# Conceptual Modules: Expressing Desired Structure for Software Reengineering

by

Elisa L. A. Baniassad

B.CSc, Technical University of Nova Scotia, 1995

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

**Master of Science**

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

**The University of British Columbia**

December 1997

# Abstract

Many tools have been built to analyze the source code of software systems. Most of these tools do not adequately support software reengineering activities because they do not allow a software engineer to analyze both existing and desired software structures.

This thesis describes the *conceptual modules* approach and supporting tool that aids the engineer in the investigation and analysis of desired structure with relation to the existing structure of source code. This approach allows a selected subset of lines of source to be treated as a logical unit. This subset is referred to as a conceptual module. The lines of code that comprise a conceptual module need not be contiguous, nor must they be related in any way in the source. Using variable dependence and control transfer information extracted from the source, the tool analyzes the conceptual module's constituent lines of code to determine its interface. Additionally, the data- or control-flow between two or more conceptual modules can be examined as a means of eliciting the relationships between the modules and between conceptual modules and the source. To allow the necessary flexibility in analysis, the functionality of the tool can be tailored through a programmatic query language component.

The usefulness of the tool has been investigated in two different ways. First, the tool was applied to several different reengineering scenarios: restructuring from procedural to object-oriented program design, re-modularizing code in an existing program with little structure, and extracting a portion of source for reuse. For each scenario, several existing program understanding tools were also applied to provide a basis of comparison between existing approaches and the conceptual modules approach. Second, the tool was successfully applied to actual reengineering tasks by two different groups of users. One group eliminated unnecessary parts of a system's source to improve efficiency and to enable parallelization of a 47,000 line, 56-file software package. The other group performed analysis on a procedural program so as to better understand how to transform the existing source into an object-oriented version.

# Contents

# List of Tables

# List of Figures

# Acknowledgements

I'd like to thank Yvonne Coady for her participation in the user-studies of this tool. I'd like to thank Rob O'Callahan for the use of Lackwit. Also, great thanks go to Ryan and Christoph, for all their help and support.

ELISA L. A. BANIASSAD

*The University of British Columbia*
*December 1997*

# Chapter 1

# Introduction

Software systems change over time. They may change because the fundamental functionality requirements for the system change, or they may change because modifications to the environment in which the software is running dictate a change in the mechanisms that produce the desired behaviour. The changes systems undergo have been referred to as software aging [17]. Typically, the changes that must be made as part of the aging process do not integrate easily with the existing program structure. As a result, software engineers must perform reengineering activities to accomplish the desired changes.

Reengineering "is the examination and modification of a system to reconstitute it in a new form and also the subsequent implementation of the new form" [4, page 15]. Reengineering is often necessary regardless of the abilities or intentions of the original software designers. Even if the original developers designed the system with change in mind, the assumptions of what would change can not generally isolate all types of change that occur over the lifetime of the system. Parnas has described the inevitability of this situation in the following way:

Even if we take all reasonable preventive measures, and do so religiously, aging is inevitable. Our ability to design for change depends on our ability to predict the future. We can do so only approximately and imperfectly. Over a period of years, we will make changes that violate our original assumptions. Documentation, even if formal and precise, will never be perfect. Reviews, will bring out issues that the designers miss, but there are bound to be issues that the reviewers miss as well. Preventive measures are worthwhile but anyone who thinks that this will eliminate aging is living in a dream world. [17, pages 283-284]

As part of reengineering activities to facilitate change, there is often a need to analyze a portion of computation that does not necessarily comply with the existing source code structure. Isolating such a piece of code for analysis and characterization is difficult to do manually because the lines of code to be extracted are often scattered around the source. Semantically analyzing all those portions of code manually would indeed be an enormous task.

For instance, when trying to reuse software, it is common that a software engineer needs to understand if a collection of lines of existing code can be extracted for reuse. Determining the interface to the desired code fragment can be difficult. It requires the engineer to undertake the arduous task of identifying the input arguments and output arguments of the code fragment. In addition, there may be data- or control-flow dependences from the isolated portion of code to many other places in the source code, and some of these places should, perhaps, also be included in the subset being extracted. Determining all this information requires complex knowledge of how the individual lines of code relate, both to each other and to the remaining code base.

To examine portions of code gathered from around the source, it is useful for the engineer to collect all the portions into one subset, and to analyze that subset as one logical unit regardless of the organization of the existing code. This grouping allows the engineer to better assess the interface to a portion of computation, as well as relationships between different computational elements. Currently, engineers may choose from a range of source analysis tools to aid these reengineering tasks [19, 10, 2, 6, 14]. These tools fall into two basic categories: reverse engineering tools and program databases.

Reverse engineering tools help the user abstract structural information from the source [14, 11]. They allow the user to create new structures out of programmatic building blocks, such as procedures, functions, and classes. Because these tools are geared towards clustering existing source structure they do not help the user ask questions about portions of computation isolated from various existing programmatic blocks. The user is confined to working within the existing structure of the program.

Program databases [18] and type inferencers [16] provide the user with the ability to ask about particular variables or procedures. These tools hold the user to a fine level of granularity. They generally produce a large amount of non-abstracted information in response to a query. This lack of abstraction can cause difficulties when applying the tools to large programs or large portions of computation.

The thesis of this research is that many software reengineering tasks can be performed in less time and with higher confidence if direct support for forming and analyzing desired software structure is provided at the source level. The user needs the ability to understand how a subset of their system relates to the rest of the source code, to characterize the interface to that subset, and to understand relationships between two or more subsets of the code. To investigate this approach, the *conceptual module* software analysis technique and tool have been developed.

The conceptual module approach and tool allow the software developer (user) to overlay a desired logical structure on the existing source code and to analyze that structure. A subset of code used in analysis is referred to as a *conceptual module*. Through construction of conceptual modules, an engineer may describe and analyze the desired structure of source code. The lines of code contributing to a conceptual module can be from anywhere in the original source, and do not have to be related in any way. Using variable dependence and control transfer information extracted from the source, the conceptual module tool analyzes the conceptual module's constituent lines to determine its inputs, outputs and calls. To support the user in examining the relationships between conceptual modules, a programmatic query interface is supplied. This interface allows users to form their own queries using an object-oriented library. Using this interface, users can access the necessary information to help them perform their specific task.

The tool has been designed as an aid to software reengineering and reuse tasks. The focus of the tool is not to fully automate reengineering activities, but rather to allow the engineer to assess a desired structure.

## 1.1 A Sample Reengineering Activity

To clarify some of the information needs of a software engineer performing a reengineering activity, the task of isolating and forming a new input filter component from GNU `sort`[1] is described. The `sort` system is in the Unix pipe-and-filter style [21]

[1]The GNU `sort` program used in this analysis was from the 1.21 version of the GNU textutils distribution.

and comprises about 5100 lines of C code split across 29 files. The majority of the code specific to the sort functionality resides in a 1700-line file called `sort.c`.

A software engineer may wish to form and extract an input pipe component from `sort` to help build a new program in the same architectural style. The target input pipe component would consist of a set of procedures acting on variables representing the state of the pipe.[2] In some cases, the code that is to be extracted into a procedure of the target component is a set of contiguous lines. In these cases, the formation of the procedure is relatively straightforward, and specialized tools can be applied to automate the task [9]. Other times, the lines of code to be included in the new procedure are split across existing procedure boundaries.

In `sort`, for instance, the engineer determines, based on a perusal of the code, that the `fp` variable declared and used in the 351-line `main` function contributes to the initialization of the input pipe functionality. By tracing the use of the `fp` variable, the engineer determines that code from the sort function also contributes to the desired initialization procedure of the target input pipe component.

When the target procedure crosses existing structural boundaries, automated support to form the component is not available. Instead, the engineer must analyze the identified lines of code to determine the interface to the desired procedure, and to determine any additional source lines that must be included to produce the desired computation. Determining this information requires the engineer to analyze the lines of code for two kinds of interactions: interactions within the lines of code representing the new structure, and interactions between the new structure and the remaining system.

[2]Although it may seem trivial to build an input filter, there are a number of subtleties that can arise. The source for GNU `sort`, for instance, deals with cases in which the input and output filenames providing data to the pipes are the same.

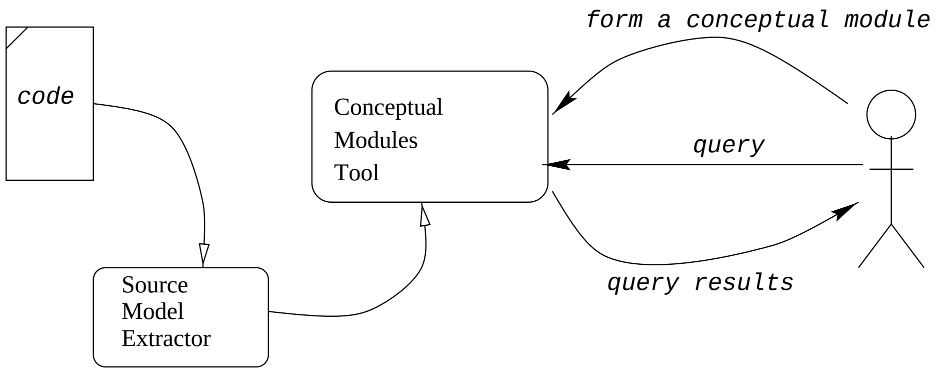


Figure 1.1: Process for using the conceptual module tool

Analysis of these types of interactions are straightforward when using the conceptual module tool. Figure 1.1 illustrates the approach. The engineer first uses a tool to extract information—a source model—from the source code. The engineer then describes the target structure as one or more conceptual modules, where each conceptual module consists of a set of source lines.

The tool provides direct support for querying about the interface of a conceptual module. Using semantic information about the source, it returns a list of input variables, output variables, and local variables of the conceptual module. It also shows where the values of the input variables were last defined outside the conceptual module, and where the output variables are next used outside the conceptual module. This information identifies the uses and definition points of the variables that link the subset of code to the rest of the source. It also produces a list of the calls made from code in the conceptual module. In the case of `sort` for example, to help form the desired input pipe initialization procedure, the engineer specifies a set of lines of code[3] in the source that provide this functionality. These lines of code can be added by specifying individual lines, ranges of lines, variables, or proce-

[3]The following lines were included in the module: 228, 239, 245-249, 1741, 1796, 2071, 2081, 2098, 2104, 2107, 2111, 2124, 2131, 2137, 2146, 2148.

dures. In this case the user would type the individual lines into the user-interface of the tool. The user interface provided for performing these additions in the current implementation of the tool is described in Section 3.4.

The tool responds with the analyzed information shown in Figure 1.2. The output shown is organized as follows. The input variables are shown as a list of variables, each with a list of line numbers following them. Each of the line numbers represents a definition of the variable that is used in the conceptual module. The output variables are shown in the same organization as the input variables. In the case of these variables, the line numbers represent the next use of the variable outside the conceptual module. The local variables are shown as a list of variable names. Finally, the calls information shows the procedures called from inside the conceptual module. These calls can potentially be to lines of code also contained in the conceptual module.

Should a user require more details on the information returned by the tool, they may make use of the query language component of the tool. For instance, the user may further inspect relationships between lines of code within the conceptual module and between the conceptual module and the source, perhaps to determine if a particular line of code indirectly affects code in the conceptual module. This can be illustrated through the following example. The user may wish to determine if a variable in the merge functionality of the `sort` program is indirectly affected by some lines of code in the sort procedure, in an endeavor to ensure that data being merged has been sorted. Specifically, the user may wish to determine if the variable `nfps` in the mergefps function is affected by the core sort functionality appearing in the sort procedure, or in the sortlines procedure.

The user makes a conceptual module `merge` and adds the `nfps` variable to

```
The input variables are:
        sort.c main.nfiles: sort.c 2073, sort.c 2041.
        sort.c main.ofp: sort.c 2143, sort.c 2140.
        sort.c main.minus: sort.c 2074, sort.c 2041, sort.c 1742.
        sort.c main.files: sort.c 2074, sort.c 2041.
        sort.c main.i: sort.c 2043, sort.c 1798.
        sort.c main.tmp: sort.c 2126, sort.c 2125.
        sort.c sort.buf: sort.c 282, sort.c 250, sort.c 239.
        sort.c main.outfile: sort.c 1989, sort.c 1982, sort.c 1742.
        sort.c errno: sort.c 589, sort.c 479, sort.c 467, sort.c 458,
                      sort.c 435, sort.c 414, sort.c 2162, sort.c 2160,
                      sort.c 2157.
        sort.c sort.nfiles: sort.c 279.
        sort.c main.argc: sort.c 1753, sort.c 1737.

The output variables are:
        sort.c main.mergeonly: sort.c 2145.
        sort.c sort.ofp: sort.c 292, sort.c 275, sort.c 260.
        sort.c main.nfiles: sort.c 2041, sort.c main.ofp, , sort.c 2156.
        sort.c main.checkonly: sort.c 2077.
        sort.c main.minus: sort.c 1742.
        sort.c instat: sort.c 2115.
        sort.c errno: sort.c 589, sort.c 479, sort.c 467, sort.c 458,
                      sort.c 435, sort.c 414, sort.c 2162, sort.c 2160,
                      sort.c 2157.
        sort.c sort.nfiles: sort.c 279, sort.c 259.
        sort.c fp: sort.c 2136, sort.c 2129, sort.c 2128, sort.c 2127.
        sort.c sort.fp: sort.c 279, sort.c 259.

The local variables are:
        sort.c sort.files

Control transfers out of input_pipe:
        call to xmalloc at sort.c 1796.
        call to check at sort.c 2081.
        call to exit at sort.c 2081.
        call to strcmp at sort.c 2104.
        call to fstat at sort.c 2107.
        call to stat at sort.c 2107.
        call to strcmp at sort.c 2107.
        call to error at sort.c 2111.
        call to xfopen at sort.c 2124.
        call to error at sort.c 2131.
        call to merge at sort.c 2146.
        call to sort at sort.c 2148.
        call to xfopen at sort.c 247.
        call to fillbuf at sort.c 248.
```

Figure 1.2: Tool output for the conceptual module input-pipe

the module. Adding a variable means that all the lines where the variable is used or its value is set are added to the conceptual module. The user then writes the query shown in Figure 1.3 to obtain a list of all lines and variables that affect the conceptual module. The user examines the returned lists to see if any of the lines or variables are in the core sort functionality. The user can see that many of the variables listed are in the sort procedure. In particular, the `lines` variable in the sort function is a core variable of the sort functionality, and line 258—the call to the sortlines procedure—is included in the list of lines affecting the values in merge.

