Conceptual Module Querying for Software Reengineering

Elisa L. A. Baniassad and Gail C. Murphy[♣]
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver B.C. Canada V6T 1Z4
+1 604 822 5169
{bani,murphy}@cs.ubc.ca

Technical Report UBC-CS TR-97-14

ABSTRACT

Many tools have been built to analyze source. Most of these tools do not adequately support reengineering activities because they do not allow a software engineer to simultaneously perform queries about both the existing and the desired source structure.

This paper introduces the *conceptual module* approach that overcomes this limitation. A conceptual module is a set of lines of source that are treated as a logical unit. We show how the approach simplifies the gathering of source information for reengineering tasks, and describe how a tool to support the approach was built as a front-end to existing source analysis tools.

Keywords

Source code analysis, software reuse, code scavenge, software structure, modularization, reverse engineering

1 INTRODUCTION

Many reengineering activities performed by software engineers require reasoning about the source code for the system. Part of the reengineering process, for instance, may involve the identification and formation of new software components from the existing code base.

A large number of tools have been built to help software engineers analyze source code. These tools provide an engineer with various views of the source. For example, cross-reference tools, such as Cscope [13] and CIA [2], help the engineer identify and view parts of the source relevant to specified program items, including variables and procedures. Program slicers allow an engineer to view an executable subset of the source contributing (or emanating from) a particular program point [16]. Reverse engineering tools help engineers identify and view higher-level structure present in source [3].

Although these tools can help the engineer understand the existing source code, the views they provide do not adequately support many of the reengineering tasks an engineer performs. Consider an engineer faced with the task of restructuring a C code base. To plan and perform this task, an engineer needs to determine the components to form, the interaction of the components with the remaining source code, and the interactions between newly formed components. Existing program understanding tools unnecessarily complicate these investigations in one of two ways. Tools that provide fine-grained information about the existing source, such as the `use` sites of particular variables, do not allow an engineer to query this information in terms of the desired reengineered structure. Tools that support the expression of the reengineered structure, such as some reverse engineering tools, restrict the queries the engineer can perform about the interactions between the components and the existing source. The engineer can thus not query the source in terms of both the existing and desired structure.

In this paper, we describe an approach that overcomes this limitation by allowing software engineers to analyze existing source in terms of conceptual modules. We define a conceptual module as a set of lines of source code that are to be treated as a logical unit. When a conceptual module is defined, an initial analysis is performed to determine the interface and internal structure of the unit. This analyzed information may be used to simplify subsequent queries about the conceptual module. The approach is supported by a tool that allows engineers to iteratively define conceptual modules for an existing code base, and to analyze data- and control-flow interactions both between a particular conceptual module and the existing source and between one or more conceptual modules.

Since our work focuses on the formation of conceptual modules and subsequent queries involving the modules, we

♣ Phone: 604.822.5169; Fax: 604.822.5485

have architected our tool as a front-end to different kinds of source code analyzers. To date, we have connected the tool to various C source code analyzers, such as Field [11], and tools built on the SUIF [17] compiler framework.

Our approach provides two benefits. First, the approach simplifies the gathering of source information for many tasks by removing the burden from software engineers of correlating and summarizing the results of multiple low-level queries. Second, the approach demonstrates how support for queries about conceptual structural information can be layered onto existing source code analysis tools.

We begin with a description of the activities performed during a sample reengineering activity—the formation of a new software component (Section 2). We then describe the conceptual module approach and tool (Section 3). Next, we show how the approach can greatly simplify the analysis of source for various reengineering activities, and describe the role it has played in some reengineering scenarios (Section 4). Section 5 discusses the role of context when performing queries during reengineering activities, and discusses design choices we made in the definition of our approach. A comparison of the approach to related work (Section 6) follows. We conclude with a summary of the paper and an outline of future work (Section 7).

2 A SAMPLE REENGINEERING ACTIVITY

To clarify some of the information needs of a software engineer trying to extract a software component, we consider the task of isolating and forming an input filter component from a system built in the Unix pipe-and-filter style, the GNU **sort** program.[1] This program 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`.

