Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies

Martin P. Robillard and Gail C. Murphy

Department of Computer Science
University of British Columbia
2366 Main Mall, Vancouver, BC
Canada V6T 1Z4

{mrobilla,murphy}@cs.ubc.ca

Technical Report UBC-CS-TR-2001-13

10 September, 2001

ABSTRACT

Many maintenance tasks address concerns, or features, that are not well modularized in the source code comprising a system. Existing approaches available to help software developers locate and manage scattered concerns use a representation based on lines of source code, complicating the analysis of the concerns. In this paper, we introduce the Concern Graph representation that abstracts the implementation details of a concern and makes explicit the relationships between different parts of the concern. The abstraction used in a Concern Graph has been designed to allow an obvious and inexpensive mapping back to the corresponding source code. To investigate the practical tradeoffs related to this approach, we have built the Feature Exploration and Analysis tool (FEAT) that allows a developer to manipulate a concern representation extracted from a Java system, and to analyze the relationships of that concern to the code base. We have used this tool to find and describe concerns related to software change tasks. We have performed case studies to evaluate the feasibility, usability, and scalability of the approach. Our results indicate that Concern Graphs can be used to document a concern for change, that developers unfamiliar with Concern Graphs can use them effectively, and that the underlying technology scales to industrial-sized programs.

Keywords

Separation of Concerns, Program Analysis, Software Evolution

1. INTRODUCTION

Achieving a suitable design for a software system involves, in part, choosing modules to localize concerns that are anticipated to change over the system's lifetime [21]. A software developer who is later asked to modify the system along an anticipated path benefits from this modularity: the developer can easily identify the code related to the change and can usually make the change in a localized way that respects the existing interfaces.

Unfortunately, software developers are all too often faced with modification tasks that do not involve localized code. Consider, for instance, a developer asked to modify the conditions under which logging occurs in a server based on Jakarta Tomcat.[1] Such a change would require the developer to consider 47 of the 148 (32%) Java source files comprising the core of Tomcat. Sometimes, such non-localized concern code is the result of inadequate design. More often, it is the result of either unanticipated modifications or of a lack of expressibility in the technology available to the original designer to express interacting or overlapping concerns. The end result is the same: software developers must handle concern code scattered across a system's source when modifying the concern, implementing a new feature that interacts with the concern, or evaluating the cost of a planned change [2], amongst other tasks.

Several approaches are available to help software developers locate and manage scattered concern code. Lexical searching tools, such as grep [1], code browsers, such as the Smalltalk integrated development environment [13], cross-reference databases, such as CIA [7], and slicers [26], can each help a developer identify relevant points in the code and can help elicit the relationships between the different parts of a program. Alternatively, a developer may be able to leverage the identification of the change from a previous modification using version differencing in a source code repository. All of these tools produce a similar result: the developer is presented with the lines of source code contributing to the concern in the system. This ad hoc, source code-intensive representation of concerns is difficult to use as the basis for reasoning about and analyzing concerns for the purpose of software evolution.

In this paper, we introduce Concern Graphs, a representation of concerns that we argue is more effective than lines of source code for the purpose of documenting and analyzing concerns. A Concern Graph abstracts the implementation details of a concern by storing the key structure implementing a concern. By storing structure, a Concern Graph documents explicitly the relationships between the different elements of a concern. This relationship information can provide a partial explanation for the inclusion of specific code elements into a concern. A Concern Graph is based on a program model that can be extracted automatically from either the source code or an intermediate representation of a program. As a result, a developer is able to manipulate and navigate a concern representation at a more abstract level than the source code without investing any effort to create the abstract representation. These properties allow a Concern Graph to be augmented incrementally from related elements in the code base, as well as collapsed to a more abstract form.

[1] Reference implementation of the Java Servlets and JavaServer Pages, release 3.2.3

```
protected void log( String s ) {
   if( name == null ) {
      String cname=this.getClass().getName();
      name=cname.substring( cname.lastIndexOf(".") +1);
      name=name + ": ";
   }
   if( cm!= null )
      cm.log( name + s );
   else
      System.out.println(name + s );
}
```

Figure 1: The `log(String)` **method of class** `org.apache.-tomcat.core.BaseInterceptor`

Our goal in creating Concern Graphs was to have a representation of concerns that requires a low to moderate amount of effort to create, a small amount of effort to manipulate and analyze, and that has a conceptually obvious and computationally inexpensive mapping back to the corresponding source code. To investigate whether these tradeoffs were possible, we built FEAT, a tool for visualizing, creating, and analyzing Concern Graphs for Java programs. As initial validation, we have used this tool to investigate whether Concern Graphs can help developers evolving a system find scattered concern code. Our results indicate that Concern Graphs can be used to document a concern for change, that developers unfamiliar with Concern Graphs can use the concept to identify a concern effectively, and that the underlying technology scales to a medium-sized industrial program.

The rest of the paper is organized as follows. Section 2 motivates the need for Concern Graphs. Section 3 describes Concern Graphs and the underlying program model. Section 4 describes the FEAT tool. Section 5 describes cases studies about the use of FEAT and Concern Graphs, Section 6 discusses outstanding issues, Section 7 presents related work, and Section 8 summarizes the paper.

2. MOTIVATION

To demonstrate the shortcomings of using source code to represent a concern, we return to the Tomcat code base. In this scenario, consider that we have been asked to make a modification to the logging functionality. From a quick perusal of the package organization of the code base, we find a `logging` package with three classes: `Logger`, `TomcatLogger`, and `LogHelper`. To make the modification, we need to determine how these classes are used. A lexical search for the keyword "log" returns, among many others, class `InvokerInterceptor`. This class declares a method, `requestMap`, which calls a method `log`. The declaration of this `log` method cannot be found in the class `InvokerInterceptor` itself and thus we must thus look at the parent class, `BaseInterceptor`.

Perusing this class, we find the declaration of the `log` method, reproduced in Figure 1. We see that after local operations on strings, the message is logged through a `cm` object. More browsing is required to determine that `cm` is a field of class `BaseInterceptor` of type `ContextManager`. This knowledge points us to the next step, the declaration of method `log` in `ContextManager`, or one of its superclasses. This time we are luckier; we find the method declaration directly in `ContextManager`. We need to repeat steps similar to those above to determine that the `log` method in `ContextManager` uses the value of a `loghelper` field of type `LogHelper` to determine how to log the message. Finally, looking at `LogHelper`, we unravel dependencies involving the three classes of the `logging` package.