The conceptual module tool enables the user to examine the code at a fine-grained level and to approach the extraction in a bottom-up manner: looking first at lines of code, and then at conceptual modules as abstractions containing those lines.

## 1.2 Overview

The remaining chapters of this thesis are organized as follows. Chapter 2 describes related work, reviewing the different existing approaches to program analysis and reengineering. Chapter 3 explains both the intent of the approach and its implementation. Chapter 4 presents sample scenarios and case studies to provide evidence that desired structure analysis is helpful for different types of reengineering tasks. Finally, Chapter 5 summarizes the work and presents possible extensions. This chapter also discusses choices made in the implementation of the tool.

## Query

```
                 /* get the already formed abstraction*/
   Abstraction A = Abstraction.Get(``merge'');
                 /* perform the initial analysis
                 true means to show the results of the analysis*/
   Analyzer.Initial(A, true);
                 /* find all the lines
                 contributing to the conceptual module's value*/
   SET chain = DefUse.find_full_chain(A);
                 /* translate the lines into variables */
   SET vars = DefUse.GetVarNames(chain);
                 /* print out the variables*/
   vars.print(``These are the variables that affect merge'');
                 /* this prints out the line numbers */
   chain.print(``These are the line numbers:'');
```

## Tool Output

```
These are the variables that affect merge
...  sort.c program_name, sort.c sort.buf, sort.c sort.i,
sort.c sort.lines, sort.c sort.n_temp_files, sort.c sort.node,
sort.c sort.ntmp, sort.c sort.ofp, sort.c sort.tempfiles,
sort.c sort.tfp, sort.c sort.tmp, sort.c sortalloc, ...
End.

These are the line numbers:
...sort.c 290, sort.c 260, sort.c 1872, sort.c 2009, sort.c 508, sort.c 283,
sort.c 1855, sort.c 276, sort.c 1477, sort.c 1298, sort.c 1457, sort.c
1454, sort.c 302, sort.c 269, sort.c 1370, sort.c 1532, sort.c 239, sort.c
1972, sort.c 1958, sort.c 492, sort.c 250, sort.c 1793, sort.c 1790, sort.c
273, sort.c 243, sort.c 1591, sort.c 1753, sort.c 1496, sort.c 1479, sort.c
266, sort.c 1358, sort.c 289, sort.c 1335, sort.c 1413, sort.c 1315, sort.c
1410, sort.c 272, sort.c 242, sort.c 1989, sort.c 235, sort.c 2036, sort.c
288, sort.c 1848, sort.c 258, sort.c 503, sort.c 2013, sort.c 228, sort.c
1369, sort.c 1349, sort.c 1525, sort.c 1346, sort.c 264, sort.c 1309,
sort.c 1306, sort.c 1997, sort.c 1798, sort.c 240, sort.c 1792, sort.c 306,
sort.c 1937, sort.c 2024, sort.c 1758, sort.c 291, sort.c 1368, sort.c
1446, sort.c 509, sort.c 1348...
End.
```

Figure 1.3: Query and result: all dependences of the merge conceptual module

# Chapter 2

# Related Work

Many program understanding approaches have been developed to aid reengineering tasks. Some approaches attempt to characterize the existing code structure. For instance, reverse engineering techniques, program databases, and type-inferencers attempt to analyze the current structure of software and to provide views of the analyzed information that are more understandable for the user. Software architecture, on the other hand, provides a formalism with which to express system design. Other approaches such as slicing allow the user to reduce the amount of existing code viewed while performing a particular task.

As discussed in Chapter 1, there is a need among software engineers performing various reuse and reengineering tasks to both break apart programmatic building blocks so as to express a desired structure made up of the blocks' constituent lines of code, and also to analyze that desired structure with relation to the existing structure. This chapter outlines the characteristics of current approaches to help engineers understand code prior to reengineering and reuse tasks.

## 2.1 Slicing

Program slicing is a method for automatically decomposing programs by analyzing their data flow and control flow. [25, page 352]

Starting from a subset of a program's behavior, slicing reduces the source of the program to a minimal form that still produces that behavior[25]. Slicing allows the user to pick a program point and compute a subset of the program that would result in the same execution at that point.

The aim of slicing is to determine how information flows through a program to obtain a specific value at a particular point. There are two basic types of slices: static and dynamic. Static slices use only statically available dependence information, relying on data-flow and control-flow dependence information. "In the case of dynamic program slicing, only the dependences that occur in a *specific* execution of the program are taken into account" [23, page 3].

Since reuse and restructuring are more often done in terms of static views of the code, the form of slicing discussed is static slicing. This section both outlines the usefulness of slicing, and compares static slicing to its prominent variants, chopping and interface slicing.

### 2.1.1 Static Slicing

In static slicing, the slice criterion "identifies one or more variables at a given line, and the slice is a subprogram whose statements might affect the value of those variables just prior to execution of that line"[10, page 2]. Most static slices are computed "by gathering statements and control predicates by way of a backward traversal of the program, starting at the slicing criterion" [23, page 2]. Forward static slices [3] may also be computed through a forward traversal of the program, from a program point.

Slicing is useful for tasks such as "program understanding, program chopping, debugging, maintenance, testing, and merging" [19, page 41]. As described by Weiser, the "usefulness [of slices] shows up in testing, parallel processor distribution, maintenance, and especially debugging" [25, page 352]. More specifically, when debugging, it is useful to know how a variable at a particular point obtained its value.

Slicing performs analysis that does not align itself with the existing source structure. Although a slice cuts across existing structure, the user has no control of the slice content. Slices consist of all lines of code that would be executed to produce the same executed result; the user cannot select single lines within a slice. This limits the flexibility of the user in expressing of desired structure since it prevents the user from isolating individual lines of code.

Slicing does not provide direct support for examining desired program structure. In most implementations of slicing tools, a user can perform set operations on slices that have been computed. A user is able to intersect several slices to determine the lines the slices have in common, however, there is no way to determine data-flow and control-flow between slices to ascertain the resultant programmatic structure expressed by the slices.

### 2.1.2 Chopping

In 1994, Jackson and Rollins introduced a variant of slicing: chopping [10]. Chopping enables the user to determine the program elements that transmit data from one program point to another, from a *source* to a *sink*. Chopping exposes and makes use of the internal structure of a program dependence graph in an intent to gain precision that was not present in the original slicing techniques.

In the original form of chopping, there was no method for inter-procedural chopping. This functionality was intentionally omitted based on the opinion that "for reverse engineering at least, analysis should be modular, respecting the structure of a program [and that] since programmers tend to approach a new program one procedure at a time, it seems that a reverse engineering tool should do the same" [10, page 5]. Using this technology, if a user wants to chop into a procedure call, they have to use the procedure call information in a chop to formulate further chops of the different procedures.

In 1995, Reps and Rosay introduced unrestricted inter-procedural chopping to generate a chop of an entire program rather than chopping one procedure at a time [19]. This approach overcomes Jackson and Rollins' original self-imposed limitation. Interestingly, Reps and Rosay do not directly address their opposition to the opinion held by Jackson and Rollins. Although Reps and Rosay clearly and actively disagree with the statement that reverse engineering tools should model a programmer's methodologies in examining only one procedure at a time, they do not contest the idea that analysis should respect the structure of a program.

Chopping is more flexible than slicing. Theoretically, a user could isolate a component by selecting a group of lines of code. The user could do this for more than one component. The relationships between the components could then be analyzed by chopping with all the lines in one component as the source and all the lines in the other component as the sink. The user could then examine the lines in the chops to understand the data- and control-flow relationships between the two components. The user could also perform these chop operations between lines within an isolated component to understand the relationships between lines within the component.

The user would, however, have to manually correlate all the information returned from these chops. To interpret the output of chops that pertain to the relationship between two components, the user would have to do extensive filtering on this output to glean the interface of each of the components and the calls made between the components. It would likely be difficult to determine the dependences on the component from the rest of the source.

### 2.1.3 Interface Slicing

Interface slicing allows the user to identify a subset of the program's interface to be reused [2]. The creators of interface slicing, Beck and Eichmann describe interface slicing, and distinguish it from traditional slicing in the following way:

> Intuitively, an *interface slice* may be viewed as a subset of the behavior of a module, just the original notion of a conventional slice. However, while a conventional slice seeks to isolate the behavior of a specified set of program variables, even across module boundaries, an interface slice seeks to isolate specified behaviors which a given module exports to its containing software system. [2, page 56]

Interface slicing was not intended as a debugging aid as was traditional slicing. Instead, it was developed as a reengineering technique to both enhance the reusability of portions of source code, and also to improve the quality of the code that is created from reused components. It does this by allowing users to single out and select the portions of a program's interface they wish to reuse. While this does address the reuse needs of some reengineering efforts, interface slicing does not allow partial reuse of computation since there is no provision for the user to take only parts of procedures or to characterize their interface.

## 2.2 Program Databases

Program databases allow the user to retrieve information about variables, procedures, functions and other elements of a program's structure. This information is generally retrieved through a query interface. This section considers two representative examples, xrefdb [18], a cross-reference server, and GraphLog [6], a visual query language.

### 2.2.1 xrefdb - Field cross-reference server

xrefdb is the Field programming environment's cross-reference utility [18]. Given an executable for a system,[1] a directory structure, or a single file, xrefdb parses the code and builds a relational database of information extracted. The xrefdb tool provides a relational algebra interface for accessing the information in the database.

Using xrefdb, a user can determine information about particular lines of code, such as calls that are made on a line, and variables altered or used on a line. Similarly, a user can ask about particular programmatic components to find out information such as all call points to a particular procedure or all uses of a variable.[2]

The xrefdb tool does not provide support for expressing a desired structure of source. It provides no mechanism for grouping lines of code and gathering data on them as a unit. More fundamentally, a user cannot talk about an arbitrary

[1]Executables must be compiled with debugging information.
[2]This information is based on the syntactic analysis

computation.

### 2.2.2 GraphLog

GraphLog [6] addresses both the need to understand the design of a system, and also the need for software visualization tools. GraphLog is a query language that is intended to ease the investigation of complex relationships between elements of a software system.

> the designer needs to be able to store all the relevant facts and relations in a database based on these, query and visualize this information, and interactively modify the packaging structure.
>
> GraphLog fits naturally within this framework for two main reasons. First, it reduces the cognitive gap between how queries are formulated and how results are displayed. Second, it improves the productivity of formulating complex queries [6, page 139]

In GraphLog, a query is expressed as a graph specifying a pattern. Given a query the system determines all instances of the pattern expressed by the graph that exist in the database. Using the GraphLog graphical language, the user may create new modules and rearrange variables between them. Pattern matching queries performed by the user operate on these modified packaging schemes as well as on the original scheme.

Although sub-routines and variables can be rearranged within and between modules, they cannot be broken up into single lines of code: fragments smaller than procedures and variables cannot be manipulated. GraphLog does not provide fine-grained support for expressing a desired reengineered structure in that there is no mechanism with which to isolate non-contiguous lines of code as one unit of computation and analyze it as such.

## 2.3 Type Inferencer: Lackwit

Type inferencing involves ignoring the types assigned variables by the programmer, and inferring the types of variables through source code analysis. Lackwit is a type inferencing tool that can find "abstract data types, detect abstraction violations, find unused variables, functions and fields of data structures, detect simple errors in operations on abstract data-types, and locate sites of possible references to a value" [16, page 338].

Lackwit achieves these goals by analyzing the inferred type of a given variable or textual expression, locating other occurrences of that type, and forming conclusions about its usage and behavior.

With these capabilities, the tool is a very useful reverse engineering technique. It provides the user a scalable approach to obtain a great deal of information about a program in terms of information that is relevant when attempting to understand source, or even modify portions of source: interactions between sub-routines, uses of variables, instances of types, etc.

As with the other reverse engineering and program understanding approaches, this tool does not enable the user to ask questions about a particular portion of the code as one logical unit and hence cannot express a desired structure for source. The fixed level of granularity means that there is no capability for breaking up variables and procedures, and no way to cluster them to form coarser-grained abstractions.

## 2.4 Reverse Engineering Tools

Reverse engineering tools help the user abstract structural information gathered from the source. Two examples of reverse engineering techniques are discussed.

### 2.4.1 Rigi

Rigi is a reverse engineering tool and technique [13]. Rigi was designed to

> assist the designers, programmers, integrators, and maintainers in defining, exploring, understanding, and comprehending the structure of large, integrated, evolving software systems [13, page ii].

Rigi is a semi-automated technique in which a user repeatedly determines criteria to cluster elements from a displayed graph of structural information. The criteria may be based on characteristics of the graph or from features of the source, such as naming conventions.

> In Rigi, parsing the subject software system results in a flat, resource flow graph that can be manipulated using a graph editor. The next phase is semi-automatic and involves pattern recognition skills, where the reverse engineer identifies subsystems in the flat graph that form meaningful abstractions. These subsystems are collapsed to build multiple, layered hierarchies of abstractions. [22, page 606]

In the Rigi environment, a user may perform pre-defined queries on the interactions between two clustered elements. There is no provision for programmatic analysis of the clusters with relation to the existing source. Similar to GraphLog, users can move nodes between clusters, but nodes represent existing structural entities, not lines of code, and thus cannot be broken apart.

