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
{bani,murphy}@cs.ubc.ca

ABSTRACT
Many tools have been built to analyze source code. 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 instance, cross-reference tools, such as Cscope [14] and CIA [3], help the engineer identify and view parts of the source relevant to specified program items, such as variables and procedures. Program slicers allow an engineer to view an executable subset of the source contributing to (or emanating from) a particular program point [17]. Reverse engineering tools help engineers identify and view higher-level structure derived from the source [4].

Although these tools can help an 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 the 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 reengineered structure restrict the queries that an engineer can perform about the interactions between the new 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. The tool provides two interfaces: a menu-driver interface simplifies the formation of conceptual modules and provides access to pre-coded queries; a programmatic interface supports task-specific analysis of conceptual modules.

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

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 [12], and tools built on the SUIF [18] 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 [6]. 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 and the uses of variables contributing to `fp`'s value, 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 from the `main` and `sort` procedures. This code is spread across multiple, non-contiguous, lines of source code.

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

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

    sort( files, nfiles, ofp );


sort( char **files, int nfiles, FILE *ofp ) {
  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

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.

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

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

[2] Although it may seem trivial to build an input filter, a number of subtleties 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.

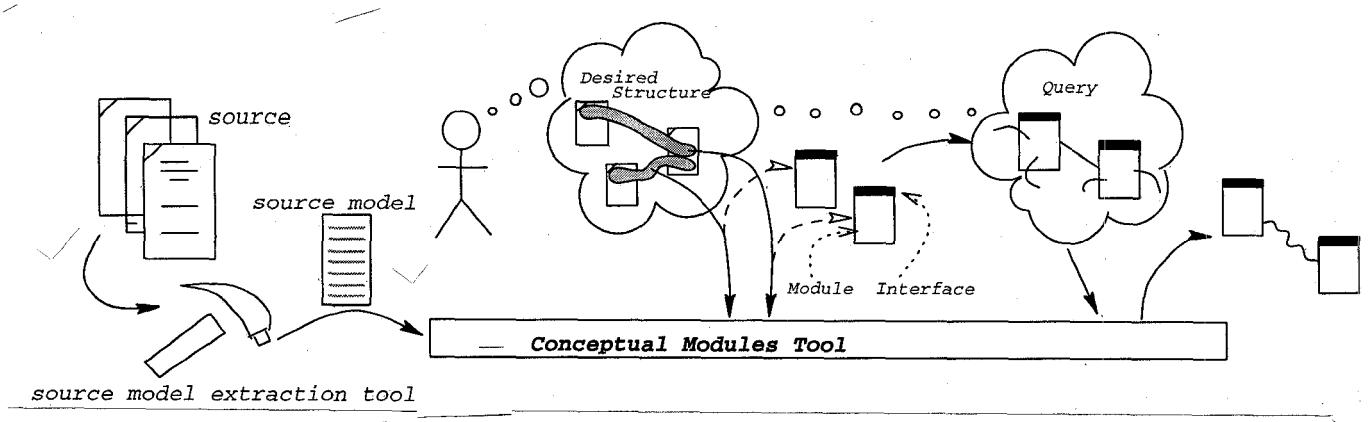

Figure 2 Process of Using the Conceptual Module Tool

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 and structure. The engineer may view the results of this analysis. The engineer may also perform queries about the relationship between the conceptual module and the existing source, and about the relationships 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 in the existing source that contribute to the functionality of the desired input pipe initialization procedure.[3] The tool responds with the analyzed information shown in Figure 3. Based on this information, the software engineer may decide to alter the lines of code contributing to the conceptual module, causing the tool to re-analyze the module with the altered information.

We describe our approach and tools in more detail in the next two sections. First, we describe the 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 existing source code 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.[4] The `use` and `def` sites are described by source line numbers.
- 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 stating a name for the module and by stating 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 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 part of 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. For example, Figure 3 shows that the `ofp` variable from the `main` procedure is an input variable to the newly formed input pipe initialization component. The figure also shows that the module includes code on line 2124 of `sort.c` that calls the `xfopen` function. The analysis used to determine this information is performed in two phases. The first phase assumes that the source model provides complete

[3] 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.

[4] A use-def pair describes a reaching definition for a specific use of a variable.

```
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:
xmalloc at sort.c 1796, check at sort.c 2081, exit at sort.c 2081,
strcmp at sort.c 2104, fstat at sort.c 2107, stat at sort.c 2107, strcmp at sort.c 2107
error at sort.c 2111, xfopen at sort.c 2124, error at sort.c 2131, merge at sort.c 2146
sort at sort.c 2148, initbuf at sort.c 239, xfopen at sort.c 247, fillbuf at sort.c 248
```

Figure 3 Output from Construction of Conceptual Module

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 direct 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` pair information.