The behavior for this small fraction of the logging concern is simple. However, we needed to browse 6 classes (and source files) to track it down. We also needed to use a lexical searching tool that introduced uncertainty into the result. And, in the end, we have a flat, low-level description of the concern.

It is not trivial to use this flat representation to analyze the dependencies between the concern and the rest of the code base. Such an analysis is needed to determine the full extent of the logging concern. It is also difficult to use such a representation to determine the interactions between different elements of a concern. Furthermore, the use of this description as a basis for understanding the concern when implementing the change requires some of the reasoning described above to be performed again.

Figure 2 provides an alternative view of the concern, consisting of structural elements contributing to the concern code. The relationships between these elements can be extracted from a program model. The source corresponding to the elements can be located by a developer (or a tool) with good accuracy. The rest of this paper describes how a developer can create and use such structural representations, in the form of Concern Graphs, to ease software evolution tasks.

```
call to BaseInterceptor.log(String) in
   InvokerInterceptor.requestMap()
method BaseInterceptor.log(String)
field BaseInterceptor.cm
method ContextManager.log(String)
field ContextManager.loghelper
class LogHelper
class Logger
class TomcatLogger
```

Figure 2: A description of the logging concern

3. DESCRIBING CONCERNS

A Concern Graph is expressed in terms of a structural model of a program. The model both abstracts and augments source code. The model abstracts the code by eliding implementation details. The model augments code by making the dependencies between different elements explicit. We first present the structural model of programs we use, and then describe how we derive a Concern Graph from this model.

3.1 Structural Program Model

The model represents the declaration and uses of various program elements of class-based object-oriented languages. Formally, such a program is expressed as a graph $P = (V_p, E_p)$, where V_p is the set of vertices, and E_p is the set of labeled, directed edges $e = (l, v_1, v_2)$.

A vertex in P can be one of three types.

- **Class vertex (C)**, represents a global class, considered without its members.
- **Field vertex (F)**, represents a field member of a class.
- **Method vertex (M)**, represents a method member of a class.

An edge in P can be one of six types, depending on the type of vertices it connects: (M,M), (M,F), (M,C), (C,C), (C,M), and (C,F). Edges are labeled with the semantic relationships they represent.

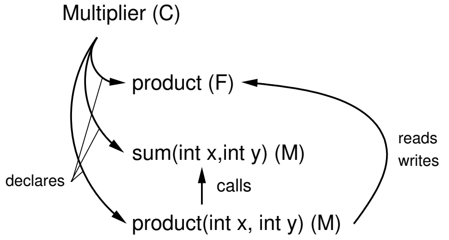

Figure 3: A representation of an multiplier class

(**calls**, m_1, m_2)	The body of method m_1 contains a call that can bind (statically or dynamically) to m_2.
(**reads**, m, f)	The body of method m contains an instruction that reads (uses) a value from field f.
(**writes**, m, f)	The body of method m contains an instruction that writes (defines) a value to field f.
(**checks**, m, c)	The body of method m checks the class of an object, or casts an object, to c.
(**creates**, m, c)	The body of method m creates an object of class c.
(**declares**, $c, \{f\|m\}$)	Class c declares method m or declares field f.
(**superclass**, c_1, c_2)	Class c_2 is the superclass of c_1.

To make this model more concrete, Figure 3 represents the program model for a simple multiplier class. From the graph, we see that class `Multiplier` declares field `product`, and declares methods `sum` and `product`. Furthermore, the method `product` reads and writes the field `product` and calls method `sum`.

3.2 Concern Graphs

We define a Concern Graph $C_{p,i} = (V_{p,i}, V^*_{p,i}, E_{p,i})$ of a program $P = (V_p, E_p)$ to be the *compacted* subset of P documenting the implementation of a concern in P. It is created by the selection of vertices and edges from P.

Defining a Concern Graph consists in defining the sets $V_{p,i}$, $V^*_{p,i}$, and $E_{p,i}$ from elements of the sets V_p and E_p. The set $V_{p,i} \subseteq V_p$ is the set of vertices representing program elements (classes, methods, or fields) that partly implement concern i. The set $V^*_{p,i} \subseteq V_p$ is the set of vertices representing program elements that are entirely devoted to the implementation of concern i (classes or methods). By convention, field vertices are placed in $V_{p,i}$. Additionally, the condition $V_{p,i} \cap V^*_{p,i} = \emptyset$ must hold. Finally, the set $E_{p,i} \subseteq E_p$ represents the edges involved in implementing the concern.

We call such a definition a *compacted* subset of P because it does not constitute the complete subgraph of P describing the concern. Information that can be extracted unambiguously from P can be left out. In particular, any vertex (and corresponding edge) representing fields or methods declared by a class vertex in $V^*_{p,i}$ need not be added to $V_{p,i}$ or $V^*_{p,i}$ (and $E_{p,i}$). Similarly, any edge $(l, v_1, v_2) \mid v_1 \in V^*_{p,i}$ need not be added to $E_{p,i}$.

The complete, or expanded, subset $C'_{p,i} = (V'_{p,i}, E'_{p,i})$ of program P corresponding to concern $C_{p,i}$ is defined as:

$$\begin{aligned} V'_{p,i} &= V_{p,i} \cup V^{**}_{p,i} \cup \{v \mid (l,u,v) \in E_p \wedge u \in V^{**}_{p,i}\} \\ E'_{p,i} &= E_{p,i} \cup \{e = (l,u,v) \mid e \in E_p \wedge u \in V^{**}_{p,i}\} \end{aligned}$$

where

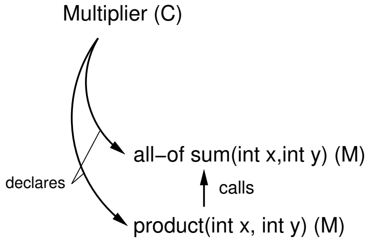

Figure 4: The summing concern in the multiplier class

$$V^{**}_{p,i} = V^*_{p,i} \cup \{v | (l,u,v) \in E_p \wedge u \in V^*_{p,i} \wedge u \in \mathcal{C}_p \wedge v \in \mathcal{M}_p\}$$

and $\mathcal{C}_p$, $\mathcal{M}_p$ are the sets of all class and method vertices in P, respectively.

Continuing with our simple multiplier example, Figure 4 shows the Concern Graph for the summing concern in the multiplier. Here we see that the only elements of class `Multiplier` that are relevant to the concern are the `product` and `sum` methods, and the fact that `product` calls `sum`. Sum is labeled *all-of* to indicate that the entire body of the method contributes to the summing concern. To be precise, sum is a member of $V^*_{\mathrm{Multiplier,\ summing}}$.

