the given pattern. A simple example of pattern is "$v_1 = $v_1 + #". This pattern's semantics is "all statements that increment a variable's value".

SCRUPLE translates both AST and user-defined patterns into Code Pattern Automata (CPA). A CPA interpreter simulates these two automata at exact same steps. If both of automata can reach final state at the same time, a match occurs.

1.2.2.2 Rigi

Rigi [Müller *et al.*, 1993][Müller *et al.*, 1994][Tilley *et al.*, 1994][Tilley, 1995][Wong, *et al.*, 1995][Storey, *et al.* 1996] is a software comprehension, reorganization and documentation tool. Rigi works in the following steps. First, Rigi applies a front-end filter to filter out architectural (procedural) information. An interactive visualization tool displays the module dependence graph. Since the dependence graph for a large system is very complex, Rigi provides the user with interaction capabilities to abstract the system by *composing/collapsing* subsystems. Often times, the user's operations rely on the topologic properties of the graphs. For example, the *coherence* of graph components and the *thickness* of interface after subsystem composition are two significant criteria that Rigi uses.

In addition to the manual interactions through GUI, Rigi allows the user to automate the processing using scripting. Scripting is very useful when the manual process is prolonged or prone to errors. A user can design his or her own routines in Tcl/Tk script for common software reengineering activities such as customized graph layout, metrics, subsystem decomposition, etc.

Rigi is essentially a syntactic-based system, but it also supports some higher-level analysis methods. It makes use of naming conventions to encapsulate specific types of subcomponents.

While syntactic analysis has many advantages over textual analysis, it still has several disadvantages; specifically it cannot deal with the following:

Syntactic variations: Programmers can write different programs to achieve similar control-flow and data-flow. For example, a SWITCH statement can be replaced by IF-ELSE statement.

Implementation variation: A single function can be implemented in many ways. For example, a queue can be implemented using either an array or a linked list.

Non-contiguousness: A semantically meaningful operation may be scattered throughout a program.

Ambiguity: Same pattern of code could have different meanings under different context.

Misinterpretation of intentions: Syntactic structure cannot always reflect the designer's intentions. For example, in a program refactoring, a database programmer may not want to encapsulate data access with event logging if he/she wants to separate the data storage and system management even though event logging and data access share similar file access patterns.

1.2.3 Semantic Analysis

Semantic methods analyze programs using *annotated* AST. As opposed to syntactic methods which are not *flow-sensitive*, semantic analyses are flow-sensitive. The annotations on the AST carry control-flow and data-flow information. Many semantic methods are based on the idea of *program slice.*

Mark Weiser first proposed program slice as a debugging tool [Weiser, 1984]. Program slice is defined as a sequence of statements that influence the value of a variable at certain point. Typically, program slice is a reduced subset of a program so as to help programmers concentrate on a relevant and hopefully smaller part of the program. Program slice is used broadly in program comprehension and debugging. There are different types of program slices (backward, forward, and condition-based slice) based on the flow direction [Ning, *et al.*, 1994], or the analysis methods (dynamic and static program slice) [Tip, 1995].

The size of a program slice may be very large and the large size limits its usefulness when

the complex dependence relations are involved. To address this problem, Jackson and Rollings proposed a procedure-level pictorial program slice to get an abstract view of program slice [Jackson and Rollings, 1994]. In a procedural-level program slice diagram, nodes represent call sites and the user focuses on understanding the interaction between procedures. Deng and Kothari *et al.* have proposed an integrated and interactive environment for program slice, called Program Slice Browser (PSB) [Deng, *et al.*, 2000][Deng *et al.*, 2001]. PSB is designed to provide the user with multi-level abstraction to cope with the complexity of large program slices. PSB provides procedure-level and block-level abstractions. PSB also enables the user to prune the program slice diagram so that the user can concentrate on the most relevant part of program slice. PSB also provides cross-referencing capability so the user can navigate through the program slice examining use of variables, pattern of data access, etc.

In addition to program comprehension, program slicing can also be used for reconstruction. *Star Diagram* by Atkinson and Browdidge *et al.* is a data-oriented software-reconstructing tool. Program slicing of given data structure is the fundamental methodology [Atkinson and Griswold, 1996][Atkinson and Griswold, 1998][Bowdidge and Griswold, 1998][Atkinson, 1999]. In their approach, a user can select interested data structure (or data type) and Star Diagram will encapsulate all instances of the specified type into an Abstract Data Type (ADT) and transform the operations either directly or indirectly related to the ADT. A GUI is provided for the user to elide irrelevant information (represented as nodes in the Star Diagram). Star Diagram also matches and groups common expressions. The user can also define the interface with the help of the GUI. In the process of reconstruction, the GUI updates and redraws the diagram guided by the user's directive. To facilitate analysis of large programs, the Star diagram uses "on-demand" analysis; in other words, complex AST is loaded in memory only when demanded by the user.

Compared with syntactic analysis, semantic analysis is more flexible because it can tolerate syntactic and implementation variation, non-contiguousness, and ambiguity. However, semantic analysis still is not effective at capturing a user's intentions or concerns.

When we examine the standard development cycle, from initial requirements, architecture

design, module or interface design, to implementation, the user's intentions moved from the higher level to the lower level. With the concretization of the design in the form of a program, most of the user's initial intentions are lost [1].

Many times, a programmer tends to be interested in only a few aspects of the program. For example, in maintaining FEM program, a user may be interested in data representing *elements* or *nodes*. Capturing the user's intentions or concerns in the process of program comprehension is a key to a good comprehension tool. Aside from the lack of support for capturing the user's intentions and concerns, program slicing techniques themselves are currently not able to obtain *precise* program slice in presence of "unstructured" statements (GOTO) and "aliasing" (pointers and arrays) [Tip, 1995]. This also implies that program slicing must be used cautiously and must be assisted with other supplemental tools.

1.2.4 Conceptual Analysis

Conceptual analyses can process *abstract concepts*. An abstract concept can be a description of a series of events or steps of a function. For example, a Finite Element Method (FEM) model mainly and necessarily consists of four stages of computation: (1) calculation of the local element matrices, (2) assembly of the global matrix, (3) solution of a sparse linear system of equations, and (4) calculation of the physical field. An abstract concept can also be a conceptual meaning of a program entity, e.g., data or data structure, class, object, etc. For instance, in scientific computing, concept *sparse matrix* is storage of physical *elements* in space. In communication systems, concepts *signal* and *state* are essential for understanding the program.

Rich and Wills[Rich and Wills, 1990] designed a concept-based system, the *Recognizer*. The Recognizer is designed to discover *commonly* used program structures, called *cliches*, used in the program. To enable flexible analysis, instead of finding patterns of source code, the Recog-

[1] To make up the lost high level information, i.e., the programmer's intentions, many programming languages provide descriptive section. Many of them are for documentations purpose. For instance, the *javadoc* can extract special comments from Java source code which describe the definition of the interface. People also have extended existing programming languages to include domain-specific features. For example, programmers use directives to instruct the parallel compiler to partition data and computing in High Performance Fortran [HPFF, 1997]. Since we focus on analysis of *executable* code in this paper, we do not consider these programming language specific features in our methods.

nizer translates program source into a directed graph representation, called *plan*. Because of this translation, the Recognizer is insensitive to syntactic variation and non-contiguousness by abstracting away from the details of the expression of the code. The graph representation is "hierarchical" in the sense that the low-level graph components can be abstracted as a graph node at higher-level. Cliches are organized as cliche library. The user describes a plan or concept using a sequence of cliches. The Recognizer combines both top-down methodology and bottom-up methodology. It also identifies familiar parts of code and obtains higher-level program abstraction by the user's specification. The Recognizer also works in the reverse direction. When the Recognizer encounters unfamiliar code, it can infer the role of the unfamiliar code by user's specification and other recognized parts.

The *Reflexion* model by Murphy *et al.* [Rich and Wills, 1990][Paul and Prakash, 1994][Murphy et al., 1995] is another reverse engineering tool that helps engineers derive high-level models from source code. The Reflexion tool obtains a source model from the source code. The user defines a conceptual model representing the user's interests and also defines declarative mappings (based on naming convention) between the source model and the conceptual model. The Reflexion tool compares the conceptual model and source model and summarizes convergences and divergences of the two models. The Reflexion tool also provides user interaction capabilities so that the user can query relations between modules. The Reflexion tool also supports dynamic information collection by instrumenting code at locations: entry/exit of class methods, entry/exit of instance methods, the places of allocation /deallocation, and the start/stop of threads. Traces (events) are collected after the instrumented code is executed. A visualization tool replays the events and displays the mapping between source model and conceptual model.

Ning and Kozaczynski *el at.* designed and implemented a software reengineering system called *COBOL/SRE* targeted at COBOL language [Kozaczynski *et al.*, 1992][Kozaczynski and Ning, 1994][Ning *et al.*, 1994]. The main mission of COBOL/SRE is to recognize program concepts by program plans and aid in program maintenance and transformation. The kernel of COBOL/SRE is the specification of program concepts - *plans*. A plan consists of two parts

- components and constraints. The components specify the features of code pattern (e.g. an assignment of a variable). The constraints specify the relations between components (e.g. data dependencies). The recognition process is hierarchical; a combination of recognized sub-concepts is abstracted as a higher-level concept. When a concept is recognized, automated transformation is possible.

Cutillo, Lanubile and Vissagio proposed function recovery method based on program slicing [Cutillo *et al.*, 1993][Lanubile and Visaggio, 1993]. There are two phases in their method. First, in the data recovery phase, by analyzing the declarations of files, reports, and I/O maps, data are distinguished among "conceptual data", "control data", and "structure data". The user defines a reference model, which is used as a template to classify the roles of data. Secondly, in the function recovery phase, the user chooses source modules where information is obtained from external sources of data (read data from a file) and sink modules where data is written to external sources. Thirdly, the transform module, transforms the data from one form into another. The function recovery is accomplished by applying program slice on source module, sink module, and transform module. A program slice can be encapsulated into a single function.

Zhao defined a formal software architecture specification to describe structures and behaviors of an object-oriented software system [Zhao, 2000]. In the user-defined specification, a "component" specifies an object while a "connector" specifies relations between objects. A "computation" of a component specifies a sequence of actions that the component could take. A "port" of a component specifies the local protocol through which the component interacts with its environment. A "role" of a connector specifies the protocol that must be satisfied by any port that is attached to that connector. A "glue" of a connector specifies how the roles of a connector interacts with each other. A "configuration" specifies the mapping between the instances and the components, and the "attachments" between the components and the connectors. The architecture specification can be represented as an information flow graph. The tool extracts parts of the system given a subset of components, connectors and configurations. The method uses forward slicing and backward slicing on the information flow graph. Irrelevant components, connectors and configurations are removed if they don't appear in either of

the slices.

The purpose of the research work conducted by Gallagher and Lyle is trying to provide software maintainers a tool that breaks program into manageable components and assisting the maintainers in guaranteeing no "ripple effects" induced by modifying the components [Gallagher and Lyle, 1991]. The method presented in the paper heavily considers "output restricted" program slice. Under their criteria, output statements are considered "critical instructions" as defined in [Kennedy, 1981]. After the programmer applies slicing on output variables, a program is decomposed into small and manageable components. The main concern of this work is that modifying decomposed components does not affect their complement (rest of the program). There are rules that determine the circumstance under which a statement in the component can be removed. In the case that changing a statement in a component unavoidably affect its complement, the user is required to either include the affected parts in the component so that the "ripple effects" are localized in the component; or create a new local variable which copies values from the sensitive variable (referred to as dependent variable in the paper) so that manipulating the agent variable will not affect its complement.

Martino, Iannello, and Zima developed a program to automatically recognize algorithms within a specific domain [Martino *et al.*, 1997]. The algorithm recognition is based on a Hierarchical Program Dependence Graph (HPDG). An HPDG is a program dependence graph augmented with syntactic information (e.g. an expression tree). In addition, an HPDG is recursively defined; in another words sub-concepts compose higher-level (more abstract) concepts. The system requires that the user specify algorithmic concepts in the form of object-oriented style rules. The recognition is composed of three temporal stages: the front end interacts with Parallel Algorithm Pattern (PAP) Recognizer in which front end program representation is converted to PDG and is stored in Abstract Representation Database by means of Prolog facts. The PAP Recognizer infers the algorithmic pattern by applying the productions of rules (implemented by Prolog clauses) to PDG, and the results of concept parsing are the recognized concept instances. A GUI tool assists user in checking recognized algorithms by clicking on the graphical concept representation.

The existing concept-based methods have their limitations. For example, according to Quilici and Ning, et al. [Kozaczynski and Ning, 1994][Quilici, 1995], the plan, i.e., the specification of program concepts, does not scale well in practice. The reasons, as presented in [Quilici, 1995], are: The recognition is driven by pre-defined library; i.e., it can't recognize the patterns of code outside the library. Secondly, There is no mechanism to specify how to interleave constraints and combine component instances when recognizing plans.

Other concept-based methods also have similar limitations. In our opinions, there are two factors contributing to the difficulty of previous concept recognition.

- The conceptual roles of data have not been specified. Similar code could have very different semantics and the roles of data provide an important context. Context is actually constraint that prescribes whether or not the discovery is reasonable and meaningful.
- The existing concept recognition is rigid in the sense that it is not flexible enough to tolerate implementation variations. The primary reason of the rigidity is the formalized pattern description. The situation gets more serious when the roles of data are not clear.

1.3 Motivation

Brooks, a distinguished computer scientist, has made the following observation "intelligence amplifying systems can, at any given level of available systems technology, beat AI systems. That is, a machine and a mind can beat a mind-imitating machine working by itself." [Brooks, 1994]. As we understand, the situation Brooks described holds especially true in understanding programs. A successful program comprehension tools should follow domain experts' comprehension process and be ready to accept domain expert's guidance on program analysis.

Domain experts, as we described in the beginning of this chapter, are programmers who are familiar with the problem that the program intends to solve and the typical implementations. Due to the similarity between problems, programs performing similar functionalities show similarities in the implementations or source code patterns. Domain in our paper refers to

categorization of problems. Programs within same domain share common characteristics, or source patterns. Domain expert possesses both domain knowledge and programming knowledge of domain-specific application. The domain knowledge refers to the knowledge of the problem; the programming knowledge refers to the knowledge of the source code patterns reflecting the domain characteristics.

A program analysis tool provides of information to the user. The usefulness of a tool relies on *precision*, *efficiency*, *flexibility* and *adaptability*. The surveyed tools in section 1.2 have deficiencies in these four respects. We believe that our research has achieved progress in these areas.

1.3.1 Precision

Precision means that the information obtained by a tool accurately reflects the facts of a program. Precision is a difficult problem. Primarily, current technology cannot achieve precision all the time. For instance, in the presence of aliasing, point-to analysis is intractable [Li and Chen, 1991]. However, by applying a few restrictions, the point-to analysis is effective in practice. In the area of programming comprehension, if the available information is sufficient to extract the conceptual architecture of the program, the point-to analysis is usable. Thus, the program analysis can take other intelligent and loose methodologies if a problem currently has no perfect solution. It is possible for us to introduce strategies other than formal methods into the program analysis. It is also possible for us to effectively use methods with carefully defined restrictions on theoretically intractable problems.

Secondly, and more importantly, the measurement of precision is ultimately the user's decision. The surveyed tools paid attention to program analysis methods, but we feel a human's perception of a program should receive at least equivalent, if not more, attention.

Our research brings focus to conceptual roles of entities in programs. The entities are the syntactic elements in a program (variables, methods, parameters, syntactic blocks, objects etc.). The conceptual roles of the entities are the human-perceptual roles of the variables, especially the domain-specific roles. For example, a variable element is an array in a program;

therefore, the syntactic role of the variable element is array. In a FEM domain, variable *elements* is actually used for storing information about the domain concept of "elements". The domain concept is the variable's conceptual role.

The conceptual roles of entities in a program are beneficial to both aspects of precision. The conceptual roles of entities are undoubtedly more coincident with the domain experts' perception of a program. The domain expert should find the analysis results more meaningful to his/her comprehension. Also, the conceptual roles alleviate the burden of pursuing a *perfect* or *exhaustive* algorithm for accurate results. If a conceptual model is known, the analysis tool based on the conceptual model can distinguish the algorithmic patterns which are consistent or inconsistent with the model. The tool marks the inconsistencies and requires domain expert to clarify the distinctions. If the conceptual model is correctly reflecting the program domain, we can anticipate that the inconsistencies are comparatively small and reside in localized code segments. Domain experts are better than anyone else including automatic tools at understanding the localized abnormalities in code. In an interactive way, an intelligent tool can achieve higher precision.

1.3.2 Efficiency

Efficiency means that the information matches the user's intentions or interests so that he or she will not be distracted by useless or irrelevant information. Program comprehension is very subjective. The measurement of the effectiveness or usefulness of understanding ultimately is determined by the user. Given the fact that we have not been able to understand the mechanisms of human mind, manual guidance (or directive) from the user is necessary. In addition, it is important for the tool to implement a mechanism to filter out irrelevant information or abstract the information.

The criteria of the filter or abstraction should be consistent with the patterns that the user used in the manual process. Some of the criteria can be objective. For example, to composite certain nodes in module dependence graph based on the graph coherence is objective. The coherence of the topology reflects the inter-relationship between modules. It is natural to

assume that the more strongly modules are connected, the more closely they are related. Other important sources for extracting the user's intentions also exist. For instance, if a naming convention is known, the names themselves carry strong semantic or conceptual information.

In addition to the graphical structure and naming conventions which have been used for years, if a conceptual model is founded, we can develop conceptual abstractions or filters. Basically, a conceptual abstraction emphasizes the program entities (which the user is interested in) while hiding other insignificant information from the user. The definition of entities that the user is *interested* in is decided either by the user or by the program domain.

We consider the graphical structures indirect reflections of the user's interests. Conceptual roles and naming conventions are direct reflections of the user's interests.

1.3.3 Flexibility

If general tools could meet the needs for program comprehension, the flexibility would be out of the question. However, people have realized that it is difficult to make a tool meet a user's expectations without the user's guidance or feedback. Rule inference is the most widely used method for tools to utilize expert's knowledge. COBOL/SRE, HPDG, the Recognizer etc. store high-level knowledge in rules. However, the traditional rules and inferences that these programs use are very complex and rigid. In the presence of program variations, it is difficult to design a rule that matches the implementation exactly before the source pattern is known. The usability of rules largely depends on the flexibility of both the rules and the inference.

In our opinions, even for domain experts, the manual comprehension is nondeterministic and iterative. In the process of mapping the conceptual model to the existing code, the human expert also has to guess first and refine the results later when more related information is available or ambiguities are eliminated. We designed our program analysis in such a way that it *emulates* and *amplifies* the human comprehension process.

We also use rules to store the domain knowledge. The rule formats are designed to be simple and most of the inferences to be straightforward. A complex inference can be accomplished by two means in our system:

- The inference process is iterative. An inference can use the results obtained from previous iterations to get new concepts. This process is similar to the manual process in which a human expert refines what he has discovered to get a more accurate view of the program.

- A plug-in mechanism allows for system functions or customized functions which perform complex operations. The complex common inferences are packaged into libraries, so the rules are still simple in format. This mechanism is similar to an expert system where the expert knowledge is accessible to an inexperienced user though they are oblivious to the details.

1.3.4 Adaptability

Our rules are different from existing rule in that the major goal of the rules is to be *common* in a *specific* domain. Being in a *common* domain means that the rules are also applicable to other programs within the same domain. That the domain is *specific* means that the analysis results should match the specific concerns in the problem as possibly inapplicable to other problem domains.

The analysis framework does provide the capability for cross-domain analysis since both the customized functions and the rules can be replaced by rules that better describe the new domain. Adaptability is the major goal of the *simple rule* plus *plug-in* design.

In Figure 1.2, *domain* is a type of problems. Programs in same domain share similar conceptual model. *Flexibility* means program analysis is able to tolerate program variations. *Precision* means program analysis should reflect the facts of a program, e.g., the conceptual model. *Efficiency* requires the information obtained should be concise and reflect the user's major concerns or interests. *Adaptability* means a domain-specific solution is applicable to all programs within same domain and the domain-specific approach can adapt to other domains.

Figure 1.2 summarizes the motivations of using domain-specific approach against domain-independent approach for program comprehension.

Our research is geared toward improving the precision and concentration of program comprehension given the context of the problem domain. The system should be flexible enough to

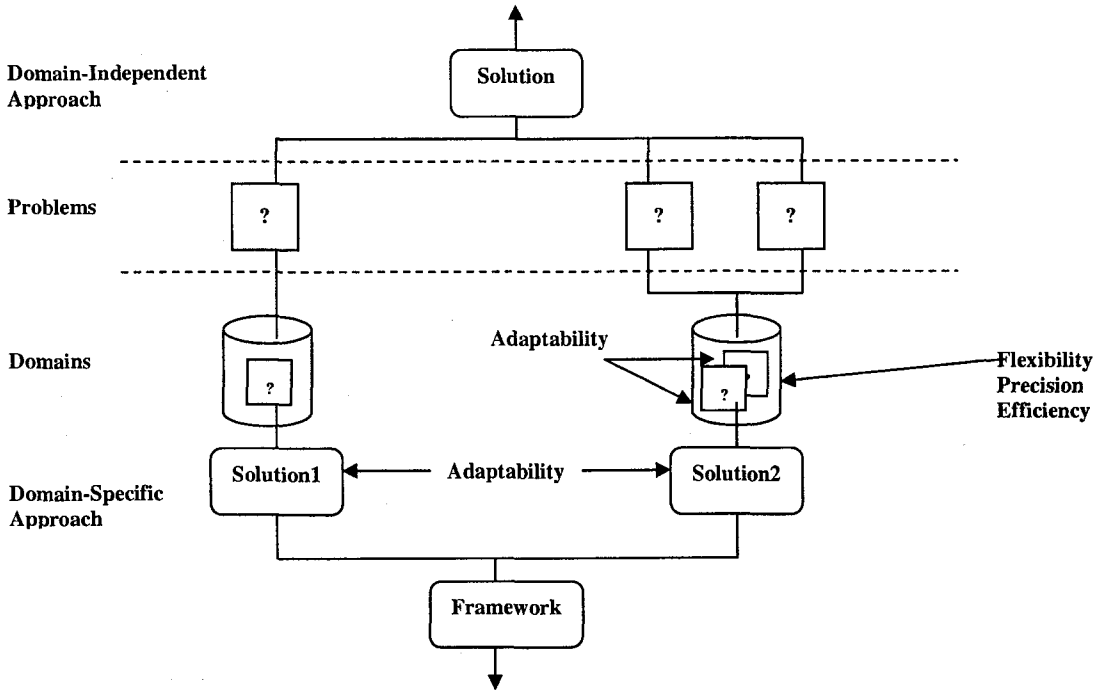

Figure 1.2 Domain-independent approach and domain-specific approach

tolerate program variations. The solution for one domain is applicable to all programs falling into the domain. In order to increase the usability of the domain-specific methods, the system should be able to easily adapt to other domains.

1.4 Outline

This dissertation is a detailed report of our research on domain-specific program analysis in past three and half years. Our research work is concretized by a program-understanding tool **SeeCORE** (Software Engineering Environment for **CO**mprehension and **R**eEngineering).

The dissertation is organized by the following chapters.

In Chapter 2, we formalize our program analysis as *Conceptual Program Analysis Model.* By looking into the manual process of how a domain expert understands a program, we formalize the manual understanding process which can be emulated and amplified by an automated tool. In this chapter, we define some important properties that such a domain-specific tool should

exhibit.

Chapter 3 introduces our software tool, SeeCORE that we developed based on domain-specific methodology. We discuss the different program analyses that we used in the tool. We also discuss how the results were saved and how the information storage was designed to allow for flexibility and inter-exchangeability. Program abstractions are very useful tools for program understanding. We also introduce several conceptual program abstractions. As an interactive tool, we show how SeeCORE interacts with the user and provides multiple views of program.

In Chapter 4, we give two case studies to demonstrate the effectiveness of SeeCORE. The two programs we analyzed using SeeCORE were purposely selected from different domains to show the adaptability of our tool.

Chapter 5 summarizes the contributions of our work.

This dissertation concludes with Chapter 6 in which we analyze the limitations of current work and point out some fields that we will continue to pursue.

CHAPTER 2. CONCEPTUAL ROLE ANALYSIS OF VARIABLE

This research is based on our observations of the program comprehension process practiced by domain experts. The foundation of this research is an analysis for recognizing conceptual roles of data or variables. Before we explain our data-centric method, it is necessary to differentiate variables and methods while recognizing the conceptual roles and explain the rationale for focusing on roles for variables.

2.1 Conceptual Role of Variables and Methods

We can categorize program artifacts into two types: data (variables) and methods. In object-oriented programming, variables and methods are integrated closely as classes or objects. Objects are like variables but objects also have dynamic behavior. In this research, we focus on procedural programs. We think that understanding procedural language is a prerequisite step towards understanding modern object-oriented programs. In his/her mind, programmer always assigns conceptual roles to significant variables or methods. In Chapter 1, we gave an example of how a domain expert understands a weather prediction code like MM5 and applies high-level domain knowledge for performing the difficult task of parallelization. A few key points bind the domain-specific concepts from weather prediction model to the actual code and that knowledge leads to a high-level parallelization strategy.