An engineer may wish to form and extract an input pipe component 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] Sometimes, 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 [5]. Other times, the code lines 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 software engineer determines that code from the `sort` function also contributes to the desired initialization procedure of the target input pipe component. Figure 1 shows a snippet of the relevant code

```
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 1 Lines of Source Contributing to Desired Input Pipe Component

from the `main` and `sort` functions. This code is spread across multiple, non-contiguous, lines of source code.

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 any additional source lines that must be included to provide 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.

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

[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.

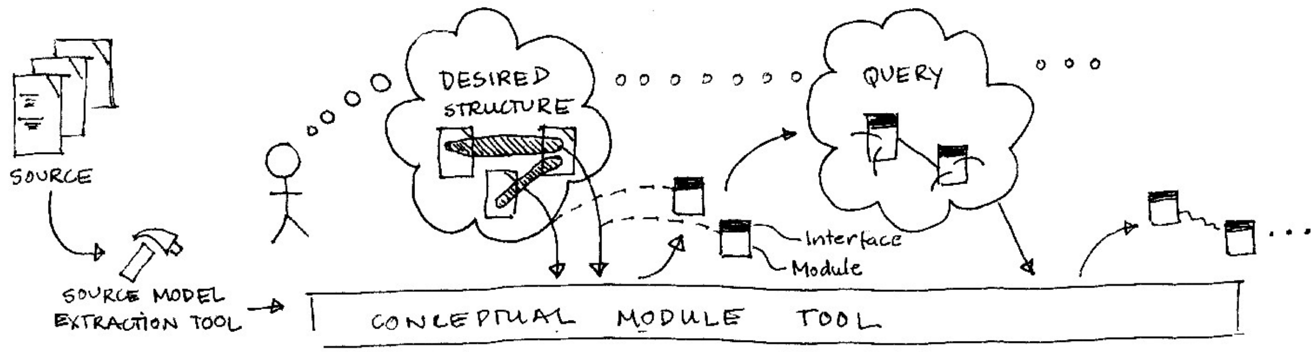

Figure 2 Process of Using the Conceptual Module Tool

3 CONCEPTUAL MODULES

The conceptual module approach and tool provides direct support for analyzing the interface of a desired component by allowing a software engineer to explicitly describe the target structure of interest before performing queries on the source. Figure 2 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. As each conceptual module is defined, the tool performs analysis to determine the module's interface. The engineer may view the results of this analysis. The engineer may also then perform queries about the relationship between the conceptual module and the existing source, and about the relationship between conceptual modules. The steps of defining a conceptual module and performing subsequent queries are performed iteratively by an engineer.

In the case of sort, for example, the engineer describes the lines of code[1] in the existing source that contribute to the functionality of the desired input pipe initialization procedure. The tool responds with the following analyzed information:

```
Input variables:
 sortalloc, main.ofp, main.minus, main.i,
 main.tmp, sort.buf, main.outfile, errno,
 sort.nfiles, main.argv, main.argc

Output variables:
 main.mergeonly, sort.ofp, sortalloc,
 main.ofp, main.checkonly, main.minus,
 instat, sort.buf, errno, sort.nfiles, fp,
 sort.fp