3.3 The Expressiveness of Concern Graphs

The tradeoff for the simplicity and abstraction of Concern Graphs is a loss of expressiveness. Loss of expressiveness stems from three principal characteristics.

First, our program model does not include intra-method program elements, such as local control flow or uses of local variables. We chose not to include this information for two reasons. First, capturing local control- and data-flow would increase the size of the program model dramatically. A larger program model would require both more storage and more processing power to manage Concern Graphs. Jackson and Rollins noted this problem with chopping: "Graphs of even the smallest chops tend to be huge" [16, p. 9]. Second, since Concern Graphs are intended to capture scattered concern code, the local information is not as relevant as the non-local information. As the designers of the CIA tool have noted, "Details of interactions between local objects are ignored because they are only interesting in a small context" [7, p. 326].

A second characteristic of our model is that it does not distinguish between different instances of edges. For example, within a method body, the model does not distinguish between two different call sites to the same method. We chose this approach to simplify the Concern Graph representation. We believe it is a reasonable choice because we have observed that when a call to a non-library method contributes to the implementation of a concern, most of the calls to that method are usually related. In situations where this is not the case, the small number of false positives have been easy to manage.

A third characteristic of our model is that it does not currently support exception handling. Exception handling introduces a particular type of control-flow that can be difficult to abstract [8, 23]. We thus chose to leave exception handling aside until we have more experience with Concern Graphs.

Section 6 discusses how these design choices play out in practice, and how they can be addressed.

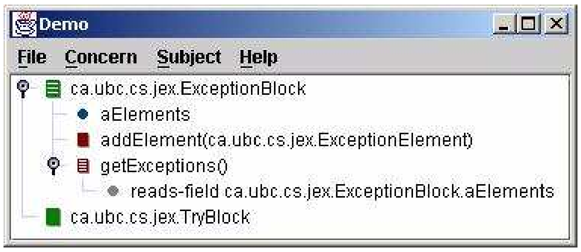

Figure 5: The FEAT window

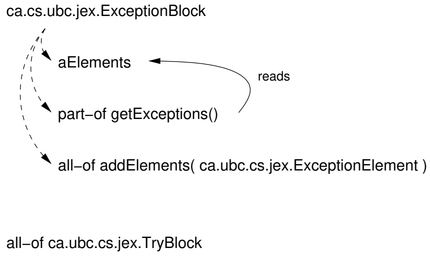

Figure 6: The Concern Graph viewed in Figure 5

4. FINDING CONCERNS IN PRACTICE

For Concern Graphs to be practical, it must be possible to create and manipulate them easily and at a low cost. To support the tasks of finding concerns in source code and of representing those concerns with Concern Graphs, we built the Feature Exploration and Analysis Tool (FEAT). FEAT allows a software engineer to navigate over an extracted model of a Java program and to build up the subset of the model that corresponds to a concern of interest. This section describe the FEAT tool, and the rationale guiding its requirements and implementation.

4.1 FEAT Concepts

FEAT support three main functions.

1. The display of a Concern Graph in a convenient and manageable form for a software developer.

2. Access to the vertices and edges of the structural program model related to vertices in the Concern Graph to support the iterative construction and modification of a Concern Graph.

3. The mapping from the vertices of a Concern Graph to the source code.

Displaying a Concern Graph

To provide software engineers with a clear, uncluttered, and unambiguous view of potentially large Concern Graphs, FEAT displays a Concern Graph as a forest of trees (Figure 5). The root of each tree in the forest is a class that contributes to the implementation of the concern. In front of each class is an indication of whether *all-of* or *part-of* the class is included in the concern (see Section 3.2). *All-of* classes are preceded by a filled rectangle; *part-of* classes are preceded by a striped rectangle. By default, classes are displayed as *part-of*.

There are two advantages to displaying a Concern Graph as a collection of trees. First, trees are easier to lay out than graphs. Second, tree nodes can be collapsed to abstract information. The use of trees does not result in a loss of information since edges severed in the graph-to-tree transformation can be inferred from the hierarchical structure of the tree. For example, a method that is the destination vertex of a *calls* edge can be accessed by expanding the class node declaring the method.

Expanding a *part-of* class displays the members of the class that pertain to the concern described. At this level, the parent-child relationship represents a *declares* edge in a Concern Graph. Fields are preceded by a circle, *all-of* methods are preceded by a filled rectangle, and *part-of* methods are shown next to a striped rectangle. Expanding a *part-of* method displays the code elements in the method body that pertain to the concern. More precisely, the elements shown are the outgoing edges in the program model. From this point, the parent-child representation, if expanded, represents one of the five types of (M,*) edges (see Section 3.1), with the edge's label and the target vertex encoded in the tree node. Superclass edges are not available directly in this view, but can be produced using a simple command.

Figure 5 shows a Concern Graph in FEAT. The figure represents FEAT's *concern view*, the main window holding the representation of the Concern Graph. Figure 6 shows the corresponding Concern Graph. The *declares* edges (dashed arrows) are abstracted by the tree structure, as members are naturally displayed as children of their declaring class. The *reads* edge is represented by the child node of method node `getExceptions()`, which is labeled `reads-field ca.ubc.cs.jex.ExceptionBlock.aElements`.

FEAT provides additional structural information in the form of tool-tip pop-ups. Flying over a *part-of* class name with the cursor shows the fraction of elements involved in the concern. Flying over a field shows the type of the field and the access modifier. Flying over a method shows the return type of the method and the access modifier.

Accessing the program model

FEAT provides a set of queries to enable users to access vertices of the program model that are related to the vertices in the Concern Graph. A user can navigate the program model in both the direct and reverse directions of the edges emanating from the vertices. A user triggers a query by right-clicking on a node in a view and choosing the appropriate query from a pop-pup-menu. The result of the query is a window similar to the main concern view (Figure 5). FEAT supports six queries.

- **Get superclass** returns the superclass of the selected class.

- **Expand class** returns all of the members declared by the selected class. In this view, the methods are displayed in their *all-of* form.

- **Grab class** returns the class declaring the element of a target node in a use edge. For example, performing this query on a node labeled `reads-field C.f` would return class `C`.

- **Fan-in** returns all the vertices in the program model that depend on the class, field, or method node that is selected. The semantics of this query depend on the type of node that is selected. For a field, the result includes all of the methods that access the selected field. For a method, the result includes all possible calling methods, including potential virtual calls. For a class C, the result is more extensive. It includes the declarations of fields of type C and methods with return type C or parameter type C. It also includes methods that access a field declared by C or of type C, and methods that call a method declared by C, or that have a return type C or parameter type C. Finally, the result includes methods checking for type C (i.e., with the keyword *instanceof*), casting to type C, or creating an object of type C.