- *Binding between index variables and spatial dimensions.* The weather prediction model mainly involves computing on grids in space and the grid data is stored in arrays. The array dimensions correspond to spatial dimensions.

- *High-level data partitioning strategy for efficient parallelization.* While the gravitational

force causes irregular data exchange between grid cells in the vertical direction, the data exchanges follow a regular pattern within a horizontal slice of the grid. This suggests that efficient parallelization can be achieved if it is based on a decomposition of the grid across horizontal directions but not the vertical direction. An array decomposition scheme can be based on this observation plus the binding between array and spatial dimensions.

- *Binding between indexing of decomposed arrays and communication patterns for message passing.* The knowledge of the underlying method can be applied to identify the communication necessary in a parallel program. For example, communication occurs between neighboring blocks (formed as a result of the grid decomposition) at the boundaries of the blocks. The access pattern $i \pm a$ where i is an index variable and a is a constant less than 3 indicates that the communication must occur. The above observations show the significant role of bindings between the spatial dimensions (a domain-specific concept) and array indexes (a program artifact). Moreover, this binding is needed to apply the high-level parallelization strategy. This example shows the meaningfulness of recognizing conceptual roles of date in program comprehension and solving difficult reengineering problems.

The above observation is not only valid in specific domains like weather prediction model or scientific applications; we have observed similar program comprehension process while dealing with other types of software. For example, in operating systems where starting with bindings between program artifacts and important concepts in OS such as index node or file control block and semantic abstract data types (e.g., *struct iblk* and *struct fiblk* in Xinu), the users can extract information from a given OS code to better understand the implementation details of a specific system. The binding between domain concepts and actual variables (or abstract data type) is critical for the precision and efficiency of program analysis to facilitate comprehension.

There are also bindings between methods and their conceptual roles in a program. For example, a FEM code typically consists of the sequential steps: calculating local element matrices; assembling the global matrix; solving a linear system of equations calculating the physical field. To understand FEM code, we also need to match specific pieces of code with

the corresponding conceptual steps.

Recognizing the conceptual roles is important for both variables and methods. The recognition process is non-trivial in either case. We focus on variables because variables are relatively simpler. More importantly, recognition of variables' roles can be very useful for recognizing the roles for methods. The following key characteristics of variables enable us in developing a practical conceptual role analyzer for variables.

A *kernel* (*central*) variable has a unique conceptual role, for example, the variables in a FEM code for *elements*, *nodes*, *coordinates*, or *matrices*. This assumption is sound because in domain-specific application, a concept is very specific and unique.

Another key characteristic is that the conceptual role is easier to recognize in case of variable because it is not scattered in different parts of the code. The same is often not true when the conceptual role is for a method. There may be many parts, scattered throughout the code, which correspond to a single logical step. The fact that the conceptual role is not scattered in case of variables, lends a great deal of convenience and advantage in designing analysis because we have to deal with simpler program patterns and the inference rules are simpler.

Another important point is that, compared to methods, the conceptual role assignment of variable is less sensitive to specific implementations. We can use different algorithms to achieve the same functionality; and even for the same algorithm, there could be many different implementations. The implementation variations result in ambiguities that present new difficulties in dealing with the conceptual role assignment for methods. If we use conceptual roles of variables as anchors, the inferences from them are more likely to result in accurate results.

It is relatively easy to describe aspects of variables by program patterns as methods have more complex aspects. Actually the conceptual meanings of methods depend on the conceptual roles of variables used in those methods. It is easier to extend the conceptual analysis from variables to methods as opposed starting with the methods first. The simplicity of the description of conceptual roles of variables is important for creating a practical tool.

2.2 Overview of Our Conceptual Role Analysis

First, it is important to be aware that our notion of "concept" is different from the notion used in concept analysis work by Gregor Snelting [Snelting and Tip, 1994]. In Snelting's work, a "concept" basically is a syntactic or semantic property of program; e.g., the relationship of classes or objects. Snelting's concept analysis derives concept lattice from syntactic and semantic information of object-oriented code. The objective is to find imperfection in class design or optimize the design by analyzing class hierarchies. In our work, a concept is a domain-specific entity, which reflects specific attributes or features of the problem. Thus, our notion concept represents domain knowledge and not the class relationships.

Our analysis of conceptual roles of variables consists of the following steps:

1. Domain expert determines the conceptual model that defines given domain-specific program characteristics. Again let us use MM5 as an example. MM5 uses Finite Difference Method (FDM) for weather prediction thus the domain-specific knowledge about MM5 includes common program characteristics resulting from FDM code.

2. The domain knowledge leads to typical program patterns that can be used to discover data items that represent significant domain entities and thus helps us in assigning conceptual roles to actual variables in the program. Domain experts can formalize rules for the role assignment.

3. Our analyzer uses the formal rules from the last step to infer the conceptual roles of variables, i.e. automatically establish the bindings between conceptual roles and actual variables.

4. Domain expert's knowledge may not be precise also variations may be possible in implementing the conceptual roles in a given program. To tolerate such variations, our analysis is *relaxed* and the inference is *fuzzy*. The analysis is iterative so that the domain expert can add new rules to enable the analyzer to disambiguate inaccurate results. The iterative analysis enables the expert to write complex inference rules by a hierarchical composition of simple rules.

5. The analysis results are shown through an integrated and interactive visualization facility that we have designed. The tool provides multiple views of program in form of summary tables and interactive diagram, etc.

2.3 Characteristics of Our Method

Our conceptual analysis method has five characteristics: domain-specific approach, knowledge-centric notion of conceptual roles, a language for writing rules for role assignment, iterative process of rule assignment and extensible knowledge representation.

Domain-specific approach. Domain-specific knowledge is the basis for defining conceptual roles and the rules for inferring the roles. The conceptual roles and inference rules are only valid within a specific domain. Recall that we use the term domain to refer to a class of codes that share common concepts and certain program patterns based on those concepts. For example, the class of FEM codes is a domain. All FEM codes contain FEM concepts such as *elements*, *nodes*, *boundary condition*, *global matrix*, and *assembly* etc. There are domain-specific rules that use program patterns to infer conceptual roles.

Knowledge-centric conceptual roles. A conceptual role is the basis of our conceptual analysis method. We focus on the conceptual roles of data or variables. In the initial phase of program comprehension, identifying the key conceptual roles is our primary objective. There is a hierarchy of roles with more complex roles being defined using simpler roles.

Inference rules. The bindings between program patterns and variables' conceptual roles are represented in rules. A rule describes a recognizable code pattern and its correspondence to a conceptual role. Domain-specific knowledge about conceptual roles and the associated patterns is required to define these rules. The knowledge includes the problem knowledge (e.g., the knowledge of the typical logical steps in FEM code) and the program knowledge (e.g., the knowledge of the typical implementations of global matrix assembly).

Program pattern recognition and role inference by using rules are the major parts of our domain-specific method. In this paper, we use the term program pattern to refer to the textual,

syntactic, and semantic pattern of source code[1]. Our automated tool emulates experts in their use of program patterns. Program patterns can be defined as fine-grained or coarse-grained.

Coarse-grained program patters are patterns at procedural- or modular level. For example, module dependence is a coarse-grained pattern. Coarse-grained patterns tend to reflect the properties of whole program. Fine-grained program patterns are the patterns that reflect localized properties, including operation patterns (increment, decrement, reduction), operand/method patterns (index patterns, naming conventions), control patterns (IF-ELSE, FOR loop), and system specific patterns (libraries, programming language's specific semantics or syntax). Compared with coarse-grained patterns, the advantage of fine-grained program patterns is precision. The lower the information level is the less ambiguities the program patterns cause. Many automated tools use fine-grained patterns to achieve accuracy. Our pattern language allows users to define fine-grained patterns. Program pattern recognition can be categorized as rigid method or relaxed method depending on its toleration of the variation between expected pattern and actual implementation. Rigid pattern recognition has serious disadvantages. It is difficult to define and use, and automated tools built on rigid pattern recognition cannot tolerate implementation variations. In rigid methods, the popularly used pattern representations are explicitly related to code structure (AST).

The program analysis should be relaxed to allow partial pattern matching. Therefore, it is appropriate to define rules with simple program patterns with partial or relaxed characteristics. For example, a pattern for identifying a linear equation solver can be defined as "a piece of code consisting of a three-level nested loop with a reduction in the inner block." This relaxed pattern includes both the Gaussian Elimination method and the Jacob Iteration method, both popularly used in scientific computing. It is easy to make up a code which conforms to the pattern but violates the intention. Our experience tells us that compared with the whole program the probability of the false hits is low. Besides, iterative analysis can assist in removing false hits later when more relevant information is available for clarifying the confusion.

[1]Our definition does not rule out the possibility of using *design patterns* which primarily are used in object-oriented programming methodology to discover domain concepts. In that sense, design patterns can be considered part of program patterns.

Both relaxed pattern recognition and rigid pattern recognition may recognize multiple roles for one single data. Recall in section 2.1, one assumption of our method is that the conceptual role for each variable should be unique. Any ambiguous conceptual roles of data in the conceptual model should eventually be eliminated. This uniqueness assumption is not in conflict with the fact that a variable could have other roles such as semantic or syntactic roles. For example in FEM code, the variable representing *global assembly matrix* is not expected to represent *nodes* or *elements* at the same time. However, it is very possible that it contains other *semantic* roles, such as *left-hand side matrix* in a linear equation.

The simple program patterns suggested in our method are designed to provide a compromise between the precision of results and the easiness for use.

Iterative analysis. Fine-grained patterns are used to precisely describe program behaviors. The patterns are localized program features and usually a complete algorithm consists of many such features. Many earlier researches have defined hierarchical pattern languages composed of fine-grained program patterns and restrictions between the patterns to describe complex concepts. Hierarchical pattern languages are difficult to use because of the complexity. Our solution is iterative program analysis. Complex inference still can be hierarchically built on simple rules using simple patterns. Functionally iterative program analysis achieves equivalent pattern recognition capability as complex hierarchical pattern recognition. The outstanding difference between our iterative analysis and hierarchical pattern recognition is that we in fact break down complex hierarchical patterns into smaller subcomponents (in form of simple patterns/rules) and do not require the constraints between the subcomponents (constraints are expressed in additional separate rules). Figure 2.1 is an example illustrating the differences.

The more important point is that since we do not anticipate the analysis being able to decisively recover all conceptual roles at the very beginning, the domain expert can add new rules to guide the tool to find more accurate or meaningful results through an iterative process. This is called conceptual role refining.

The conceptual roles of variables can be refined in the iterative process. Most of existing

	Hierarchical Pattern Recognition	Iterative Pattern Recognition

Rules	Pattern p {
	Subcomponent A: <pattern a>
	Subcomponent B: <pattern b>
	}
		Pattern p {
		subcomponent A, B
		}
		Pattern A{
		: <pattern a>
		}
		Pattern B{
		: <pattern b>
		}

Recognition	: If <pattern a> & <pattern b> -> pattern p	If <pattern a> -> pattern A If <pattern b> -> pattern B If A&B -> pattern p

Figure 2.1 Hierarchical pattern recognition vs. iterative pattern recognition

researchers do not take any potential changes of inferences into account. We emphasize that the roles of variables are not fixed when they are annotated. The refining of roles of variables has two perspectives:

1. A variable in program could be annotated with more conceptual roles which are supposed to be closer to the conceptual roles in the conceptual model. Basically, the system is enriching itself during iterative analysis.

2. A previously existing role of a variable could be ruled over by more recent inferences. The rule base is not guaranteed to be self-consistent. By default, most recent inferences have higher priority. This conforms to the typical human comprehension.

Iterative analysis allows the user to refine his/her rule definitions. Iterative analysis also provides the opportunity to enable the user to interactively control the analysis. Combined with relaxed and simple pattern definition, the iterative analysis produces results in stages so as to diminish the danger of losing useful partial recognitions.

Extensible information representation. For iterative program analysis, User also could add new rules to discover more conceptual roles of variables at any time. The information representation should be extensible to adopt new discoveries. The information store (results) is expected to absorb new discoveries and grow. Figure 2.2 illustrate such scenarios.

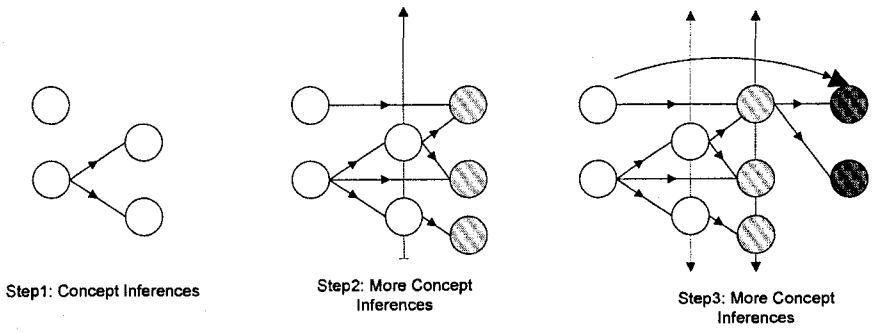

Figure 2.2 Iterative analysis and extensible information store

The information store also should be exchangeable because there is no mandatory order for adding rules or for inferences. The user can add any rule at any time that he or she believes would be useful for enhancing the precision of the analysis. We cannot anticipate the analysis path to reach certain conceptual role discovery. We could manage analysis result using a relational database. The inferred concepts can be saved as fields associated with records (variables). Though a relational database satisfies extensibility and exchangeability, it does not support structural information representation which is required for program analysis. We select XML as the intermediate representation. XML is structural information representation yet satisfies the needs for extensibility and exchangeability. We organize our program intermediate representation in multiple XML files (like tables in a database) and keep track of references among the files. We developed a series of XML query utilities which can retrieve structural information.

Rules for conceptual role analysis are domain-specific, but the language for formulating the rules is domain-independent. By defining different sets of rules, different domains can be handled.

Though concept-level analysis methods have been addressed by earlier researchers (see

section 1.2.4), we have not yet seen a clear recognition of conceptual roles of variables or program entities for program analysis. The major contributions of this research is recognizing the importance of concepts behind program entities and developing a practical conceptual role analysis tool for program comprehension.

CHAPTER 3. SEECORE - AN INFRASTRUCTURE FOR CONCEPTUAL PROGRAM ANALYSIS

In chapter 2, based on our observation of how domain experts comprehend and reengineer domain-specific code, we establish the Conceptual Analysis Model. The objective of our research is to develop an automated domain-specific tool to prove that the conceptual analysis model is an effective approach for domain-specific program comprehension and reengineering.

The comprehensive tool that we have developed is named **SeeCORE**, which stands for **S**oftware **E**ngineering **E**nvironment for **CO**mprehension and **Re**Engineering. In addition to our kernel innovation - conceptual analysis, we also integrate program visualization into our infrastructure.

3.1 Architecture

SeeCORE is a research project conducted at the Software Engineering Laboratory in the Department of Electrical and Computer Engineering at Iowa State University. This software is a framework for building a series of domain-specific tools to help programmers understand and reengineer legacy systems. The design of SeeCORE strictly follows the guidelines of the conceptual analysis model.

In SeeCORE, program analysis is divided into two levels. Lower-level analysis, *program-centric analysis*, is independent of program domain. The program-centric analyzer extracts syntactic or semantic features from Abstract Syntax Tree (AST). The analysis is inter-procedural and the results are stored in XML repository. The higher-level analysis, *knowledge-centric* or *conceptual analysis*, is dependent on the specific program domain. In SeeCORE, we provide a rule-based knowledge-centric analysis, in which domain expert can define customized rules to

express his interests or concerns.

The XML repository is the kernel information store of the tool. We use XML because it is an extensible and exchangeable information representation format. Extensibility and exchangeability are very important to our tool because our analysis is iterative. Analysis results are gradually refined.

Figure 3.1 illustrates the architecture of SeeCORE. The detailed workflow of SeeCORE is described by following steps.

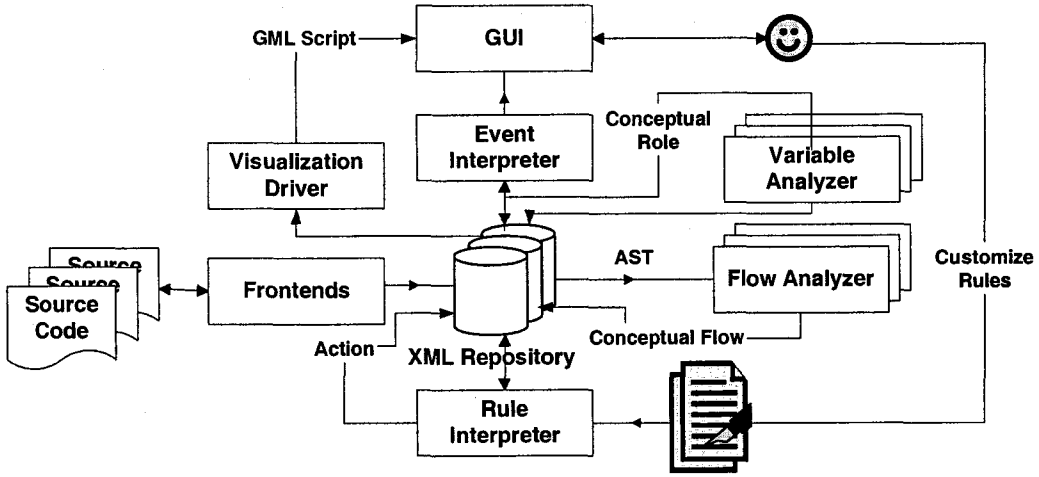

Figure 3.1 SeeCORE architecture

The EDG compiler[EDG, 2002] parses the source code (C or Fortran code). The binary output of the EDG frontends is unreadable for human being and very programming language specific. We developed a conversion tool *EDG-to-XML* utility, which converts the binary output to a XML common intermediate representation.

Variable analyzers extract the syntactic or semantic features associated with uses of variables. The analysis is inter-procedural and provides a summary of the uses of variables across the program.

The rule interpreter loads rules defined either by system or by users. The rules are usually designed specifically for the program domain. Following the instructions of the rules, the interpreter recognizes semantic or conceptual patterns stored in the XML repository and annotates

conceptual roles. These rules can be used to discover critical program variables. These rules can also be used to identify the conceptual relationships in the problem domain. The inferences are translated into SQL-style queries.

Our conceptual flow analyzers capture the behaviors of program with respect to the *mission-critical* variables. The *conceptual program skeleton* is a program abstraction that focuses on the definitions of critical variables and the "transition" between the definitions. The conceptual program skeleton helps domain experts determine the main computational steps of program.

Visualization drivers convert the analysis results to GML-like scripts. GML (Graph Manipulation Language) is employed to decouple the visualization components and the internal analyzers to increase the flexibility.

In addition to the architecture of SeeCORE, we have also developed auxiliary components that are important to the usability of the tool, such as program slice browser, program slice abstraction, etc. Limited by the page size we cannot show all of them in one architecture diagram. We will respectively supplement subsidiary diagrams in dedicated sections.

In order to explain technical details, we start to use a small program *mydemo.c* as an example. There is no specific background behind this code except for demonstration purpose only. The programming style is similar to scientific application. The complete source code can be found in Appendix A.

3.2 Analysis Frontends

Instead of writing our own parsers, we decided to leverage existing parsers to obtain AST tree. EDG frontends developed by Edison Design Group provides common intermediate representations (mainly AST) for C/C++ and Fortran code. The intermediate representations are dumped into binary files. We developed an EDG-to-XML utility that converts the binary representations to XML representations. The reasons we choose XML as the representation format are:

Enable extensible information storage. XML is an appropriate platform for our iterative analysis and conceptual role refining. To ease the burden of the analyzers on the backend,

the AST obtained from the frontends should be represented in a compatible format, i.e. XML.

Provide concise comprehension oriented information. EDG frontends basically are compilers. The original intention of the frontends was to enable optimized compilation. Many properties or attributes from original AST are not directly related to program understanding, e.g., the size of storage unit. The flexibility of XML representation enables us to easily wipe out useless information from high-level program analysis's point of view because most of the redundant information is represented as XML *attributes* in *elements*. Deleting redundant information is not only for saving space since XML descriptive tags already take substantive space. We prefer simple AST mainly because it is easier to maintain and verify.

Use existing tools. The XML community has developed many useful and accessible tools. Leveraging existing technology to maximize the outcome is one objective that we try to attain in implementing our tool. For instance, the EDG-to-XML utility uses XSLT technology (Extensible Stylesheet Language Transformation) [XSLT, 2002]. We wrote an XML tree manipulation script in XSLT to remove redundant information and reorganize AST structure.

Figure 3.2 illustrates the workflow of the EDG frontend and the EDG-to-XML utility.

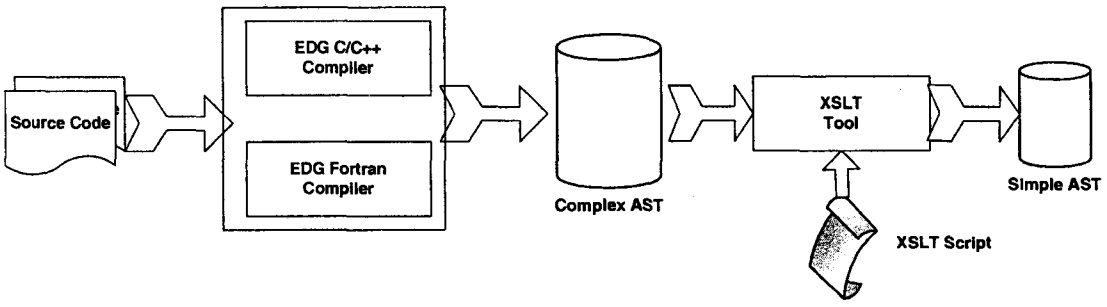

Figure 3.2 Frontends and EDG-to-XML utility

3.3 Semantic Analysis

Variable analysis is fundamental to our system. The variable analyzer's input is a language-independent AST from the frontends and the EDG-to-XML utility. The output is the semantic summary of variables. There are two major components for variable analysis. The role analyzer

extracts elementary semantic roles of data and the relation analyzer extracts dependences between data. We name this process semantic analysis because the analyzers recover the roles of variables by primarily exploiting the syntactical information, e.g., the control structure, the index position, etc. In addition, the semantic analyzers use semantic information that is programming language dependent. For example, the analyzers extract semantic roles *file*, *input*, and *output* by using the semantics of library functions such as $fopen$, $fread$, $fwrite$, and $fscanf$ etc.

Figure 3.3 illustrates the architecture of the semantic analysis.

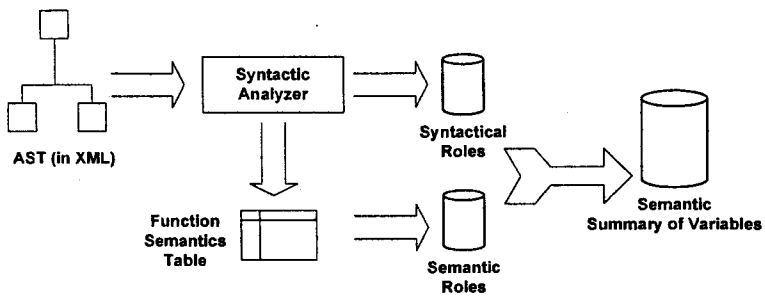

Figure 3.3 Program semantic analysis

3.4 Variable Role Analysis

Role analyzer summarizes the syntactic and semantic roles of variables in program. A summary of roles of a variable is represented as variable semantic feature vectors. Each semantic feature vector denotes the semantics of a single use of a variable.

We represent the semantics of a use in a feature vector. The feature vector is implemented in a data structure called *ProgramFeature*. A ProgramFeature consists of a vector of semantics as summarized in Table 3.1:

As shown in Figure 3.3, the syntactical analyzer traverses the AST and annotates the semantics of variable in feature vectors. If a variable is used in a function, the analyzer looks up a function semantics table and retrieves its semantic roles.

In the example of Table 3.2, the use of array X in *mydemo.c* has three semantic feature

Table 3.1 Major semantic roles of a variable

Semantic Role	Description	Semantic Role	Description
File	Used in a file operation.	*Loopcontrol*	Used in a loop control head.
Input	Used in an input statement.	*Loopbody*	Used in a loop.
Output	Used in an output statement.	*Subscript*	Used in as a subscript of an array.
Read	Read in an expression.	*Write*	Used to save the result of an expression.
Return	Used to carry return value in a function.	*Ifcontrol*	Used in a branch control.
Param	Used as a parameter.	*Expr*	Used in an expression

vectors. Each one of them is associated with a single use.