The second type of query checks for an *indirect* relationship. This query forms chains of `use-def` information from the use-def pairs contained in the variable dependence relation to determine all definition points of variables in a particular 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.

Simply determining the presence of a relationship between two conceptual modules is generally not sufficient to help a

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 that 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.

software engineer perform a reengineering task. Often, the engineer needs to understand the particular program items, for instance, variables involved in the relationship. To provide the engineer flexibility in probing the details of a relationship, we provide a program interface that provides a software engineer access to

- input, output, and local variable names,
- definitions and uses of conceptual module variables,
- lines of code spanned by the module,
- calls made to and by code in the conceptual module, and
- relationship information between conceptual modules including common line numbers, variables and calls [2].

Most of the primitives return an array of strings; this format allows the results of one query to be easily used in subsequent queries. An example of the use the program interface appears in Section 4.2.

3.3 Tool Support

A tool to support the formation and querying of conceptual modules has been implemented in Java [1] and Perl [16]. As described earlier, this tool provides both a menu-driven interface and an interface programmable in Java.

We have also built a source model extraction tool upon the SUIF compiler framework [18]. This framework provides access to an intermediate representation of a multi-file software system. Similar to other SUIF tools, the components of the extractor tool 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 [13], and a fourth, which uses the sharlit data-flow analysis framework [15], 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 [12] 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 compared the use of our tool to several existing tools in the context of two reengineering scenarios. We have also conducted a case study that investigated the use of our tool in an actual reengineering task.

For the comparison scenarios, we applied four program understanding tools representing different technologies to two tasks. In addition to our conceptual module tool, we applied the Unravel slicing tool [8], the Lackwit tool that is based on type-inferencing [11], and the cross-reference database tool, xrefdb, that is distributed as part of the Field programming environment [12]. The task for the first scenario considered the component formation and extraction outlined in Section 2, namely the creation of an input pipe component from the GNU `sort` program. The second scenario considered a restructuring task: the re-modularization of a legacy C program, `adventure`. A different source model extractor was used to provide data to the conceptual module tool for each task. The source model for the first task was extracted using the tool we built on the SUIF framework. The source model for the second task was extracted using Field and was post-processed to include the cross-product of `use-def` pairs.

For the case study, the tool was used to help a graduate student identify and remove unwanted code in a binary decision diagram package consisting of over 45,000 lines of C code. The student was removing the code to help parallelize the package.

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 of source lines, less than ten, 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 took 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 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 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 Field xrefdb tool, we queried 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 these 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 considering only 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 help the user perform the task. For instance, a variable that is not relevant to the task may be listed as an output variable because it appears as one of the arguments to a procedure call that is on a line included in the conceptual module. It would be helpful to selectively elide this information at various 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.[5] 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.[6]

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 all of these procedures.

It was difficult to apply a slicing tool to this problem because many variables were 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 only able to compute pairwise intersections 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 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 5 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 module, the tool provided a direct way to access the information of interest.

4.3 Case Study : Extracting a Subset of CUDD

The CU Decision Diagram Package (CUDD)[7] provides functions to manipulate multiple forms of decision diagrams. The system is comprised of 47,796 lines of commented C code. A Computer Science graduate student

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

[6] 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.