- **Fan-out** returns all of the outgoing edges for the selected method. Fields do not have outgoing edges.

- **Transitive fan-out** returns all the vertices that the selected method transitively depends on, including fields, methods, and classes.

Mapping to source code

A Concern Graph alone is not sufficient to perform a maintenance task on a system. The source contains useful information such as names, comments, and intra-method structure, and in the end, it is the source that must be modified. To help a developer access this information, FEAT permits a developer to view the source code corresponding to a Concern Graph element.

To access the source, a developer selects an element in the tree view and uses a pop-up menu to request to view the source. For *all-of* methods, the code viewing option brings up a code viewer positioned on the first line of the method. For *part-of* methods, the viewer highlights all lines that correspond to the outgoing edges from the method node. For example, if a method m contains a single *writes* edge to field f, all lines of code where field f is defined in m are highlighted. Similarly, a request to view the code for a single edge, such as `C.m writes-field D.f`, will highlight all the lines in method `C.m` where field `D.f` is defined.

Manipulating the program model

FEAT also allows a developer to manipulate elements of the Concern Graph or query results. For example, a user can remove an element from a view, move an element from an query result window to the main concern window, highlight an element, for instance, to remember that it has been considered, or convert a method or a class from *part-of* to *all-of*. A user may also compare the results of a query window with the main concern window to determine which elements returned by a query are already part of the concern.

Other issues

The program model available in FEAT depends upon configuration information provided to the tool. Configuration information consists of a list of packages to be considered when analyzing targets of virtual method calls, a list of packages to be considered when doing fan-in queries, and a list of packages to elide in query result windows. This configuration information can be specified at any point during a FEAT session: FEAT queries will use the last defined configuration information.

4.2 Implementation Details

FEAT version 1.7.1 supports the Java language. In comparison to traditional source code analyzers and cross-reference tools, such as CIA [7], or the framework of Canfora and colleagues [5], FEAT does not rely on a program database. Rather, it uses the compiled representation (bytecode) of the program directly, and is thus more similar to browsers for languages such as Smalltalk [13], or Trellis [19].

FEAT is implemented in Java. It uses IBM's Jikes Bytecode Toolkit [17] to represent and manipulate Java classes at run-time. Section 5.3 discusses the memory consumption issues related to this approach.

To resolve the targets of virtual method calls, FEAT uses a simplified class hierarchy analysis algorithm [9] that executes over all of the classes in a set of packages specified by the user. The mapping of an edge to source code is performed by extracting line numbers for the bytecode instructions corresponding to the edge under investigation.

A user starts finding a concern with FEAT by providing a seed, usually a single class. This seed can be found by browsing code or system documentation, using lexical searches, or other means. Currently, the seed is expressed through a text file using a small declarative language similar to the description in Figure 2.

5. CASE STUDIES

To investigate whether Concern Graphs are useful to represent concerns when performing maintenance tasks, we undertook a set of case studies. The first case study focused on whether Concern Graphs and FEAT could be used to support effectively a complete change task involving a scattered concern (Section 5.1).

A second case study focused on whether the Concern Graph approach as incorporated in FEAT could be used by non-inventors of the approach, to strengthen and question observations made during the first case study (Section 5.2).

A third case study investigated whether the approach scales (Section 5.3).

5.1 Summarization Case Study

In the first case study, an author of this paper took the role of a maintenance programmer to perform a modification to AVID, a Java visualization software system developed at the University of British Columbia. AVID comprises 12 853 uncommented, non-blank lines of code organized in 177 classes and 16 packages. The participant for this case study had no previous exposure to the code of AVID. [2]

The task

To visualize the execution of Java programs, AVID requires, amongst other inputs, a file containing summarized information about the events generated during the execution of a Java program [25]. This summary file is generated by a summarizing program. The program takes as input an event trace file and produces a summary file that contains information such as the number of calls and the number of objects allocated or deallocated up to a certain point in the trace file, as determined by some user-defined checkpoint frequency. The summary files also contain information about the age of objects at allocation and deallocation time.

The object age information is voluminous, and experience with the visualizer showed that this information was not always used. Being able to generate and read summary files that did not include this object age information was thus a desirable change for AVID, and we chose it for our first case study. As an indication of the scale of this task, the graph representing the model of the summarization program has at least 246 application-specific vertices. [3]

Finding the concern code

In performing this task, the participant used FEAT to discover the concern code that was to be modified, and to save a representation of this code as a memory aid when later performing the change.

The discovery process that was carried out by the participant can be divided into four slightly overlapping phases. A preparatory phase consisted of understanding the application domain and of

[2] The participant was involved in the AVID project as a user of the technology.

[3] This figure does not include the vertices corresponding to the Java library code, which were automatically filtered by FEAT.

seeding the concern. This phase did not involve Concern Graphs or FEAT. A second phase consisted of discovering the part of the code where the writing to the summary files was triggered. A third phase involved understanding and describing the reading and writing mechanism for summary files. A fourth phase consisted of the discovery of a finer implementation detail based on the Concern Graph that was created, while making the change.

To understand the application domain, the participant spoke briefly with an original developer of the system. This developer explained, at an abstract level, the functioning of the visualizer and the use of summary files. This discussion did not involve viewing source code or explicitly mentioning actual data structures. The only exception is that the original developer mentioned the entry point to the summarizing program, class `PrimarySummarization`. This class was used as a seed to the concern and thus, when the participant started using FEAT, the Concern Graph consisted solely of this class name.

In the second phase, the participant looked for the major program elements involved in reading and writing to the summary files as a means of gaining an understanding of the format of the files. Using FEAT, loaded with the single entry-point class `PrimarySummarization`, the participant expanded the class and added the `main` method to the concern description. A fan-out query on the `main` method revealed all of the elements used by `main`.[4] These elements consisted of objects being created, and one call to method `summarize` of class `EventSummarizer`. This element was added to the Concern Graph because it was the only non-library method call. The participant then analyzed the `summarize` method more closely, using both the result of FEAT's fan-out query and the source code viewer. From this information, the participant determined that the only points that could involve writing to the summary file where through calls to `Info.write`, `Summary.write`, and 2 `store` methods. The participant added these elements to the Concern Graph. Figure 7 shows the Concern Graph at this point. To produce this Concern Graph, the participant needed only to find and select the `main`, `summarize`, `write`, and `store` method vertices. Dependent vertices and edges, such as *declares* relationships, are automatically included by FEAT. Furthermore, it was only necessary to view the source code of method `summarize`.