Table 3.2 Feature vector of variable X in *mydemo.c*

<Variable Name="X" Type="2" passByValue="true">

<ProgramFeature block="0" callLink="0" curNodeID="47" expr="1" file="0" finalNodeID="47" ifcontrol="0" input="0" loopbody="1" loopcontrol="0" output="0" param="0" read="0" return="0" subscript="0" write="1" />

<ProgramFeature block="0" callLink="0" curNodeID="93" expr="1" file="0" finalNodeID="93" ifcontrol="0" input="0" loopbody="1" loopcontrol="0" output="0" param="0" read="1" return="0" subscript="1" write="0" />

<ProgramFeature block="0" callLink="0" curNodeID="109" expr="1" file="0" finalNodeID="109" ifcontrol="0" input="0" loopbody="1" loopcontrol="0" output="0" param="0" read="1" return="0" subscript="1" write="0" />

<Variable/>

The first feature vector corresponds to statement "$X[i] = i$"; the second one corresponds to "$value[X[i]][Y[i]] = scale * 3$"; and the third feature vector is the semantic information about statement "$value[X[i]][Y[i]] = scale * 2$". In addition to the semantic or syntactic annotation listed in Table 3.1, there are a few other annotations, such as *callLink* and *finalNodeID*. These annotations are not directly related to variable's semantics; they are used for maintaining procedure call context because our program analysis is inter-procedural analysis.

The semantic role analysis is also accumulative. A variable can and usually does hold more than one single semantic role. Each single role reflects one perspective of a variable use.

Some of semantic roles hint that the variable is related to other variables. For example, role *file* implies that there should be another variable is used as file pointer in a file operation statement. This type of information is very important because the close relationships between variables are useful for inferring the conceptual roles of variable.

Variable relationship analysis is local dependence analysis driven by the semantic feature vector. For each role of the feature vector, there is a corresponding attribute that stores the relevant variables. For example, if role File is annotated, the concerned file pointer is saved as the value of a new attribute *filep*. If a variable is used in a loop control statement, i.e., one use of the variable is annotated *loopcontrol*, the upper bound and lower bound of the iteration as well as the control variable are also saved along with *loopcontrol*. For scientific computing, this is valuable for inferring the relationship between an array that represent points in space and another array that represents the coordinates.

Local dependence relation analysis is not sufficient because domain experts are usually more interested in critical variables active in the whole program. Temporary variables used in functions may block the chance to discover the relationship between critical variables simply because they are not directly related. We use program slicing to make up the deficiency of local dependence analysis. However, it is worth pointing out that the advantage of local dependence analysis taking the syntactic structure into consideration is not obtainable in program slicing.

3.5 Conceptual Analysis

When a programmer writes a program, he or she assigns concepts to the data. The concepts are domain-specific; the concepts are knowledge centric, i.e., *node* is a concept in FEM method. Similar syntactic or semantic patterns can have different interpretations under different domain contexts. There is a mapping or binding between the domain concepts and the program patterns. For instance, a branch control statement that contains a field of a widely used structure can mean a state transition in networking program, or it can mean convergence

checking in scientific computing. We believe that a program comprehension tool should take domain knowledge into account during program analysis.

Domain knowledge is very specialized. A programmer without the proper background has difficulty understanding a domain-specific program. A family of applications in the same domain shares many common characteristics or patterns. Based on the knowledge of such patterns, a domain expert can make good judgments in comprehension process. Our tool incorporates the expert's domain knowledge to help the programmer to understand a program in a more relevant way.

3.5.1 Overall Analysis

A conceptual analysis is composed of three major parts.

1. **Semantics Database.** The XML information storage is organized as a database. Semantic inferences or queries are applied on the semantic annotations in the store.

2. **Rules.** Rule defines the binding between domain-specific concept and program pattern. Our tool supports pluggable domain-specific rules that can be customized. The rules are written in simple format. The grammars of rules are shown as Figure 3.4.

3. **Interpreter.** The interpreter loads the rules and applies them to records in the semantic database. If a semantic/conceptual pattern matching occurs, the interpreter takes the action on the rule's right side. In the grammar, *term* is usually a semantic annotation, e.g. *file* or *input*. The extension of term - *function* usually represents complex semantic patterns or actions. The tool provides pre-defined functions, and user can also write customized functions. One of the design concerns in our system is to keep the rules as simple as possible. As we showed in Chapter 2, the complexity of rule format has a great impact on the usability of tool. Instead of introducing complex elements in our pattern language, we provide a plug-in mechanism to enable powerful pattern recognition yet simple rule format.

rule: terms '->' action

terms: terms logic_op term| term | '!' term

action: term | function '(' parameters ')'

term: annotation | function '(' parameters ')' | annotation '=' constant

logic_op: '|' | '&' | '>' | '<' | '>=' | '<=' | '<>'

annotation: string | constant

function: string

parameters: parameters ',' term | term

*string: ['a'-'z'|'A'-'Z'] ['a'-'z'|'A'-'Z'|'0'-'9']**

constant:['0'-'9']+ | '"' ['a'-'z'|'A'-'Z'|'0'-'9'] '"'*

Figure 3.4 Rule grammar

3.5.2 Plug-in Mechanism

Term is considered an elementary element in rule. *Function* is far more complicated. Function usually contains a sequence of complex operations, such as AST traversal, dependence tracking, etc. *Function* is typically used for these purposes:

1. **Simplify the rule format.** Writing complex predicatives such as *ALL* and *ANY* as functions is easy. Many AI languages have similar expressions for predicatives.

2. **Hide complex operations from end users.** The information storage is organized as a database to provide a simple and uniform interface for the end users. The end users are not expected to write complex operations often. The tool provides frequently used patterns as a system library with the hope that the end users can easily assemble those patterns to perform more specific complex operations.

3. **Hide internal representation from end users.** AST is very complex and program-oriented. End users prefer to concentrate on conceptual level information and look into specific source code only when necessary. However, it is also often times unavoidable to use AST information to aid in inference. One of the uses of function is to provide the

end user a simple interface that he or she can use to query AST without knowing AST's specific structure in the XML repository.

In the plug-in implementation, each function needs to register at the startup time. When the rule interpreter recognizes a function, it marshals the parameters[1], calls the function, and expects integer- type return value(s). The interpretation works in a recursive style because a parameter could be another term. The uniform interfaces between the functions and the interpreter and the parameter marshal and return value(s) do not increase the complexity of the interpreter at all.

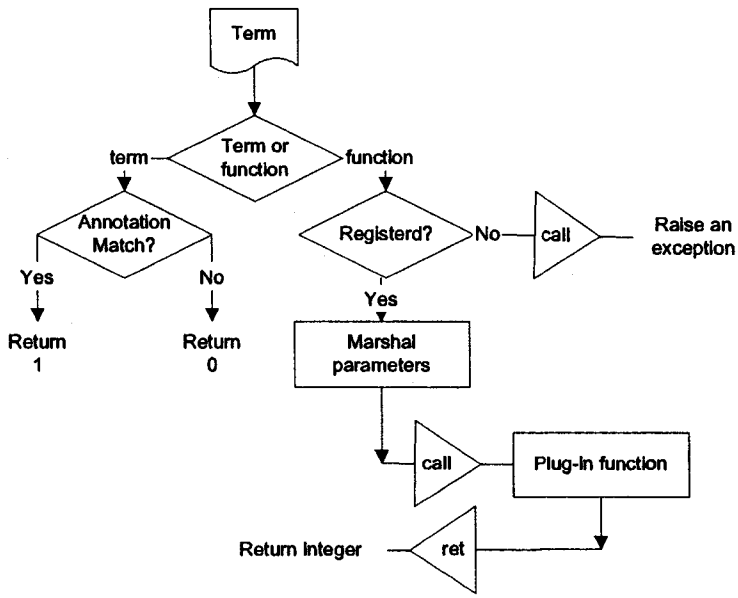

Figure 3.5 Plug-in mechanism

Executing a rule is similar to executing a SQL statement. Here, the database is the XML repository, SELECT is the pattern matching, and UPDATE is the action.

The following two simple examples illustrate how the rule's similarities and how they are executed in the system.

[1]We can compare our parameter marshalling to RPC (Remote Procedure Call) marshalling. In RPC, the parameters needed by remote procedures are uniformly packaged and sent over to the server via network. Our rule interpreter parses the rules, assembles the parameters and packaged them into a linear data structure (vector or list) and sends to the plug-in functions. The rule interpreter is not responsible for the meanings of those parameters.

Example 1: *subscript* → *dimension*

The rule is applied to each single use of variables. Once the rule finds a record annotated with "subscript", it adds one more concept annotation "dimension" to the record. This is a simple example of aliasing. It is preferable to use domain-specific annotations. For example, compared with term "subscript", "dimension" may be a better annotation because it shows the specific conceptual meaning of an index variable in scientific application.

We have shown the actual format as used in our system. To simplify things for the reader, we also give an equivalent SQL-like statement.

Equivalent SQL: *UPDATE xml_repository.alluses SET dimension='1' WHERE subscript='1'*

Example 2: *computing < avg("computing")* → *redundant*

Determining variable redundancy is far more complicated. This rule is a part of a group that helps determine if a variable is redundant. The rule tells the system to evaluate the number of times that a variable is used for arithmetic calculation. If the number is below average, the variable is considered a *redundant* variable. In this example, "*computing*" is a derived annotation because elementary analyzers do not directly extract it. The *avg* is a *function* that calculates the average number of variable annotations with the given attribute.

Equivalent SQL: *UPDATE xml_repository.allvariables SET redundant='1' WHERE computing < SELECT avg("computing") FROM xml_repository.allvariables*

3.5.3 Analysis Specifics

There are three types of program analysis in our tool.

3.5.3.1 Type I: Itemized Analysis

Itemized analysis basically is local pattern recognition where the patterns are the semantic annotations obtained from fundamental data analyzers. The itemized analysis does not consider the control flow. For example, above example "*subscript* → *dimension*" is a rule belonging to this category.

3.5.3.2 Type II: Synthetic Analysis

In synthetic analysis, the analyzer summarizes all uses of a variable in a program and applies synthetic rules to the summary instead of individual semantic features.

Above, Example 2 illustrates synthetic analysis. The analyzer summarizes all uses of a variable and counts how many have been used for arithmetic calculations. The synthetic analysis does not consider dependence between variables. Synthetic analysis provides the user a global view of variable uses.

3.5.3.3 Type III: Crosscutting Analysis

Crosscutting analysis is mainly used for analyzing control-dependence and data-dependence between variables. Crosscutting analysis can be applied to either a single variable (from itemized analysis) use or a summary of all variable uses (from synthetic analysis). The objective of crosscutting analysis is to infer conceptual or semantic roles from other relevant variables.

In scientific computing, a branch statement is most frequently used to either check some bounds or compare a variable against a threshold value. We want to distinguish these two roles using the rule presented in the following Example 4. The meaning of the rule is actually "if a variable appears in a branch statement and if ALL relevant variables in the same branch statement have been annotated boundary, then the variable is considered a bound"; and naturally, the branch statement is a bound checking.

Example 4. $ifcontrol\&all(marks(rels(\text{"}ifcontrol\text{"}), \text{"}boundary\text{"})) \rightarrow boundary\&!control$

Equivalent SQL:

1. *SELECT ifvars, name FROM xml_repository.allvaribales WHERE ifcontrol > '0' INTO rels_ifcontrol*

2. *UPDATE xml_repository.allvariables x SET boundary='1', control='0' WHERE NOT (ANY(SELECT boundary FROM rels_ifcontrol r WHERE r.name=x.name) = '0')*

The function *rels* is a crosscutting analysis. The function *rels("ifcontrol")* returns relevant variables in a branch statement. The function *marks* checks if returned variables have been annotated "boundary". If true, then the evaluation is set to 1. The predicative *all*, also a

function, determines if all evaluations returned from function *marks* are 1.

Table 3.3 shows in *mydemo.c* variable X's conceptual annotations after applying the rules.

Table 3.3 The conceptual annotations of X after applying the rules

<Variable Name="X" Type="2" array="2" block="0" computing="6" critical="2" dimension="4" expr="3" file="0" ifcontrol="0" input="0" loop1="6" loopbody="3" loopcontrol="0" output="0" param="0" passByValue="true"peripheral="0" read="2" redundant="0" return="0" significant="0" subscript="2" write="1">
<ProgramFeature block="0" callLink="0" computing="2" curNodeID="47" expr="1" file="0" finalNodeID="47" ifcontrol="0" input="0" loop1="2" loopbody="1" loopcontrol="0" output="0" param="0" read="0" return="0" subscript="0" write="1">
</ProgramFeature>
<ProgramFeature block="0" callLink="0" computing="2" curNodeID="93" dimension="2" expr="1" file="0" finalNodeID="93" ifcontrol="0" input="0" loop1="2" loopbody="1" loopcontrol="0" output="0" param="0" read="1" return="0" subscript="1" write="0">
</ProgramFeature>
<ProgramFeature block="0" callLink="0" computing="2" curNodeID="109" dimension="2" expr="1" file="0" finalNodeID="109" ifcontrol="0" input="0" loop1="2" loopbody="1" loopcontrol="0" output="0" param="0" read="1" return="0" subscript="1" write="0">
</ProgramFeature>
</Variable>

In Table 3.3, *critical*, *dimension* and *loop1* are the syntactic or conceptual roles of X that the tool discovered. Note that only those attributes with non-zero value are valid conceptual roles. For example, *peripheral* is a new attribute added to the XML file; however, its value is zero and that means that the annotation is not justified.

Conceptually, the three analysis methods above reflect three perspectives of manual comprehension: the domain expert recognizes distinct local patterns, such as computing patterns, array access patterns, etc. (itemized analysis); the domain expert collects global information to evaluate the roles or significance of a variable (synthetic analysis); and the domain expert infers conceptual roles based on the relationship between known variables and unknown variables (cross-cutting analysis).

The main work of our research is recovering the conceptual roles from data in programs. Our definition of concept is different from "concept analysis" in Gregor Snelting's work [Snelting

and Tip, 1994]. Snelting's concept is a maximal set of objects sharing common attributes where each object has all attributes. There, the concept analysis is basically structure analysis. In our paper, a concept is a domain-specific element, which reflects specific attributes or features of the problem. Thus, our method is different from the series of clustering techniques following the concept analysis.

Our definition of concept is close to the term used in DESIRE [Biggerstaff *et al.*, 1994]. The recovering processes are also similar. The main clues that DESIRE used to recover concept are naming convention, patterns of module dependencies and assistance from DM-TAO. The difference between our tool and DESIRE is that our program analysis is iterative analysis based on semantic/conceptual annotations and domain-specific rules. Our concept recovery may need syntactic patterns at expression level.

Many program analysis tools employ syntactic or semantic analysis (called "program structure (PS) level" and "function/file/data (F/F/D) level" in paper [Woods, *et al.*, 1999]). SNiFF+ [SNiFF, 2002] extracts semantic information like "function calls function" by traversing and manipulating its parser-oriented structure; Rigi [Wong, 1998] analyzes module dependency topology (e.g., dependency and coherence). It supports a built-in scripting language that the user can use to automate abstraction according to dependency graph properties; ASF+SDF meta-environment [Brand, *et al.*, 1998][Brand, *et al.*, 1997][Klint, 1993] allows the user to implement queries over AST, which actually define the structural semantics. In Dali [Kazman and Carri' ere, 1999], the user can write queries in SQL to understand the language semantics and structure semantics. As its extension, CORUMII [Woods, *et al.*, 1999] added dynamic analysis. The query languages used in Reflexion [Murphy and Notkin, 1997] and Richner *et. al.*'s work [Richner and Ducasse, 1999] are mainly used for capturing hierarchical relationships, interactions between classes, and instantiations of classes.

Our work is different from the above works in following ways:

1. Since we are recovering conceptual roles behind data, our query language deals with the relations between data, including index relation, value-passing relation, control relation, and the relation between data and libraries.

2. Our queries manipulate domain-specific conceptual annotations in addition to language, structure, and function information.

3. Our analysis is iterative; therefore, the analyzers gradually enrich and refine the results.

3.6 XML Information Storage

The information storage of our system is essentially an XML database. All analysis results are saved in XML files. The primary reason we chose XML as the platform for our information store is that XML format is extensible and exchangeable. Since our analysis is iterative, the results are expected to be refined either by manual or by rules. The refining could be update of existing concepts or addition of new concepts. The second reason that we chose XML format is that SeeCORE consists of many components, and each component performs independent function. XML format information storage expedites information exchange between components. For example, data flow component needs information of the code structure and read/write information. Instead of being tightly coupled with the control component and read/write component, the dataflow component code is totally separated from the other components. The data flow component simply takes the output of read/write component and control component (CMP.XML file as shown in below Table 3.4) and retrieves only the information useful for getting the dataflow (save in CMP.FLOW.XML file).

Table 3.4 shows the output of each component. Since we mainly process C code and Fortran code, we define information store at function level. The namespace of the information store for each function consists of <source>.<function>.<type>.XML, where source is the name of the source code, function is the function name, and type is the type of the information, i.e., RW, CMP, VAR, GML, and FLOW.

Each XML file can be considered a table in the database. The correlation between the tables is illustrated in Figure 3.6. Note that the rules can be applied iteratively. Thus the conceptual roles of variables are gradually refined.

Table 3.4 XML information store

Component	XML File	Contents
Frontends	<source><function>.XML	AST in XML format
Read/write component syntactic analysis component	<source>.<function>.CMP.XML	Control structure and read/write information
Semantic analysis component	<source>.<function>.VAR.XML	Semantic roles of variables and variable summary
Rule inference component	<source>.<function>.VAR.RULE.XML	Conceptual roles of variables
Dataflow analysis component	<source>.<function>.CMP.FLOW.XML	Concept-oriented data flow
Visualization component	<source>.<function>.GML.XML	GML style graph representation

3.7 Program Abstractions

In some cases, before the rules are applied to program, it is difficult to estimate the forthcoming results. Sometimes, the analyzers obtain too much information to handle. Because of this, it is necessary to furnish an abstraction mechanism to filter out irrelevant or unnecessary information. Program slicing is such an abstraction that extracts statements that transitively impacts a definition of one or a few variables.

Mark Weiser introduced program slicing to facilitate program comprehension and debugging [Weiser, 1984][Weiser and Lyle, 1986]. Since then, program slicing has continued to be a topic of considerable research [Binkley and Gallagher, 1996][Tip, 1995]. Different notions of program slices as well as a number of methods to compute them have been proposed in the literature [Tip, 1995]. An important distinction is static vs. dynamic slice. A static slice is computed without making assumptions regarding a program's input, whereas a dynamic slice relies on some specific test cases for the input [Tip, 1995]. We use static slices in our research.

Viewing a program slice simply as a sequence of lines is inadequate [Jackson and Rollings, 1994]. We designed Program Slice Browser as an interactive visualization tool with novel features that assists the user in making effective use of program slicing. Program Slice Browser

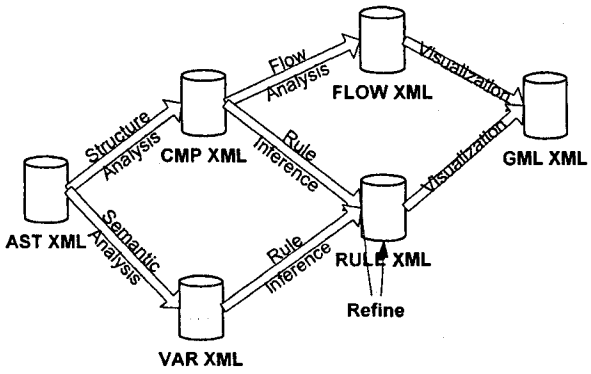

Figure 3.6 Correlations between XML files

also provides new abstraction mechanisms, navigation capabilities, and integration with other components for visualizing software.

3.7.1 Program Slice Browser

Program Slice Browser (PSB) is designed as a part of SeeCORE. We have also implemented PSB in our former Software Reengineering Environment (SRE) [Deng *et al.*, 2000]. PSB is a part of software visualization environment, which consists of a code browser, a Block-Level Abstract Syntax Tree (BLAST) viewer, template viewer, and PSB itself.

We have used an integrated design and a combination of syntactic and semantic abstractions as the foundation for creating a powerful visualization environment. The user can visualize the software at different levels using the integrated environment: the actual code, syntactic structure of the code, program slice (to focus on specific parts of the program), or the semantic structure of the code. Note that these directly correspond to the four visualization components integrated into the environment. These components are integrated with a *cross-referencing capability* that helps in relating different views of the software., The user can get a handle over the complexity of large software and navigate micro-level details by using compacted and simplified views. Syntactic and semantic forms of compact views are provided through BLAST viewer and template viewer respectively. Simplifications are possible through abstraction mechanisms.

3.7.1.1 Program slice model

We consider backward slices as defined in [Ning, *et al.*, 1994]. The mechanics for slicing, as implemented in our SRE, is described in this section. A program slice, with reference to given variables and a given line in the program, is defined as follows.

Definition: Given a variable vector V in which each element of v_i is a variable to be checked, and a location point p (a line number) of program, the program slice $SLICE(V,p)$ is the sequence of statements which influence the definition of each variable $V_i \in V$ at point p. Formally, $SLICE(V,p)$ can be defined recursively as described below.

1. $Slice(V,p) = \sum_{V_i \in V} sSlice(V,p)$
2. $sSlice(V_i,p) = D_i$ where at statement d_i, the variable V_i is defined as a constant and that definition can reach p, Or
3. $sSlice(V_i,p) = q \bigcup Slice(readSetVector(q),q)$, where V_i is defined at statement q and the definition can reach p. $readSetVector(q)$ is the vector of variables used in statement q.

The program analysis for slicing in our system is inter-procedural analysis. If a variable is passed to a function as parameter, we kept track of it down into the function. In implementation, we considered a function a composite statement that contains multiple reads and writes. In intra-procedural analysis, we calculated the read set and write set for each function. Then in inter-procedural analysis, the read set and write set of functions are treated the same way as those of simple statements except that we must consider the impact of global variables and aliasing at call sites.

Precise program slice is a difficult problem. In this paper, we focus on addressing the user's capability to interact and extract useful information from complex program slice and apply it to solve practical problems. Other research papers [Ottenstein and Ottenstein, 1984][Agrawal *et al.*, 1991][Landi and Ryder, 1992][Ball, 1993][Ball and Horwitz, 1993][Choi *et al.*, 1993][Choi and Ferrante, 1994][Agrawal, 1994] have addressed the problem of more precise analysis of program slice in the presence of "unstructured" control flow, e.g. GOTO statements and

"aliasing", e.g. arrays and pointers. We applied essentially the same strategy as Lyle [Lyle, 1984]to deal with the array analysis.

3.7.1.2 Interactive program slice diagram

To begin with, PSB displays a slice as a directed graph with three types of nodes corresponding to statements (that are part of the slice) within a selected subroutine, variable references that are outside the subroutine, and calls made to other subroutines that should be considered as a part of the program slice. The tool supports two initial layouts. The user can also change the placement of those individual nodes.

- In *multi-column* layout nodes are spread in fixed columns to prevent the edges from overlapping. The user can set the number of columns on a toolbar.

- The *upside-down-tree* layout is the layout in which nodes are positioned according to the data flow pattern. Applying the breadth first search algorithm creates the tree layout. The root node, placed at the bottom, is the point where the program slice begins, and the nodes at the top are the points where the program slice ends. The user can adjust the tree layout by setting the parameters *width*, *height*, and the *spread factor* (the bigger the factor is, the farther the distance between horizontal nodes).

Figure 3.7 is an example of a program diagram displayed using upside-down-tree layout. In Figure 3.7, three components of program slice browser are presented. In the main window, a program slice is displayed as an upside-down tree. The small toolbox (upper-left) allows user to determine the parameters width and height. The toolbox also accommodates the user with four buttons: *recovery*, *procedure-level abstraction*, *block-level abstraction*, and *elimination* buttons. The abstractions are discussed in a later section. The third window (lower-left) is the slice control window. In this example, the window indicates that a program slice of variable *ui*3 starting from line 46 is computed.

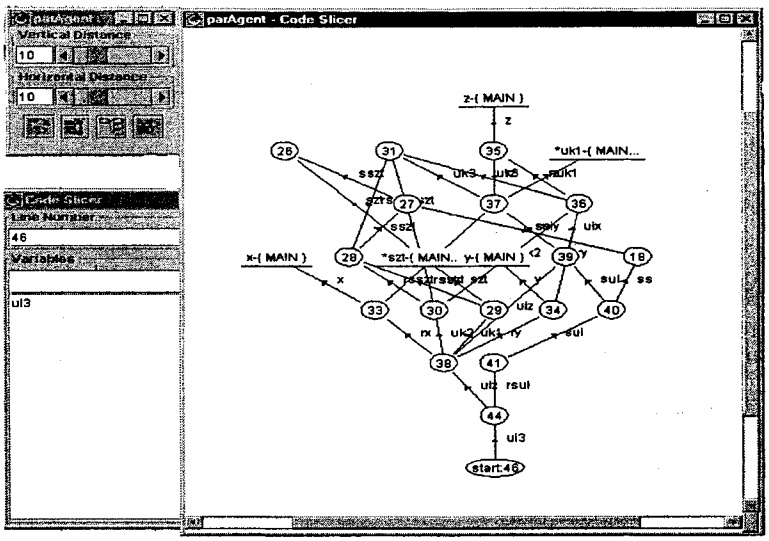

Figure 3.7 A program slice diagram

3.7.1.3 Semantic program slice abstraction

We used program slice abstractions to make it easier for the user to manage and understand complex program slices [Nishimatsu *et al.*, 1999]. These abstractions can be of different types. The papers [Jackson and Rollings, 1994][Sosic and Abramson, 1997] consider procedure-level abstraction to facilitate program understanding. Additional abstractions are needed to facilitate debugging, and performance analysis. PSB is designed to incorporate four types of abstractions corresponding respectively to code blocks, procedures, graphs, and domain-specific semantic entities.