Local variables:
 main.files, main.nfiles, sort.files

Control transfers from Input_Pipe_Init
module:
 main to xmalloc at sort.c 1796
 main to check at sort.c 2081
 main to exit at sort.c 2081
 main to strcmp at sort.c 2104
 main to fstat at sort.c 2107
 main to stat at sort.c 2107
 main to strcmp at sort.c 2107
 main to error at sort.c 2111
 main to xfopen at sort.c 2124
 main to error at sort.c 2131
 main to merge at sort.c 2146
 main to sort at sort.c 2148
 sort to initbuf at sort.c 239
 sort to xfopen at sort.c 247
 sort to fillbuf at sort.c 248
```

Based on this information, the software engineer may decide to alter the lines of code contributing to the conceptual module, and to re-compute the analysis with the altered information.

We describe our approach and tools in more detail in the next two sections. First, we describe source model on which the analysis is based. Then, in Section 3.2, we describe the query language. In Section 3.3, we describe the tools built to support the approach.

3.1 Source Model

The model extracted from an existing source code base is targeted at a procedural language and consists of three relations:

- a *variable dependence* relation that describes `uses`, `defs`, or `use-def` pairs for a variable. The `use` and `def` sites are described by source line numbers.

[1] The following lines were included in the module: 228, 239, 245-249, 1741, 1796, 2071, 2073-4, 2041, 2081, 2796, 2098, 2104, 2107, 2111, 2124, 2131, 2137, 2146, 2148, and 2796.

- a *control transfer* relation that identifies the source line numbers containing call sites and, in each case, the procedure called.
- a *procedure* relation that describes the source line number on which the definition of each procedure starts.

3.2 Query Language

The query language supports the definition of conceptual modules (Section 3.2.1) and the investigation of interactions involving conceptual modules (Section 3.2.2).

3.2.1 Forming a Conceptual Module

A conceptual module is formed by providing a name for the module and a set of lines of existing code contributing to that module. A software engineer may specify lines of code to include in a conceptual module in two different ways: by specifying particular lines of code, or by specifying pieces of logical structure in the existing code that are then automatically converted into lines of code. For example, in `sort`, an engineer may choose to include the `fp` variable in a conceptual module formed to represent the target input pipe initialization procedure. The tool will scan the source model to determine the lines of code where the variable is either used or defined; the tool will then use these lines as the definition of the conceptual module. The engineer may also add a procedure to a conceptual module. In this case, all of the lines of code deemed to be inside the procedure, based on the procedure relation, are included in the conceptual module.

Given the lines of code, the tool determines the input variables, local variables, and output variables of the conceptual module, and the calls made from the conceptual module to procedures in the existing source. The analysis used to determine this information is run in two phases. The first phase assumes that the source model provides complete information, meaning that the variable dependence information includes `use-def` pairs. A second phase of analysis, called inferred analysis, is used when only `use` or `def` information is provided. Table 1 describes the rules applied during both phases of the analysis.

3.2.2 Querying with Conceptual Modules

The information presented to a software engineer based on the formation of a conceptual module summarizes the interaction of a conceptual module with the existing source. As the engineer builds up the conceptual structure, there is also a need to support queries between conceptual modules. We have found four types of queries useful when applying the tool to some reengineering scenarios.

We refer to the first type of query as a *direct* relationship query. This query checks whether one conceptual module, A, provides a definition for a variable which is used in a second conceptual module, B. If so, we say there is a direct relationship from A to B. This query is based on, and requires, `use-def` information.

The second type of query checks for an *indirect* relationship. This query uses the chains of `use-def` information contained in the variable dependence relation to determine all definition points of variables named in a conceptual module, B. If a definition point is contained within another conceptual module, A, we say there is an indirect relationship from A to B.

The last two queries allow the software engineer to check definitional relationships about the conceptual modules. The *overlapped* query determines if two conceptual modules share any lines of code. The *contains* query determines if the source lines comprising one conceptual module are a subset of the source lines comprising another conceptual module.

Table 1: Conceptual Module Analysis Rules

Rule	Description