In the next phase, the participant discovered the details of the reading and writing protocol for summary files. Specifically, the participant explored the outgoing edges in the program model of the methods discovered in the previous phase to determine what elements actually performed the reading and writing operations, and then explored the incoming edges to analyze the context in which these operations were performed. This phase was more iterative than the first, and included, in alternation, viewing source code through the automatic highlighting feature of FEAT, and exploring dependencies through the query capabilities of FEAT. Using this process, the participant discovered that the code pertaining to the reading and writing of summary files was located in the methods `add`, `read`, and `write` of classes `Info`, `Summary`, `CategoryInfo`, `CategorySummary`, and `CategoryManager`, and a handful of helper methods in the same classes. Once the complete mechanism was discovered, it was possible to determine, by looking at the corresponding source code, that only a subset of the methods identified dealt with the reading and writing of object age information. Only these methods were added to the Concern Graph.

The second through the third phases required approximately 90 minutes to complete. The Concern Graph produced included 3 fields and 18 methods scattered across 7 classes.

[4]The results did not include filtered elements, which for this case study included all the classes of packages `java.lang`,`java.io`, and `java.util`.

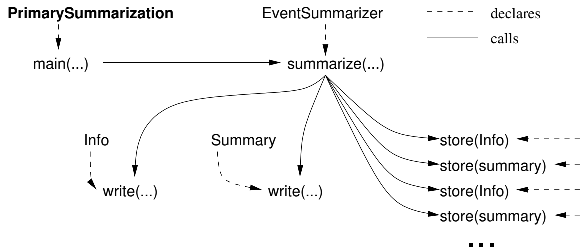

Figure 7: Finding the important parts

Making the change

To implement the change, the participant visited the source file corresponding to each class in the Concern Graph once and implemented the changes needed to that class. Of the 18 methods present in the Concern Graph, 12 had to be modified to implement the required change. Of the remaining 6 methods, 4 had object age-related code that did not need to be changed due to specific implementation details. The 2 other methods were left in the Concern Graph as structural "bridges" between different parts of the code. For example, method `summarize` (see Figure 7) was left in the Concern Graph as a pointer to the `read` and `write` methods, even if no code in `summarize` actually had to be changed. These methods could have been omitted, as they can be obtained easily with FEAT queries.

To test the change, the modified summarization program was used to generate new summary files both with and without the object age information, and these files were used in visualizing event traces. This allowed the participant to discover that one of the assumptions made about the behavior of the concern was wrong. This assumption was that the first read operation on a summary file would be done through the method `read` of class `Info`. Execution of the program revealed that the first read operation was in fact performed through the `read` method of class `Summary`. To remedy this situation, the participant used the Concern Graph in a final phase, to find the site of the first read operation to the summary file. The participant iteratively performed fan-in queries, investigating the resulting call sites with the code browser until the context of the calls was determined.

In subsequent testing, the participant successfully visualized event traces using the new format of summary files. Making the change and testing it required approximately 150 minutes.

Results

We draw five observations about Concern Graphs based on our use of FEAT for this change task. The observations are presented from the point of view of the case study participant, an experienced software developer.

OBSERVATION 1. *The granularity of the Concern Graph was sufficient to describe a concern for the purpose of the software change task.*

The participant did not need to consult any other documentation prior to implementing the change. The general behavior of the code learned as part of creating the Concern Graph was still fresh in memory, and the behavior that was not understood at the time of performing the change could be discovered in minimal time through queries. The Concern Graph also pointed to the target source code with sufficient accuracy. Section 6 discusses the accuracy of the mapping to source in more details.

OBSERVATION 2. *Most of the source code viewed while finding a concern was relevant to the concern.*

An explanation for this observation is that the details of code not related to the concern under investigation were usually discarded at the level of the Concern Graph model.

OBSERVATION 3. *The number of false positive was low.*

In the context of this case study, a false positive is an code element included in the Concern Graph that did not implement the object age feature. In this case study, only 2 out of the 19 methods identified in the Concern Graph were not directly related to the concern. We posit that this low false positive rate is a result of the queries returning elements that are structurally dependent, as compared to text searching tools that can return unrelated items. In this study, the false positives that did occur were methods implementing parts of the object age concern that were not directly impacted by the change.

OBSERVATION 4. *The number of false negatives was low.*

The participant made a single pass through the source files to implement the change. Only one method had to be added to the concern description while performing the change. Our explanation for this observation is that most of the concern code interacts structurally, so the cross-referencing capabilities of FEAT allowed the participant to identify the extent of the concern.

OBSERVATION 5. *The program model was not useful in helping to understand highly algorithmic code.*

The participant determined the reading and writing protocol for summary files by reading the source code and the comments of a few specific files. The Concern Graph was not helpful in understanding this behavior because it did not capture information about the behavior of the concern. We chose not to include such information in a Concern Graph to minimize the effort required to create a graph. Section 6 discusses this tradeoff in more details.

5.2 Usage Case Study

We performed a second case study to explore whether developers unfamiliar with Concern Graphs and FEAT would be able to effectively identify a concern related to a program change task using the approach and tool. We describe the study, present the results, and compare them to the observations that arose from the first case study.

Study setup

In this case study, participants were asked to identify the code contributing to a specified concern in the context of a program change task. The participants were not asked to perform the change. The target for this task was the Jex system version 1.1 [22]. Jex is a static analysis tool that produces a view of the exception flow in a Java program. Jex is written in Java and consists of 57 152 uncommented, non-blank lines of code organized in 542 classes and 18 packages.

The participants were asked to identify the code in Jex that handles Java anonymous classes. The context for identifying this code was to change Jex to support a version of the Java language that did not include anonymous classes.

Using FEAT, a Concern Graph for this concern was produced by the developer of Jex. The elements in this Concern Graph span 8 classes in 3 different packages. A subset of this Concern Graph, consisting of one class and one method, was provided as a seed to the participants of the case study.

The three participants in this study had diverse backgrounds: one was a senior undergraduate student who had worked in two different companies as part of a co-operative work program, one was a graduate student with previous work experience as a software developer, and one was a developer for a telecommunication company. All the participants had some experience with Java, although only one was actively involved in development work with Java at the time of the case study. The participants had no previous exposure to either the source code of Jex or the FEAT tool.

Prior to performing the task, the participants completed a 30 minute training session with the tool, during which they had assistance from the developer of FEAT. The participants were then asked to produce a description of the anonymous class handling concern that was as complete and as precise as possible. The participants were instructed to perform the task using only FEAT. In particular, code viewing was to be done only through FEAT's code highlighting function.