We have used the following notation. For formal definitions of abstractions, we represent a slice S as follows:

S=$\{(e,d,V)|(e,d)$ is the edge from e to d in S and e is dependent on d because there is single (or multiple) variable in vector V which is defined in d and used in e and the definition at d can reach $e\}$

Block-level Abstraction. Blocks are syntactic units. There are three distinct types of blocks in our system - ordinary, call, and control blocks. A block level abstraction encapsulates statements belonging to a block and represents them as a single node of the program slice

diagram. This new node is labeled according to the block type and the source code line numbers defining the boundary of the block, e.g. DO[10, 23]. Formally, the block-level abstraction on a slice S and a block B is a binary operation defined as follows:

$BLOCK(S, B) = (S - \{(e, d, V) | e \in B \text{ or } d \in B\}) \bigcup \{(c, d, V) | \exists (e, d, V) \in S$ where $e \in B$ and $d \notin B$ and c is the new node representing the block $B\} \bigcup \{(e, c, V) | \exists (e, d, V) \in S$ where $d \in B$ and $e \notin B$ and c is the new node representing the block $B\}$

The user can enable this abstraction by clicking on the "block level abstraction" button on the PSB toolbar and then using the BLAST viewer to select the block he or she wishes to encapsulate. This interaction is shown by the edge labeled "1" in Figure 3.8.

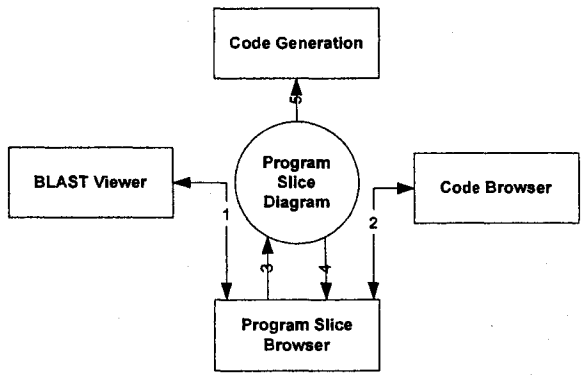

Figure 3.8 Interaction between program slice browser and other modules

This abstraction encapsulates information internal to a block and simplifies a program slice by reducing the number of nodes and the number of edges. The nodes belonging to each selected block are merged together to form a new node. This new node retains the dependence relation between the merged nodes and rest of the nodes, but all relations within the block are hidden.

This abstraction is useful for top-down debugging. Debugging is done at beginning at block-level; the internals of the block are checked after a defective block is identified. For complex legacy codes, top-down debugging is often desirable. Checking at the statement-level for the entire slice can be very time consuming (especially considering the instrumentation inserted into a loop that will be repeated many times) and tedious. With top-down debugging,

it is possible to limit the statement-level checking to only a small part of the slice.

Block-level abstraction is also useful in bottom-up debugging. The expert who is familiar with the program can use bottom-up debugging. Usually, a block of code is a meaningful computation unit. The expert can easily verify the correctness of a block of code with high-level knowledge. Once he or she identifies the correctness of block, the verified block can be abstracted so that internal details are hidden from the expert. This helps the expert concentrate on the remaining unverified parts.

Procedure-level Abstraction. A procedure level abstraction simplifies a program slice by retaining only the information about procedure calls. For example, suppose in statement 10, a common variable A is read, and in statement 20, a local variable B is read. In procedure DUMMY, both A and B are changed because they are passed into DUMMY as actual parameters. After the procedure level abstraction, the nodes representing statement 10 and 20 in the slice diagram are removed. A new edge labeled "A, B" points to the "DUMMY" node indicating variables A and B are changed in DUMMY. In this way, programmers can easily grasp the dependence relationships between current procedure and other procedures. This is especially useful in case of Fortran programs because of the pervasive use of *common* (global) variables and the use of *reference parameters.* The user can enable this abstraction by clicking the "procedure level abstraction" button on the PSB toolbar.

Formally, the procedure level abstraction on a slice S is a unary operation defined as follows:

$PROC(S) = (S - \{(e, d, V)|$ neither e nor d is a call block$\}) \bigcup \{(x, y, V)|\exists$ a path p from the call block x to the call block y in S and V is the dependence vector of along p that may be changed in $y\}$.

We will describe a scenario to illustrate the use of procedure level abstraction for relative debugging [Sosic and Abramson, 1997]. Relative debugging is a debugging method that finds defects by comparing the results of the debugged code against the results of a referenced code. Relative debugging usually is used in software reengineering, such as parallelization.

The subject of analysis is a subroutine called HZETA from an electromagnetic application program written in Fortran. Suppose that a discrepancy is observed between the serial and

parallel outputs for the variable H. Using the code browser, the programmer can determine that H is modified in the subroutine HZETA. However, we cannot rule out the possibility that the problem lies within modification of another variable in a different subroutine. That modification has a bug and it is indirectly (possibly through a long chain of subroutine calls) causing the observed discrepancy between serial and parallel programs. In practice, it is very difficult to perform this type of analysis manually. This scenario is in fact motivated by a similar problem we have faced in debugging a widely used regional climate model code called MM5. This model has about 100K lines of code with more than 200 subroutines.

The procedure level abstraction is very useful for relative debugging in scenarios such as the one described above. To illustrate the use, Figure 3.9 shows a PSB display after the procedure level abstraction has been applied. Note that the objective is to analyze the subroutine of HEZTA to gain a procedure-level understanding of how the variable H is updated. The PSB display in Figure 3.9 shows us that all the variables (relevant to the computation of H) are coming from MAIN. Thus, the user can ignore other subroutines and concentrate on HZETA and MAIN to solve the debugging problem.

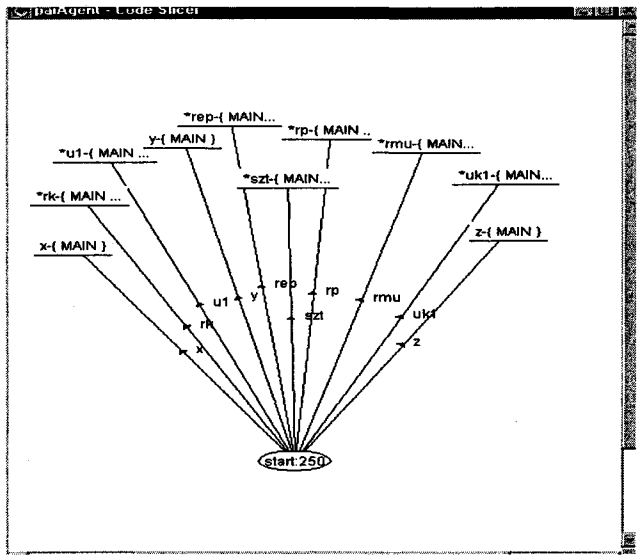

Figure 3.9 Procedure-level abstraction on the program slice diagram

Graph-level Abstraction. A graph-level abstraction is used to simplify a slice based on

graph characteristics. Weiser and Lyle called *program dicing* in [Lyle and Weiser, 1987]. The objective of this abstraction is to enable the use of graph characteristics and algorithms to find efficient solutions for debugging problems. Relative debugging is similar to the link-fault-detection problem in networks in that it is viewed as a problem of identifying faulty links in a program slice diagram. Then, efficient graph algorithms can be applied to find the bugs, now viewed as faulty links.

We will present examples to illustrate how this abstraction can be used. Consider the program slice shown in Figure 3.10. One can easily observe that the graph has two "super nodes" (each representing a group of ordinary nodes) connected by one link. This graph characteristic can be exploited to make debugging efficient. By checking the correctness of SZT on connecting link "129→70" it is possible to eliminate the top super node and thus quickly reduce the size of the debugging problem. In addition to simple visual inspection, this technique can be further improved by making use of clustering algorithms to form appropriate super nodes.

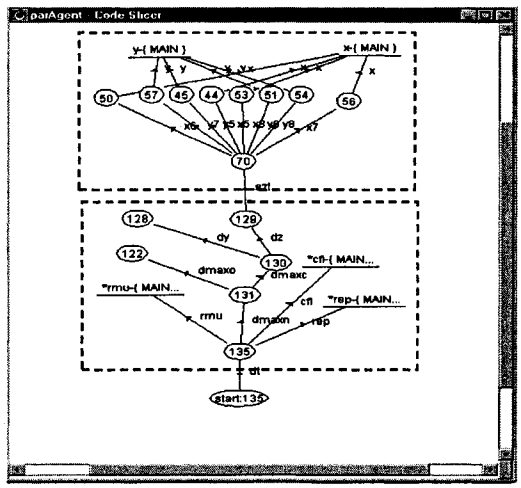

Figure 3.10 "Super nodes" in the program slice

Concept-level Abstraction. A concept-level abstraction is provided to extract information from a slice using domain-specific knowledge encapsulated in the form of a *template* for a class of programs. We believe that the semantic-level abstractions will be useful in context of

a variety of application domains. However, to have a focus and validate our ideas with real-life test cases, we are using domain knowledge associates with FEM and FDM and applying that knowledge to analyzing legacy scientific application in case studies.

We proposed a *template viewer* to enable conceptual abstractions. The template is defined in terms of *concepts* and *logical steps* that manipulate these abstract structures. The template represents high-level understanding of the program, e.g. in FEM code *element* is a concept and *equation solver* is a logical step. These concepts and steps may be dispersed through the program and they are bound to the program during the interactive diagnostic phase. Conceptual abstractions are based on either concepts or logical events that are identified through a domain-specific template. The user clicks on an abstract data structure seen through the template viewer. Then, all the nodes (in a program slice) corresponding to modifications of this abstract data structure are grouped together as a single node. This new node is labeled with the name of the concept.

Another conceptual abstraction is based on logical events/steps. The user clicks on a logical event seen through the template viewer. Then, all the nodes (in a program slice) corresponding to statements or subroutine calls that are bound to the specified logical event are grouped together as a single node. This new node is labeled with the name of the logical event. Conceptual abstractions simplify the slice and provide more useful information as the user can relate the slice to meaningful data structures and/or logical events.

The primitive template viewer we developed in our tool SeeCORE formally is called *Program Skeleton.*

3.7.1.4 A Quantitative Experiment

PSB is a visualization tool. Its abstraction mechanisms are useful and we have provided some illustrations of their use. This section describes a simple quantitative experiment in which PSB is applied to understand and analyze a program slice. The experiment is based on a real-life debugging experience. The subject of analysis is a subroutine called HZETA from an electromagnetic application program written in Fortran. HZETA contains 175 lines of code

including comments and blank lines and 29 common variables. Our BLAST tree consists of 20 blocks. The program slice is based on array variable H, to track its modification starting at the end of the subroutine HZETA.

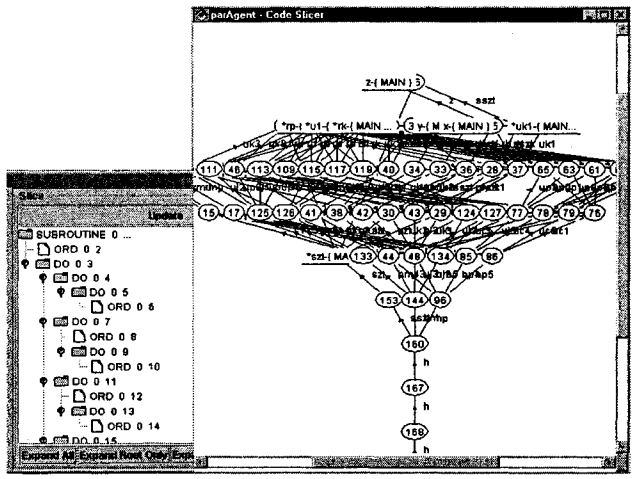

Figure 3.11 An example of program slice diagram and BLAST viewer

Figure 3.11 shows the program slice diagram and the BLAST viewer. As seen, the slice (right) is very complex. It contains altogether 65 nodes, and 168 arcs.

First, consider the effect of procedure-level abstraction on the slice diagram. Figure 3.9 shows us that only MAIN and HZETA itself are responsible for modification of H. It also tells us that 10 out of 29 common variables contributing to modification of H are possibly modified in MAIN.

Next, consider the effects of block-level abstractions. The block labeled "ORD 0 6" in the BLAST viewer is an ordinary block from line 26 to line 48. This block is inside a two-level nested loop. This loop is to compute unit vectors of a locally orthogonal system. All 20 statements within the selected block (excluding two blank lines) belong to the slice. Suppose that after a review of this piece of code, the programmer decides that it is not necessary to debug the code within the selected block. Programmer checks the block-level abstraction toggle button and click the "ORD 0 6" node in the BLAST viewer to instruct the browser to merge all 20 statements into one super node labeled DO[26, 48]. After this abstraction, the

slice diagram contains 45 nodes and 106 arcs.

From line 102 to line 145 is another two-level nested loop block "DO 0 11". This DO loop is used to compute transformed variables in a contra-variant component. After this block is abstracted, the reduced diagram contains only 33 nodes and 67 arcs.

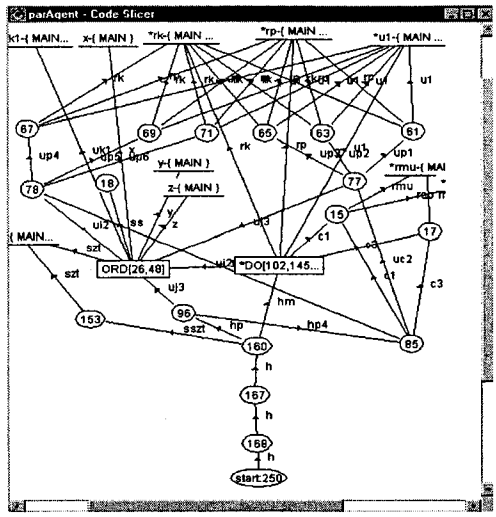

Figure 3.12 The final snapshot of the program slice diagram after block abstraction and elimination

Finally, suppose the programmer determines that the temporary variable HP5 at line 86 is computed correctly. The programmer checks the elimination toggle button and clicks the node 86 to issue an elimination instruction to cut off a branch of the tree. After applying this graph abstraction, only 30 nodes and 54 arcs are left. The final result is shown in Figure 3.12.

We have summarized the quantitative results of the experiment in Table 3.5.

3.7.2 Program Skeleton

The objective of the concept-oriented flow analysis is to identify the important computational steps in a program. These steps are represented as a *conceptual program skeleton.* A conceptual program skeleton is an abstraction that captures the behaviors of the program to show how the critical variables are computed. Computing the conceptual program skeleton involves performing a series of *conceptual program slicing.* Each conceptual program slice starts

Table 3.5 Summary of the effects of the abstractions on subroutine HZETA

Operations	Effects
Original program slice diagram	65 nodes, 168 arcs.
With procedure-level abstraction	11 nodes, 10 arcs.
With first block-level abstraction	45 nodes, 106 arcs.
With a successive block-level abstraction.	33 nodes, 67 arcs.
With a successive graph-level abstraction	30 nodes, 54 arcs.

with a definition of a critical variable and ends in places where it reaches critical variables.

The difference between the conceptual program slice abstraction and the program skeleton is that the program skeleton is an architectural view of program. Each step (node) in the program skeleton is a small program slice; the program skeleton can be considered an abstraction of a traditional program slice that covers the whole program.

3.7.2.1 The Skeleton Algorithm

The relationship between steps in program skeleton reflects the transition between critical variables. We start with a traditional program slice using an algorithm similar to "system dependence graph" in [Sinha *et al.*, 1999]. The traditional slice is modified to obtain the conceptual slice. The skeleton extraction algorithm is shown in Figure 3.13. The input to the algorithm is *concerned_set*. Typically, *concerned_set* contains conceptually critical variables. The basic idea of the algorithm is that the analyzer follows the control flow forward. At each point l where a critical variable a is computed, the analyzer gets a backward program slice starting from $< l : a >$ and ends where it is either defined as constant or assigned to another critical variable. Finally we get a sequence of steps where the critical variables are computed. Each step consists of an execution sequence. The execution sequence reflects the value transition and transformation between the critical variables.

We do not differentiate individual elements in an array or fields in a struct. We assume

```
Function concept_skeleton
Input: concerned_set
Output: a vector of slices
 1 Follow the control flow
    until reaching the end of the program{
 2    Get the write set of the current statement
 3    For each definition in the write set {
 4      If it is in the concerned_set {
 5        Get read set of the statement
 6        For each use in the read set {
 7          If the read variable is in the concerned set {
 8            Add the definition  & the use to a slice
 9          } else {
10            Follow the traditional backward slice
              until all branches either reach the end
                 or reach a critical variable {
11              Add all involved variables except for
                the redundant variables to a slice
12            } // end follow backward slice
13          }
14        } // end for each use
15      }
16    } // end for each definition
17    attach the slice to the slice vector
18  } // end for the control flow
```

Figure 3.13 The program skeleton algorithm

that the concepts of an array or struct are atomic - one single variable does not contain multiple concepts (they can contain multiple semantic or syntactic annotations). But it is worth pointing out that

- Fields or elements within one variable could mean different concepts. The solution would be that we modify the read/write analyzer as if variables are accessed at the field-level or element-level.
- Occasionally in scientific application, an element at specific position in an array is used for special purpose (means different concept), so the slicer should be able to detect the unusual array access patterns.

3.7.2.2 Assessment of Concept-Oriented Program Slice Skeleton

The philosophy behind the program skeleton is that users usually are interested in those critical variables that represent the fundamental elements in the conceptual mode. For instance, in finite element method code, the fundamental concepts include *elements*, *nodes*, *coordinates*, *global assembly matrix*, etc. The logical steps or dynamic behaviors of the concepts, e.g., equation solver, sparse matrix assemblies, etc. are computing surrounding those conceptual elements. In actual code, there are other auxiliary steps to perform these significant logical steps such as using temporary variables. However, unless the user is specifically interested in the detailed algorithms, he or she would prefer to get a general idea of the program architecture. Program slice skeleton produces a natural way to perform top-down methodology. Because program slice skeleton is based on the significance of variables defined by conceptual rules, the program skeleton reflects the conceptual steps (logical steps) that a program performs at the macro-level.

From the perspective of program abstraction, program skeleton only reflects the dependence between critical variables. Ideally, traditional program slice extracts relatively smaller parts of programs. However, according to many researchers' (including our own) experience, the dependence could transitively be extended to cover a large part of the program. Considering the complicated dependence relationship between statements, the traditional program slice often is far more complex than intuitively expected. Program skeleton functionally blocks the massive dependence transition by critical variables. Thus, the picture is simplified to describe the definitions of critical variables and the flow transition between the critical variables.

The criteria we use to evaluate a program slice skeleton include:

- **Transitions between critical variables.** Critical variables are special variables that represent fundamental concepts in the conceptual model. The transitions between the concepts in the conceptual model are covered by the transitions between the critical variables. The critical variables are a small number of the variables; thus, the transitions are simpler and bounded by the critical variables.

- **Inter-relationship between critical variables.** One important application of program slice skeleton is discovering the inter-relationship between the critical variables. This inter-relationship is a significant signature for recovering the conceptual roles of variables in program in addition to computing patterns.

- **The abstraction of the program.** Program slice skeleton should show the architecture of the program. In the skeleton, the nodes are either the definitions of critical variables or the transitions between critical variables. To accommodate this, the program slice skeleton is interactive. The user can click on the nodes and go to the source code if he or she wants to investigate the details.

In Chapter 4: Case Studies, we will explain program skeleton in details in the finite element method case study.

3.7.3 Variable Concept Web

One of the challenges in program comprehension is retrieving the complex relationship between variables. Thanks to the conceptual roles of variables, the relationship between critical variables could more closely conform to the conceptual model in the user's mind. The traditional tools can retrieve dependences between variables as well. Due to the lack of the guidance from the user, a general analyzer usually returns massive and complex dependences. The massive information is of little use for recovering the simple conceptual model. Many interactive tools such as Reflexion and Rigi can accept the user's instructions on specific code. Compared with such tools, our system is extensible to other programs because the domain-specific rules are applicable to all programs within same domain.

In Appendix B, the screenshot shows the concept web of *mydemo.c*. The *control-dependence* between variable *value* and *condition* has been used to infer one of the conceptual roles of *condition - boundary_cond*, which means that the variable *condition* is used to determine the "boundary" of the surface or body given the context of scientific application.

In order to retrieve the specific type of the dependences, the algorithm shown in Figure 3.13 has been altered according to the algorithm in Figure 3.14. The major change is that when

applying backward program slicing, the type of dependence is passed down to all involved variables. We do realize that the dependence type could change in the backward program slicing. For example, suppose we have code "$tmp = b; a[tmp] = d;$" and a, b, and d are critical variables. The dependence between a and d is value-dependence. The dependence between a and b changes from *index-dependence* (between a and tmp) to *value-dependence* (between tmp and b). Our criterion is to select the *index-dependence* as the type of the dependence between a and b. The reason is that the type of dependence primarily means the *impact* of relevant variable. In the example, the type of dependence between a and b is dominated by the impact of tmp to a.

```
Function concept_skeleton
Input: concerned_set
Output: a vector of slices
 1 Follow the control flow
   until reaching the end of the program{
 2   Get the write set of the current statement
 3   For each definition in the write set {
 4    If it is in the concerned_set {
 5     Get the read set of the statement
 6     Determine the type of the dependence
 7     For each use in the read set {
 8      If the read variable is in the concerned set {
 9       Add the definition, the dependence type
           & the use to a slice
10      } else {
11       Follow the traditional backward slice and
           pass on the type of the dependence
         until all branches either reach the end
           or reach a critical variable {
12         Add all involved variables except for
             the redundant variables to a slice
13           Add the inherited dependence type
               to the slice.
13       } // end follow backward slice
14      }
15     } // end for each use
16    }
17   } // end for each definition
18   attach the slice to the slice vector
```

Figure 3.14 Augmented program skeleton algorithm

Below, Table 3.6 presents the conceptual role of variable *condition* in *mydemo.c* after

applying the rules. Especially, variable *condition* is annotated as "boundary_cond" because variable *condition* is related to a critical variable *value*, and the type of the dependence between variable *condition* and *value* is control-dependence. This pattern is expressed in rule "$critical\&percent(depinference(\text{"}critical\text{"}, \text{"}control\text{"}), \text{"}0.6\text{"}) \rightarrow boundary\_cond$". The complete program skeleton file for *mydemo.c* can be found in Appendix C.

Table 3.6 Roles of *condition* in *mydemo.c*

<Variable Name="condition" Type="2" array="2" block="0" boundary_cond="1" computing="4" critical="2" expr="2" file="0" ifcontrol="0" input="0" loop1="4" loopbody="2" loopcontrol="0" output="0" param="0" passByValue="true" peripheral="0" read="1" redundant="0" return="0" significant="0" subscript="0" write="1">

<ProgramFeature block="0" callLink="0" computing="2" curNodeID="59" expr="1" file="0" finalNodeID="59" ifcontrol="0" input="0" loop1="2" loopbody="1" loopcontrol="0" output="0" param="0" read="0" return="0" subscript="0" undetermined="2" write="1">
</ProgramFeature>

<ProgramFeature block="0" callLink="0" computing="2" curNodeID="79" expr="1" file="0" finalNodeID="79" ifcontrol="0" input="0" loop1="2" loopbody="1" loopcontrol="0" output="0" param="0" read="1" return="0" subscript="0" undetermined="2" write="0">
</ProgramFeature>

</Variable>

3.8 Integrated Software Environment

Integrated environment is essential to a success of a software engineering tool. Our research work aims to integrate various software engineering/reengineering techniques into one integrated software engineering environment. In addition to our original designs, we refer to other successful tools, including those commercially successful IDEs. The primary objective of the environment is to provide a generic analysis process. The analysis is driven by domain-specific rules and additional domain-specific features, e.g., specific interactions or particular views if needed.

3.8.1 Project Manager

A user of SeeCORE starts with creating a project. A project contains a list of source code that the user hopes to analyze and understand. In addition to the source code, the project also includes domain-specific rules that are applicable to the program. The system does not provide an interface for editing rules. The user can add or delete rules during analysis.