[7] CUDD was written by Fabio Somenzi (University of Colorado, Boulder); the version used was Release 2.1.2.

```
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 5 Query for Common Definition Points of Multiple Conceptual Modules

wanted to extract one form of diagram—the ZDD diagram—from the package while leaving another—the BDD diagram functionality—unaffected as a step in creating a parallel version of the package.

A search for the string "Zdd" indicated that at least 2000 lines must be considered as part of this task. The student's initial approach was to examine each line of code returned by the search. Using this approach, the student spent twenty hours removing 1000 lines of code. To speed the task, we helped the student use our tool to construct two conceptual modules: the first comprised the ZDD-related code targeted for deletion; the other contained the BDD functionality to be preserved. We then used direct and indirect relationship queries, and queries about the calls between the two conceptual modules to determine if there were any dependences from the BDD to the ZDD functionality. The queries reported just under 200 such dependences. An examination of the code causing the dependences found that all the lines could safely be moved from the BDD module to the ZDD module as the lines represented non-BDD functionality. We then repeated the queries to validate that all dependences had indeed been severed. The queries took 7 minutes to run on a SUN Sparc 5 (SunOS 5) with 64Mb of memory.

After removing the 2200 lines of code in the ZDD conceptual module, regression testing was performed on the remaining code. This testing did not find any affect on the BDD functionality from the removal. The student used our tool for a total of four hours and was able to finish the extraction of the remaining 1200 lines of code in six hours, resulting in a significant time-savings over the original approach.

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 as a basis for 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 the return of a large number of false positives with respect to the task. This situation arose 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 reengineering task, there is a need to perform queries over groups of structural items such as all of the procedures, calls or variables in a chosen portion of code. For instance when restructuring `adventure`, we needed to perform a query about all of the variables referenced within a block of code. None of the tools we used as part of the scenario, and none of which we are aware, provide support for this type of grouped query. Instead, the user must perform a series of queries, and combine the results, 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 use existing text editors and analysis tools like grep to easily identify source to map to a conceptual module. The use of line numbers in the source model also enhances the flexibility of the tool, making it possible to connect the tool to different source model extractors; different extractors can agree on line numbers whereas they may differ in interpretations of abstract representations such as abstract-syntax trees.

To date, any imprecision resulting from the use of line numbers to identify source of interest has not hindered us in completing our desired tasks. Further investigation is needed to determine if this is true for a wider range of reengineering tasks.

5.5 Role of the Source Model

Our approach supports a range of source models: a source model may comprise either `use-def` pair 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 can interpret the results of applying our tool to that 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 that are aimed at overcoming the limitations described earlier. We also compare our approach to reverse engineering tools.

Consens et al. introduced the GraphLog query language to ease the investigation of complex relationships between elements of a software system [5]. In GraphLog, a query is expressed as a graph, easing the expression of queries involving transitive closure. Given a query, the system determines all instances of the given pattern existing in the database. Similar to other database approaches we have discussed, GraphLog does not provide any direct support for expressing a desired reengineering structure; an engineer must manually track how the results of queries map to a desired structure. GraphLog, however, could be incorporated into the conceptual module tool as a replacement for the existing programmable query language. The use of GraphLog might simplify the investigation of how conceptual modules relate to each other and to the existing source.

Chopping, a generalization of slicing, addresses the difficulty of expressing query context when slicing. A chop identifies a subset of the statements of a program that account for all influences of a given set of definitions (source) on a given set of uses (sink) [7]. Although chopping does not provide any direct support for expressing desired components and analyzing their interface and internal structure, chopping could be used to investigate relationships between components by performing chops on code assigned to each component. Since chops include control dependence information that is

not currently included in the source models used by the conceptual module tool, chopping may identify relationships not indicated by the use of the conceptual module tool. Chopping could thus be used to extend the analysis functionality of the conceptual module tool.

Reverse engineering tools, similar to the conceptual module approach, help a software engineer analyze and understand structural aspects of a software system. The analysis performed by reverse engineering tools is intended to help an engineer abstract structural information gathered from source so as to better understand the existing software structure. In contrast, our approach helps an engineer overlay a fine-grained structure onto the existing source and then ask questions about how the new, overlayed structure interacts with the existing structure. Two reverse engineering approaches that do provide some support for investigating the interaction between the two structures are the Rigi system [9] and the MITRE Software Architecture Recovery Tool (ManSART) [19].

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 recover semi-automatically 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 that extract and analyze information from an abstract-syntax tree 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 [10]. 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. An important feature of the technique is its mapping language. This language 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 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 and tool. This approach allows a software engineer to express a desired reengineered structure in terms of the existing source, and to then perform queries about the existing source in terms of the reengineered structure. We have shown how the 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, our approach may provide a suitable framework on which to perform architectural design conformance checks. For example, the query language can be used to determine, for a system built according to a pipe-and-filter architecture, if an output pipe ever flows data back to an input pipe. Such a flow may break the invariants of the architectural style.

ACKNOWLEDGMENTS

We thank Robert O'Callahan and Daniel Jackson for the use of LackWit, Yvonne Coady for her participation in the case study, and David Notkin for comments on an earlier

draft of this paper. We also thank the anonymous reviewers for their helpful comments.

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. Baniassad, E.L.A. Conceptual modules: Expressing desired structure for software reengineering, MSc. Thesis, University of British Columbia, Vancouver, December 1997.
3. 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).
4. Chikofsky, E.J. and Cross II, J.H. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1): 13-17, (1990).
5. 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.
6. Griswold, W.G. and Notkin, D. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3): 228-269, (July 1993).
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. Reiss, S. Connecting tools using message passing in the Field program development environment. *IEEE Software*, 7(4): 57-66, (1990).
13. Steensgaard, B. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32-41, Petersburg Beach, FL, January 1996.
14. 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.
15. Tjiang, S.W.K. and Hennessy, J.L. Sharlit—a tool for building optimizers. In *Proceedings of ACM the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 82-93, July 1992.
16. Wall, L. and Schwartz, R.L. Programming Perl, O'Reilly & Associates Inc., 1991.
17. Weiser, M. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4) 352-357, (July 1984).
18. 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)
19. 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.