The participants were asked to report the time required to perform the task, their final Concern Graph, a usage log automatically generated by FEAT, and their confidence in the quality of the result, in terms of estimated percentage of the concern they had missed. Two additional participants were involved in prototyping the study. The experiences of these participants caused us to adjust the content of the training session to ensure participants understood how to use the tool. The results of the prototype participants are not included in the results reported.

Results

We analyzed two types of data from the study: the completeness of the Concern Graph produced (quantitative), and the usage patterns of the participants (qualitative).

We used the Concern Graph produced by the author of Jex as a benchmark. One of the 8 classes in this Graph was marked as *all-of*. Of the remaining 7 *part-of* classes, the Concern Graph includes 1 field and 15 methods: 6 methods are labeled *all-of*; 12 code elements, such as the use of a field, are specified as part of the concern in the remaining 9 methods. Figure 8 shows a view of this Concern Graph. The first level of indentation represents classes. The second level of indentation represents class members, and the third level of indentation represents the uses of class members in method bodies.

Table 1 shows how many of these elements were identified by the study participants. Participant 1 found almost all of the concern, corroborating Observation 4 from the first study. The elements missed by this participant were the result of minor inconsistencies in building the Concern Graph. For example, the participant included the call to method `JexFile.isAnonymous` in method `JexVisitor.addExternalNonVirtualCallExceptions`, but failed to include the declaration of method `isAnonymous` itself in the Concern Graph. This situation could be avoided automatically if FEAT included the targets of edges in the Concern Graph. Participants 2 and 3 missed a higher number of elements. The majority of their false negatives resulted from of a failure to see that one field, `aNextAnonymous` of class `TypeDeclarationCollectorVisitor`, was involved in implementing the concern. This field was found by the expert and participant 1. The expert found the field because, in the source code, the field was referenced close to the call to the creation of an `AnonymousJexFile` object in method `visitClassDeclaration`. Reference to this field was also visible in the results of a fan-out query. Once field `aNextAnonymous` is discovered, a fan-in query on the field returns 5 out of the 7 elements of class `TypeDeclarationCollectorVisitor` related to

```
class JexFile
  all-of method isAnonymous
class Workspace
  all-of method getExceptionFromAnonymousClasses
all-of class AnonymousJexFile
class JexLoader
  all-of method getExceptionsFromAnonymousClasses
  all-of method getTypes
class JexPath
  method main
    calls JexPath.getAnonymousJexFiles
  all-of method getAnonymousJexFiles
class JexFileCollection
  method dump
    calls JexFile.isAnonymous
  method writeJexFiles
    checks AnonymousJexFile
class JexVisitor
  method addExternalNonVirtualCallExceptions
    calls JexFile.isAnonymous
  method addVirtualCallExceptions
    calls JexVisitor.addAnonymousVirtualCallExceptions
  all-of method addAnonymousVirtualCallExceptions
class TypeDeclarationCollectorVisitor
  field aNextAnonymous
  method visitNewObjectExpression
    writes TypeDeclarationCollectorVisitor.aNextAnonymous
  method visitClassDeclaration
    creates AnonymousJexFile
    calls AnonymousJexFile.<init>
    reads TypeDeclarationCollectorVisitor.aNextAnonymous
    writes TypeDeclarationCollectorVisitor.aNextAnonymous
  method <init>
    writes TypeDeclarationCollectorVisitor.aNextAnonymous
  method visitTypeDeclarationStatement
    reads TypeDeclarationCollectorVisitor.aNextAnonymous
```

Figure 8: The anonymous class handling concern in Jex

Table 1: Concern completeness results

Participant	1	2	3
Classes found (8)	7	6	8
Field found (1)	1	0	0
Methods found (15)	13	7	11
Code elements found (12)	11	3	7
False positives	2	0	0

the concern.

The number of false positives in the Concern Graphs produced by the participants was low. Of the three participants, only one produced a Concern Graph with false positives: this graph had 2 false positives which were clients of the functionality described by the concern rather than elements of the concern. This data corroborates Observation 3 from study one.

The participants each produced a Concern Graph in less than 50 minutes. We find the quantitative results of this case study encouraging because the participants, who all had minimal training with the concept of Concern Graphs and the FEAT tool, were able to narrow down, in a short amount of time, an unfamiliar code base of 57 KLOC to a Concern Graph that captured many of the pertinent parts of the concern.

Analysis of the usage logs collected from the use of FEAT by the participants show that approximately 80% of the source code viewed while finding a concern was relevant to the concern (Observation 2). This measure is approximative because viewing an element opens the entire source file. As a result, it is possible to view different elements in the same file.

Similar to the first case study, the participants in this study were unable to use Concern Graphs to capture system behavior. Moreover, they were unable to use the approach to represent subtle aspects of the structure (Observation 5). For example, even though both participants 2 and 3 viewed code related to method `JexLoader.getTypes`, neither of these participants incorporated this method in their Concern Graph. The `getTypes` method belonged in the Concern Graph because it was a private method performing specific services for loading anonymous Jex files. To discover this information, participants had to observe that the caller of the method was part of the concern, and that there was no other caller of the method.

We cannot evaluate observation 1 in the context of this study, since the participants did not actually perform the change.

5.3 Scalability Case Study

To evaluate whether the technology supporting Concern Graphs scales for use on medium-sized industrial systems, we applied FEAT to NSC release 2.1, a large network provisioning code base developed by Redback Networks Canada, Inc. The NSC code base comprises 233 packages and 1489 classes. It depends on approximately 9 MB of third-party libraries.

The approach taken in the FEAT tool is to load the entire program model into memory. This approach allows users to quickly perform dependency analyses on any parts of a program, and to dynamically reconfigure the environment used to evaluate the queries.

In the case of the NSC code base, it was not possible to load all of the application classes and their dependent classes into the memory available on the analysis machine.[5] It was thus necessary to selectively restrict the dynamic model of the program. We accomplished this restriction by configuring FEAT to fully load only a user-defined set of classes. Other classes were loaded as stubs that included some information about the class and its members but that did not include the entire bytecode model. A consequence of this tradeoff is that any class loaded as a stub could not be queried for dependencies to a program element, except if these dependencies could be detected without the bytecode (e.g., field types, method parameter types). In practice, this approach does not influence the results of the queries if the classes loaded as stubs do not transitively depend on the application classes of interest, which is generally the case with library code and low-level application code. Loading some classes as stubs does not influence the completeness of the class-hierarchy analysis that is performed to determine the potential targets to virtual calls because this analysis requires only method signatures.