3.8.2 Program-Oriented Analysis

The program-oriented analysis parses the source code and generates XML intermediate representation. Ideally, the intermediate representation should be independent of programming languages. In practice, each programming language has its own distinct characteristics. We tried to design a common intermediate representation that consists of common representation for general features such as branch, function call, loop etc., plus language-specific features. Our representation is expected to be a superset of popular programming languages' features. Currently we mainly consider C/C++ and FORTRAN code.

We use EDG-to-XML utility to transform the EDG intermediate representation into the common intermediate representations. Since EDG is essentially a compiler, its intermediate representation contains many features that are not useful for program reengineering. Our representation is concise, but it contains sufficient information for program analysis.

Just like typical integrated development environment, the project manager creates a series of scripts to drive the EDG frontends to generate XML files for a large project in batch mode.

3.8.3 Multiple Views of Program

The semantic and conceptual program analysis offers multiple views of a program. The essence of those views is the semantic or conceptual roles of data/variables. There are relationships among the views and mainly the relationships are related to conceptual roles of variables. The system provides cross-reference capability that enables the user to easily switch between different views.

3.8.3.1 Variables Semantic View

The variables semantic view is a global summary of semantic roles of variables in a program. The main functions of the variables semantic views are:

Providing a variable semantic summary

Variables are the major representation of concepts in the conceptual mode. The system collects all uses of variables in the program and annotated with semantic roles for individual uses. The user can quickly review the summary and understand how the variables are used. The patterns of the uses are important hints to the conceptual roles of the data.

Annotating semantic roles of variables

The variables are annotated with semantic roles according to the use patterns. The semantic roles of variables are domain-independent. Thus the semantic variable analysis can be used for different domains. Some statistical analysis can be done on the semantic roles. For instance the ratio of a variable used for file input/output against calculation can be a sign whether it is a log given the context of a network monitor.

Navigating the code

The cross-reference capability allows the user to navigate the code. Particularly the system provides a static *stack frame* for the inter-procedural use of variables. If a variable is not used in current method but a method transitively called, the stack frame can help the user keep track of the usages of the variable across multiple methods.

It worth pointing out here that in our analysis, we consider all parameters to be reference parameters. We do this because the type of the parameter passing does not impact the semantic or conceptual roles that a variable holds. Similarly, although the different types of parameter passing do impact the program slicing, the primary purpose of our program slice used in here is not for debugging. The main goal of program slicing techniques in our research is to retrieve the dependence relationship between variables to infer the conceptual roles of data.

3.8.3.2 Variable Concept View

Variable Concept View shows the conceptual roles of variables after applying domain-specific rules. One example of a variable concept view is given in Appendix C. The objective of a variable concept view is visualizing the rule inferences and conceptual roles of variables.

The GUI of variable concept view is very similar to variable semantics view. In addition to the semantics view, the concepts view particularly visualizes conceptual summary of variables and conceptual rule inferences.

Conceptual summary

In the conceptual view, the system offers a tabular summary of the variables from the perspective of the conceptual roles. Technically, the criteria of the conceptual roles can be any variable categories. However, the style of the visualization strongly suggests the conceptual roles used in the tabular summary should be those orthogonal conceptual roles. One particularly useful category is conceptual significance. In a domain-specific program, there are many variables used in the code. But usually only a small percentage of them are reflected in the conceptual model on the user's mind. So we could differentiate the variables by their relationship with those kernel variables in the model. A user generally focuses on understanding the concepts of kernel variables first, then shifts to other concerned variables or specific code.

Rule inferences

Rule preserves domain expert's domain knowledge. A rule describes a code pattern that suggests strong semantic or conceptual meanings. It is favorable to show user how a conceptual role was inferred. In our software, newly discovered conceptual roles are visually emphasized to seize the user's attention. At the same time, the inference window shows the rules that were used in the inferences.

3.8.3.3 Program Slice Browser

Program Slice Browser is an interactive program slice visualization tool that was originally used for debugging. Since then, program slice has been extended by researchers to assist program comprehension. In our software, program slice browser presents users a program

navigation tool. The program slice diagram shows the data dependences between statements represented as nodes. The users can traverse the tree and examine source code by clicking on the nodes.

We provide both forward and backward program slice. The forward slice browser allows the user to evaluate the influence of a certain definition; the backward slice browser allows the user to trace back the dataflow to find out those definitions that contribute to the current update.

Appendix F shows both examples of forward and backward program slice.

3.8.3.4 Program Skeleton

Program skeleton is a visualization tool that presents the architecture of a program. Program skeleton shows the program architecture in sequential steps. Each step shows a computation involving significant variables. The significance of a variable is determined by the domain knowledge stored in rules.

Program skeleton can be thought of as partitions of global program slice. As we said in previous sections, one drawback of traditional program slice is the complexity and extension of data dependences. Many abstraction techniques have been suggested to simplify program slice, and program skeleton is a novel abstraction designed for architecture recovery

Appendix G presents a demo of program skeleton. The demo shows the major computing steps in the finite element method code Eddy. The program skeleton view is composed of two parts:

Sequential Steps

Sequential steps are represented as a sequence of computing of conceptually significant variables. Each step is a special program slice segment which starts with the definition of a significant variable and ends with either constant (implies that the data flow cannot traced back further) or another significant variable. The conceptual meaning of each step is that the cores of computing are those critical variables. The relationship between steps shows the dataflow transition between those kernel variables. For human comprehension, conceptual

model is usually not complex. Thus, the architecture should only reflect the relatively simple conceptual model rather than expose too many details.

Programs Slice Segments

Each step corresponds to a small program slice segment. The program slice segment is the computing bounded by significant variables. The boundaries of the program slice are either dead ends (where the dataflow ends) or boundaries of adjourning steps.

The program skeleton is interactive: when the user click on the steps, the skeleton shows the program slice segment on the right window (shown in Appendix G), and when the user click on the list of the program slice segment, the source viewer loads the source code and highlights the lines in the slice (as shown in Appendix H).

3.8.3.5 Variable Concept Web

In addition to the architecture view of the program, the program skeleton also facilitates Variable Concept Web. A concept web is a diagram that illustrates the relationships between critical variables. One such example is shown in Appendix I.

The variable concept web is a simple yet sufficient map of the domain expert's conceptual model. In actual code, the relationships between variables are very complex. For human comprehension, the expert's conceptual model of the program domain is very simple and his or her primary focuses are the relationships between those kernel variables. The auxiliary variables that do not reflect in the conceptual model are hidden from the user. If the user wants to study the details, the interactive environment assists the user in navigating the source code.

Programmatically, the relationship between variables can be value-passing/computing, array-index, file-read/write, or control dependence. The program-oriented relationships finally are translated into domain-specific relationships. For example, given the context of the FEM model, the conceptual relationships are mathematical (in a formula), area-coordinates, source-results etc.

3.8.3.6 Correlation between the Views and the Rules

So far, we have shown the multiple views of programs that our tool SeeCORE provides. The core of those views is the conceptual model in domain expert's mind and his/her domain-specific knowledge stored in rules. When the domain expert adjusts the rules, the views accordingly are updated to reflect the changing model. The framework is supposed to be generic across different domains because it is the human expert's comprehension process that the software emulates.

We conclude this chapter with Figure 3.15. Figure 3.15 illustrates the correlations between the multiple views of a program. Conceptual roles of data, program skeleton, and concept web together form the program model. The domain-specific rules decide the conformation between the conceptual model in domain expert's mind and the program model recovered by the tool. We are continuing to dedicate efforts to improve our program visualizations and provide domain experts more friendly GUIs to enable pleasant understanding of domain-specific programs.

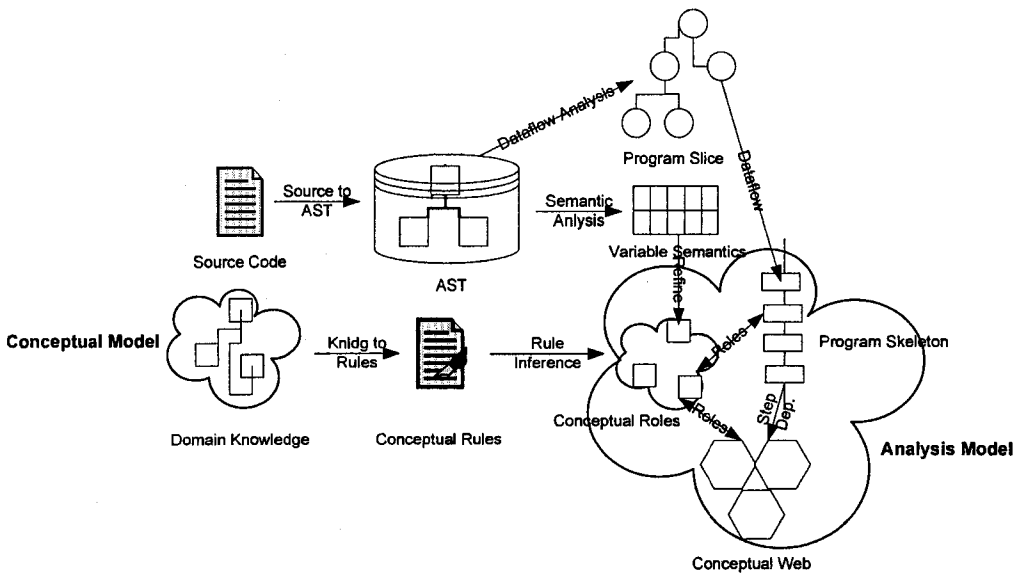

Figure 3.15 Correlations between multiple views

CHAPTER 4. CASE STUDIES

To demonstrate the effectiveness of our domain-specific tool SeeCORE, we have selected two case studies: a Finite Element Method (FEM) code *Eddy* and an operating system *Xinu*.

The FEM is a widely used method; it consists of standardized logic steps. FEM code contains many domain-specific concepts that are closely related to underlying mathematical or physical models. Without applying the domain-specific knowledge, it is difficult to bind a FEM code to the standardized FEM model.

Xinu is a small operating system developed for educational purposes. An operating system code is different from scientific application in many ways and its different purpose and different concerns are reflected through different program patterns. The OS code has different subsystems for performing distinct functions. The operations performed by the OS are often driven by events; the event handler and the event dispatcher are typically coupled by using function pointers. The OS manages computer resource and the information needed to mange a resource is stored in a so-called control structure, such as a process control block, a file control block and a device control block, etc.

By choosing two diverse examples of application domains, we demonstrate that our tool is versatile. We evaluate the changes needed to accomplish the adaptation.

4.1 FEM Code Case Study

In scientific engineering field, numerical methods, such as finite element methods, finite difference methods and boundary element methods, are popularly used. The FEM code Eddy, is for a simulation model of eddy current in a coil. It is for calculating the impedance values of all probe positions. This code reflects typical domain-specific knowledge including the

conceptual roles of data and the logical steps to be found in any FEM code. In our case study we focus on extracting conceptual roles of data from Eddy.

Domain knowledge is hierarchical where a domain inherits the properties from its super domain. Because FEM is a subcategory of numerical methods, we divide the domain-specific knowledge into two parts - knowledge applicable to any numerical modeling code and the specific knowledge applicable only to FEM.

4.1.1 Knowledge Pertinent to Numerical Modeling

The generic steps in any numerical simulation are: initializing, saving results, and performing simulation. Other characteristics representations used for geometric or material properties. The code is expected to reflect the following domain-specific concepts:

- Field values
- Spatial properties
- Temporal properties

Table 4.1 describes the correspondence between the domain-specific concepts and the program patterns.

In addition to the domain knowledge, programming knowledge also is important to understand a program. An experienced domain expert is able to relate program patterns to domain concepts or use program patterns to infer domain concepts. Table 4.2 summarizes some common programming knowledge that is useful to infer the bindings between numerical method domain concepts and program entities.

4.1.2 Domain-Specific Knowledge of FEM Code

FEM codes have additional characteristics. In FEM, a body or structure is discretized into smaller elements (i.e., triangular elements, rectangular elements etc.); each element consists of

Table 4.1 Domain concepts and program patterns in numerical methods

#	**Domain Concepts from Numerical Methods**	**Program Patterns**	**Domain Knowledge of Numerical Methods**
1.1	Field values	Arrays	Arrays are used to store the field values.
1.2	Spatial dimensions	Array Dimensions	A subset of array dimensions corresponds to spatial dimensions.
1.3	Temporal dimensions	Array Dimensions	An array dimension corresponds to temporal dimension.
1.4	Data exchange between neighboring grid cell	Special array index access patterns	Elements of an array represent points in space.
1.5	Spatial bounds	Loop bounds	Loop bounds are auxiliary variables with the index variables representing the coordinates of the grid cells
1.6	Temporal bound	Loop bound	If a loop is iterated on the temporal dimension, the loop bound indicates the boundary of simulation time.
1.7	Threshold	Branch control	Branch control could be a threshold comparison.

finite number of corner points called the *nodes* or *nodal points*; the properties of the elements are formulated and combined to obtain the properties of the entire body. In FEM code, solving the problem for the entire body is reduced to assembling the solutions of individual small elements whose stress-strain relationships are more easily approximated. Necessary boundary conditions are imposed on a subset of the elements [Desai and Abel, 1972][Martin and Carey, 1973][Reddy, 1993][Chandrupatla and Belegundu, 2002][FEAO, 2002].

To recognize the conceptual roles listed in Table 4.3 in a FEM program, the analyzer must use the distinct algorithmic patterns of FEM code. The FEM algorithmic patterns are determined by FEM logical steps. An FEM code has to implement the following standard steps.

Program entities in this paper mainly refer to variables or methods. It worth pointing out

Table 4.2 Programming knowledge and program patterns in numerical methods code

#	Program Patterns	Programming Knowledge of Numerical Methods Code
2.1	File input	File input operations are used for loading initial values. It typically is pre-processing steps in numerical methods.
2.2	File output	File output operations are used for saving results. It typically is post-processing step in numerical methods.
2.3	Global variables	Global variables are usually used for representing conceptual elements in the domain model.
2.4	Frequently used variables whose active scope is throughout the whole program	If a variable is used frequently and its active scope is throughout the whole program, it is highly probable that the variable carries out important compute results which directly used by the critical variables that represent domain concepts.
2.5	Loops in the program	Usually in a numerical method code, certain loops are compute-intensive parts. The level of nesting is indicative of the significance or the type of computing.

that a concept could be represented implicitly by other related concepts. One example is that coordinates can represent *nodes* and the connectivity between the *nodes* represents *elements*. Our automated tool is not designed to provide such complex inferences; the tool is limited to discovering those variables that could potentially represent concepts. The tool provides multiple views of the program including the relationship between variables, the relationship between variables and major steps, and the relationship between major steps to assist the domain expert in recovering the conceptual model.

4.1.3 Specific Rules for the FEM Code

We have designed 20 rules for binding the conceptual roles to the variables in the program. The 20 rules primarily represent the knowledge about FEM codes summarized in previous Table 4.1, 4.2, 4.3 and 4.4. The domain-specific rules are described below in Table 4.5. It

Table 4.3 Domain knowledge and program patterns in FEM

#	**Domain Concepts of FEM Code**	**Program Patterns**	**Domain-Specific Knowledge of FEM Code**
3.1	Nodes	Array elements	The set of nodes is represented by arrays.
3.2	Elements /Connectivity	Array representing connectivity matrix	We may not see an explicit entity representing elements. The relationship between the elements and the node is represented by a connectivity matrix.
3.3	Boundary	Arrays used in branch control	The arrays that store boundary nodes are typically used in branch control.
3.4	Global matrix	A large multi-dimensional array are used in a loop that solves a system of global linear equations	The global matrix is used in an equation solver.
3.5	Vector	Array used in nested loops	The vector is used in the global equation solver.
3.6	Field values	Arrays that used in post-processing steps, i.e., saving values to files after the equation solving.	The results of FEM code usually are the field values of the entire body or surface.

Table 4.4 FEM modular steps and program patterns

#	**Logical Steps of FEM (Modular Steps)**	**Program Patterns**	**Domain-Specific Knowledge of FEM Code**
4.1	Mesh generation	Loops where elements are computed.	Mesh generation determines the connectivity.
4.2	Assembly	Loops	Assemble the global matrix using nodes and connectivity matrix.
4.3	Global Matrix solver	Nested loops that compute global matrix and vector	Global matrix solver is a linear equation solver.
4.4	Calculation of field values	Calculation of the variables which are annotated field values.	The compute results are the field values of the entire body.

worth noting that the terms or names used in the rules may not bear exactly the same meaning in the scientific papers. Basically the user who writes rules can choose names or terms he/she likes as long as they are consistently defined and used in the analysis. Some of the rules seem trivial, such as *subscript* → *dimension*. The main purpose of this type of aliasing rules is to make analysis results more friendly and readable to domain users.

We try as possible to define the conceptual rules based on the semantic atoms available in program feature vectors obtained by the semantic analysis. However, there are a few important program patterns which are hard to be described by simple program feature vector alone. We extended our simple rules to embed the plug-ins which act as stored procedures to recognize complex program patterns. For instance, *scope*, *relatives* are such embedded plug-ins.

In Table 4.5 and 4.6, two plug-in functions have been used. *Relatives* is a plug-in which finds out other variables that are relevant to current variable as to the specific semantic patterns. In the specific rule in the table, we are interested to know what else variables have been used with the current variables in IF conditions. *Percent* is a plug-in which indeed is a fuzzy function. To allow the analyzer to find more meaningful results, we sometimes must relax the pattern recognition criteria.

In this case study, we divided importance of a variable into five levels: *critical*, *significant*, *peripheral*, *redundant* and *undetermined*; the categorization is not absolute - it can be customized by changing the rules. The user has control over the categorization. Categorizing variables according to their significance is very valuable. The categorization conforms to the philosophy that in a complex program, the backbone of the system is a small set of variables while the rest variables are either auxiliary and subject to the programmer's style.

4.1.4 Experiment Results

The FEM code consists of three modules: input module, mesh generator and the solver. Input module and mesh generator have simple code structure. We focus on the solver because it contains all the FEM concepts and the main logic steps. The statistics for the FEM code is shown in Table 4.7.

Table 4.5 Specific rules for FEM model

#	Rules	Interpretation of Bindings	Corresponding Knowledge
5.1	file & input → source; file & output → result; source → significant; result → significant;	Due to the large amount of data, typically data are initialized by reading from configuration files and written back to result files. Thus, file input is initialization; file output is the final step. The variables involved in these file operations are important.	2.1, 2.2
5.2	subscript → dimension;	Any use of a variable as a subscript is flagged as a candidate for representing a spatial or temporal dimension.	1.2, 1.3
5.3	result → property;	Results of a finite element method code usually are the field values.	1.1, 2.2
5.4	read & write → reduction	The term reduction is used to indicate that a variable is read and written, e.g. $V = V + A[I]$; Reduction typically indicates that the code is for an equation solver.	4.3
5.5	loop3 & reduction → mx; mx & array → matrix;	If an array is computed in a three-level nested loop and the computing pattern is reduction then the array is a candidate for the global matrix	2.5, 3.6, 4.3
5.6	loop2 & reduction → vr; vr & array → vector;	If an array used in a two-level nested loop and the computing pattern is reduction, then the array is a candidate for the vector.	2.5, 3.6, 4.3
5.7	! significant & scope ( ) < 2 → redundant; computing < avg ("computing") → peripheral & ! significant;	If a variable is not used or rarely used (the frequency of the uses is extremely low compared with other variables), it is considered peripheral variables (i.e., not significant) If a peripheral variable's active scope is limited within one function, it is considered redundant variable. Annotations peripheral and redundant are used to categorize those variables unlikely represent conceptual roles.	2.4

Table 4.6 Specific rules for FEM model (continued)

#	Rules	Interpretation of Bindings	Corresponding Knowledge
5.8	ifcontrol & ! loopcontrol → control; loopcontrol & ! ifcontrol → bounds; ifcontrol & loopcontrol → control; ifcontrol & loopcontrol → bounds;	If a variable is used in a control structure, it could be either threshold or bound.	1.5, 1.6, 1.7
5.9	ifcontrol & all (marks (relatives ("ifcontrol"[1]) , "bounds")) → bounds & ! control	This rule differentiates between threshold and boundary - if a variable is only related to other recognized "bounds" this variable is a bound and the branch statement is bound checking.	1.7
5.10	loopbody & count (parents ("ForLoop" , "2")) = 3 → loop3; loopbody & count (parents ("ForLoop" , "2")) = 2 → loop2; loopbody & count (parents ("ForLoop" , "2")) = 1 → loop1;	Loops are important patterns for scientific computing (especially for the FEM, the level of nesting is one of the most outstanding signs for locating an equation solver).	2.5
5.11	critical & any (depinference ("vector" , "index")) → connectivity	The vector usually represents the properties of the nodes in FEM code. The index of the node array is possibly related to the physical positions of the nodes or the connectivity between the nodes.	3.2
5.12	critical & percent (depinference ("vector" , "control") , "0.6") → boundary_cond; critical & percent (depinference ("matrix" , "control") , "0.6") → boundary_cond	Inside a surface or body, computing on internal nodes is uniform. Special checking or computing is done only for the node on boundary. The checking is expected to be control/branch statement.	2.5, 3.3, 3.4

Table 4.7 Facts of Eddy

	Input Module	Mesh Generator Module	Problem Solver Module
Line of code	163	332	1646
Files	1	2	13
Functions	1	5	41
Libraries	7	8	16
Variables	21	31	70

4.1.4.1 Variable Categorization

As described above, we have defined rules for categorizing the variables. The result of our categorization is shown in Table 4.8. The actual variables in each category are shown in Table 4.9.

Table 4.8 Division of the significance of variables

	Input		**Mesh Generator**		**Problem Solver**	
	Count	**Percentage**	**Count**	**Percentage**	**Count**	**Percentage**
Critical	2	10%	7	21%	25	36%
Significant	6	28%	4	12%	4	6%
Peripheral	12	57%	10	29%	29	41%
Redundant	1	5%	10	29%	7	10%
Undetermined	0	0%	3	9%	5	7%
Total	21	100%	34	100%	70	100%

In our tool, we are more interested in pursuing critical variables because one hypothesis of our analysis is critical variables carry conceptual roles in the domain model. The results in Table 4.8 have shown that categorizing variables by applying semantic and conceptual rules is meaningful since the user can focus on rather small number of variables in complex program.

In Input module, the main task is to initialize the field values of the materials (in this code, the field values are complex numbers representing density).

Table 4.9 Categorization of critical variables in Eddy

Module	Category	Actual Variables
Input Module	**Critical**	*QV* *ifp*
	Significant	*NZElem* *Length* *Width* *Area1* *Area2* *IS*
	Peripheral	*i* *IsSymm* *NCPS* *NumMat* *Vyn* *VP* *Turns* *InRad* *Thick* *CX* *RX* *FName*
	Redundant	*NOC*
Mesh Generator	**Critical**	*NP* *NBound* *NPBgn* *NPEnd* *XOrd* *YOrd* *mfp*
	Significant	*InRad* *Width* *NB* *IB*
	Peripheral	*i* *j* *Thick* *Length* *N1* *N2* *N3* *NumMat* *NumBC* *ifp*
	Redundant	*DIST_X* *DIST_Y* *j1* *j2* *j3* *imax* *imin* *idiff* *Delta_X* *Delta_Y*
	Undetermined	*i1* *i2* *i3*
Problem Solver	**Critical**	*NP* *NPBC* *NCoil* *NPP* *Mat* *ImpElm* *NBound* *XOrd* *YOrd* *Area* *Phi* *SK* *A* *QV* *Q* *ZA* *ZB* *ZProbe* *RX* *CX* *ifp* *cfp* *vfp* *mfp* *impfp*
	Significant	*AEl* *NCond1* *N1* *N2*
	Peripheral	*NRUNS* *NumNP* *NumEl* *Vyn* *VP* *IsSymm* *NumMat* *NZElem* *NCPS* *Cond* *Depth* *Width* *InRad* *Thick* *NCond* *ACross* *OmegaR* *Pi* *FName* *i* *i2* *NZ* *NumBC* *IB* *IBound* *LU* *IS* *SknDpth* *Current1*
	Redundant	*XSCALE* *YSCALE* *NOC* *j* *j2* *IC* *ICG*
	Undetermined	*Omega* *Z1* *Z2* *ZSum* *Current*

- QV is the current density of the materials representing in complex numbers, and ifp is the handle for the file in which the material properties are stored.

- The remaining variables are auxiliary attribute of QV, e.g., geometric properties of the body (used as auxiliary variables for performing computations). In mesh generator module, the major task is to create the coordinates of the nodes and the connectivity between the nodes.

The mesh generator module generates mesh composed of elements.

- $XOrd$ and $YOrd$ are the coordinates of the nodes. NP is the connectivity matrix. $NBound$, $NPBgn$ and $NPEnd$ together contain the information of the nodes on the boundary.

- $i1$, $i2$, $i3$ are labeled "undetermined" because these variables are involved in the computation of significant variables, however, by using our rules we cannot confirm them as critical variables. The user can decide whether or not these "undetermined" variables are critical or not.