Local Variable	A local variable is identified in two phases. In the first phase, the `use-def` tuples of the variable dependence relation pertaining to the lines in the conceptual module are considered. Variables for which all of the `use` and `def` sites are in the conceptual module become local variables. The second phase deals with variable dependence tuples describing only `use` or only `def` sites. It considers all such tuples involving input and output variables of the conceptual module previously identified. If all known `use` and `def` sites of a variable are on lines included in the conceptual module, the variable becomes a local variable.
Input Variable	An input variable is one that is used on a line contained inside the conceptual module, but for which there exists a definition on a line which is not contained in the module.
Output Variable	An output variable is one that is defined on a line contained inside the conceptual module, but for which there exists a use on a line that is not contained in the module.
Control Transfer	A control transfer is a call to a procedure not included in the conceptual module for which a call site is included in the conceptual module.

Simple determining the presence of a relationship between two conceptual modules is generally not sufficient to help a software engineer perform a reengineering task. Often, the engineer needs to understand the particular program items, for instances, variables, involved in the relationship. To provide the engineer flexibility in probing the details of a relationship, we provide a programmable interface to our tool. In the case of the containment relationship, for instance, a user may wish to ask if a contained conceptual module has any input or output variables in common with its container. The tool provides two primitives, `inVarsInCommon` and `outVarsInCommon`, that provide access to this information. Each of these primitives returns a set of variable names shared by the two conceptual modules.

Through the programmable interface, a software engineer also has access to the following conceptual module information: input, output, and local variable names, the definitions and uses of those variables, the lines of code the module spans, and the calls made by code in the module. Primitives are also provided to navigate `use-def` chain information in the source model, and to examine how two conceptual modules relate, including relations through common variables, common line numbers, and common calls. Each of the primitives returns an array of strings; this format allows the results of one query to be easily in subsequent queries

3.3 Tool Support

A tool to support the formation and querying of conceptual modules has been implemented in Java [1] and Perl [15]. This tool provides a simple textual interface that may be used to form the modules and query about defined relations. A user may also program queries in Java.

To support the investigation of the approach, we also built a source model extraction tool and wrote some scripts to form an appropriate source model from an existing extractor.

The source model extraction tool was built on the SUIF compiler framework [17]. This framework provides access to an intermediate representation of a multi-file software system. Similar to other SUIF tools, the tools to extract a source model act as filters on the SUIF intermediate representation. Four filters were built: one filter transforms line information about loops to support extraction, a second extracts control transfer information, a third performs points-to analysis using Steensgaard's algorithm [12], and a fourth, which uses the sharlit data-flow analysis framework [14], extracts variable `use-def` information and records information about the definition of procedures.

Scripts were also written to transform the output of the Field cross-reference database for use with the conceptual module query tool. These scripts support the formation of two kinds of source models. The first kind of source model includes information about `uses` and some `defs`, but no `use-def` pairs. The second kind of source model includes the cross-product of all `use` and `def` points for a given variable.

4 USING CONCEPTUAL MODULES

To evaluate the effectiveness of our approach, we applied our tool and several existing tools to two reengineering scenarios. The first scenario considers the component formation and extraction outlined in Section 2, namely 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`.

We applied four program understanding tools representing different technologies to these tasks. In addition to our conceptual module tool, we applied the Unravel slicing tool [7], the Lackwit tool that is based on type-inferencing [10], and the cross-reference database tool, xrefdb, that is distributed as part of the Field programming environment [11].

A different source model extractor was used to provide data to the conceptual module tool for each task. The source model used for the first task was extracted using the tool we built on the SUIF framework. The source model used for the second task was extracted using Field and was post-processed to include the cross-product of `use-def` pairs.