To verify that FEAT was operating correctly given these optimizations, we used it to identify the code corresponding to a port to a new error handling framework that had been added in a previous version of NSC. By differencing the code in the versions recorded before and after the change, we were able to determine that the code we identified using FEAT corresponded to the change.

6. DISCUSSION

The Concern Graph representation trades simplicity of creation and understanding for precision. We discuss the rationale behind the choices we made in choosing the Concern Graphs representation, and in developing the technology to support it. We also discuss how the representation can apply to programs implemented in non-object-oriented languages.

6.1 Properties of Concern Graphs

Concern Graphs are intended to provide a convenient abstraction for the purposes of reasoning about a concern or analyzing a concern. Concern Graphs have three important properties.

[5]The machine used had 256MB of memory.

Concern Graphs are **compact**. Concern Graphs support a compact and local view of a concern by eliding irrelevant (non-concern) code. A developer can see all of the program elements implementing a concern in one location. This locality has been helpful in organizing a change task. A developer can still access related, non-concern code through simple queries over the program model.

Concern Graphs are **simple**. Concern Graphs abstract the details of a class' implementation, distilling syntactically rich statements to simple keywords such as *calls*, or *reads*, and their target code element. This abstraction captures the essence of the relationships between different code elements, making it easier for a developer to focus on the concern. When necessary, elements can be mapped to source code to access the details.

Concern Graphs are **descriptive**. The explicit documentation of the relationships between different program elements belonging to a concern removes the need for a developer to mentally perform the first few steps of program text compilation to derive this information.

The price to pay for the convenience of this abstraction is a non-exact mapping of Concern Graphs to the source code. In practice, this imprecise mapping results in false positives when mapping a significant edge of a Concern Graph to source code, and false negatives when unimportant code has been filtered. In our experience, when making a change based on a Concern Graph, these limitations have not impeded the task. For example, in our first case study, the mapping of the use of a field to the source code indicated lines of code that were not part of the concern. The number of false positives we have had to contend with has been in the range of zero to five instances. Even with a minimal understanding of the concern, it was easy to filter these false positives and it was preferable to spend time doing the filtering rather than having to spend more up-front time dealing with a larger, but more precise, program model. Typically, false negatives have been the result of concern code that involves basic programming language constructs, such as integer arithmetic, or libraries elements, that were not modeled by the Concern Graph. It has also been straightforward to find such code given that it is usually located "close" to modeled code elements. Further investigation is needed to determine whether the current granularity of Concern Graphs is sufficient for more general use. One extension we are currently considering is to distinguish between identical edges having a different projection in the source code.

6.2 Structural Focus

As mentioned earlier, Concern Graphs do not support more behavioral aspects of the implementation of a concern, such as object protocols, or pre- and post- conditions. In the definition of Concern Graphs, we decided to capture structure at the expense of behavior. Our rationale for this decision was to make Concern Graphs as inexpensive as possible to create and maintain. To compensate for the lack of behavior information, the FEAT tool provides a developer with direct access to the source code implementing a concern element. We have observed that developers used this feature in the case studies.

Alternatively, we could provide more support for a developer to describe behavior: for instance, a developer might specify an object protocol [4]. We plan to investigate the annotation of Concern Graphs with additional behavioral information, obtained either from user input or instrumented executions of programs, to complement our current structural description of concerns. Such annotations might help improve the usefulness of Concern Graphs during a change task by reminding a developer of the behavior learned as the concern was uncovered, or by providing documentation of the behavior for a different developer.

6.3 Automatic Analysis

During the case studies, it became apparent that there are situations in which the concern description building process could be automated. While searching for the code of a concern, it was observed that certain elements often occur in pairs. For example, when a method was declared to be *all-of* in a concern, usually all of its call sites were also part of the concern. Similarly, when a constructor call was added to a concern, usually a *creates* edge had to be added as well. Automating such portions of the concern building task might help a user eliminate false negatives in a concern description.

Data-flow-oriented queries may also be helpful when trying to identify the code related to a concern. For example, such a query may be useful to discover where the value supplied as an argument in a method call originates, or which objects, rather than types, actually interact in the program. Localized data-flow analyses might help answer the first kind of question. The value-point relation, which is used as the basis for the Ajax tool [20], might help answer the second kind of question. Each of these approaches could likely be integrated with suitable performance into a FEAT-like tool.

6.4 Tool Support for Other Languages

We believe Concern Graphs could be extended to other programming languages, including procedural languages such as C. The unifying framework for Concern Graphs is the idea of global program elements and their relationships. To port Concern Graphs to another language would mean defining the global program elements representing vertices, the corresponding edges, and defining abstractions representing the edges' label. For the C language, the abstraction described by Chen *et al.* [7] would be a good starting point.

6.5 FEAT Limitations

Other issues that arose when using Concern Graphs were more related to the FEAT tool. One issue was the quality of the program model provided by FEAT. Although FEAT extracts the program model directly from bytecode, the result is neither exact not complete. First, because of the dynamic binding of method calls in Java, FEAT can only provide a conservative estimate of the explicit *calls* edges linking two method vertices. Second, FEAT cannot elicit the *calls* edges implemented through reflection, because these calls cannot be detected statically. Finally, FEAT cannot detect relationships between elements that are expressed in the source code but that are lost through compiler optimizations, such as accesses to final fields (inlined in the bytecode), and calls to inlined methods.

A second issue relates to the use of FEAT to elicit concern descriptions. Most users reported that they sometimes "got lost" in the navigation, not remembering which elements they had investigated and which ones were still untouched. We had provided a marking functionality in FEAT to allow users to highlight concern nodes to provide tracking support, but this feature was not sufficient. The participants of the case studies suggested that visual support for the graph navigation process might address this issue. Such an approach would imply representing program graphs visually, which might be challenging for large programs. Instead, we are considering visualizing the iterative process.

7. RELATED WORK

Many program understanding and reverse engineering approaches have been developed to help a developer discover the code related to a maintenance task.

Slicing [15, 26], chopping [16], and their extensions to object-oriented programming (e.g., slicing class hierarchies [24], or objects [18]) compute the code related to the task based on the input of one or two relevant program points by the developer. These tools determine the code of interest based on a detailed program model that typically includes both intra- and inter-procedural control- and data-flow. As opposed to Concern Graphs, the result of slicing is not abstracted, but is instead presented in terms of the program code. Applications of slicing to program maintenance must provide strategies to deal with resultant size of the information and the details it includes. An example of such a strategy is the system of rules proposed by Gallagher and Lyle [12].