### 2.4.2 ManSART

ManSART is a semi-automated approach for recovery of architectural descriptions from a system's code base [27]. ManSART displays graphical views of recovered structure to the user.

The architectural views are created by recognizers that extract and analyze information from an abstract syntax tree (AST) of the system. The views include links back to the source contributing to a component or connector in the view. To facilitate the use of views created, a set of view manipulation operators have been defined that can, among other things, merge views and build hierarchies of views. These manipulation operators allow a user to access the source information through a predefined set of tests called *containment analysis*. These tests determine when an element of a view contains or overlaps another based on the underlying source information.

It is only through these pre-defined set of tests, however, that an engineer can query the relationship between the abstracted and existing structure. This means that the user cannot tailor the functionality of the tool to suit their own specific program analysis needs. There is also no means for analyzing or expressing the desired source structure.

## 2.5 Reflexion Models

To aid the user in understanding the structure of a system's source code, the Reflexion Model technique[15] summarizes information extracted from the source in terms of a high-level box-and-arrow diagram of the system specified by the engineer. By placing source entities in named "buckets", the user may define their own high-level model of the source, and then may rely on the tool to show how the model maps to the source structure. This technique is iterative in that the user may successively apply the technique to achieve a more accurate model of its structure.

This approach acts both as an iterative program understanding technique, and a target structure analysis technique. Using the approach, the user may move existing structural elements around and between the buckets. The user may also examine the interactions and dependences between buckets with relation to the new, or target, structure of the code.

The user cannot define their own queries to analyze the structure of the system, and no data-flow analysis is performed.

## 2.6 Software Architecture

The goal of software architecture is to provide the engineer with a means of formally describing a high-level view of a system[21]. This provides the benefit of being able to check for certain properties about a system at a high-level even before code is written. The designer can use an architectural description language [5] to represent the design of the existing program. The user may then use automated tools to perform analysis, such as model-checking, on the representation.

In some respects, it seems as though this approach should give a user the ability to abstract out certain portions of code, either to build new systems out of existing components[8] or to make a component out of that portion and then perform analysis to see how it fits into the existing system.

However, this top-down approach has the following two limitations. First, the user needs to begin with a complete architectural description of a system to be able to abstract out a component of it. A complete description is needed to enable analysis of the relationship between the portion to be extracted and the rest of the existing source. Because the full description is required, the user cannot just look at the existing source code and decide which parts should be collected and treated as a logical unit, instead they must work from a higher-level view of the system.

Second, since the architectural description does not have explicit ties to portions of source code, once a component has been abstracted and analyzed, the user must still map the analysis back to the low-level code. This mapping may be error prone, because the architectural description of the source may not truly reflect the existing system. Also, if the user is dealing with a complicated abstraction it is likely that portions of computation may be missed both when performing this mapping and when abstracting out the computation.

Software architecture, thus, does not help the user to navigate through source code in a bottom-up manner. This limits the use of architectural information to help a user better understand the interactions of a particular portion of source code to the rest of the system, or to other subsets. When attempting to reuse, remove, or re-organize source code, it is imperative that the user be provided with fine-grained information about the impact of any changes. By definition, high-level analysis techniques cannot provide this information since they are concerned with providing a user a view of the overall design of a system.

# Chapter 3

# Conceptual Module Approach and Tool

The conceptual module approach allows a software engineer to identify lines of non-contiguous code from around a system's source, to treat those lines as one logical unit or conceptual module, and then to make tailored queries to examine the conceptual modules—both individually, and with relation to each other. The approach provides a fine-grained, bottom-up approach to source code analysis to aid reengineering.

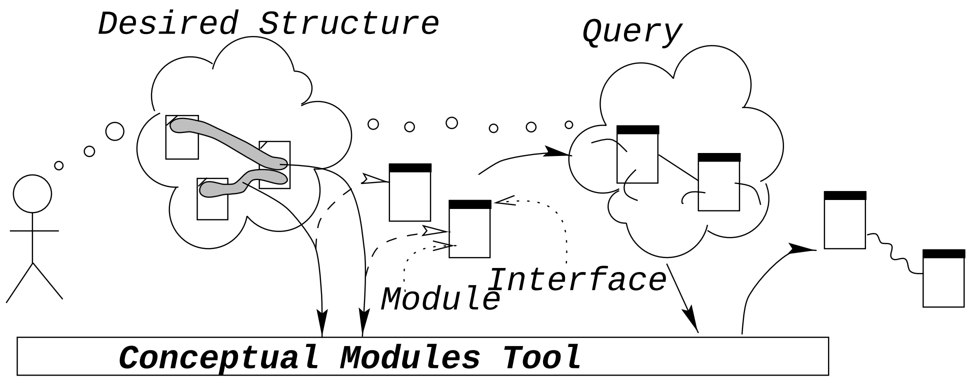


Figure 3.1: Iterative process of using the conceptual module tool

Figure 3.1 illustrates the process for using the approach as embodied by the tool. Before using the tool, a source model of the code is formed using a source code analysis tool, described further in Section 3.2.1. This source model is used internally for analysis by the conceptual module tool. When this is done, the software engineer examines the source code and identifies a collection of source lines to be treated as a unit. Next, the engineer specifies a conceptual module to represent that collection. As each conceptual module is defined, the tool performs analysis to determine the module's interface. This interface may provide sufficient information to help perform the reengineering task at hand. Sometimes, though, the user may require information about the relationship between a defined conceptual module and the existing source, and about the relationship between the conceptual modules. The user may form queries to derive this information. The steps of defining a conceptual module and performing subsequent queries are performed iteratively by the user.

The conceptual module tool consists of several interconnected components: a user interface, a query interface, an analyzer and an intermediate representation of the source.

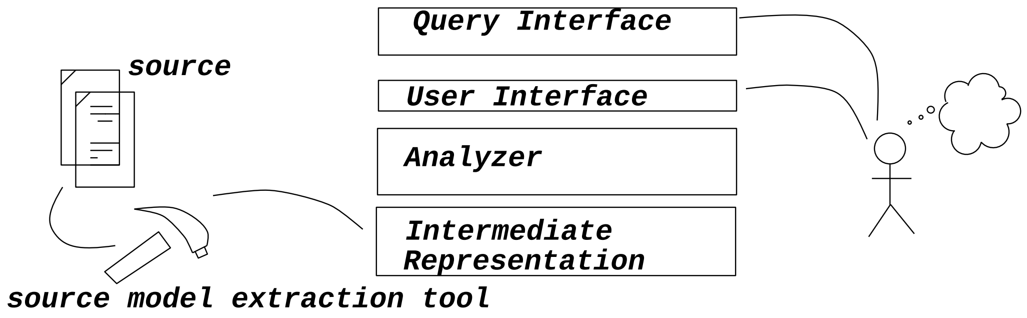


Figure 3.2: Components of the conceptual module tool

The user interacts with the tool through the user interface and the query interface as shown in Figure 3.2. The user interface is menu-driven, comprising commands for conceptual module construction, refinement and analysis. The query interface is programmatic and is provided so that the user can create their own analysis methods for task-specific purposes. Both the user interface and the query interface use the analyzer component. This component contains the general algorithms for examining control-flow and data-flow within and between conceptual modules. Performing this analysis requires the knowledge of interactions in the source. The intermediate representation is a model of the source and is created by a source model extractor which takes as input ANSI C code.

This chapter discusses the intent and implementation of the conceptual module tool. First it discusses the mechanisms for constructing and refining conceptual modules. Then it discusses the tool's analysis features. Finally, it describes the user and query interfaces.

## 3.1 Conceptual Module Construction and Refinement

A user creates a conceptual module by providing a name for the module. A conceptual module, when first constructed, contains no lines of code. The user may add lines of code to the conceptual module in several ways: by adding individual lines of code, adding ranges of lines, or adding procedures or variables. If the user adds an entire procedure, all the lines of code within the named procedure will be added to the conceptual module. In the case of adding a variable, all the lines on which the variable is either used or modified are added to the conceptual module.

The ability to add procedures is useful because there are many instances when a user needs to investigate structure that extends beyond the structure that is provided in the programming language. For example, when transforming a procedural program to an object-oriented one, the user may place existing procedures into the proposed classes. In this way the user can quickly form target member methods of classes embodied by conceptual modules. The ability to add a variable is useful since reengineering activities often involve the encapsulation of some computation involving a variable. In the `sort` example presented in Chapter 1, when isolating the input pipe of a pipe-and-filter program, the user wanted to add all variables related to input variables and files so as to quickly collect a portion of computation contributing to file input and output.

A conceptual module is refined based on the analysis the tool performs. This analysis is described in detail in Section 3.2. Based on the results of analysis, the user may choose to change the set of lines that make up a conceptual module. They may choose to add lines, variables, or procedures, or they may decide that certain lines do not belong in a conceptual module. For example, upon examining the interface of a conceptual module, the user may decide that a variable listed as an input or output variable should be fully included in the conceptual module, and so may add that variable to it. Conversely, the user may note that a particular line in a conceptual module causes a dependence between it and another conceptual module. To break the dependence, the user may choose to delete this line from one conceptual module.

## 3.2 Analysis

In reengineering activities, it is useful to fully understand the interface of a subset of the code, and how it relates to both the rest of the source and to other subsets. Precise needs differ depending on the subtleties of the task at hand. Rather than providing a fully automated approach, the conceptual module tool allows the user to specify queries they would like to perform on conceptual modules.

Analysis of conceptual modules may be broken into three major categories: single conceptual module analysis, conceptual modules relationship analysis, and conceptual modules to source analysis.

**Single conceptual module analysis** This analysis consists of determining the interface to a conceptual module: its input, output and local variables as well as its calls. An example of this was given in Section 1.1. More detail is given in 3.2.2

**Relationship analysis** This analysis involves examining the dependences between multiple conceptual modules. Through this feature the user can learn if two conceptual modules directly or indirectly relate, if their constituent lines overlap, or if one contains the other. For example, a software engineer who is restructuring procedural code to object-oriented code might want to understand the relationship between two of the proposed classes to ensure that the correct class relationships, such as particular uses relationships, are preserved. After creating a conceptual module for each class, the engineer can analyze the relationship between the conceptual modules, examining data-flow and control-flow between them. The mechanisms for performing this analysis are described in Section 3.2.4.

**Conceptual module to source analysis** This analysis involves looking at how one or more conceptual modules relate to the remainder of the existing source. For instance, a user who is re-structuring from a procedural program to an object-oriented program needs to know how the source uses the code in a conceptual module to determine if members of a posited class should be assigned private or public status.

The mechanisms for performing this analysis are discussed in Section 3.2.3 and in Section 3.4.1.

### 3.2.1 Intermediate Representation

The conceptual module tool uses an intermediate representation of the source to perform conceptual module analysis. The intermediate representation is made up of three relations:

- variable dependences are represented by the vardep relation,
- control transfers are represented by the ctrltxf relation, and
- procedure starts are represented by the procstart relation.

The tool can use this information to compute the input, output and local variables of a conceptual module, the calls to and from a conceptual module, and relationships both between conceptual modules and between conceptual modules and the remaining source.

The format of the three relations are each described.

**Variable Dependence** Variable dependence information is displayed in the vardep relation. Tuples of this relation are of the format:

| | | | | | |
|---|---|---|---|---|---|
| **vardep** | `file-name` | `use-line#` | `variable name` | `file-name` | `def-line#` |

A tuple in this relation shows the line at which a variable is used (the file name at use-line#) and the line at which the variable was last set (the file name at def-line#). This tuple can also be referred to as a *use-def pair* [7], since it contains a pair of line numbers, one the use line and the other the definition line for a particular variable.

This relation is used to perform most of the analysis done by the tool. It is used for computing input, output and local variables to a conceptual module, and for determining direct and indirect relationships between conceptual modules. There can be multiple definitions for each use point, and multiple uses for each definition point.

**Control Transfers** This relation conveys the information about transfer of control in the source code.

The format of a tuple in this relation is:

| | | | |
|---|---|---|---|
| **ctrltxf** | `file-name` | `line#` | `procedure-name` |

The current implementation of the conceptual module tool handles only procedural programs without gotos. Hence, the tool assumes that all control transfers are procedure calls. Under this assumption, the control transfer relation shows that a call to a procedure (procedure name) is made at a particular line number of a particular file (file name at line #).

This relation is used for computation of the calls made in the conceptual module, and is also used in conjunction with the `procstart` relation to find calls made to lines in a conceptual module.

**Procedure Starts** This relation simply describes the line number of the first line of a procedure definition. The format of a tuple in this relation is:

| | | | |
|---|---|---|---|
| **procstart** | `procedure-name` | `file-name` | `line#` |

With this relation, the tool can determine the range of lines of code comprising a particular procedure, and thus can determine which procedure each line of code is in. This information is used in conjunction with the `ctrltxf` relation for determining the calls into a conceptual module.

### Production and Quality of the Intermediate Representation

To obtain flexibility in analyzing source, the source model extractor used to produce the intermediate representation is separate from the conceptual module tool. Existing tools and existing frameworks can be used as the source model extractor. Two different source code analysis tools have been used to date to form intermediate representations: a tool built on top of the SUIF framework [26] and the program database tool xrefdb[18].

Because the SUIF-based tool performs data-flow analysis, it provides complete information about variable dependences. xrefdb computes calls, procedure-starts, use points and definition points of variables but can not link a use to a definition point since it performs only a syntactic analysis of the source. It is at times beneficial to use xrefdb over the SUIF-based tool since SUIF must be able to fully compile a program to analyze it. The xrefdb tool, on the other hand, is more useful when a full compile is not possible. Also, xrefdb can typically handle a larger program than the SUIF-based tool. Since the SUIF-based tool performs data-flow analysis it has a practical program size limit of a few hundred thousand lines of code, while xrefdb performs its computation through one parse of the program.