4.1 Scenario 1: Extracting a Component from `sort`

As described in Section 2, the `sort` program is built as a pipe-and-filter system. 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. We applied the tools to this task after identifying, based on a perusal of the source, a modest number—less than ten lines—of source that should be included in the component.

The Unravel tool supports the computation of backward slices given a variable name and a program point (line of code). For this task, we wanted to compute backward slices on variables from the pre-identified lines of code. The slices we computed in this way were large. In all cases, because the slice computations were several hours, we interrupted the computation and viewed partial slices. Each of the partial slices was over 750 nodes in size. Qualitative inspection of these slices did reveal 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 sort filter, not the input pipe, functionality.

With Lackwit, we computed and viewed graphs showing the procedures affecting the values of particular variables. These graphs were useful in determining the procedures in which potentially relevant code might be located, but they did not provide specific information about relevant source lines. A graph we computed for the buf variable in the fillbuf procedure, for instance, included 23 procedures. 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 5 of the procedures as containing code relevant to the task at hand.

In the case of the xrefdb tool computed by Field, we queried the tool for the lines comprising all references and all declarations of variables identified of interest. With these queries, we identified 126 lines of source code for qualitative assessment. 30% of these lines were assessed to be relevant to the task.

As described earlier in the paper, we applied the conceptual module tool 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, we visited the definition points reported in the analysis for the input variable, sort.buf, and found additional lines of source to include in the module. To form the desired procedure, we iterated through this process approximately six times.

We found it straightforward to apply the conceptual module tool to this task because, at any point, we were only considering limited information about the source, such as the definition points of input variables or use points of output variables. This information was determined and provided in the context of the desired structure. The conceptual module tool performed the filtering that we had to do manually when using the other techniques.

Although the conceptual module tool provided support for many aspects of the task, it sometimes includes information in the analyzed interface that does not contribute to the performance of the task. For instance, the tool will report input variables that are used in a procedure call included as a source line that the software engineer intends to remove or rename. It would be helpful to selectively elide this information at some points during the tool's use.

4.2 Scenario #2: Restructuring `adventure`

The adventure program is an exploration game that has been distributed as part of the Unix operating system for many years.[1] The game was originally written in Fortran and was later converted to C. The source now consists of approximately 8,000 lines of C code distributed across 13 files.[2]

A substantial amount of the functionality of the game resides in a 525-line main procedure where control-flow between labels is used to move a player through the game. The restructuring task was to form procedures to encapsulate different states of the game. We began by trying to encapsulate three labeled areas of the main function as procedures. We then wanted to understand how the desired procedures interact through state information. For instance, we wanted 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, we wanted 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, we computed only a few sample slices. As was the case for sort, the slices were large, 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 because the vast majority of the functionality was included in the 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 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, the tool is unable to report information about interactions between different variables.

We applied our tool by forming conceptual modules for each of the desired procedures consisting of the identified source lines. We then wrote a user-defined form of indirect query to determine if there were common definition points for the target procedure. Figure 3 shows the query. For each conceptual module, this query computes the use-def chains of the input and local 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 by the

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

[2] We slightly modified the distributed version of the source 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.