In the problem solver module, the main task is to solve the global system equations to obtain the properties of the entire body.

- Among the redundant variables: $XSCALE$ and $YSCALE$ are constant scales between input data and materials (e.g. input is in nanometers while materials quantities are in millimeters). NOC, j, and $j2$ are not used in the program; IC and ICG are index variables of a loop that controls the number of simulation runs. These variables are used only once or twice; and more importantly, have little relationship with other variables.

- Most of the peripheral variables, as we can infer from their names, are physical parameters, such as number of nodes, number of elements, number of materials, thickness, depth, etc. We consider these variables peripheral variables because the information stored in these variables is very simple; these variables accompany other variables in computing and seldom carry sources or results directly related to the calculations.

- The significant variables AEl, $NCond1$, $N1$, and $N2$ represent respectively the area of an element, the number of conductors in a coil, and the number of elements in two coils. They can also be considered physical parameters. The difference between these four and other peripheral variables is that these four parameters are read from an input file and written to the final results; this implies that the user is interested in them.

- Among the undetermined variables, *Omega* is an important parameter used widely in the program; $Z1$, $Z2$, $ZSUM$ and *Current* are results computed in each iteration. However, the tool did not flag the results as critical variables because the tool detected that those results are stored in other variables which are flagged critical. Therefore, the tool considers them "actors" - they are involved in important computing, but their main purpose is convenience.

- All variables containing important physical concepts have been labeled "critical". This important concept includes nodes, elements, physical boundary nodes, materials, surface area, source file pointer, and destination file pointer. These are listed in Appendix J.

4.1.4.2 Program Skeleton

In the experiment where only critical variables are considered, the tool found 44 concept-oriented program slices. Among them, 17 slices reflect the transition between critical variables. After removing reflexions (the transitions that only involve the variable itself), the tool found 12 transitions that are useful for understanding the relationship between entities in the problem.

Table 4.10 Quantitative summary of conceptual program skeleton

	Total number of slices	**Transitions between critical variables**	**Transitions between critical variables after removing reflexions**
FEM code Eddy	44	17	13

The conceptual program skeleton also shows the dependence relationships between the critical variables. The results are summarized in Table 4.11 which have been used to aid conceptual inference when there is confusion about variables' conceptual roles.

Table 4.11 Relationship between critical variables discovered by conceptual program skeleton

#	**Definition of Critical Variable**	**Related Critical Variable**	**Type of Dependence**
1	*NPBC*	*NBound*	index
2	*SK*	*XOrd*, *YOrd*, *Area*, *Mat*, *Eddy.Main.RX*, *Eddy.Main.CX*	value
3	*SK*	*NP*	index, control
4	Area	*SK*, *Eddy.Main.RX*, *Mat*, *XOrd*, *YOrd*, *NP*	value
5	Q	*NP*, *Area*, *QV*, *XOrd*, *Mat*, *SK*	value
6	*Q*	*NP*	index
7	*A*	*Q*	value
8	*A*	*NPBC*	control
9	*ZA*	*NP*, *XOrd*, *ImpElm*, *A*	value
10	*ZB*	*NP*, *XOrd*, *ImpElm*, *A*	value
11	*ZProbe*	*NP*, *XOrd*, *ImpElm*, *A*	value
12	*Phi*	*A*	value

If we applied traditional program slice on variable *Phi*, we would see that almost all critical variables are involved. In our concept-oriented program slice, only variable *A* is involved in the definition of *Phi*. The slice is much simpler and the physical meaning of this slice is also more obvious - get the magnitude of the complex magnetic vector potential value.

4.1.4.3 Recognized Conceptual Roles

We include both the manually compiled conceptual roles and automatically recovered conceptual roles in Appendix J. The manual compilation was created by the programmer and the

author. Our tool has successfully recovered FEM specific concepts *matrix*, *connectivity*, *boundary condition*(represented by the variable that stores the positions of nodes on the boundary), *physical properties* and *spatial dimensions*. Our tool also recognized all variables that appeared in Table 4.11 as critical variables. The tool could not differentiate specific field values, such as reluctivity, impedance, etc unless the user could further define rules that bind program patterns to more specific roles.

The manually compiled concept annotation table does not include all used variables in the code. The left out variables are either auxiliary variables as described in previous section 4.1.4.1, or file pointers whose conceptual roles are difficult to describe in FEM domain.

4.2 Another Case Study: Xinu Operating System

Xinu is an operating system designed and implemented by Douglas Comer *et. al.* at Purdue University [Comer, 1988]. Xinu, as a small yet full-fledged operating system, has been used extensively for educational purpose and also as base for creating special purpose operating systems for some telecom companies. Xinu is an open source program. The version of Xinu that we have used in the case study was obtained from Dr. Kothari at Department of Electrical and Computer Engineering at Iowa State University. The code has been used in the operating system course for years and has been incrementally modified and enhanced by students. Thus, the program is a typical legacy code - it has evolved for years; it involved many people working at different stages; it is mixed up with different programming styles, both good and bad.

4.2.1 Domain Knowledge of Operating System

The standard functionalities of an operating system (OS) include process, memory, disk and other devices, and file system management. An OS programmer is primarily interested in knowing how these components are implemented and organized. These components (such as the process control module, file management system, etc.) are the domain-specific concepts known to the OS programmer and commonly used in practice to understand and modify an OS code. These domain-specific concepts and the organization of OS code is quite different

from the domain-specific concepts that we have discussed in scientific application analysis.

Device management data structure typically are control blocks. A control block is a structure (*struct* in C/C++) that contains information necessary for managing a device. For example, a file control block contains buffers for in-core copies of the current index and data blocks necessary for accessing the file. Each device has its own control block and its structure and contents are customized for the specific type of device. Furthermore, an operating system has only one process control, one memory management, one file system management, one device management, etc. In the actual code, this is reflected as the control blocks are defined as global variables. The names vary but every OS code has certain key data structures such as process table, device table etc. Each device table entry is reserved for a device and it has a pointer to the control block for that device.

Table 4.12 lists statistical facts about the Xinu code and we summarize the above observations in Table 4.13 "Domain knowledge and program patterns in operating system code".

Table 4.12 Facts of Xinu

	Lines of Code	**Number of Functions**	**Number of Root Functions**	**Number of Global Variables**	**Number of Local Variables and Parameters**
Xinu	8751	186	69	61	715

4.2.2 Specific Rules for the Xinu Code

After static module dependence analysis, among the 186 functions, Xinu has 69 functions that appear as the root of a call order tree. This is quite a different organization than the FEM code: the FEM code is organized as a single call order tree, whereas the Xinu code is organized as a forest of multiple call order trees. The code within a tree gets called in response

Table 4.13 Domain knowledge and program patterns in operating system code

#	**Domain Knowledge of Operating System**	**Program Patterns**	**Note**
13.1	Resource control structures	Defined as a *struct*, referenced from the device table	A data structure for managing a resource
13.2	Distinct type of resources management	Global variables, local variables or parameters declared specific struct	Specific struct type is related to the specific resource management.
13.3	Event driven mechanism in OS	There is a dispatch table and it has function pointers	Many asynchronous OS operations are driven by events, such as interrupt, semaphore, inter-process communication, and device management etc. Event handler typically is implemented as function pointer. A dispatcher calls event handler when an event occurs.
13.4	Multiple management instances	Arrays	Multiple instances are organized as tables (arrays). Using array is simpler and faster than link list. It is a typical program pattern in operating system, for instance, process tables, directory table, device table, etc.
13.5	Inter-relationships between resources	Control or value dependence between variables	One resource management could be related to other resource managements. This relationship could be discovered by dependence analysis of the resource management blocks.

to an event (interrupt or a system call) and thus it does not have an "explicit" link to the main program. The link is through a function pointer. We designed 64 rules to analyze Xinu. The rules can be grouped into three categories.

Table 4.14 Domain-specific rules designed for Xinu analysis

#	Rules	Interpretation of Bindings	Corresponding Knowledge
14.1	global ( ) → significant; Type = "2" → a_table; a_table & significant → critical;	Global variables are significant; array means multiple instances of management object. They are categorized as critical variables.	13.2, 13.4
14.2	TypeName = "routine_address" → handler & critical; TypeName = "devsw" → device_table & critical; TypeName = "dstab" → disk_table & critical; ...	Specific struct routine_address is bound to event handler. Struct devsw is bound to device switch table entity. Struct dstab is bound to disk control table.	13.1, 13.3
14.3	device_table → abstract ("device_table"); disk_table → abstract ("disk_table"); arp_cache → abstract ("arp_cache"); ...	If a variable is declared the type of specific struct, it is considered an instance of the management data structure. In other words, all local variables, including parameters declared the same type of the specific data structure, are grouped together as an identical resource management.	13.1, 13.2

To save space, we only list a few representative rules in Table 4.14. The complete list of the rules can be found in Appendix L.

Specifically, we highlight the followings regarding the rule designs.

- *Global ()* is a function used to determine whether or not a variable is active in the global scope.

- We enhanced the SeeCORE system by adding a few new attributes to the program feature vector. TypeName was added to indicate the data type of a variable.

- Category 14.2 lists important struct defined in Xinu code. We have found 29 bindings between the struct and the domain-specific concepts. These bindings were done manually using *grep* utility.

- In addition to the rules in category 14.2, the tool enables grouping of all variables with a specific type into a single category corresponding concept. These rules are given in category 14.3. Note that; *abstract()* is a function that creates an abstract variable which represents the concept. The conceptual meaning behind this group of rules is that in operating system typically specific struct is used for specific resource.

4.2.3 Experiment Results

We have applied the rules to analyze Xinu code. The results are encouraging. We were able to derive a categorization of variables and recover the relationships between dispatchers and event/device handlers.

4.2.3.1 Variable Categorization

Among totally 776 variables (including parameters), the tool recognized 63 variables as critical variables. Among them, 35 event handler variables are device handlers [2]; 15 variables are tables or collective objects; and the rest of the 13 variables (not included in Table 4.15) are pointers that point to the control blocks listed in Table 4.15.

Not only are we interested in the number of critical variables, we are more interested in the specific roles of the critical variables. As we can see the critical variables summarized in Table 4.15 cover all areas in an operating system including process management (*proctab*), semaphore management (*semaph*), memory management (*mc_block*, *memlist*, *ptfree*), device

[2]We consider function pointers as variables too because we are also interested in the relationships between event dispatcher and event handlers.

table (*devtab*), ports (*ports*), network (*eth*), disk table (*dstab*), file table (*fltab*), console (*tty*, *keyboard_buffer*), buffer pool (*bptab*), and interrupt dispatch table (*intmap*).

Table 4.15 Categorization of critical variables in Xinu

Concept	**Critical Variables**
Event handler	*console_init console_shutdown ioerr console_read console_write console_getc console_putc console_ioctl ionull dsinit dsopen dsread dswrite lfinit lfclose lfread lfwrite lfseek lfgetc lfputc ethinit ethshut ethread ethwrite ethinter dgmopen dginit dgclose dgread dgwrite dgcntl real_main old_clk_isr int_handler clkinit*
Process table	*proctab*
Semaphore	*semaph*
Memory control block	*mc_block*
Free memory list	*memlist*
Device switch table	*devtab*
Queue	*q*
Ports	*ports*
Free queue list	*ptfree*
Buffer pool	*bptab*
Ethernet control block	*eth*
Disk control block	*dstab*
File control block	*fltab*
File control block	*fltab*
Interrupt dispatch table	*intmap*
TTY control block	*tty*
Keyboard buffer	*keyboard_buffer*

4.2.3.2 The Relationships between Device Switch Table and Device Handler

A difficult part of Xinu for a beginner to understand is its device management. In Xinu, function pointers are used in high-level device driver routines to point to device-specific routines

initialization, open, read, write, seek, close, shutdown, etc.

Our analyzer has recovered the relationships between device switch table and device handler in Xinu. In Xinu, device switch table is implemented as pseudo device which is shared by distinct devices including file, disk, and Ethernet. That our tool could recover such specific implementation owes to the domain-specific knowledge 13.1, 13.2 and 13.5 as explained in Table 4.13.

The above results have been visualized by using *Veras* graph component. A demo of part of the relationships between the device dispatch table and the device handlers is given in Appendix M. The graph shows that there are 31 event handlers (i.e. device handlers) are related to the device switch table, including console, disk, file, Ethernet, and datagram device and how the device switch table divides device operation into high-level operation and low-level operation.

4.2.3.3 The Relationships between Control Blocks

The control blocks in an operating system are related to each other. The relationships are usually difficult to obtain due to the huge code size. Function pointers increase the difficulty to understand the implicit links between the control blocks. Even the user, with the knowledge of the relationships between the control blocks, may still find it difficult to relate the concepts in his or her mind to specific parts of the code.

Our tool is able to extract the relationships between the control blocks. In order to help the user focus on the relationships between the control blocks, we developed an abstraction utility that filters out those variables without control block concept annotation (including device handlers) from the visualization. The simpler picture is shown in Appendix N.

As shown in Appendix N, the file operation in Xinu is divided into high-level operation and low-level operation. The high-level operation involves index node(*index_node*), directory(*directory*) and file control block(*file_blk*). The low-level operation involves disk control block(*disk_blk*), buffer pool(*buffer_pool*) and queue node(*queue_node*).

We conclude the Xinu experiments with Table 4.16. Table 4.16 summarizes the complete

relationships between the control blocks that SeeCORE has recovered.

Table 4.16 The relationships between the control blocks in Xinu

Control Block	Related Control Blocks	Note
device_table	*directory*, *disk_blk*, *eth_blk*, *datagram_blk*, *file_blk*.	Xinu's unique device management.
directory	*device_table*, *disk_blk*, *index_node*	Directory management.
disk_blk	*directory*, *device_table*, *file_blk*, *buffer_pool*, *q_table*	Disk management.
file_blk	*disk_blk*, *file_dscrp*, *buffer_pool*, *q_table*	File management.
ports	*queue_node*, *q_table*	Ports management.
eth_packet	*udp*, *ip*, *datagram_blk*, *buffer_pool*, *queue_node*, *q_table*	Network management.
datagram_blk	*device_table*, *eth_packet*, *net_buffer*	Network management.
buffer_pool	*queue_node*, *q_table*	Buffer management. Queue is a basic data structure used by many other components.

In Table 4.16, some of the concepts, e.g., *buffer_pool*, *queue_node*, and *q_table*, are basic data structures and are related to many other control blocks.

The conceptual meanings of the variables are very evident by their naming conventions. A user with certain operating system background should be able to easily relate the results to the conceptual model in his or her mind.

CHAPTER 5. CONTRIBUTIONS AND CONCLUSIONS

This thesis demonstrates that a tool incorporating domain-specific knowledge can provide accurate and efficient domain-specific program analysis. The case studies on the two very different domains have shown that though the high-level knowledge usually is unique in specific domains, it is possible to implement a generic framework that can be easily adapted across different domains.

5.1 Contributions

The contributions of this thesis are in three areas - the conceptual model based on variable roles, a mechanism for recovering the bindings between actual code and conceptual model, and SeeCORE, the prototype implementation of the domain-specific analysis framework.

5.1.1 Conceptual Model

Our program comprehension concept model is based on our observation of how domain experts understand domain-specific programs; we recognize the importance of conceptual roles of variables in the program understanding. The conceptual model consists of five elements - domain context, conceptual roles of variables, program patterns, iterative analysis, and extensible information representation. Domain experts use the first four elements to understand domain-specific programs and they are incorporated into our domain-specific analysis tool. The extensible information representation is required for the tool to implement iterative analysis.

The advantage that our conceptual model has over the other existing methods is that we directly incorporate the domain experts' knowledge into program analysis. Thus, the results are expected to conform to the specific users' concerns or interests. Most of the existing

program comprehension tools focus on information about the organization of the program, design patterns, other program characteristics such as the topology of dependence graph, the inheritance relationships between classes or objects. Undoubtedly, these are important and useful for understanding programs. However, they are insufficient to provide domain-specific understanding which usually are needed by domain experts.

5.1.2 Iterative Analysis

One distinction between our program analysis tool and the other tools is that our analysis is iterative. The iterative analysis has two components: approximate analysis and refined analysis. Implementation variations minimize the usefulness of rigid pattern descriptions in practice. We keep the pattern definitions simple. The main purpose of the analysis is to find as many candidates as possible that can potentially represent domain-specific concepts. The tool iteratively applies the rules to refine the analysis results. Compared with other methods, our iterative analysis is more flexible in tolerating implementation variations.

5.1.3 Prototype Implementation

We implemented SeeCORE as a prototype to demonstrate the effectiveness of our domain-specific methods. SeeCORE provides an analysis platform to incorporate domain knowledge. The functions of SeeCORE include control flow analysis, data flow analysis, variable analysis, domain-specific rule inference, program skeleton abstraction, and variable relationship analysis. SeeCORE also provides a simple visual environment for showing analysis results. These visualizations include variable summary, program skeleton, and variable concept web.

We have successfully used SeeCORE to analyze two programs from two different domains - a finite element method code Eddy and an operating system code Xinu. The results of the experiments have shown that SeeCORE is able to infer finite element method concepts in Eddy and describe the relationships between operating system components implemented in Xinu. More importantly SeeCORE captures the event driven behavior of Xinu to recover the relationships between the operating system components that cannot be recovered using

structural analysis.

5.2 Conclusions

The conceptual role analysis of variables represents one point in program comprehension methodologies. The conceptual model is used routinely by domain experts to understand programs however automated analysis based on the conceptual roles of data has hardly been exploited. We constructed a conceptual model that centers on the conceptual roles of data in a program. To assist the programmers in relating a conceptual model in their minds to actual code, we developed an integrated program comprehension tool to provide the bindings between the model and the code. The tool is well suited to analyze programs by incorporating domain knowledge stored in rules.

The experimental results obtained from the SeeCORE prototype are encouraging. We have received positive feedback from our research peers and domain experts. With the hope that this methodology can be applied to other domains, we are going to extend the prototype and use it to analyze more applications in different domains.

Software is a valuable investment. Even in a rapidly changing environment, managing, improving, and enhancing existing code are daily exercises in industries. We hope that our work benefits the area of program comprehension research and can be used in practice from education to software development.

CHAPTER 6. FUTURE WORK

There are several possible directions for future work. In this chapter, we mainly consider extending the program analysis and improving the usability of our tool SeeCORE.

6.1 Extending the Program Analysis

Currently, the dependence analysis in SeeCORE is mainly data-flow and control-flow analysis. Flow analysis is able to capture the execution paths through a program. The deficiency is that the current analysis does not directly derive code structure. Understanding code structure, such as module dependence and variable-function dependence, is especially important for object-oriented programs. The structural analysis can complement the current flow analysis. For example, had we used structure analysis, we could show the relationships between the control blocks (the data structures) and the event / device handlers (the methods) in Xinu.

6.2 Improving the Project Management

Project organization capabilities are important for managing the source code and the analysis results. Consistency and integrity checks should be there in the information storage management. In our program analysis, if an information file is missing or corrupted, the failure could be cascading because our analysis is inter-procedural and iterative. A storage management tool should be developed to guard the consistency and integrity of the information store.

6.3 Checking Consistency of Rules

At this stage, users can add customized rules at will. The module for creating rules only checks the correctness of syntax and the availability of plug-in functions. Because of this, users may write inconsistent rules. When the rule set is big, it is difficult for the user to manually detect or find conflicts. Formal methods can be applied here to check the consistency of rules.

6.4 Improving the Software Environment

There are improvements that can be done in the software environment to increase the usability of SeeCORE.

Variable Summary. Currently, the variable summary is shown as a tree list or summary table. It is not easy for the user to find interesting variables when the program becomes complex; also a complex program usually contains a large number of variables that do not fit in one window. Interaction capabilities such as querying, filtering, zooming are very useful functions to enable the user to easily capture the interesting parts.

Program Skeleton. The program skeleton describes the global flow of a program. We would prefer to show the skeleton on the dependence graph, call order tree or BLAST viewer so that the user can easily relate the global flow to the syntactic structure. A better synchronized view of the source code should be provided so that the user can easily see the source code as well as the program skeleton at the same time.

Variable Concept Web. We show the conceptual annotations associated with each variable in SeeCORE. We should also show conceptual connections (the dependences) between variables. A friendly user interaction can be by clicking on the edge between two variable nodes the tool visualizes how and where the two variables are related to each other.

APPENDIX A. *mydemo.c*

```
1 const int NUMNODE = 100;
2 int X[NUMNODE];
3 int Y[NUMNODE];
4 int value[NUMNODE][NUMNODE];
5 int condition[NUMNODE];
6 int scale;
7
8 int main(){
9         int i;
10        int tmp;
11        scale = 2;
12        for (i=0; i<NUMNODE; i++){
13                X[i] = i;
14                Y[i] = i;
15                condition[i] = i%2;
16        }
17
18        for (i=0; i<NUMNODE; i++){
19                tmp = condition[i];
20                if (tmp == 0){
21                        value[X[i]][Y[i]] = scale*3;
22                } else {
23                        value[X[i]][Y[i]] = scale*2;
24                }
25        }
26
27        return 0;
28 }
```

APPENDIX B. Concept Web of *mydemo.c*

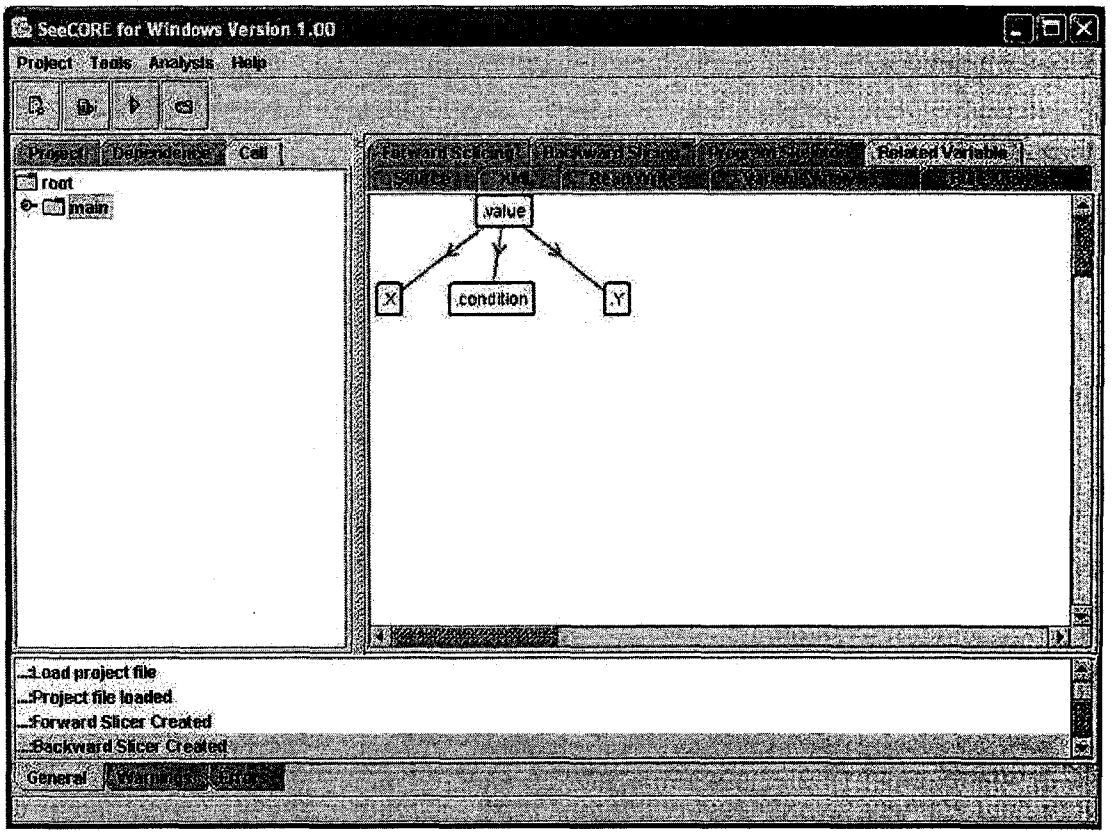

Figure B.1 Variable concept web of *mydemo.c*

The above Figure B.1 displays the concept web of *mydemo.c*. The web shows that variable value is connected with variable X, Y and *condition*. The links between the variable nodes represent specific types of dependences between the variables. Specifically, the dependences between X and *value* as well as Y and *value* are *index-dependence*; the dependence between value and condition is *control-dependence*. The types of dependences have been stored in the

skeleton XML file. Once one of the variables' conceptual role is recognized, the dependence is used to infer the conceptual roles of other related variables.