To build up use-def pairs from uses and definitions produced by xrefdb, a script was built to produce the cross product of all the uses and definitions.

### 3.2.2 Single Conceptual Module Analysiss

The interface of a conceptual module is made up of input, and output variables of the conceptual module. Calls made by lines of code in the conceptual module are not technically part of the interface, but are reported as part of the interface for the convenience of the user. The process for computing an interface for a conceptual module is described below.

#### Relating Variables to Conceptual Module

The process of assigning local, input and output variables involves sweeping through the intermediate representation, examining the lines of code that relate to a conceptual module, and maintaining variable names, uses and definitions that pertain to those lines of code.

The three types of variables are defined as follows. Input variables are those that are used in the conceptual module but have definition points outside the conceptual module. Output variables are those which are defined in the conceptual module, but have use points outside the conceptual module. Local variables are those for which all uses and definition points are contained inside the conceptual module.

Input, output and local variables are computed one conceptual module at a time. They are computed in two readings or *sweeps* of the intermediate representation. The first sweep involves examining each of the variable dependences in the intermediate representation to form a preliminary list of input, output and local variables. The second sweep reexamines the intermediate representation to refine, if necessary, the lists of variables. This refinement is required in the case of an incomplete source model. As mentioned in the previous section, the source model may contain only uses and definitions, and not full use-def pairs. This will be explained in more detail in the description of the second sweep of the intermediate representation.

**First Sweep** In the initial sweep of the intermediate representation, each variable dependence tuple is examined. From this variable dependence information, three lists are built: `LocalVars`, `InputVars`, and `OutputVars`. The variable dependences in the intermediate representation are compared against the lines of the conceptual module. Each variable dependence tuple is examined individually, and in turn. The tests that are applied to each vardep tuple are shown in Figure 3.3.

```
If the use-line and the def-line are in the conceptual module,
     then place the variable name on the LocalVars list.

If the use-line is in the conceptual module, but the def-line is not,
     then if the variable is on the LocalVars list
          remove it from the LocalVars list
     place the variable on the InputVar list.

If the def-line is in the conceptual module, but the use-line is not,
     then if the variable is on the LocalVars list
          remove it from the LocalVars list
     place the variable on the OutputVar list.
```

Figure 3.3: First sweep of variable assignment

**Second Sweep** The second sweep deals with incomplete use-def information in the intermediate representation. As shown below, this incomplete information is signaled by the keyword "unknown" in the vardep tuple in place of the file name and -1 in place of the line number. The presence of 'unknown' means that only uses and definitions, but potentially not full use-def pairs will be present in the intermediate representation. A complete use-def pair tuple format is:

```
vardep filename line varname deffilename defline
```

while a *use-only* variable dependence tuple looks like:

```
vardep filename line varname unknown -1
```

and a *definition-only* variable dependence tuple looks like:

```
vardep unknown -1 varname deffilename defline
```

In a use-only tuple, the known line number is the use point, while in the definition-only relation the known line number is the definition point.

In the first sweep of this analysis an open world assumption is made: if the word 'unknown' is encountered in place of a definition point of a variable, then there must be some existing definition-point that is not included in the intermediate representation, and similarly for an unknown use-point. In this second sweep, the closed world assumption is made: if all the known line numbers pertaining to a variable are in the conceptual module, then that variable should be considered local to the conceptual module. Basically, this step involves re-analyzing the intermediate representation by taking another look at the results of the more conservative analysis of the first sweep to pull in local variables.

In the second sweep the three variable lists compiled in the first sweep are examined. Once again, every vardep line in the intermediate representation is processed, but this time, information is compiled on each of the variables that appears in any of the three variable vectors. It is then determined what information is known about each variable, and decided if the variable should be classified as input, output or local.

Since this inferred analysis doubles the time it takes for variable analysis, and since it is only necessary if the source model is incomplete, it can be turned on or off by the user. This promotes awareness of the user about the quality of information that is being given to the analysis tool, and also ensures the user is aware of the quality of information that is being returned to them. The second sweep consists of the steps shown in Figure 3.4.

```
Place all variables in the LocalVar, OutputVar and InputVar lists
     on a promote_to_locals list.
Repeat for each variable dependence tuple of the
     intermediate representation
{
    if the variable is on the promote_to_locals list
        if the tuple is a use only and thus has only a use line defined
        and the use-line is not in the conceptual module
             remove the variable from the promote_to_locals list.
        if the tuple is a def only and thus has only a def line defined
        and the def-line is not in the conceptual module
             remove the variable from the promote_to_locals list.
}
remove all variables on the promote_to_locals list
     from the InputVar list or the OutputVar list
place all the variables on the promote_to_locals list
     on the LocalVar list.
```

Figure 3.4: Second sweep of variable assignment

At the end of these two sweeps, three lists of variables exist that conform to the definitions in Table 3.2 using the sets in Table 3.1.

### Computing Calls Information

Calls information refers to calls made to or by a conceptual module. A call is made to a procedure by a conceptual module if a control transfer line for that procedure is in the lines of that conceptual module. If there is a control transfer to a procedure whose lines are included in a conceptual module, then there is a possible transfer of control into that conceptual module.

Computing all the calls into a conceptual module involves determining the mapping between lines in the conceptual module and existing procedures, and then

| **Set name** | **Definition** |
|---|---|
| $A$ | A set of line numbers that comprise a conceptual module called $A$. |
| $uses(v)$ | A set of all line numbers of the uses of a variable $v$. |
| $defs(v)$ | A set of all line numbers of the definitions of a variable $v$. |
| $use\_pts(v, l)$ | A set of line numbers that are the use points of $v$ for the definition point, line $l$. This returns all line numbers in the event of incomplete use-def information. |
| $def\_pts(v, l)$ | A set of lines that are the definition points of $v$ for the use point, line $l$ This returns all line numbers in the event of incomplete use-def information. |
| $variables$ | A set of all variables in the intermediate representation. |
| $used(a)$ | A set of all $variables$ used on line $a$. |
| $lines$ | A set of all lines. |

Table 3.1: Definitions for variable dependence analysis

determining where calls are made to those procedures.

To determine the procedure containing a given line of code, the following method is used: First, all the procstart relations are loaded into the tool, and are sorted by line number in descending order. It is not necessary that they be sorted by file name also, since when comparing the line numbers, the file name is also checked. Then, the "file-name line-number" pairs are compared to each of the procstart tuples in descending order until a procstart tuple is found where the file-name is the same and the line-number is greater than the file name and line-number of the procstart tuple.

| **Variable Type** | **Definition** |
|---|---|
| Local Variables | $LocalVariables(A) =$ $\{v \in V \mid defs(v) \subseteq A \wedge uses(v) \subseteq (A)\}$ |
| Input Variables | $invars(A) = \{v \in V \mid \exists l \in A \bullet$ $l \in uses(v) \wedge def\_pts(v,l) \not\subseteq A\}$ $InputVariables(A) = invars(A) - LocalVariables(A)$ |
| Output Variables | $outvars(A) = \{v \in V \mid \exists l \in A \bullet$ $l \in defs(v) \wedge use\_pts(v,l) \not\subseteq A\}$ $OutputVariables(A) = outvars(A) - LocalVariables(A)$ |

Table 3.2: Set definitions for input, output and local variables of a conceptual module

### 3.2.3 Determining Relationships Between Conceptual Modules

As illustrated in Figure 3.5 there are four types of relationships between conceptual modules:

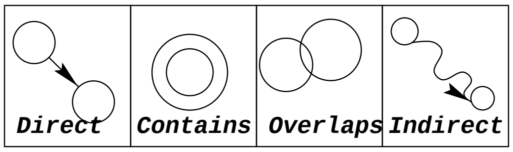
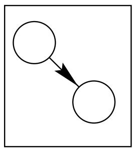


Figure 3.5: Types of relationships between conceptual module

- a direct relationship where one conceptual module directly affects another,
- a containment relationship where one conceptual module contains another,
- an overlaps relationship where one conceptual module contains some lines of another, and
- an indirect relationship, where one conceptual module affects some line of code that then affects another conceptual module.

### Direct Relationship



Given two conceptual modules, $A$ and $B$, this process involves looking at the line numbers for one conceptual module to see if any of those lines, $l$, are definition lines in the other conceptual module. Extracting this information entails examining each use-def pair in the intermediate representation to determine if the following expression is true.

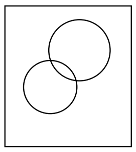
$$\exists l \in A \wedge \exists v \in V \mid l \in defs(v) \wedge (use\_pts(v, l) \cap B) \neq \emptyset$$

There is a direct relationship from $A$ to $B$ if there is a definition of $v$ in $A$ and a use of that definition in $B$.

The process for determining this relationship in the tool is to examine the vardep tuples whose use-lines are in $B$ to see if any of the definition points for those uses are in $A$.

No relationships between computations can be computed in the event of incomplete use-def information in the intermediate representation. This is because there can exist no use-def pair in the intermediate representation to link two conceptual modules if there are no use-def pairs.

### Overlap



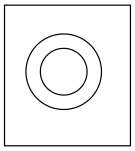
When one conceptual module is made up of some lines that are also in another conceptual module then it is said that the two conceptual modules overlap each other.

The tool's process for determining if the overlap relationship holds is to examine all the lines in the first conceptual module to see if they appear in the second conceptual module. If any of the lines appear in both conceptual modules, then there is an overlap between them.

So, for two conceptual modules, the first made up of lines $A$ and the second made up of lines $B$, the two overlap if the following expression holds:

$(A \cap B) \neq \emptyset$

### Containment



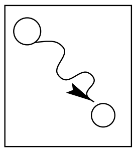
When all the lines that make up one conceptual module are a subset of the set of lines that make up another conceptual module, then it is said that the first conceptual module is contained in the other.

The process for determining this is to look at all the lines in the first conceptual module and see if they are also in the lines that make up the second conceptual module. If they are, then the first conceptual module is contained in the second conceptual module.

So, for two conceptual modules, the first made up of lines $A$ and the second made up of lines $B$, the first is contained in the second if the following expression holds:

$A \subseteq B$

### Indirect



If a conceptual module modifies some variable which then affects another conceptual module, it is said that there is an indirect relationship between the two conceptual modules. To determine if one conceptual module indirectly affects another, it is necessary to compute a backwards chain of one conceptual module and to examine if any of the lines of code in that chain are in the other conceptual module. *used(a)* is a set of all *variables* used on line $a$, and *lines* is a set of all lines. A line of code *line* is in the backwards chain for a particular line $L$ if the following expression holds true:

$$depends(a,b) \Rightarrow a \in lines \wedge b \in lines \wedge (\exists\, v \in V \bullet v \in used(a) \wedge b \in def\_pts(v,a))$$

$$line \in chain(L) \Rightarrow depends(L, line) \vee$$
$$(\exists\, M \in lines \wedge depends(M, line) \wedge M \in chain(L))$$

A line is in the backwards chain of a conceptual module comprised of lines $A$ if

$$(L \in A) \mid (line \in chain(L))$$

The definitions for the *def_pts(a,b)* relation in the above equation is shown in Table 3.1.

As in the computation of direct relationships between conceptual modules, chain information cannot be computed if there is no complete use-def information in the intermediate representation.

The process for computing a use-def chain between two conceptual modules is described in Figure 3.2.3:

```
Form a list:  definitions, of all lines in the conceptual module
       (this can be any list of variables -
              input, output, or local).
Repeat until:
       the list stops growing
       or a line in the definitions list
       is in the other conceptual module.
{
       usepoints = use points of definitions
       new_def_pts = definition points of usepoints
       definitions = definitions ∪ new_def_pts
}
```

Figure 3.6: Pseudo-code computation of an indirect relationship between two conceptual modules

### 3.2.4 Determining Relationships Between Conceptual Modules and the Source

This type of analysis involves asking questions about how a conceptual module relates to lines of code in the source. This analysis can involve just one conceptual module, or it can involve a comparison of how multiple conceptual modules relate to the source. Because the questions that can be asked about this type of relationship are unbounded, the programmatic query language presented in Section 3.4.1 is provided so that users of the tool can tailor queries to determine the information they need.

Queries in this category of analysis may involve determining if two conceptual modules have common definition points in the source, if two conceptual modules affect a common line of code, or if a particular line of code in the source has an affect on a conceptual module.

## 3.3 Tool Implementation

The functionality of the tool resides in 8 Java[1] classes and 7 Perl[24] scripts. The classes comprise approximately 3000 lines of commented Java code. The scripts are used for accessing the intermediate representation.

In the original implementation of the tool, all intermediate representation access was done by loading the entire intermediate representation into a Java Vector[1] of records, scanning the records, and applying the analysis algorithms. However, this caused the program to both consume excessive amounts of memory and to run very slowly, thus reducing the tool's ability to deal with very large (over 100,000

[1]A Vector is a kind of Java array.

line) intermediate representations. A faster mechanism was needed for the retrieval.

With the current implementation of the tool, loading the intermediate representation for `sort.c`[2] takes about 5 seconds. Loading the intermediate representation of a larger program, `GNUplot`[3] takes 30 seconds. Perl was chosen as the implementation language to manipulate the intermediate representation for its ability to sweep through files very efficiently.

Figure 3.7 shows the relationships between the classes and the scripts. Arrows show calls made between classes or scripts. Grey areas illustrate the components that make up the system. These were shown at a higher level in Figure 3.2

### 3.3.1 Classes

The following Java classes are used in the implementation of the tool.