```
public static void advChain()
 throws java.io.IOException, java.lang.InterruptedException,
         java.io.FileNotFoundException
        {
                SET common = new SET(); // Create a new vector of strings
                // Get the first conceptual module in the list of modules of interest
                Module first = (Module)Module.ModuleTable.elementAt(0);
                // Get the use-def chains for all input and local variables of that module
                common=DefUse.GetFullUseDefChain(first);

                // For the rest of the modules...
                for(int i=1; i<Module.ModuleTable.size(); i++) {
                   // Get the use-def chains for the next module
                   Module current = (Module)Module.ModuleTable.elementAt(i);
                   SET  curr_chain= DefUse.GetFullDefUseChain(current);

                   // Interect the chains to determine common definition points (variable name
                   // and line numbers)
                   common = DefUse.INTERSECTION(common, curr_chain);
                }
                common.print();    // Print out the common definition points
        }
```

Figure 3 Query for Common Definition Points of Multiple Conceptual Modules

module, the tool provided a direct way to access the information of interest.

5 DISCUSSION

The reengineering scenarios highlight some of the effects the context and form of specifying a query, and the format for reporting results from a query can have on the usability of a tool to support reengineering tasks. We discuss each of these aspects in relation to our tool. In this section, we also consider our design choices of using line numbers to drive the tool, and the format of the source model.

5.1 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 in the query results as occurred when applying Lackwit to the task on `sort`.

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. If information about the interaction between the conceptual module and the rest of the code or other conceptual modules is needed, the software engineer may perform further queries based on the definition of that module. The local analysis performed on the conceptual module can also help in these situations by reducing the number of subsequent queries that need be performed. For example, an engineer tracing all variables affecting the input variables to a module may ignore the local and output variables of the module.

5.2 Query Form

Often, when performing a software reengineering task, there is a need to perform queries over groups of structural items. For the task on `adventure`, 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 we used, and most of which we are aware, provide support for this kind of grouped queries. Instead, the user must perform both a series of queries, and 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, the use of conceptual modules over information extracted from the xrefdb database eliminated the need for the multiple queries applied when directly using xrefdb.

5.3 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` 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.

5.4 The Use of Line Numbers

We chose to base our conceptual module tool on line numbers for two 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.

5.5 Role of the Source Model

Our approach supports a range of source models: a source model may comprise either `use-def` chain information, or uncorrelated `use` and `def` information. We use the analysis function of our tool to "smooth-out" these differing forms of source model information. We believe 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 implemented 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.

6 RELATED WORK

In the presentation of the reengineering scenarios and the discussion, we have compared our approach to a number of existing technologies, including program databases, program slicers, and type inferencing tools. In this section, we consider the relationship of our approach to particular program database and slicing approaches aimed at overcoming the limitations described earlier, and we compare our approach to reverse engineering approaches.

Consens et. al. describe the application of the GraphLog query language to software engineering as a means of easing the investigation of complex relationships between elements of a software system [4]. In GraphLog, a query is expressed as a graph. Given a query, the system determines all instances of the given pattern existing in the database. Similar to the other database approaches we have discussed, GraphLog does not provide any support for expressing a desired reengineered structure. As a result, an engineer must manually track how the results of query map to the desired structure. In contrast, the conceptual module tool allows an engineer to express the desired structure, removing the need for this tracking.

A generalization of slicing that provides additional support for expressing the context of the slice is chopping. Given a set of definitions, a source, and a set of uses, a sink, a chop of "a program identifies a subset of its statements that account for all influences of source on sink" [6]. By specifying a region of the program of interest, an engineer may find a chop easier to interpret than the slices we applied in our reengineering scenarios. Chops, however, are confined to a single procedure, restricting their use in many reengineering tasks. Similar to slices, chops function to identify statements of interest to the engineer; the engineer is responsible for interpreting and analyzing those statements and determining such information as the interface to a new desired component.

Similar to reverse engineering tools, the conceptual module approach helps a software engineer analyze and understand structural aspects of a software system. However, whereas reverse engineering tools help an engineer abstract structural information gathered from source, our approach helps an engineer overlay a fine-grained structure onto the existing source and then ask questions about how the overlayed structure interacts with the existing structure. Two reverse engineering approaches that provide limited support for investigating the interaction between the two structures are the Rigi system [8] and the MITRE Software Architecture Recovery Tool (ManSART) [18].

The Rigi system is a semi-automated reverse engineering 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 on features of the source, such as naming conventions. In the Rigi environment, a user may perform pre-defined queries on the interactions between two clustered elements. For instance, a user may request an "exact interface" report on a node that provides information similar to the analysis we perform on a conceptual module. However, in contrast to the conceptual module tool, the Rigi environment does not provide any capability for the user to use this information in subsequent queries about the interaction of the node with other nodes or with the existing system.