APPENDIX C. *mydemo.main.skeleton.xml*

```
<?xml version="1.0" encoding="UTF-8" ?>
<root fileName="C:\DATA\Yunbo\cray\SeeCORE\mydemo\xml\mydemo.main.skeleton.xml">
<mainFunctionInfo Function="main" Source="mydemo" />
<Definition Function="main" Line1="13" Line2="13" Name=".X" Source="mydemo">
<Slice Function="main" Line1="13" Line2="13" Name="mydemo.main.i" Source="mydemo" />
</Definition>
<Definition Function="main" Line1="14" Line2="14" Name=".Y" Source="mydemo">
<Slice Function="main" Line1="14" Line2="14" Name="mydemo.main.i" Source="mydemo" />
</Definition>
<Definition Function="main" Line1="15" Line2="15" Name=".condition" Source="mydemo">
<Slice Function="main" Line1="15" Line2="15" Name="mydemo.main.i" Source="mydemo" />
</Definition>
<Definition Function="main" Line1="21" Line2="21" Name=".value" Source="mydemo">
<RelatedVariable Name=".Y" TypeOfDep="index" />
<RelatedVariable Name=".condition" TypeOfDep="control" />
<RelatedVariable Name=".X" TypeOfDep="index" />
<Slice Function="main" Line1="21" Line2="21" Name=".X" Source="mydemo" />
<Slice Function="main" Line1="21" Line2="21" Name=".Y" Source="mydemo" />
<Slice Function="main" Line1="21" Line2="21" Name=".scale" Source="mydemo" />
<Slice Function="main" Line1="21" Line2="21" Name="mydemo.main.i" Source="mydemo" />
<Slice Function="main" Line1="21" Line2="21" Name="mydemo.main.tmp" Source="mydemo" />
<Slice Function="main" Line1="19" Line2="19" Name="mydemo.main.i" Source="mydemo" />
```

```
</Definition>
<Definition Function="main" Line1="23" Line2="23" Name=".value" Source="mydemo">
<RelatedVariable Name=".Y" TypeOfDep="index" />
<RelatedVariable Name=".condition" TypeOfDep="control" />
<RelatedVariable Name=".X" TypeOfDep="index" />
<Slice Function="main" Line1="23" Line2="23" Name="mydemo.main.i" Source="mydemo" />
<Slice Function="main" Line1="23" Line2="23" Name="mydemo.main.tmp" Source="mydemo" />
<Slice Function="main" Line1="19" Line2="19" Name="mydemo.main.i" Source="mydemo" />
<Slice Function="main" Line1="23" Line2="23" Name=".scale" Source="mydemo" />
<Slice Function="main" Line1="23" Line2="23" Name=".X" Source="mydemo" />
<Slice Function="main" Line1="23" Line2="23" Name=".Y" Source="mydemo" />
</Definition>
</root>
```

APPENDIX D. Plug in Functions

Table D.1 Plug-in functions

Category	Function Name	Parameters	Return Values	Description
Logic Function	*All*	A set of Boolean values.	Boolean	Predicative. It returns true only all input Boolean values are true.
	Any	A set of Boolean values.	Boolean	Predicative. It returns true if any of the input is true.
Statistical Function	*Avg*	A semantic / conceptual annotation	Integer	It returns the average number of the variables that are annotated the given annotation across the whole program.
	Count	A set of integer values	Integer	It returns the sum of the input.
	Percent	<a list of Boolean values, threshold>	Boolean	It is an approximation function which returns true if the percentage of true values in the input is greater than the threshold.
Syntactic Function	*Governs*	<syntax node, syntactic role>	Boolean	It evaluates true if the syntactic node's parent node holds the syntactic role.
	Parents	<syntactic role, scope>	A vector of syntactic nodes	The function returns the ancestor syntactic nodes that contains the given syntactic role. It can be either the immediate ancestor or all ancestors depending on the given scope.
	Scope	None	Integer	It returns the scope (measured in blocks) that a variable is active.

Table D.2 Plug-in functions (continued)

Category	Function Name	Parameters	Return Values	Description
Conceptual Function	*Marks*	<A list of variables, conceptual role>	A vector of Boolean values	The function examines the input variables and evaluates true or false respectively depending on whether the variable has been annotated the given role.
	Rels	Conceptual role	A set of variables	It returns a set of counterpart variables relevant to the conceptual roles.[1]
	DepInference	<conceptual role, type of dependence>	A vector of Boolean values	The function returns true if the types of the dependence between the analyzed variable and other variables annotated the given conceptual role matches the given dependence type.

[1] Some semantic/conceptual roles imply paired variables. For example, a variable with role *file* means that it is once used in a file operation which implies another variable used as *file pointer*.

APPENDIX E. SeeCORE - Variable Concept View

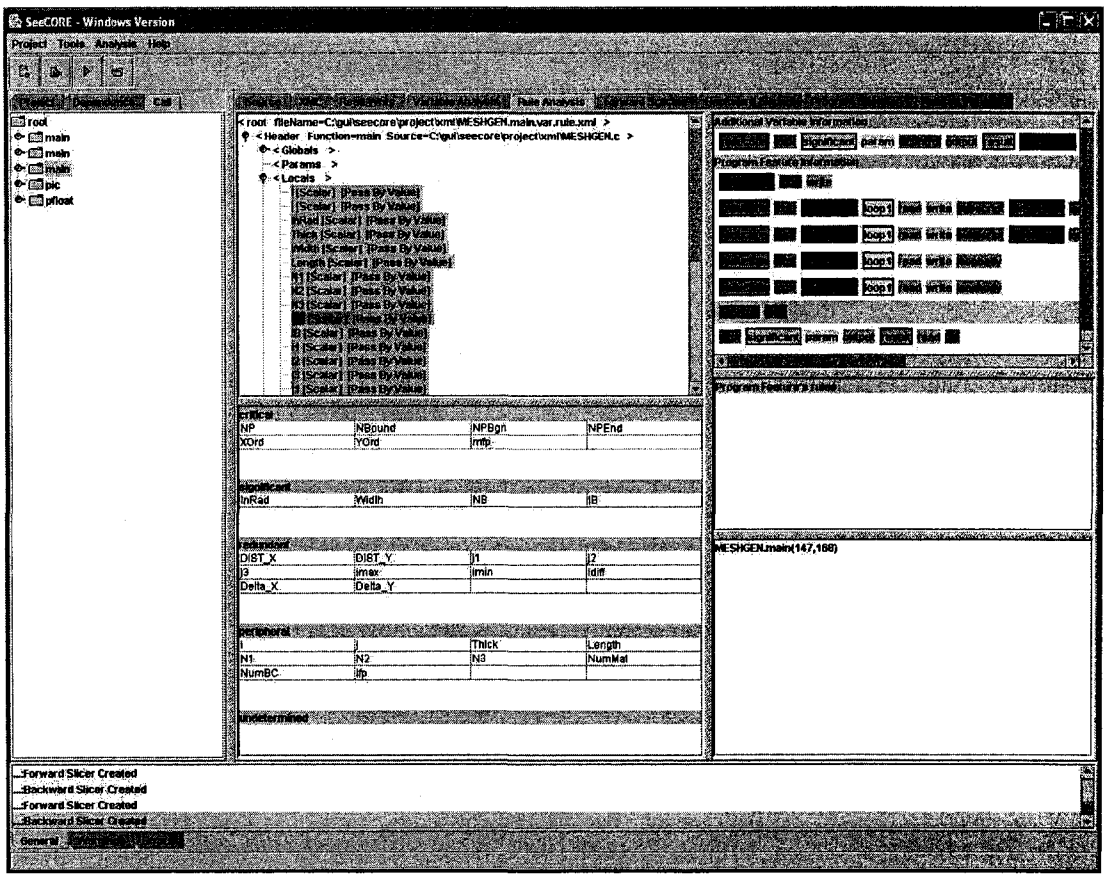

Figure E.1 SeeCORE variable conceptual view

The concept view of variables not only shows the results - conceptual annotation of the variables but also illustrates the inference steps.

The table in the middle shows the conceptual significance of the variables. In this demo,

there are five variable categories[1]: critical, significant, redundant, peripheral, and undetermined.

On the right-most of the main frame, there are three windows, from top to bottom, respectively contain:

- Conceptual roles of variable NB. The first line shows the *synthetic* conceptual roles of NB. The second group contains conceptual roles annotated for all individual uses. The boxed annotations are conceptual roles discovered by rules. The un-boxed ones are semantic roles.
- The rule used for inferring the new conceptual roles.
- The stack frame which corresponds to the source location that variable NB is used.

[1]Note: The variable category can be customized by rules. It is possible that a different category of variable is more applicable to another domain.

APPENDIX F. SeeCORE - Forward/Backward Program Slice Browser

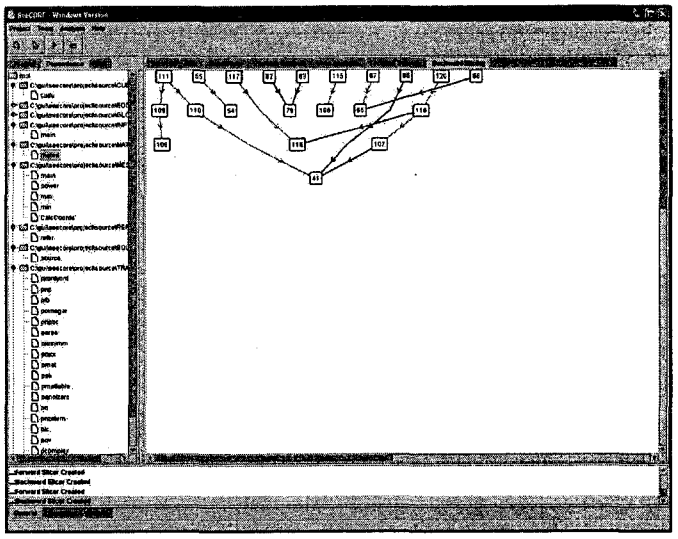

Figure F.1 Backward program slicer

Backward program slice browser allows a user to trace the dataflow back to find out what previous definitions contribute to the update.

Forward program slice browser allows the user to evaluate the influence of a statement (definition).

Both forward and backward program slice browsers are interactive. When the user clicks on the node (denotes statement), a source viewer will load the source code and highlight the statement.

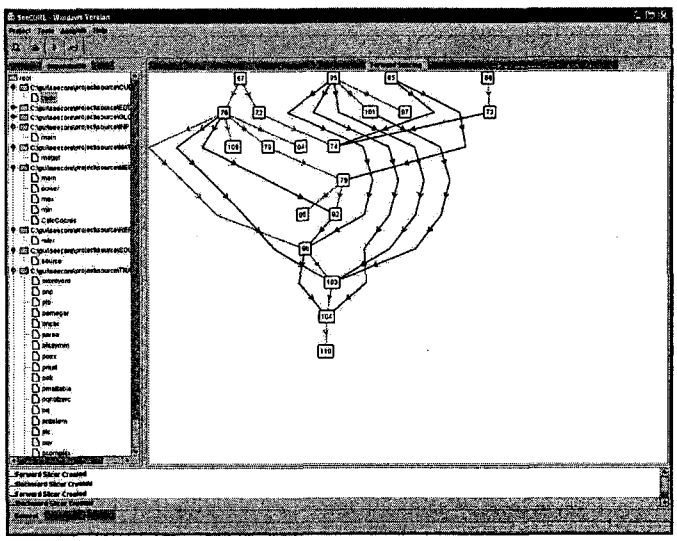

Figure F.2 Forward program slicer

APPENDIX G. SeeCORE - Program Skeleton

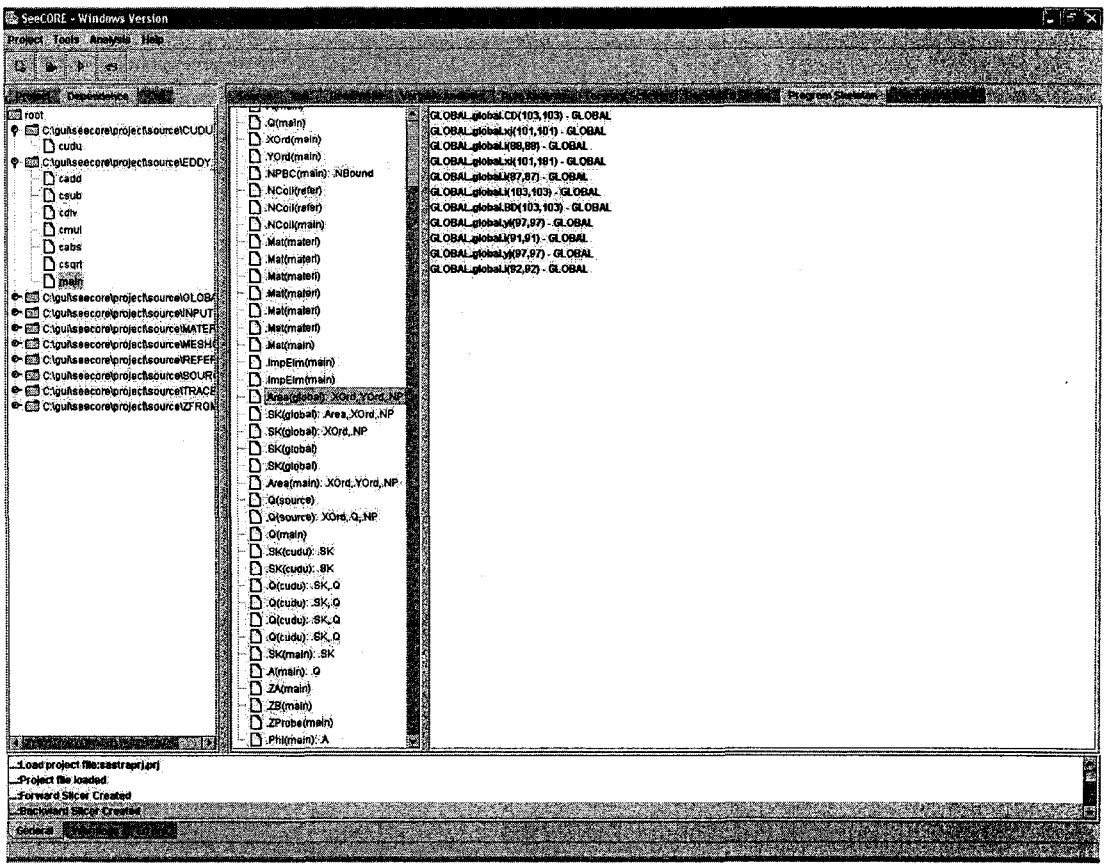

Figure G.1 Program skeleton

The program skeleton shows the computing steps[1] of a FEM code (in the center window). Each step is a computing of a conceptually significant variable. One computing could involve multiple statements (shown in the right window).

[1]The steps displayed here exclude mesh generation simply because the analyzed code has a separated executable module which primarily does the mesh generation.

Program skeleton is interactive. User can click on the steps to see the involved statements. The user can also click on the involved statements then the source viewer will load the source code and highlight corresponding lines.

The demo shows one computing step of a significant variable Mat. The concept of Mat in the code is "materials in the geometry". The step in the figure is part of the global matrix assembly.

APPENDIX H. SeeCORE - Skeleton Navigation

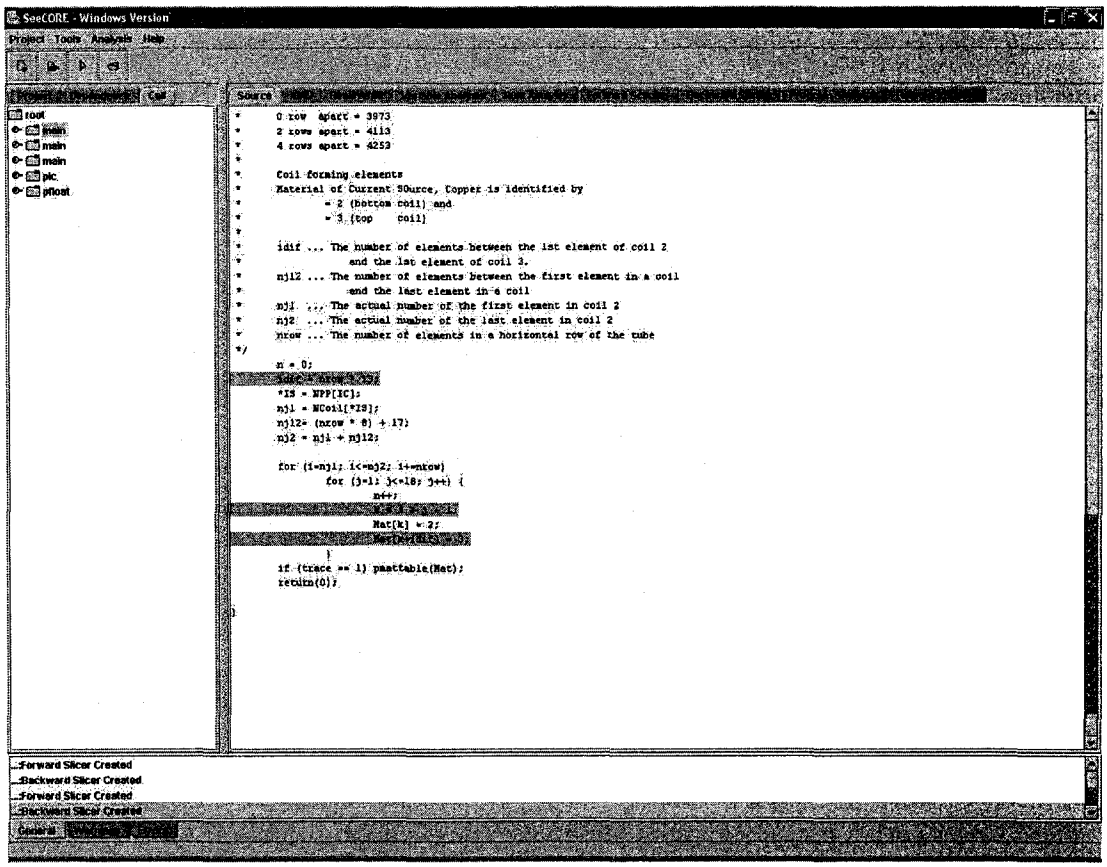

Figure H.1 Program skeleton navigator

Program skeleton is an interactive tool. When user clicks on the nodes, the tool loads source file and highlights related statements.

APPENDIX I. SeeCORE - Concept Web

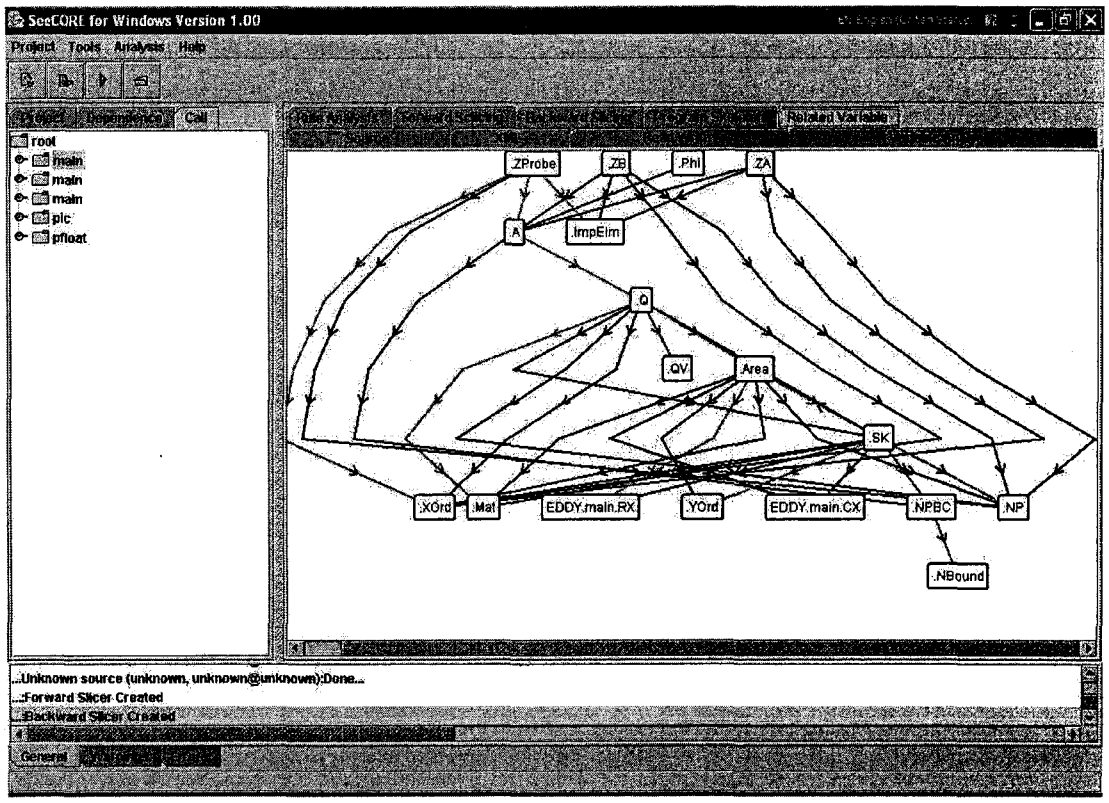

Figure I.1 Concept web

The concept web shows the relationships between the kernel variables which are reflected in the conceptual model. In actual code, a kernel variable could have connections to many other variables including significant ones and insignificant one. Naturally, when a human expert understands a program, his primary focus is those significant variables. The concept web of the significant variables is a simpler but more meaningful map of the conceptual model.

The concept web as shown in Figure I is the diagram illustrating such relationships. In the figure, each variable has been annotated conceptual roles. Syntactically, the relationship between the variables can be any of value-dependence, control-dependence, index-dependence, etc. Conceptually, given the context of finite element method, the relationships are translated into either area vs. coordinates, global matrix vs. vector, global matrix vs. material properties, or material properties vs. coordinates, etc.

Each link between the variables denotes a dependence. The inference based on the dependence can be written in rules and used to infer more conceptual roles.

APPENDIX J. Conceptual Roles of Data in the FEM Code - Eddy

Table J.1 Human compiled conceptual roles of variables in Eddy

Variables	Conceptual Roles of Data
A	The array storing the magnetic vector potential values.
Q	The right hand side vector of the global matrix equation.
$ImpElm$	Records the elements present in the eddy current coil.
$Xord, Yord$	Mesh X and Y coordinates.
$NBound$	Contains boundary node numbers
NP	Connectivity between nodes in elements
Mat	Materials used in the geometry
SK	Complex global matrix
$NCoil$	Coil reference element numbers
$Area$	Area of elements
Phi	Magnitude of the complex valued magnetic vector potential
$ZA, ZB, ZSum, ZProbe$	Impedance values. These are the results.
QV	Complex current density in material
$NPBC$	Indicates which nodes are on the boundary.
RX, CX	Reluctivity in element

Table J.1 includes almost all key elements in the finite element code Eddy. Table J.2 summarizes the conceptual annotations which have been recognized by our tool. In addition to those annotations shown in the table, the syntactic/semantic/conceptual roles recovered by the tool include particular computing patterns (e.g., reduction), active scope of the variables, control structure (e.g., the nested level of loop) etc.

In Table J.2, *dimension* refers to the property related to space, e.g. coordinates, connectivity between points in space etc. *undetermined* means that the tool cannot determine

Table J.2 Programmatically recognized conceptual roles of variables in Eddy

Variables	Conceptual Annotations of Data
A	Critical
Q	Critical, Vector, Property
$ImpElm$	Critical, Property, Result
$Xord, Yord$	Critical, Source, Dimension
$NBound$	Critical, Source Dimension
NP	Critical, Source, Connectivity, Dimension
Mat	Critical, Property, Result, Dimension
SK	Critical, Matrix
$NCoil$	N/A
$Area$	Critical
Phi	Critical, Property, Result
ZA, ZB	Critical, Property, Result
$ZSUM$	Undetermined
$ZProbe$	Critical, Property, Result
QV	Critical
$NPBC$	Critical, Boundary_cond
RX, CX	Critical, Source

whether the variable is important or not. A typical scenario is that there is no explicit pattern to infer the variable is a critical variable; however, apparently it is being involved in important computing. A variable annotated *boundary_cond* is a variable related to nodes on the boundary.