- *Abstraction* holds the member attributes concerning a particular conceptual module including the conceptual module's name and the lines of code that comprise it. It also tracks, as a class variable, a table of all the defined conceptual modules, and contains mechanisms for locating conceptual modules and for saving conceptual modules to a file. It also maintains the variables, and control transfers that pertain to a conceptual module, and provides mechanisms for retrieving and changing those pieces of information.

- *InputMechanism* is the main user interface class.

- *Analyzer* is the main analysis class for the tool. This class contains almost all of the interface and relationship analysis functionality except the low-level

[2]`sort` comprises 29 files and 5100 lines of C code. The intermediate representation for `sort.c` is approximately 5000 tuples

[3]`GNUplot is made up of 12000 lines of C code, and its intermediate representation is 500,000 tuples`

methods for compilation of use-def chains.

- *DefUse* provides the methods that compute use-def chain information both for one conceptual module and also between conceptual modules. The backwards chain information is described in Section 3.2.3.

- *USER* class has one main method, *QUERY*, which is the method in which the user places their own functionality (user-defined queries).

- *variable* is a class that contains information about variables in the system, in particular, the input, output, and local variables contained in the conceptual module's lists.

- *Ctrltxf* is a class that contains information about control transfers retrieved from the intermediate representation.

- *Procstart* is a class that contains information about procedure starts retrieved from the intermediate representation.

### Scripts

The tool includes the following eight scripts. All but the first, *SetUpFiles*, script are written in Perl. The *SetUpFiles* script is a C-shell script.

- *SetUpFiles* uses `grep` for extracting the control transfer and procedure start relations from the intermediate representation.

- *varlistaddition* is used for getting the line numbers of the uses and definitions of variables.

- *defsforuses* takes in a list of use line numbers, and returns the definition lines for those uses. This script is used when determining chain and relationship information between conceptual modules.
- *rel* intersects and analyzes two lists of lines to identify a direct relationship between two conceptual modules.
- *chaining* takes one step back in the use-def chain.
- *linestovars* takes line numbers and returns the variables used or defined on those lines.
- *varanalysis* performs analysis on a set of lines comprising a conceptual module, determining local, input and output variables.
- *inferred_analysis* uses the variable information produced by `varanalysis` to do a second sweep through the intermediate representation to promote any variables to local variables if all lines of code which are known about a particular variable are housed in the conceptual module.

## 3.4 User and Query Interface

The user interface of the tool is for general purpose construction, refinement, and analysis of conceptual modules. It consists of the menu items shown in Figure 3.8. It is further described in Appendix 1. The queries available through the user interface rely on the same classes and methods as those used by the query interface. The user interface provides queries through a menu since they are used often and need to be accessed on a regular basis. The conceptual module tool was implemented to test the usefulness of the conceptual modules approach. Its user-interface is currently sufficient for analyzing conceptual modules but was not the focus of this research.

Since the requirements of users differ greatly depending on the reengineering task being performed, users are often interested in very different details about conceptual modules. In addition to basic and frequently accessed information about conceptual module interfaces and relationships, the user may inspect the details of all computed information. For instance the user may wish to ask:

- if the conceptual modules have any common definition points?
- if the conceptual modules have any common effects?
- if the conceptual modules have any variables in common?

To allow the user to tailor methods for manipulation and analysis of conceptual modules, a programmatic query interface and analysis library is provided. To use the library functions of the tool, the user must provide functionality in the `QUERY()` method which resides in the Java file: `USER.java`.

### 3.4.1 Query Library

This section gives information about the member functions and attributes that the tool makes visible to the user. It includes detailed lists of the primitives provided by the tool, and information about how the primitives are implemented (including interface information).

**Abstraction Member Access** These member methods and attributes of the class `Abstraction` are available to the user for use in the programmatic query mechanism.

- *Name* is a name of a conceptual module and is of type String.
- *AbstractionTable* is a list of all conceptual module, and is of type Vector of conceptual modules.
- *Lines* is a list of line #'s in conceptual module and is of type Vector of Strings each look like: "filename linenum".
- *LOCALVars* is a list of a conceptual module's local variables, and is of type Vector of Strings.
- *INVars* is a list of a conceptual module's input variables, and is of type Vector of Strings.
- *OUTVars* is a list of a conceptual module's output variables, and is of type Vector of Strings.
- *CtrlsOUT* is a list of control transfers out of the conceptual module, and is of type Vector of Ctrltxfs.
- *Procstarts* is a list of all procedure starts in the intermediate representation, and is of type Vector of Procstarts.

**Direct Relationship Methods** The following methods are available to the user for retrieving direct relationship information:

- `public static void Analyzer.getRelationship(Abstraction A,Abstraction B)`

  This method takes two conceptual modules as parameters, and writes (to standard output) information about whether one provides variables to the other, and whether there is any overlap of lines between the two. It also writes out whether the two conceptual modules are equal, or whether one contains the other.

There is currently no programmatic access to the result of this method.

- `public static SET Analyzer.inVarsInCommon(Abstraction A,Abstraction B)`

  This method takes two conceptual modules as parameters and returns a SET[4] of input variables that the two conceptual modules have in common. This is done by first computing the input variables for each conceptual module, and then by comparing all the definition points of those variables for commonalities.

- `public static SET Analyzer.outVarsInCommon(Abstraction A,Abstraction B)`

  This method takes two conceptual modules as parameters and returns a SET of output variables that the two conceptual modules have in common. This is done by first computing the output variables for each conceptual module, and then by comparing all the use points of those variables for commonalities.

### Indirect Relationship Methods

- `public SET Analyzer.get_full_chain(Abstraction A)`

  This method returns the SET containing all the lines upon which conceptual module `A` depends.

- `public static void DefUse.FindChain(Abstraction A, Abstraction B)`

  This takes two conceptual modules as parameters and writes to `stdout` a backwards dependence path from the second to the first.

[4]A `SET` is a Vector of Strings with additional print methods.

To inspect a reciprocal relationship it is necessary to call this method twice, the second time switching the order of the conceptual modules passed in.

**Abstraction Construction and Refinement Methods** When a user is programmatically analyzing one or more conceptual modules, they may wish to construct new modules, or refine those they are analyzing. The following methods support this functionality.

- `public Abstraction(String n)`

  This method creates a conceptual module with name, "Name"

- `public void addLine(String L)`

  This method adds the string "Filename line#" of line number L to the conceptual module Lines Vector which is of type Vector of Strings.

- `public void AddLines(SET lines)`

  This method adds a set of lines to a variable

- `public void AddRange(String Line1, String Line2)`

  This adds the lines ranging from Line1 to Line2. Both lines specified must appear in the same file.

- `public void AddProcedure(String Mod, String Proc)`

  This method adds the lines in a procedure, including the declaration line, to a conceptual module.

- `public void AddVariable(SET vars)`

  This method adds a SET of variables to a conceptual module

- `public void RemoveLines(SET lines)`

  This method removes the SET of lines from a conceptual module

- `public void RemoveVariable(SET vars)`

  This method removes all the variables in the SET vars from the conceptual module.

- `public static int LocateAbstraction(String name)`

  This method returns the conceptual module table index, -1 if not found.

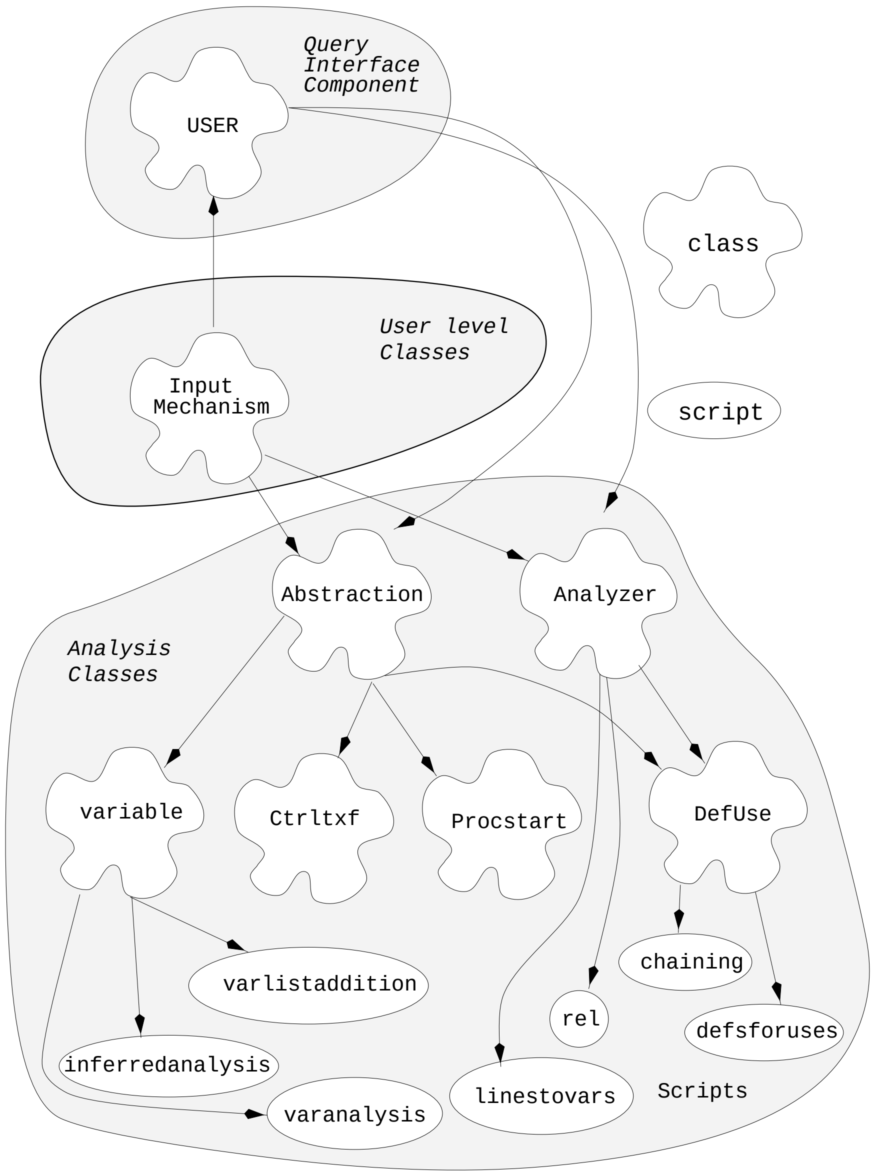
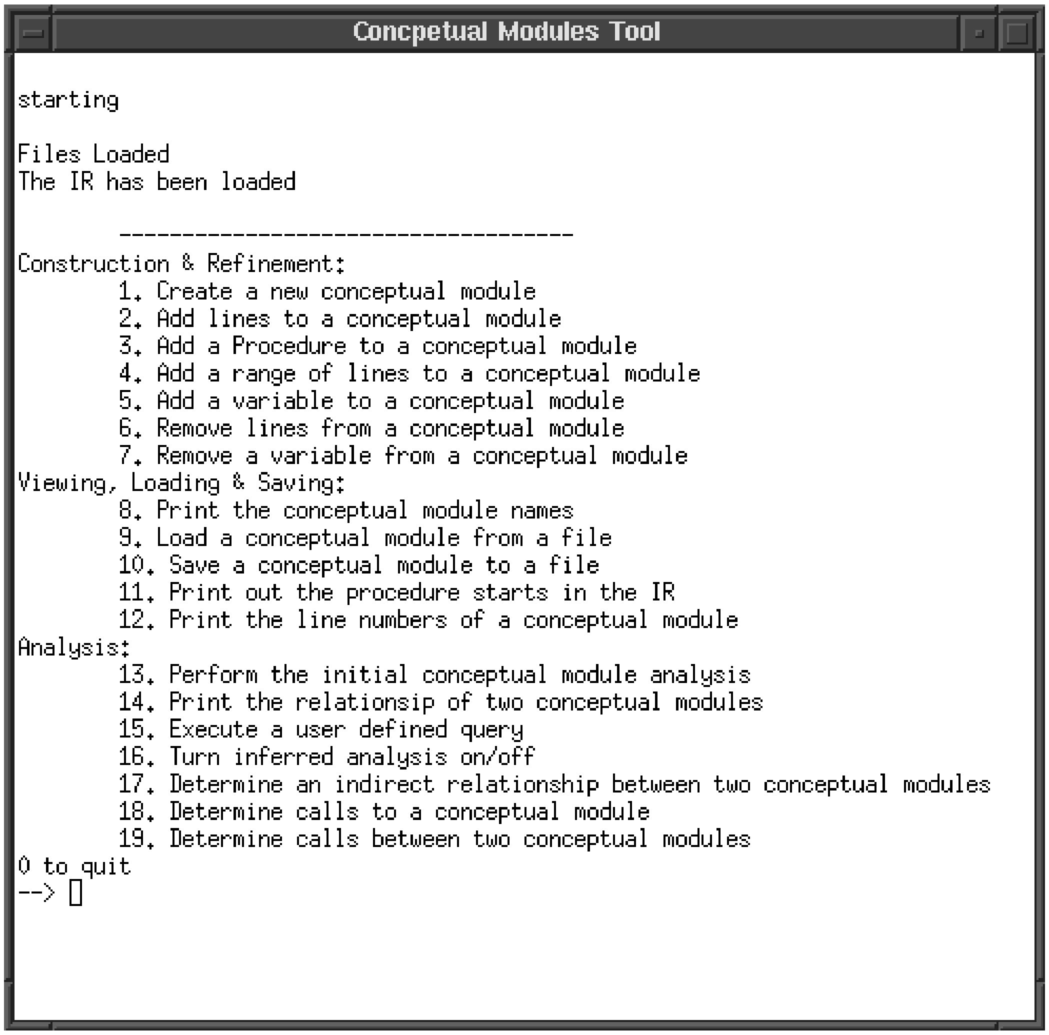


Figure 3.7: Relationships between classes and scripts



```
starting

Files Loaded
The IR has been loaded

        ------------------------------------
Construction & Refinement:
        1. Create a new conceptual module
        2. Add lines to a conceptual module
        3. Add a Procedure to a conceptual module
        4. Add a range of lines to a conceptual module
        5. Add a variable to a conceptual module
        6. Remove lines from a conceptual module
        7. Remove a variable from a conceptual module
Viewing, Loading & Saving:
        8. Print the conceptual module names
        9. Load a conceptual module from a file
        10. Save a conceptual module to a file
        11. Print out the procedure starts in the IR
        12. Print the line numbers of a conceptual module
Analysis:
        13. Perform the initial conceptual module analysis
        14. Print the relationsip of two conceptual modules
        15. Execute a user defined query
        16. Turn inferred analysis on/off
        17. Determine an indirect relationship between two conceptual modules
        18. Determine calls to a conceptual module
        19. Determine calls between two conceptual modules
0 to quit
-->
```

Figure 3.8: User Interface window of the tool

# Chapter 4

# Validation

To examine the usefulness of the conceptual module tool, the tool was used both in actual restructuring and reengineering tasks by other users in conjunction with the author, and in sample scenarios by the author. In the case studies, the participants worked through reengineering and restructuring tasks they were performing with the help of the conceptual module tool. The sample scenarios involved asking certain specific types of questions using both existing program understanding tools and the conceptual module tool. The results of using the different tools were then compared.