The ManSART environment provides support to semi-automatically recover architectural descriptions from a system's code base. Similar to Rigi, ManSART displays graphical views of recovered structure to an engineer. These views are created by recognizers which extract and analyze information from an 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 the views created, a set of view manipulation operators have been defined that can, among other things, merge views and build hierarchies. These manipulation operators allow a user to access the source information through a pre-defined set of tests called containment analysis. Similar to the containment and overlap queries in our approach, 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 sets, however, that an engineer can query the relationship between the abstracted and existing structure.

In providing a mechanism to view existing structure in terms of a new structure, the conceptual module approach is also similar to the software reflexion model technique [9]. The reflexion model technique summarizes information extracted from the source in terms of a high-level box-and-arrow diagram of the system specified by the engineer. A key feature of the technique is its mapping language which eases the specification of the association between the source and the high-level model. These two techniques are complementary. The reflexion model technique may be used to determine a coarse-grained mapping between the source and the target reengineered structure. The queries supported by the conceptual module approach may then be used to investigate and refine the boundaries of the new target components. These queries require the conceptual module tool to have knowledge of the semantics of the source model; in contrast, the reflexion model tools process the source and high-level models in a syntactic manner.

7 SUMMARY AND FUTURE WORK

A software engineer performing a reengineering activity must typically understand and manage three forms of information:

- the structure of the existing source,
- the structure of the desired reengineering source, and
- the relationship between the reengineered and existing structures.

Existing source code analysis and reverse engineering tools do not provide adequate support to the engineer in all of these dimensions.

In this paper, we have described the conceptual module approach that allows a software engineer to express the desired reengineered source in terms of the existing source, and to then perform queries about the existing source in terms of the reengineered structure. The approach can support the description of many desired reengineered structures because a conceptual module is defined on the basis of lines of code. In particular, the approach can support reengineered structures in which components consist of non-contiguous source lines.

We have also shown how the conceptual module approach can simplify the gathering of information from source during reengineering activities. This simplification is a result of filtering applied by the tool based on the context of the defined and analyzed conceptual modules. This approach augments, rather than replaces, existing techniques and tools for source analysis and program understanding. We described, for instance, the use of the approach in conjunction with information analyzed by the cross-reference database tool distributed with the Field programming environment.

In addition to providing support for reengineering, the conceptual approach may provide a suitable framework on which to perform architectural design conformance checks. For example, the query language can be used to determine, in a system built according to a pipe-and-filter architecture, if the output pipe ever flows data back to the input pipe. Such a flow may break the invariants of the architectural style. The approach may also be used to improve the scalability of some source code analyses. For instance, slicers might be built to take advantage of the analyzed information in the conceptual module by focusing on input and output variables and ignoring local variables.

ACKNOWLEDGMENTS

We thank Robert O'Callahan for the use of LackWit. We also thank David Notkin for comments on an earlier draft of this paper.

This research was funded in part by a Natural Sciences and Engineering Research Council of Canada research grant, and in part by funding from the University of British Columbia.

REFERENCES

1. Arnold, K and Gosling, J. The Java Programming Language, Addison-Wesley, 1996.
2. Chen, Y.F., Nishimoto, M.Y. and Ramamoorthy, C.V. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3): 225-234, (March 1990).

3. Chikofsky, E.J. and Cross II, J.H. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1): 13-17, (1990).

4. Consens, M., Mendelzon, A. and Ryman, A. Visualizing and Querying Software Structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 138-156, May 1992.

5. Griswold, W.G. and Notkin, D. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3): 228-269, (July 1993).

6. Jackson, D and Rollins, E. A new model of program dependences for reverse engineering. In *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 2-10, New Orleans LA, December 1994.

7. Lyle, J.R. and Binkley, D. Program slicing in the presence of pointers. In *Proceedings of the 1993 Software Engineering Research Forum*, pages 255-260, Orlando, FL, November 1993.

8. Muller, H.A. and Klashinsky, K. A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80-86, April 1988.

9. Murphy, G.C. Notkin, D. and Sullivan, K. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18-28, Washington, D.C., October 1995.

10. O'Callahan, R. and Jackson, D. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 19th International Conference on Software Engineering*, pages 338-348, Boston, MA, May 1996.

11. Reiss, S. Connecting Tools using Message Passing in the Field Program Development Environment. *IEEE Software*, 7(4): 57-66, (1990).

12. Steensgaard, B. Points-to Analysis in Almost Linear Time. In *Proceedings of the Twenty-third Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32-41, Petersburg Beach, FL, January 1996.

13. Steffen, J.L. Interactive Examination of a C Program with Cscope. In *Proceedings of the USENIX Winter Conference*, pages 170-175, Berkeley, CA, January 1985.

14. Tjiang, S.W.K. and Hennessy, J.L. Sharlit—A Tool for Building Optimizers. In *Proceedings of ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 82-93, July 1992.

15. Wall, L. and Schwartz, R.L. Programming Perl, O'Reilly & Associates Inc., 1991.

16. Weiser, M. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4) 352-357, (July 1984).

17. Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S.M. and Hennesy, J.L. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12): 31-37, (December, 1994)

Yeh, A.S., Harris, D.R., and Chase, M.P. Manipulating Recovered Software Architecture Views. In *Proceedings of the 19th International Conference on Software Engineering*, pages 184-194, Boston MA, May 1997.