APPENDIX K. Eddy's Program Skeleton

Table K.1 Eddy's program skeleton

Program Skeleton	Comments
eddy.main.:eddy.main.vfp<- eddy.main.:eddy.main.impfp<- eddy.main.:.NPP<- eddy.main.:.A<- eddy.main.:.Q<- eddy.main.:.XOrd<-.XORD; eddy.main.:.YOrd<-.YORD; eddy.main.:.NPBC<-.NBound;	Load initialization data from file. Form the mash, Form the boundary condition.
refer.refer.:.NCoil<- refer.refer.:.NCoil<- eddy.main.:.NCoil<-	Generate the starting reference position of the source coils.
materl.materl.:.Mat<- materl.materl.:.Mat<- materl.materl.:.Mat<- materl.materl.:.Mat<- materl.materl.:.Mat<- materl.materl.:.Mat<- eddy.main.:.Mat<-	Assign the particular material property to each element.
eddy.main.:.ImpElm<- eddy.main.:.ImpElm<-	Build the connection between the probe positions (points in time dimension) and the elements
global.global.:.Area<-.XOrd;.YOrd;.NP; global.global.:.SK<-. EDDY.main.RX,.NP,.YOrd,.Area,.Mat,.XOrd,.SK; global.global.:.SK<- .NP,.Area,.Mat,.XOrd,EDDY.main.CX,.SK,.NP; :.NPBC: .NPBC global.global.:.SK<-.SK; .NPBC; global.global.:.SK<-.SK; .NPBC;	Compute the area for each element Assemble the global matrix
eddy.main.:.Area<- .SK;.XOrd;.YOrd;.Area;.NP;EDDY.main.RX;.Mat;	Compute the area for each element

Table K.2 Eddy's program skeleton (continued)

Program Skeleton	Comments
source.source.:.Q<- source.source.:.Q<-.NP;.Area;.QV;.XOrd;.Mat;.Q eddy.main.:.Q<- .NP;.XOrd;.Area;	Form the vector
cudu.cudu.:.SK<-.SK; cudu.cudu.:.SK<-.SK;	Incorporate the boundary condition (we cannot see the variable annotated with boundary condition here because currently the program skeleton does not visualize dependence relation)
cudu.cudu.:.Q<-.Q;.SK; cudu.cudu.:.Q<-.Q;.SK; cudu.cudu.:.Q<-.Q;.SK; cudu.cudu.:.Q<-.Q;.SK;	Global matrix solver
eddy.main.:.SK<-.SK; .Q;	Global matrix solver (continue)
eddy.main.:.A<-.Q; .NPBC; eddy.main.:.ZA<- .NP;.XOrd;.ImpElm;.A; eddy.main.:.ZB<- .NP;.XOrd;.ImpElm;.A; eddy.main.:.ZProbe<- .NP;.XOrd;.ImpElm;.A eddy.main.:.Phi<-.A;	Calculate the physical properties

APPENDIX L. Complete List of the Domain-Specific Rules Designed for Xinu Analysis

Table L.1 Xinu rules

Domain-Specific Rules	**Comments**
global ( ) -> significant Type = "2" -> a_table a_table & significant -> critical	O.S. code common patterns
TypeName = "routine_address" -> handler & critical TypeName = "devsw" -> device_table & critical TypeName = "dstab" -> disk_table & critical TypeName = "dsblk" -> disk_blk & critical TypeName = "arpent" -> arp_cache & critical TypeName = "dgblk" -> datagram_blk & critical TypeName = "flblk" -> file_blk & critical TypeName = "etblk" -> eth_blk & critical TypeName = "netq" -> net_buffer & critical TypeName = "ptnode" -> queue_node & critical TypeName = "bpool" -> buffer_pool & critical TypeName = "tty" -> tty & critical TypeName = "dir" -> directory & critical TypeName = "pentry" -> process_blk & critical TypeName = "fdes" -> file_dscrp & critical TypeName = "arppak" -> arp_pack & critical TypeName = "sentry" -> semaph & critical TypeName = "qent" -> q_table & critical TypeName = "intmap" -> interrupt & critical TypeName = "mblock" -> mem_blk & critical TypeName = "epacket" -> eth_packet & critical TypeName = "iblk" -> index_node & critical TypeName = "ip" -> ip & critical TypeName = "icmp" -> icmp & critical TypeName = "iblock" -> index_blk & critical TypeName = "freeblk" -> free_blk & critical TypeName = "memusage" -> mem_usage & critical TypeName = "pt" -> ports & critical TypeName = "udp" -> udp & critical	Specific *struct* are specific control block.

Table L.2 Xinu rules (continued)

Domain-Specific Rules	Comments
device_table -> abstract ("device_table") disk_table -> abstract ("disk_table") arp_cache -> abstract ("arp_cache") datagram_blk -> abstract ("datagram_blk") file_blk -> abstract ("file_blk") eth_blk -> abstract ("eth_blk") net_buffer -> abstract ("net_buffer") % tty -> abstract ("tty") directory -> abstract ("directory") process_blk -> abstract ("process_blk") disk_blk -> abstract ("disk_blk") queue_node -> abstract ("queue_node") buffer_pool -> abstract ("buffer_pool") file_dscrp -> abstract ("file_dscrp") arp_pack -> abstract ("arp_pack") semaph -> abstract ("semaph") q_table -> abstract ("q_table") interrupt -> abstract ("interrupt") mem_blk -> abstract ("mem_blk") eth_packet -> abstract ("eth_packet") index_blk -> abstract ("index_blk") index_node -> abstract ("index_node") ip -> abstract ("ip") icmp -> abstract ("icmp") index_blk -> abstract ("index_blk") free_blk -> abstract ("free_blk") mem_usage -> abstract ("mem_usage") ports -> abstract ("ports") udp -> abstract ("udp")	All variables declared with the specific control *struct* type are merged into single concepts.
critical -> ! significant & ! peripheral significant -> ! peripheral undetermined -> ! significant	It is favorable to resolve category conflicts so that the categorization is not confusing.

APPENDIX M. Visualization of Xinu Device Dispatcher and Device Handlers

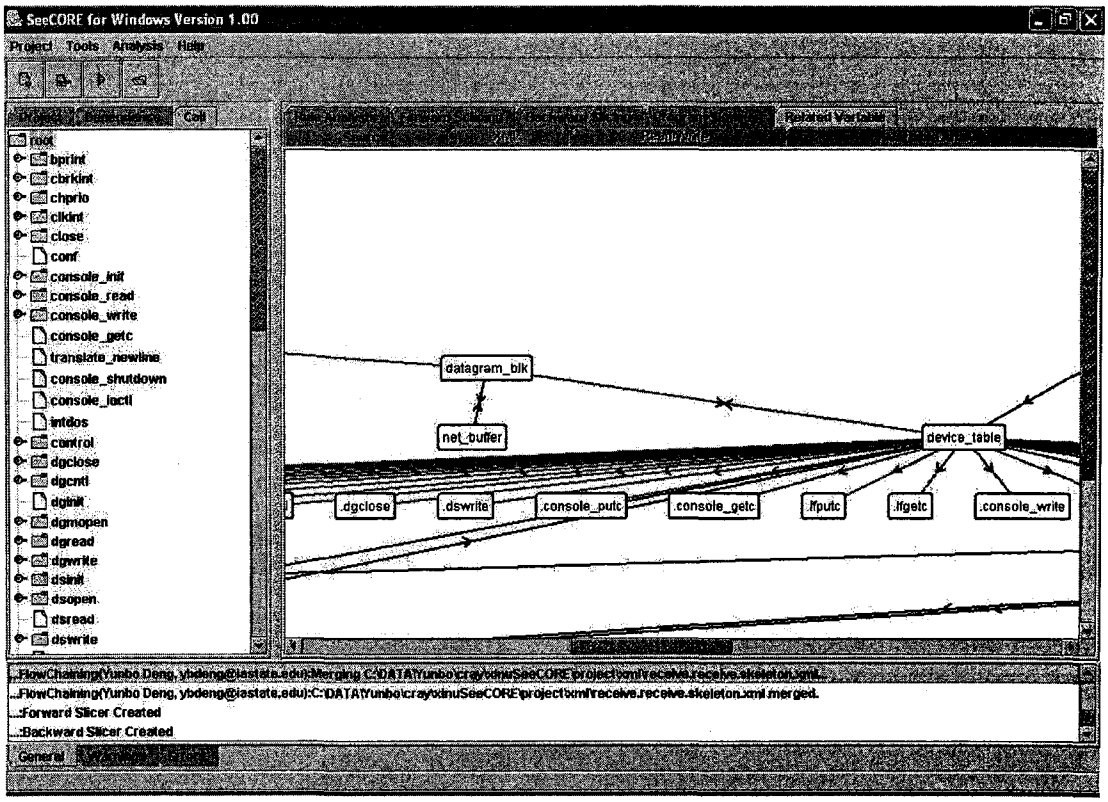

Figure M.1 Xinu device handlers and dispatcher

Limited by the page size, only part of the relationships between the device switch table (device dispatch table) and device handlers are shown in the figure. The graph shows the relationships between the device table and the datagram close (*dgclose*), disk write (*dswrite*), console character output (*console_putc*), console character input (*console_getc*), file character

output (*lfputc*), file character input (*lfgetc*), and console write (*console_write*). The complete graph actually includes all relationships between the device switch table and the device handlers.

In addition, in the graph we can also see the relationship between the device table and the datagram control block (*datagram_blk*) and the relationship between the datagram control block and the network buffer.

APPENDIX N. Visualization of the Relationships between Control Blocks in Xinu Code

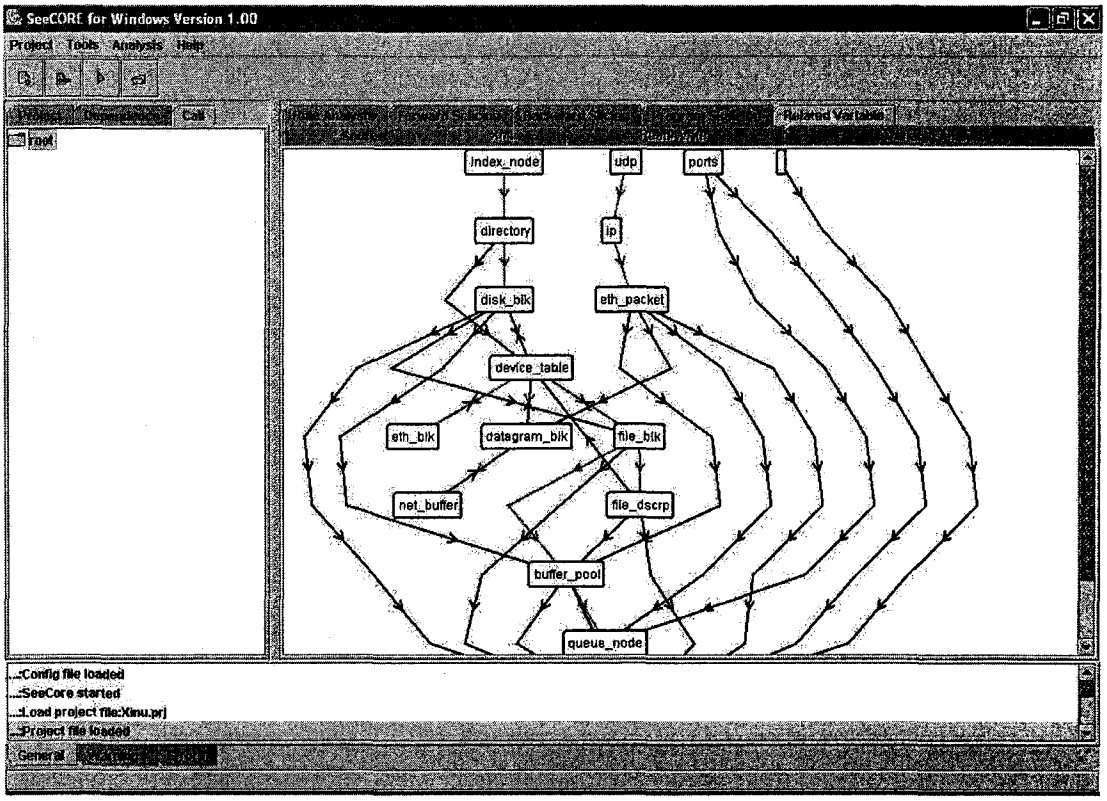

Figure N.1 Visualization of the relationships between Xinu control blocks

The graph mainly shows the relationships between the control blocks in Xinu. Especially it shows the relationships between the device switch table and the datagram, Ethernet, file, disk, and directory. Xinu uniquely treats console, datagram, Ethernet, file, and disk all as devices. It is one of the most difficult parts for a beginner to understand Xinu code.

BIBLIOGRAPHY

[Adelson and Soloway, 1985] J. L. Adelson, & E., Soloway. The role of domain experience in software design. *IEEE Transactions on Software Engineering, SE-11*(11), 1351–1360. 1985.

[Agrawal, 1994] H. Agrawal. On slicing programs with jump statements. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.* Orlando, Florida. 1994.

[Agrawal *et al.*, 1991] H. Agrawal. R. Demillo, and E. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceeding of the ACM Fourth Symposium on Testing, Analysis, and Verification.* pp. 60-73. 1991.

[Anthes and Warne, 1978] R. A. Anthes, and T. T. Warne. Development of hydrodynamic models suitable for air pollution and other mesometeorological studies. *Mon. Weather Review, #106*, pp. 1045-1078, 1978.

[Atkinson, 1999] D. C. Atkinson. The design and implementation of practical and task-oriented whole-program analysis tools, Ph.D. Thesis, Technical Report CS99-618, Department of Computer Science and Engineering, University of California, San Diego, April 1999.

[Atkinson and Griswold, 1996] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools, In *IEEE Proceedings of the 18th International Conference on Software Engineering*, Berlin, pp. 16-27, March, 1996.

[Atkinson and Griswold, 1998] D. C. Atkinson and W. G. Griswold. Effective whole-program analysis in the presence of pointers, In *Proceedings of the ACM SIGSOFT 1998 Symposium on the Foundations of Software Engineering*, November 1998.

[Ball, 1993] T. Ball. The Use of Control-Flow and Control Dependence in Software Tools. Ph.D. Thesis. University of Wisconsin-Madison. 1993.

[Ball and Horwitz, 1993] T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging.* 1993.

[Biggerstaff *et al.*, 1994] T. Biggerstaff, B. Mitbander, D. Webster. Program understanding

and the concept assignment problem. *Communications of the ACM* Volume 37, Issue 5. May 1994.

[Binkley and Gallagher, 1996] D. Binkley, and K. Gallagher, Program Slicing. *Advances in Computers*, Volume 43, 1996.

[Bowdidge, 1995] R. W. Bowdidge. Supporting the restructuring of data abstractions through manipulation of a program visualization. Ph.D. Thesis, Technical Report CS95-457, Department of Computer Science and Engineering, University of California, San Diego, November 1995.

[Bowdidge and Griswold, 1998] R. W. Bowdidge and W. G. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Transactions on Software Engineering and Methodology*, 7(2), April 1998.

[Brand, *et al.*, 1997] M. Brand, A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In *Proceedings of the Fourth International Working Conference on Reverse Engineering.* pp. 144-153. 1997.

[Brand, *et al.*, 1998] M. Brand, A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proceedings of the Sixth International Conference on Program Comprehension*, 1998.

[Brooks, 1994] F. Brooks, Jr. The Computer Scientists as Toolsmith - II. *Computer Graphics*, Vol. 28. pp. 281-287. November 1994.

[Chandrupatla and Belegundu, 2002] Tirupathi R. Chandrupatla and Ashok D. Belegundu. Introduction to Finite Elements In Engineering, 3rd ED. Prentice Hall. 480 pp. ISBN: 0-13-061591-9. 2002.

[Choi *et al.*, 1993] J.D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-introduced aliases and side effects. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages.* ACM. pp 232-245. 1993.

[Choi and Ferrante, 1994] J.D. Choi and J. Ferrante. Static slicing in the presence of GOTO statements. *ACM Transactions on Programming Languages and Systems.* May 1994.

[Comer, 1988] Douglas E. Comer. Operating System Design - The XINU Approach, PC-Edition. Prentice Hall. ISBN 0-13-638180-4. 1988.

[Cutillo *et al.*, 1993] Cutillo, Lanubile and Vissagio, Extracting Application Domain Functions from Old Code: A Real Experience, *IEEE Second Workshop on Program Comprehension*, Capri, Italy, pp. 186 - 192. July 1993.

[Deng *et al.*, 2000] Yunbo Deng, Suraj Kothari, *et al.* ParAgent - A software reengineering

tool for parallel computing, In *Twelfth IASTED International Conference on Parallel and Distributed Computing System*, Las Vegas, November 2000.

[Deng *et al.*, 2001] Yunbo Deng, Suraj Kothari and Yogy Namara. Program slice browser. In *IEEE Ninth International Workshop on Program Comprehension*, Toronto, Canada, May 2001.

[Deng and Kothari, 2002a] Yunbo Deng and Suraj Kothari. Recovering conceptual roles of data in a program. In *Proceedings of International Conference On Software Maintenance.* pp 342- 350. 2002.

[Deng and Kothari, 2002b] Yunbo Deng and Suraj Kothari. Using conceptual roles of data for enhanced program comprehension. In *Proceedings of Ninth Working Conference on Reverse Engineering.* pp 119-127. 2002.

[Desai and Abel, 1972] C. S. Desai and J. F. Abel. Introduction to the Finite Element Method: A Numerical Method for Engineering Analysis New York. 477 pp. ISBN 99-0021371-8. 1972.

[EDG, 2002] Edison Design Group. http://www.edg.com/

[Ernst, 1995] Michael D. Ernst, Slicing pointers and procedures (abstracts), Microsoft Research technical report MSR-TR-95-23, January 13, 1995.

[FEAO, 2002] Finite Element Analysis Online. http://www.fea-online.com/

[Gallagher and Lyle, 1991] K. B. Gallagher and J. R. Lyle, Using program slicing in software maintenance. *IEEE Transaction on Software Engineering*, Vol.17, No.8, pp.751-761, 1991.

[HPFF, 1997] High Performance Fortran Language Specification (Version 2.0), High Performance Fortran Forum, January 31, 1997.

[Jackson and Rollings, 1994] Daniel Jackson and Eugene I. Rollings. Abstraction mechanism for pictorial slicing, In *IEEE Third International Workshop on Program Comprehension.* Page(s): 82 -88. 1994.

[Kazman and Carri' ere, 1999] R. Kazman and S. J. Carri' ere. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering.* April 1999.

[Kennedy, 1981] K. Kennedy. A survey of data flow analysis techniques. In *Program Flow Analysis: Theory and Applications.* S.S. Muchnick. And N. D. Jones. Eds. Englewood Cliffs, NJ: Prentice-Hall. 1981.

[Kiczales *et al.*, 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-oriented programming.

In *proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag LNCS n.1241.Finland. June 1997.

[Klint, 1993] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176-201, 1993.

[Kothari *et al.*, 2002] S. Kothari *et al.* Software tools and parallel computing for numerical weather prediction models, In *Proceedings of Third International Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications*, Fort Lauderdale, Florida. 2002.

[Kozaczynski *et al.*, 1992] W. Kozaczynski and J. Q. Ning and A. Engberts, Program concept recognition and transformation, *IEEE Transactions on Software Engineering*, 18(12), pp. 1065–1075, 1992.

[Kozaczynski and Ning, 1994] W. Kozaczynski and J.Q. Ning. Automated program understanding by concept recognition. *Automated Software Engineering*, 1(1):61–78, March 1994.

[Landi and Ryder, 1992] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the 1992 ACM Conference on Programming Languages Design and implementation.* San Francisco. 1992.

[Lanubile and Visaggio, 1993] F. Lanubile and G. Visaggio, Function recovery based on program slicing. In *Proceedings of International Conference on Software Maintenance*, pp. 396-404. Montreal, Quebec, Canada. 1993.

[Li and Chen, 1991] J. Li and M. Chen. Data alignment phase in compiling programs for distributed memory machines. *Parallel and Distributed Computing*, 13:213-221, 1991.

[Lopes and Kiczales, 1998] Cristina Videira Lopes and Gregor Kiczales, Recent developments in AspectJ. In *ECOOP'98 Workshop Reader.* Springer-Verlag LNCS n. 1543. Belgium. 1998.

[Lyle, 1984] J. Lyle. Evaluating Variations on Program Slicing for Debugging. Ph.D. Thesis. University of Maryland. 1984.

[Lyle and Weiser, 1987] J. Lyle and M. Weiser. Automatic bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications.* pp. 877-883. Beijing, China. 1987.

[Martin and Carey, 1973] H. C. Martin and G. F. Carey. Introduction to Finite Element Analysis - Theory and Applications. McGraw-Hill. New York. 386 pp. ISBN 0-07-040641-3. 1973.

[Martino *et al.*, 1997] B. Di Martino, G. Iannello, H. Zima. An automated algorithmic

recognition technique to support parallel software development. In *Proceedings of IEEE International Workshop on Parallel and Distributed Software Engineering*, Boston (USA), 17-18 May 1997.

[Mitra *et al.*, 2000] S. Mitra, S. Kothari, J. Cho, A. Krishnaswarmy: ParAgent: A Domain-Specific Semi-automatic Parallelization Tool. *HiPC 2000.* pp141-148. Bangalore, India. 2000.

[Murphy *et al.*, 1995] G. Murphy, D. Notkin and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT '95: Third ACM SIGSOFT Symposium on Foundations of Software Engineering*, Software Engineering Notes 20(4), pages 18–28, Washington, DC, October 1995.

[Murphy and Notkin, 1997] G. Murphy and D. Notkin. Reengineering with Reflexion models: A case study". *IEEE Computer*, 17(2):29-36, Aug. 1997.

[Müller *et al.*, 1993] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, James S. Uhl., A reverse engineering approach to subsystem structure identification, *Journal of Software Maintenance*, Vol. 5, No. 4, pp. 181-204, December 1993.

[Müller *et al.*, 1994] H. A. Müller, K. Wong, and S. R. Tilley. Understanding software systems using reverse engineering technology. *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings* (ACFAS 1994). 1994.

[Ning, *et al.*, 1994] J. Q. Ning, and A. Engberts and W. Kozaczynski. Automated support for legacy code understanding, *Communications of the ACM*, 37(5), pp. 50-7,1994.

[Nishimatsu *et al.*, 1999] A. Nishimatsu and M. Jihira and S. Kusumoto and K. Inoue, Call mark slicing: An efficient and economical way to reducing slice, In *Proceedings of the International Conference on Software Engineering*, Los Angels, CA. 1999.

[Ottenstein and Ottenstein, 1984] K. Ottenstein and L Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* 1984.

[Paul and Prakash, 1994] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463-475, 1994.

[Quilici, 1995] Alex Quilici. Toward practical automated program understanding. In *Proceedings of the 1995 IJCAI Workshop on AI and Software Engineering (AISE-95)*, August 1995.

[Rich and Wills, 1990] Charles Rich and Linda Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software.* Vol 7 No. 1. pp 82-89. January, 1990.

[Richner and Ducasse, 1999] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings of the International Conference of Software Maintenance (ICSM).* pp. 13-22. August 1999.

[Reddy, 1993] J. N. Reddy. An Introduction to the Finite Element Method. McGraw-Hill Book Co., New York. 495 pp. ISBN 0-07-051346-5. 1984, 1993.

[Robillard and Murphy, 2000] Martin P. Robillard and Gail C. Murphy. An exploration of a lightweight means of concern separation. Position paper for the *ECOOP'2000 Workshop on Aspects and Dimensions of Concerns*, June, 2000.

[Sinha *et al.*, 1999] S. Sinha, M. Harrold, G. Rothermal. System-Dependence-Graph-based slicing of programs with arbitrary interprocedural control flow. *In International Conference on Software Engineering.* 1999.

[Snelting and Tip, 1994] G. Snelting, G. Tip. Reengineering class hierarchies using concept analysis, In P*roceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 99-110, November 1994.

[SNiFF, 2002] SNiFF+ Available online http://www.windriver.com/products/html/sniff.html by March 2002. [Storey, *et al.* 1996] M.-A.D. Storey, H.A. Müller and K. Wong. Manipulating and documenting software structures. In *Software Visualization*, Vol. 7. 1996.

[Sosic and Abramson, 1997] R Sosic and D.A. Abramson. Guard: a relative debugger, *Software - Practice & Experience.* Vol 27(2). pp. 185-206. February 1997.

[Tilley *et al.*, 1994] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, pages 501-520, December 1994.

[Tilley, 1995] S. R. Tilley. Domain-retargetable reverse engineering. Ph.D. Dissertation, Department of Computer Science, University of Victoria, 1995.

[Tip, 1995] Frank Tip. A survey of program slicing techniques, *Journal of Programming Languages*, vol. 3, 1995.

[W3C, 2002] W3C: Extensible Markup Language (XML) http://www.w3.org/XML/

[Weiser, 1984] Mark Weiser, Program slicing, *IEEE Trans. On Software Engineering*, SE-10(4), pp. 352-357. July, 1984.

[Weiser and Lyle, 1986] M. Weiser and J. Lyle, Experiments on slicing-based debugging tools, In *Proceedings of the 1st Conference on Empirical Studies of Programming.* pp. 187-197. 1986.

[Wong, *et al.*, 1995] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural redocumentation: A case study. *IEEE Software*, pages 46-54, January 1995.

[Wong, 1998] K. Wong. The Rigi User's Manual - Version 5.4.4. Available online http://www.rigi.csc.uvic.ca/Pages/publications.html by March 2002.

[Woods, *et al.*, 1999] S. Woods, S. J. Carri' ere, R. Kazman. A semantic foundation for architectural reengineering and interchange. In *Proceedings of International Conference on Software Maintenance.* pp. 391-398. Sept. 1999.

[XSLT, 2002] XSLT - Extensible Stylesheet Language Transformation http://www.w3.org/TR/xslt

[Zhao, 2000] J. Zhao. A slicing-based approach to extracting reusable software architectures. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering*, pages 215-223. February 2000.