This chapter provides an account of the case studies, and outlines the results from each of the sample scenarios.

## 4.1 User Testing: Case Study

Two small case studies were conducted to better understand the usefulness of the conceptual module approach in a real-task setting. The first group of users were removing large portions of a program while attempting not to affect certain other portions of that program. The second group was analyzing a procedural program to better assess how to restructure it according to an object-oriented design. Each group spent roughly four hours using the tool.

### 4.1.1 Extracting a Functional Subset

The first case study was conducted with a group of graduate students in Computer Science. The students' project was to parallelize the system's computation, and to remove unwanted computation so as to enhance parallelizability and efficiency. The students were deleting large portions of a 56 file program.

The application being modified was the CU Decision Diagram Package (CUDD).[1] The CUDD package provides functions to manipulate Binary Decision Diagrams (BDDs), Algebraic Decision Diagrams (ADDs) and Zero-suppressed Binary Decision Diagrams (ZDDs). The system is comprised of 47,796 lines of commented `C` code. The group intended to eventually remove the ADD and ZDD functionality from the program. The task in this case study was the first phase of this removal: extraction of the ZDD functionality while leaving the BDD functionality unaffected.

Their initial plan with this restructuring task was a common one: to delete or comment out the lines of code and then attempt to compile the program, execute it, and perform regression testing.

To discover how much code the users wished to remove, the users relied on a consistent variable, sub-routine and type naming convention, removing all lines that contained any program element with a certain name. Before they began using the conceptual module tool, the group used `grep`[2] to search for these naming patterns. From this search they determined that approximately 2000 lines of code should

[1]CUDD: CU Decision Diagram Package, Release 2.1.2, written by Fabio Somenzi of the Department of Electrical and Computer Engineering, University of Colorado at Boulder.

[2]`grep` is a Unix lexical search tool.

be targeted for removal. Initially, they had intended to examine each line of code returned by `grep` to see if it might affect the portion of code to preserve. The `grep` tool could not completely help them with their task since it did not provide them the semantic detail they needed to perform the extraction.

> ...the time-line was too tight to wade through all that code, and not only did I want to rip out what we did not need (just to reduce the size of the monster) but I also had to make some structural changes to the most fundamental data structure involved, and I needed to know WHERE to focus these changes... — *Yvonne Coady, conceptual module tool user*

To help them with this task, the conceptual module tool was used to construct two conceptual modules: the first comprised the lines of code targeted for deletion, and the second comprised the lines of functionality to be preserved. Three types of analysis on these conceptual modules were then performed:

- simple relationship analysis to determine the direct relationships between the two conceptual modules,
- transitive relationship analysis to determine if there was an indirect relationship between the two conceptual modules,
- calls-in analysis to see if any of the lines of code to be preserved invoked those targeted for deletion.

These three queries enabled the users to directly ask about whether there was any relationship between the two conceptual modules. Running the three queries took approximately ten minutes, including the time to load the 70,000 line intermediate representation.

These queries reported many relationships found between the two conceptual modules. However, when analyzed, it was found that the code causing the links between the two conceptual modules actually represented code that should be deleted. When those lines of code were moved to the conceptual module of lines to be deleted, the relationships between the two conceptual modules were severed. The users were then able to continue their restructuring task with increased confidence that the deletion of the targeted lines of code would not affect the portion of code they wished to preserve.

> ... not only did [the conceptual module] tool verify the independent nature of the ZDD functionality and allow me to rip out all that code, BUT, the process of using your tool forced me to analyze and understand the code in a way that I had not been doing — and that ultimately it very quickly gave me the perspective I needed. — *Yvonne Coady, conceptual module tool user*

After the removal of the lines of code in the ZDD conceptual module, the group performed regression testing on the BDD functionality in the package and found it to be unaffected by the removal of those lines. The users estimated that they spent over 30 hours on extraction; and four of those hours were spent using the conceptual module tool. The first half of the extraction was done before using the conceptual module tool, consuming approximately 20 of the 30 hours. After using the conceptual module tool, they finished the remaining extraction in around 6 hours.

During the study, we placed the inputs to the tool into a file and made use of the UNIX input redirection facility to direct the input from the file into the tool.

### 4.1.2 Moving to Object-Oriented Design

The second case study involved an undergraduate student in Electrical Engineering who was faced with converting a large C program to C++. The student had a minimal understanding of the existing code when asked to perform this task.

The student wished to determine if a postulated class diagram resembled the existing module packaging scheme for the system. We made use of the ability to add procedures to conceptual modules and also captured some of the main files in the system into conceptual modules to form proposed classes, we then performed initial analysis, relationship analysis, and calls analysis on the conceptual modules to determine the dependences on and between those conceptual modules.

After determining that there was no overlap between conceptual modules, the user was able to assume that the variables local to the conceptual modules should be private variables to those conceptual modules, and that the procedures that were called from outside a conceptual module should be maintained as public or protected methods.

The student had a set of inter-class data-flow and control-flow relationship constraints; for instance, control and data-flow should go only from one class to another, but not vice-versa. To test if these constraints were maintained, relationship analysis was performed to see what variables were being modified between conceptual modules, and also what cross-conceptual module control transfers existed. This helped the student refine which procedures should belong to which proposed class, and if any portions of computation should be moved around from one conceptual module to another to maintain the privacy of certain variables.

In one case, it was found that leaving a portion of computation in the class in which it was originally placed would violate the uses relationship constraints between two classes. Enclosing that functionality in another conceptual module and reapplying the analysis showed that moving the functionality to another class would maintain the relationship without breaking other desired properties of the system. The student was able to move code around between conceptual modules to find the necessary relationships. The process also helped improve the student's understanding of the existing code.

## 4.2 Sample Scenarios

To further study the effectiveness of the conceptual module approach, the tool and several existing program understanding tools were applied to three reengineering scenarios. The first scenario considers the component formation and extraction outlined in the introduction: the creation of an input pipe component that includes an initialization procedure from the GNU `sort` program. The second scenario considers a restructuring task: the re-modularization of a legacy C program, `adventure`. The last scenario considers the formation of a class from a module in the `GNUplot` program.

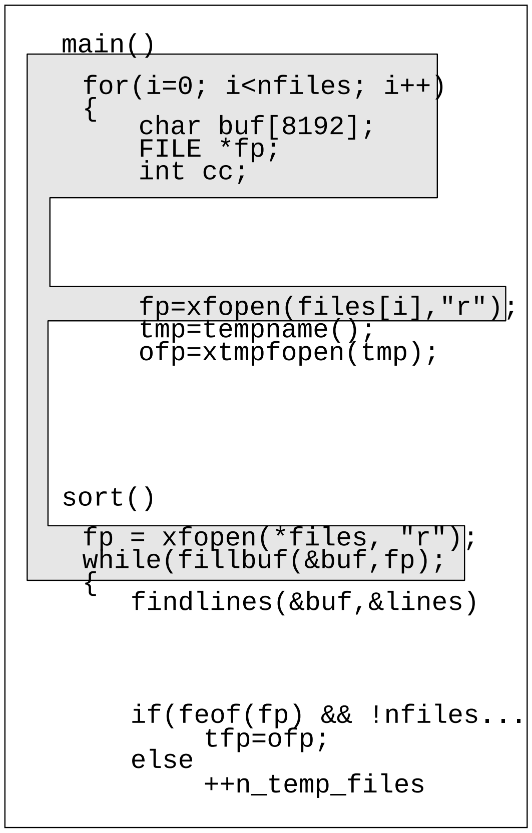
The sample activities were all performed with several program understanding tools: the conceptual module query tool, Unravel [12]—a slicing tool, Lackwit[16]—a type inferencing tool, and xrefdb[18]—a program database tool.

Each program understanding tool provides the user different types of information. To provide some basis for comparison, the measure applied was the amount of information returned to the user which was of interest, and the amount of work that was required to filter that information.

### 4.2.1 Gnu sort - Component Extraction

As mentioned above, the initial test performed with the conceptual module tool was to extract an input pipe component from the Unix sort program.

As described in Chapter 1, the `GNU sort` program is built as a pipe-and-filter system. The program consists of one filter, the sort functionality, and two pipes: the input pipe and the output pipe from the sort filter.



```
  main()
    for(i=0; i<nfiles; i++)
    {   char buf[8192];
        FILE *fp;
        int cc;



        fp=xfopen(files[i],"r");
        tmp=tempname();
        ofp=xtmpfopen(tmp);




  sort()
    fp = xfopen(*files, "r");
    while(fillbuf(&buf,fp);
    {
        findlines(&buf,&lines)



        if(feof(fp) && !nfiles...
            tfp=ofp;
        else
            ++n_temp_files
```

Figure 4.1: Selected non-contiguous lines of source

The task in this scenario consisted of identifying the existing source lines that should be included as part of an initialization function for a desired input pipe component. The tools were applied to this task after identifying, based on a perusal of the source, a modest number—less than 10 lines—of source that should be included in the component. Figure 4.1 shows a snippet of the relevant code from the main and sort functions. This code is spread across multiple non-contiguous lines of source code.

The Unravel tool supports the computation of backward slices given a variable name and a program point (line of code). For this task, the aim was to compute backward slices on variables from the pre-identified lines of code. The resulting slices would provide the lines of source that contributed to the values of those variables at the program point. The slices computed in this way were large. In all cases, because the slice computations took several hours, the process had to be interrupted; partial slices were used in substitution. Each of the partial slices was over 750 nodes in size. Qualitative inspection of these slices revealed some procedures of interest, however, most of the source lines were not relevant to the input pipe component. For example, most lines in the sortlines procedure were included in one of the slices; these lines contribute to the functionality of the sort filter, not the input pipe functionality.

To assess the usefulness of Lackwit for this type of task, the graph capability of Lackwit was employed. Lackwit has a feature that produces a graph summarizing the information about a single component of a variable.

> The nodes of the graph represent the global declarations, and the edges represent the use of one declaration by another in the text of the program. Arrows point from the using declaration to the used declarations.[16, page 342]

For instance, a graph computed for the `buf` variable in the `fillbuf` procedure for instance, a variable central to the sort functionality, included 23 procedures. These graphs were useful in determining the procedures containing potentially relevant code, but they did not provide specific information about relevant source lines. As indicated by the graph, all but one of these procedures could potentially alter the value of the variable. A qualitative evaluation of these procedures identified five of the procedures as containing code relevant to the task at hand.

In the case of the xrefdb tool, the lines comprising all references and all declarations of variables identified of interest were queried. With these queries, 126 lines of source code for qualitative assessment were identified. 30% of these lines were assessed to be relevant to the task by comparing them with the final set of lines of code targeted for extraction.

As described earlier, the conceptual module tool was applied to this task by forming a module comprised of the pre-identified lines of source. The analysis of those lines performed by the conceptual module tool was then used to drive further investigation of the source. For example, through examination of the definition points reported in the analysis for the input variable, `buf`, it was found that additional lines of source should be included in the module. To form the desired procedure, this process was iterated approximately six times.

It was straightforward to apply the conceptual module tool to this task because, at any point, only limited information was being considered about the source, such as the definition points of input variables or use points of output variables. This information was determined and produced in the context of the desired structure. The conceptual module tool performed the filtering that had to be done manually when using other techniques. For example, after collecting the pre-identified lines of code into a conceptual module, the tool provided a succinct list of the input, output and local variables rather than providing subsets of the program's execution for each variable as was returned by the Unravel tool.

### 4.2.2 Adventure - Clarifying Commonalities

The `adventure` program is an exploration game that has been distributed as part of the Unix operating system for many years.[3] The game was originally written in Fortran and was later converted to C. The source consists of approximately 8,000

[3]Version 6 of adventure was used in this analysis.

lines of C code distributed across 13 files.[4]

A substantial amount of the functionality of the game resides in a 525-line main procedure where gotos between labels are used to move a player through the game. The restructuring task was to form procedures out of lines of code between goto labels so as to encapsulate different states of the game. The goal was then to understand how the desired procedures interact through state information and, in particular, to determine variable definitions shared by these procedures.

It was difficult to apply a slicing tool to this problem because of the number of variables of interest. Essentially, it was necessary to compute the intersection of backward slices on each variable mentioned in each target procedure. For the target procedures in `adventure`, this would have involved computing 38 slices. As the Unravel tool was unable to intersect this large number of slices, only a few sample slices were computed. As was the case for `sort`, the slices were large, consisting of over 700 nodes, making it difficult to wade through the reported information to determine the program points of interest.

It was also difficult to apply the Lackwit tool to this task because of the granularity of the information reported. The graphical view of the type information used for `sort` that reports on the affect of procedures on the values of variables was not useful in this case. It was not useful because the vast majority of the functionality was included in one main procedure.

The Lackwit tool also provides the capability to report a list of variables sharing values with the variable of interest. For `adventure`, the results from these queries were difficult to interpret and to filter because they returned a significant

[4]The distributed version of the source was modified slightly to permit analysis. For example, as distributed, the source contains multiple declarations for global variables. These declarations were restructured. No substantive changes were made to the main function that is the target of this scenario.

amount of information. For instance, querying on the `wzdark` variable of one of the desired procedures returned 231 related variables. The xrefdb tool was also not well-suited for the task. Since the tool reports cross-reference information extracted from a syntactic parse of the source code, the tool is unable to report information about data-flow between different variables.

The conceptual module tool was applied to the task of forming conceptual modules for each of the desired procedures consisting of the identified source lines. Then, a user-defined form of indirect query was written. To determine if there were any common definition points for the target procedure.

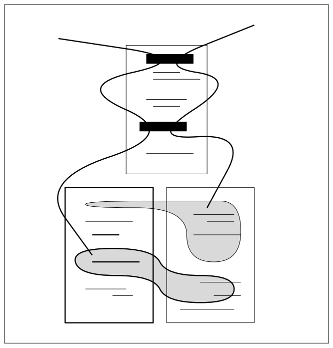


Figure 4.2: Common definition points between two conceptual modules

For each conceptual module, this query computes the use-def chains of the input variables of the module, and intersects all resultant chains to produce a list of variables and definition points common to all the conceptual modules. Local variables are considered to handle cases of module overlap. By allowing the engineer to focus on `use-def` chains of collections of variables encapsulated in the module, the tool provided a direct way to access information of interest. (Figure 4.2)

### 4.2.3 GNUplot - Assessing New Structure

The third task involved assessing the difficulty of restructuring a module from the `GNUplot`[5] program into a class. Specifically, the objective was to understand which procedures from a module should become methods of a target class, and furthermore,

[5]The GNUplot version analyzed was 3.50.1.17

whether the potential methods would become public or private members of the class. The module considered was a contouring module specified in the file `contour.c`. The `GNUplot` program comprises approximately 12,000 lines of `C` code. The contour module contains just over 900 lines of non-blank, non-commented source.

Neither Unravel nor Lackwit were able to analyze the source for `GNUplot`. As a result, only xrefdb and the conceptual module tool were used to perform analysis on this file.

The results of using xrefdb for this task were similar to those reported for the task on `adventure`. This tool was used to find lines of code that might be relevant to the task based on the references and declarations of variables. To start determining which target methods might be private, it was necessary to understand which existing procedures operated on data local to the target class. Determining this information, though, required access to use-def information to determine relationships of variables. This information is outside the scope of the functionality provided by xrefdb.

To answer these questions with the conceptual module tool, the source in the `contour.c` file was examined and a conceptual module was created for each contour-related procedure. (The file also contained several functions providing general geometry functionality.) Then a conceptual module was created to represent the target contour class. This class-level conceptual module was comprised of the source contributing to each method-level conceptual module. To determine potential private methods, the query shown in Figure 4.3 was performed to determine the methods in the class conceptual modules that used or defined data local to the container conceptual module. This query was straightforward to write.

```
\\ All conceptual modules defined are pointed to by the
\\ AbstractionTable structure.
\\ Only the container and the class conceptual modules
\\ are defined.

\\ get the container module for all the conceptual modules
Abstraction Container = Abstraction.Get(e);
for(int i=0; i<Abstraction.AbstractionTable.size(); i++)
{
    if(i!=e) { \\ if it is not the container conceptual module
           \\ get the conceptual module at 'i'
         Abstraction Contained = Abstraction.Get(i);
           \\ perform initial analysis on that conceptual module
         Analyzer.Initial(Contained, true);
           \\ compute the common input variables of the contained and
           \\ the container conceptual modules
         SET inLines = Analyzer.inVarsInCommon(Contained, Container);
           \\ compute the common output variables of the contained and
           \\ the container conceptual modules
         SET outLines = Analyzer.outVarsInCommon(Contained, Container);
           \\ Print out the input variables list preceded by the words
           \\ ``Public Variables:''
         inLines.print(``Public Variables:'');
           \\ print out the output variables list
         outLines.print();
     }
}
```

Figure 4.3: Query to compare inputs and outputs of two conceptual modules

# Chapter 5

# Summary

This thesis has presented the conceptual module source code analysis approach. The conceptual module tool, developed to support this approach, allows the user to overlay a desired logical structure on the existing structure of source code and then to build, analyze and refine that structure. In contrast to many other tools that have been built to analyze source, the conceptual module approach supports reengineering activities because it allows the engineer to simultaneously perform queries about both the existing and the desired source structure.

The conceptual module tool provides the user the capability to build conceptual modules out of lines of code drawn from anywhere in the original source. The lines can be treated as one logical unit. To give the user the necessary querying flexibility to perform analysis suited to their task, a programmatic query interface is supplied. Through this interface, users can write their own queries using a library of pre-defined methods, tailoring those queries to the needs of the specific task.

Two user studies were performed. In the first study, described in Section 4.1.1, unwanted functionality in a software package was analyzed for removal so as to streamline the package and enhance parallelizability. The users were able to ask specific questions about the relationship between the code targeted for removal and the functionality they wished to preserve. The tool was also able to give them feedback about lines of code that could be removed which they had not originally targeted. The identified code was removed from the software package and regression testing was performed on the remaining functionality. The functionality was found to be preserved.

In the second study, accounted in Section 4.1.2, a system's code was analyzed prior to restructuring the code from a procedural design to an object-oriented one. The user was attempting to evaluate how to place existing code in desired classes. By performing relationship interface analysis on the target classes, the user was able to determine whether a certain class structure would preserve the correct relationships between classes such as uses and contains relationships. The tool was not only able to give information about the class interfaces including public and private data structure and methods, but was also helpful in refining the portions of code that would be placed in each class.

The tool was also applied in three scenarios. These scenarios were pre-defined to test the effectiveness and conciseness of the output of the tool when faced with certain common types of reengineering and restructuring tasks. For `sort`, the task was to extract the input pipe component and to test the extracted component's functionality using a test driver. For `GNUplot`, the task was to determine the private and public members of a class that was to be created from a module containing several procedures. For `adventure`, the task was to understand the common definitions between different blocks in the code. In each of these cases, the use of the conceptual module tool proved to be effective. When compared with program databases, type inferencers and slicing tools, the conceptual module tool reduced the amount of superfluous information reported to the user and radically cut down the number of queries that had to be performed.

## 5.1 Discussion

This section discusses issues regarding the conceptual module approach. It first examines the need for users to be given information with relation to the context of their query. Then it discusses the form and report format of queries. Next, it details the reasons for basing the tool on line number, as well as the role of the source model in analysis. Finally, it considers the need for control dependence information to be included in the intermediate representation.

### 5.1.1 Query Context

Many existing tools do not allow the software engineer to adequately express the context of the query being performed. Context is expressed in two parts. First, it can be beneficial for a software engineer to identify the region of the program over which the query is being made. For instance, a slicing tool typically allows a user to specify a particular program point of interest and then to determine the direction—forward or backward—of the slice. In a similar way, the conceptual module tool provides an engineer control in specifying this aspect of context since a conceptual module is defined in terms of particular lines in the source. In contrast, type inferencing tools like Lackwit are based on the analysis of the use of variables over the entire program. A consequence of a lack of context specification in query formation can be a return of a large number of false positives with respect to the task being performed. This situation arose when applying Lackwit to the task on `sort` described in Section 4.2.1.

Second, it can be beneficial to a software engineer to restrict the region of the program over which query results are reported. An engineer, for instance, may not be able to efficiently interpret slices comprised of hundreds of nodes; the set of statements contributing to the slice that are within a certain distance from the program point may be sufficient. The conceptual module tool provides some control to the user over this aspect of context by reporting localized results of the analysis of the lines of code contributing to the module. The user can then tailor queries to directly control the scope of their analysis.

### 5.1.2 Query Form

Often, when performing a query task, there is a need to perform queries over groups of structural items. For the task on `adventure` described in Section 4.2.2, it was desirable to perform queries about all of the variables within a block of code and then to combine the results, perhaps using set operations. None of the existing tools surveyed, and most of those commonly known, provide support for this kind of grouped queries. Instead, the user must perform a series of queries and perform the desired combination operations manually.

The conceptual module approach demonstrates how support for grouped queries can be added as a front end to an existing tool. In the `sort` scenario described in Section 4.2.1, the use of the conceptual module approach using information extracted from the xrefdb database eliminated the need for the multiple queries applied when directly using xrefdb.

### 5.1.3 Query Report Format

The Lackwit tool is characteristic of a number of program understanding tools that report results in terms of the existing source structure, such as describing the procedures affecting the value of a variable. There is an underlying assumption with these tools that the existing structure will be sufficient to help an engineer interpret the results. However, when applied to systems like `adventure` (Section 4.2.2) that have little structure, the results are either meaningless, as was the case in the computed variable graphs, or they are overwhelming, as when perusing the textual lists of variable dependences.

The conceptual module tool addresses this problem by reporting query results in terms of the target, rather than the existing structure. The engineer may thus choose the appropriate structure in which to view the results.

### 5.1.4 Line Number Granularity

The conceptual module tool is based on line numbers for three reasons. One reason is that a user of the tool can easily identify source by line numbers to map to a conceptual module. The specification of this correspondence would likely be more difficult if a finer-grained representation, such as an abstract-syntax tree, were used. The use of line numbers in the source model to identify pieces of the system also enhances the flexibility of the tool by making it possible to connect the conceptual module tool to different source model extractors. Line numbers are also a close approximation of statements.

### 5.1.5 Role of the Source Model

The conceptual module approach supports a range of source models: a source model may comprise either `use-def` chain information, or uncorrelated `use` and `def` information. The analysis function of the tool is used to "smooth-out" these differing forms of source model information. The combination of the use of a source model, as opposed to directly analyzing the source, and an analysis capability to smooth differences in the source models, provides a software engineer with significant flexibility. An engineer can choose a source model extractor suitable for the system being studied, and yet can interpret the results of applying a tool to the source model in a consistent manner.

The conceptual module tool is dependent on the relations comprising the source model. Currently, these relations are oriented at representing systems implementing in a procedural language. Extensions to relations in the source model and the analysis performed in the tool would be necessary to apply the tool to reengineer systems written in other kinds of languages.

One class of language of interest are object-oriented languages. In the current implementation of the tool, the structure of the conceptual modules created closely resembles the procedural structure of the programs analyzed. Enabling a user to analyze an object-oriented system with the conceptual module tool would require a substantial number of additional analysis routines and the ability to model object-oriented structural features in conceptual modules. The user would have to be able to incorporate object-oriented concepts—such as class hierarchies and private, public and protected members—into the conceptual modules they create. To perform thorough analysis of the conceptual modules with relation to the existing source, the source model would also have to include object-oriented information such as class declarations, public, private and protected member declarations, and sub-class relationships.

### 5.1.6 Control Dependence Information

The analysis of the conceptual module tool is currently limited by its inability to consider control dependence information. Enhancing the conceptual module tool to consider control dependence information in its analysis would enable a user who is extracting code for reuse to better understand how the code relates to control structures such as loops. If a user were examining lines based on some naming convention, as the users were doing for the CUDD package.[1] and then grabbing lines based on names, it is possible that the user may grab a line that is contained in a loop. The user may or may not be interested in this information, however, the user should be alerted that the line of code was intended to be used inside a loop. The user may then decide to include the loop statement in the conceptual module to maintain the control binding of that line of code.

The addition of this functionality would require a fourth relation in the intermediate representation: ctrldep, the control dependence relation. Analysis functionality exploiting this relation would need to be added to the tool. These modifications would include adding chaining functionality to bring control dependent lines into back-chains. Also, in the relationship analysis, modifications would be needed to identify if one conceptual module was control dependent on another. In the construction of conceptual modules there could also be functionality allowing the user to opt to include lines upon which lines in the conceptual module depend.

[1]This example is described in Section 4.1.1.

## 5.2 Extensions

This section discusses possible extensions to the conceptual module approach and tool. These extensions would take the form of additional or changed functionality of the tool.

### 5.2.1 Architectural Conformance

Because of the complexity of present day software systems, the original design of a program may not be adhered to throughout the process of its implementation[20]. Additionally, over the life of a software system, the software developers working on the system will typically change. Together, these two conditions often mean that the true design of a system is not known. True design refers not to the design that was originally intended for the system, but to the design currently embodied by the software.

Based on these unknowns, it is beneficial to be able to check how well a system conforms to the design it seems to follow. Typically, there are a set of rules the software would have to follow so as to conform to a particular design. In a pipe-and-filter architecture, for instance, pipes may not be allowed to cause feedbacks between filters. Architectural conformance involves examining source at the code level to determine if it complies with the rules for a particular architectural model, and if it does not, reporting the code causing the violation [20].

In addition to providing support for reengineering, the conceptual module approach may provide a suitable framework on which to perform architectural design conformance checks.

For example, the query language could be used to determine the architectural conformance of a pipe component in a pipe-and-filter system, or could perhaps be used to try to verify the relationships between layers in a layered architecture. To perform architectural conformance checks using the conceptual module approach and tool, a user would create conceptual modules for the components and connectors in the architectural model, and then would formulate queries based on the rules for that model to test if the code adhered to the architectural rules. The user would then be able to ascertain from the output of the tool which lines of code, if any, violated the principles of the architectural model. The user would make iterative use of the tool, refining which lines of code fit into which conceptual module, and would also be able to return to the source to change to it to better fit the architectural model.

The relationship information presently provided by the tool may be sufficient for embarking on this type of analysis. For instance, the user can test a group of conceptual modules forming a pipe-and-filter architecture for a feedback loop (transitive closure) by using the indirect-relationship query mechanisms presently in place. As another example, the user can query the calls made from one layer of an architectural model to a higher layer to see if that lower level uses the higher one.

To provide the user the correct functionality for architectural conformance checking, more user testing is needed to determine the types of information required for verifying the architectural conformance of a component or a connector.

### 5.2.2 Result Reporting

Multiple variable definitions and multiple calls can appear on the same line of code. Sometimes, a user is interested in isolating just one of the variables or just one of the calls on a line. Including the entire line in a conceptual module, causes control-flow and data-flow dependences to the conceptual module because of the unwanted portions of the line. When analyzing that conceptual module, the tool will translate those control-flow and data-flow dependences into calls and input, output or local variables, all of which are due not to the variable or call of interest, but to the other calls or variables appearing on that same line.

In most of these cases, the user can remember the lines of code that cause the superfluous information, and can ignore all the information that is produced by the inclusion of that line.[2] This inclusion of unwanted variables or calls, however, can lead to annoying and repetitive information, and at times can snow-ball, and pull in so many pieces of information that the analysis mostly returns information the user is not interested in.

One possible solution to this problem is to allow users to expel lines of code from the intermediate representation, so that they will not be taken into account at all. This approach, though, can cause implementation problems, because many of the functions of the tool may be required to be radically altered to account for omission of lines of code. For example, this omission would preclude chaining through those lines of code. The result would be that the user would have to make complex interpretations of information returned from all the relationship type queries.

Another possibility is some way to allow users to break apart lines of code. This would cause a fundamental change in the current implementation of the tool since it is currently completely based on lines of code. Thorough user testing and investigation of this is necessary to attempt to strike a balance between the convenient and familiar line-number granularity, and the provision of the ability to break

[2]It is quite clear in the user interface, which line of code caused the inclusion of a particular call or variable.

apart lines to access their constituent variables and calls.

In both schemes, great care would have to be taken to ensure the awareness of users about what portions of which lines are omitted from view. Further user testing is necessary to fully understand the most convenient way for users to request this omission, and the safest way to keep them informed of their outstanding omission requests.

### 5.2.3 Considering Data Structures

During the case study described in Section 4.1.1 involving the CUDD users, a situation arose in which the users wished to include all of the lines of code relating to a field in a data structure. Currently, the conceptual module tool provides no support for data structure analysis. However, in this case, the users were able to continue as they had been doing and select lines of code based on a consistent variable naming convention. It would have been more convenient if the tool had provided direct support for the inclusion of whole or portions of data-structures in the conceptual modules.

The need for this analysis also arose when performing the object-oriented analysis case study described in Section 4.1.2. The ability to create conceptual modules out of lines of code using particular types in the system would have significantly simplified the task of assigning lines of code to proposed classes.

Currently, the intermediate representation does not contain information about types or data-structures so the tool itself is presently unable to perform this type of analysis. For this analysis to be possible, the intermediate representation would have to include data structure, including field information and type information, in its variable dependence relation. Rather than just including this information as a portion of the variable name in the vardep relation, it would speed up analysis if it were included as a separate field in the relation—the `type` field.

# Bibliography

[1] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[2] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *Proceedings of the 15th International Conference on Software Engineering*, pages 509–519. IEEE Computer Society Press, April 1993.

[3] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.

[4] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[5] Paul Clements. What is Software Architecture? And Why Do I Care? In *NOAA Symposium on Software Engineering*, September 1994.

[6] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 138–156. IEEE Computer Society Press, May 1992.

[7] Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting A Compiler.* The Benjamin/Cummings Publishing Company, Inc., 1998.

[8] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, pages 179–185, April 1995.

[9] William G Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transactions of Software Engineering and Methodology*, 2(3):228–269, July 1993.

[10] Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. In *Proceedings of the ACM SIGSOFT '94 Symposium*

*on the Foundations of Software Engineering*, pages 2–10. ACM Press, December 1994.

[11] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering*. IEEE Computer Society Press, 1997.

[12] J.R. Lyle and D. Binkley. Program slicing in the presence of pointers. In *Proceedings of the 1993 Software Engineering Research Forum*, pages 255–260. ACM Press, November 1993.

[13] H. A. Muller. *Rigi-A Model for Software System Construction, Integration and Evolution based on Module Interface Specifications*. PhD thesis, Rice University, Houston, TX, 1986.

[14] Hausi A. Müller, S. R. Tilley, M. A. Orgun, B. D. Corrie, and N. H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In *SIGSOFT'92: Proceedings of the Fifth ACM SIGSOFT: Symposium on Software Development Environment*, pages 88–98, December 1992.

[15] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, October 1995.

[16] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 19th International Conference on Software Engineering*, pages 338–348. ACM Press, May 1997.

[17] David Lorge Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, pages 279–290, Sorrento, Italy, May 1994. IEEE Compuser Society Press.

[18] S.P. Reiss. *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development*. Kluwer Academic Publishers, Amsterdam, Netherlands, 1995.

[19] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 41–52. ACM Press, October 1995.

[20] R. W. Schwanke, V. A. Strack, and T. Werthmann-Auzinger. Industrial software architecture with gestalt, 1996.

[21] Mary Shaw and David Garlan. *Software Architecture*. Prentice Hall, Upper Saddle River, NJ 07458, 1996.

[22] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. Rigi: A visualization environment for reverse engineering. In *Proceedings of the 19th International Conference on Software Engineering*, pages 606–607. ACM Press, May 1997.

[23] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[24] L. Wall, T. Christiansen, and R.L. Schwartz. *Programming Perl*. O'Reilly & Associates, 1991.

[25] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*. IEEE Computer Society Press, March 1981.

[26] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary Hall, Monica Lam, and John Hennessy. The SUIF compiler system: a parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Stanford University, Computer Systems Laboratory, May 1994.

[27] Alexander S. Yeh, David R. Harris, and Melissa P. Chase. Manipulating recovered software architecture views. In *Proceedings of the 19th International Conference on Software Engineering*, pages 184–194. ACM Press, May 1997.

# Appendix

# Appendix A

# Menu Items of the Tool

The user interface of the tool consists of both a query interface and a menu-driven interface. The menu-driven interface is shown in Figure A.1. Each of the menu items is described in more detail below.

It should be noted that when "line-number" is referred to, the string "filename line number" is actually required. For example, when adding line 1241 of file `foo`, the user is required to enter: `foo 1241`. This is not true when adding and removing ranges of lines.

When "variable name" is mentioned, the string "filename variable name " is actually required. For instance, when adding variable `bar` of procedure `raz` in file `foo` the user is required to enter: `foo raz.bar`.

A When "procedure-name" is mentioned, "filename procedure name" is actually required. Thus, when adding procedure `raz` of file `foo` the user is actually required to enter `foo raz`.

1. *Create a new conceptual module.*

   Allows the user to enter a name for a new conceptual module.

2. *Add lines to a conceptual module.*

   The user is prompted to type a conceptual module name and a list of line-numbers terminated by "END_LINES". Those lines are then added to the conceptual module.

3. *Add a procedure to a conceptual module.*

   The user is asked to type in a procedure-name and a conceptual module name, and the lines in that procedure are added to the conceptual module.

4. *Add a range of lines to a conceptual module.*

   The user enters the filename, then the first line, then the last line of the range. These lines are then added to the conceptual module.

5. *Add a variable to a conceptual module*

   The user adds a list of variable-names terminated by "END_VARIABLES", and then those variables are added to the conceptual module.

6. *Remove lines from a conceptual module.*

   The user gives a line-number and it is removed from the conceptual module.

7. *Remove a variable from a conceptual module.*

   The user enters a variable-name and it is removed from a conceptual module.

8. *Print the conceptual module names.*

   Prints out the names of all the currently defined conceptual modules.

9. *Load a conceptual module from a file.*

   The user is prompted for a conceptual module name and a filename, and it puts the contents of the file into the conceptual module.

10. *Save a conceptual module to a file.*

    The user is prompted for a conceptual module name and a filename, and the contents of the conceptual module are written out to that file.

11. *Print out the procedure starts in the IR.*

    Writes to standard output all the procedures in the intermediate representation.

12. *Print the line numbers of a conceptual module.*

    The user is prompted for the name of a conceptual module, and then all the lines comprising that module are written to standard output.

13. *Perform initial analysis on a conceptual module.*

    The user is prompted for the name of a conceptual module, and then single conceptual module analysis is performed for that conceptual module.

14. *Print the relationship of two conceptual modules.*

    The user is prompted for a conceptual module name, and direct relationship analysis is performed as well as overlap analysis, and contains analysis.

15. *Execute a user defined query.*

    Causes the method `USER.QUERY()` to be called.

16. *Turn inferred analysis on/off.*

    Toggles the inferred analysis outlined in Section 3.2.2 between on and off.

17. *Determine an indirect relationship between two conceptual modules.*

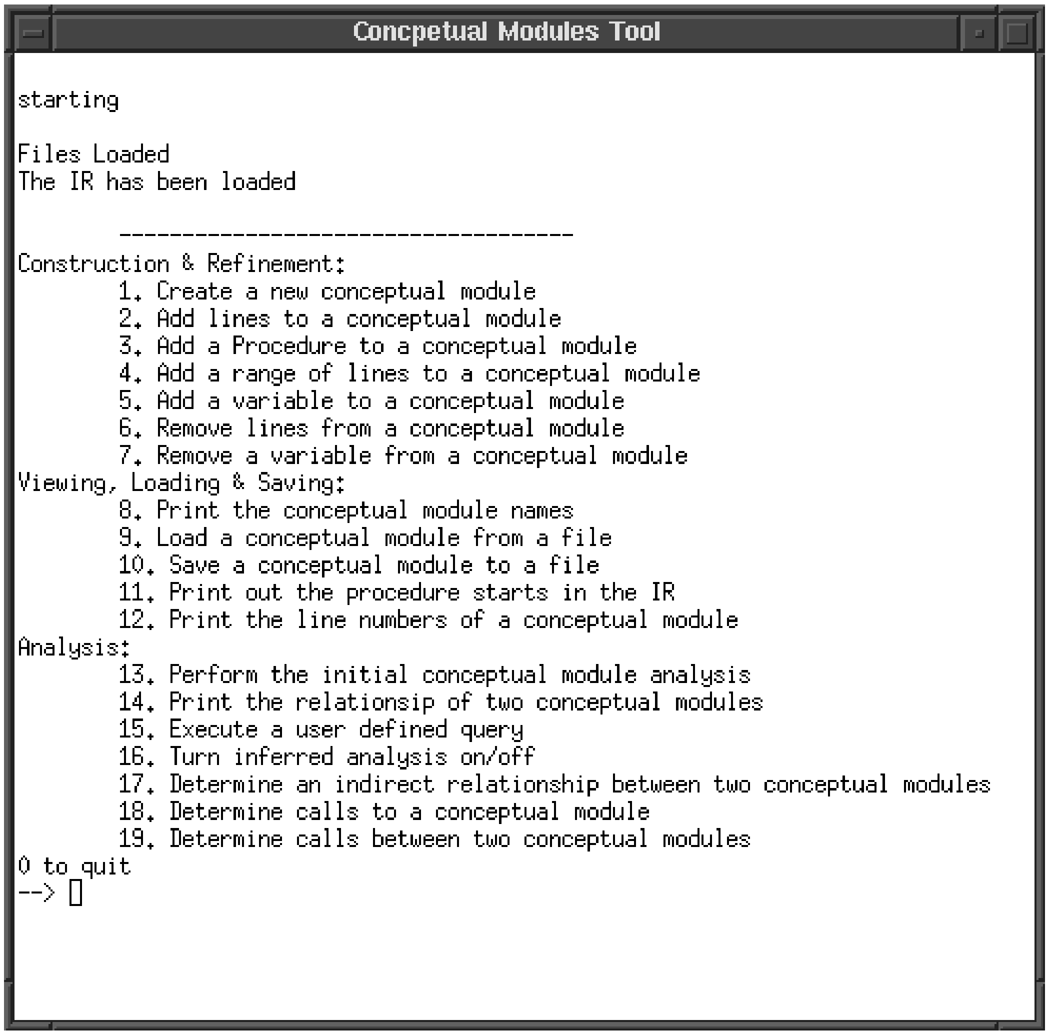
    Prompts the user for two conceptual module names, and determines if there is an indirect relationship from the first to the second.

18. *Determine calls into a conceptual module.*

    The user is prompted for a conceptual module name, and performs calls-in analysis (as described in Section 3.2.2) on the conceptual module.

19. *Determine calls between two conceptual modules.*

    The user is prompted for two conceptual module names, and determines if lines in one conceptual module calls the other.



```
starting

Files Loaded
The IR has been loaded

        -----------------------------------
Construction & Refinement:
        1. Create a new conceptual module
        2. Add lines to a conceptual module
        3. Add a Procedure to a conceptual module
        4. Add a range of lines to a conceptual module
        5. Add a variable to a conceptual module
        6. Remove lines from a conceptual module
        7. Remove a variable from a conceptual module
Viewing, Loading & Saving:
        8. Print the conceptual module names
        9. Load a conceptual module from a file
        10. Save a conceptual module to a file
        11. Print out the procedure starts in the IR
        12. Print the line numbers of a conceptual module
Analysis:
        13. Perform the initial conceptual module analysis
        14. Print the relationsip of two conceptual modules
        15. Execute a user defined query
        16. Turn inferred analysis on/off
        17. Determine an indirect relationship between two conceptual modules
        18. Determine calls to a conceptual module
        19. Determine calls between two conceptual modules
0 to quit
-->
```

Figure A.1: User interface window of the tool