Source code browsers, such as provided in Smalltalk [13] or Trellis [19], and program databases, such as CIA [7], enable a developer to access cross-reference information for source code elements, such as methods. The context for collecting the information by such tools is limited: it is not possible to accumulate the results of cross-referencing queries in a network of program elements. As a result, a developer must manually build a list of program elements pertaining to a concern and manage the context in which these elements are used and queried. Conceptual Modules [3] can help alleviate this problem by capturing segments of program code related to a concern, and providing support for querying these as an entity. However, as opposed to Concern Graphs, conceptual modules do not provide an abstraction above lines of source code.

A number of approaches have been developed to specifically address the problem of finding concerns in source code. Aspect Browser helps developers finds concerns using lexical searches of the program text [14]. It uses the Seesoft [10] concept and a map metaphor to graphically represent the location of code implementing concerns. In comparison, FEAT supports the use of structural queries to discover concerns: we believe structural querying may allow a developer to discover more of a concern faster. However, lexical searching still plays a role in our approach; for instance, we have used lexical searches to define a seed (see Section 4.2). Chen and Rajlich have proposed a technique for finding features (concerns) in source code in which a user systematically traverses a program dependency graph [6]. The dependency graph used in their technique is similar to the program model we use in that it also considers global program elements. However, in addition to forcing a user through a particular consideration of program elements, their approach is limited in that it does not allow users to find concern elements that are related through a non-concern element.

These last two approaches for finding concerns use static program information. Other approaches use information about a program's execution. Wilde *et al.* [27] used carefully designed test cases to locate user functionality in legacy Fortran code. Eisenbarth *et al.* [11] propose to use dynamic information to derive mappings between features and components using concept analysis. Currently, neither Concern Graphs nor the program model from which a Concern Graph is extracted contain information from the program's execution. Dynamic program information could help produce a more complete program model in Java by introducing *calls* edges that result from introspection. More generally, dynamic program information could be used to annotate a Concern Graph with profile or behavioral information.

8. SUMMARY

Many maintenance tasks involve non-localized changes to a system's code base. Often, the non-localized changes correspond to a single conceptual feature, or concern, that was not modularized. We have observed that existing techniques are not sufficiently expressive for documenting the concern prior to making the change, nor abstract enough to use as the basis for analysis.

In this paper, we proposed Concern Graphs, a representation of concerns that is intended to overcome these limitations. Concern Graphs localize an abstracted representation of the program elements contributing to the implementation of the concern. The representation makes dependencies between the contributing program elements explicit. A straightforward mapping between the abstracted program elements and the source allows a developer to recover needed implementation details.

We argued that Concern Graphs are both appropriate for expressing concerns during a program change task and for using as the basis for manipulation and analysis. In this paper, we explored a specific concern analysis task: concern discovery. We presented FEAT, a tool that uses Concern Graphs to support the analysis of dependencies between a concern and the rest of the program.

Three case studies demonstrated the usefulness of Concern Graphs for maintenance tasks. In a first case study, we demonstrated that a Concern Graph was sufficiently precise to describe the program points to change, allowing us to focus on the critical parts of the program. A second case study showed that developers unfamiliar with the Concern Graph representation could use the representation and supporting technology to find a concern scattered in source code. A third case study showed that the technology underlying Concern Graphs scales to industrial-sized systems.

In the future, we plan to investigate the use of Concern Graphs for other analysis tasks, such a concern overlap analysis.

ACKNOWLEDGMENTS

We would like to thank Ron Westfall, Amar Shan and Ian Forster of Redback Networks Canada for making the industrial case study possible. We also thank the original evaluators of FEAT and case study participants for their useful comments: Andrew Catton, Brian De Alwis, Stéphane Durocher, Eric Dyke, and Reid Holmes. Additional thanks go to Albert Lai for helping to fine-tune the performance of FEAT, Stéphane Durocher for his help with the formalization of Concern Graphs, Thad Heinrichs for his help with the summarization case study, and Brian De Alwis for his useful comments on this paper. The Jikes Bytecode Toolkit was provided freely by IBM through the Alphaworks project. This work was funded by an NSERC graduate fellowship and research grant.

REFERENCES

[1] A. V. Aho. Pattern matching in strings. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 325–347, New York, 1980. Academic Press.

[2] R. Arnold and S. Bohner. *Software Change Impact Analysis*. IEEE Computer Society, 1996.

[3] E. L. Baniassad and G. C. Murphy. Conceptual module querying for software reengineering. In *Proceedings of the 20th International Conference on Software Engineering*, pages 64–73. ACM, May 1998.

[4] S. Butkevich, M. Renedo, G. Baumgartner, and M. Young. Compiler and tool support for debugging object protocols. In *Proceedings of the 8th International Symposium on the Foundations of Software Engineering*, pages 50–59. ACM, November 2000.

[5] G. Canfora, A. Cimitile, U. De Carlini, and A. De Lucia. An extensible system for source code analysis. *IEEE Transactions on Software Engineering*, 24(9):721–740, September 1998.

[6] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 241–247. IEEE Computer Society, 2000.

[7] Y.-F. Chen, M. Y. Nishimoto, and C. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[8] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, pages 21–31. ACM, September 1999.

[9] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-oriented Programming*, volume 952 of *LNCS*. Springer-Verlag, August 1995.

[10] S. Eick, J. Steffen, and E. Summer, Jr. Seesoft—a tool for visualizing line-oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[11] T. Eisenbarth, R. Koschke, and D. Simon. Derivation of feature component maps by means of concept analysis. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 176–179. IEEE Computer Society, March 2001.

[12] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[13] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1984.

[14] W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 265–274. ACM, May 2001.

[15] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[16] D. Jackson and E. J. Rollins. A new model of program dependence for reverse engineering. In *Proceedings of the 2rd ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 2–10. ACM, December 1994.

[17] The Jikes bytecode toolkit. IBM, March 2000. http://www.alphaworks.ibm.com/tech/jikesbt.

[18] D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *Proceedings of the International Conference on Software Maintenance*, pages 358–367. IEEE Computer Society, November 1998.

[19] P. D. O'Brien, D. C. Halbert, and M. F. Kilian. The Trellis programming environment. In *Proceedings of the Conference on Object-oriented Programming, Systems, and Applications*, pages 91–102. ACM, October 1987.

[20] R. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, November 2000.

[21] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[22] M. P. Robillard. The Jex home page. http://www.cs.ubc.ca/˜mrobilla/jex.

[23] S. Sinha and M. J. Harrold. Analysis of programs with exception-handling constructs. In *Proceedings of the International Conference on Software Maintenance*, pages 348–357. IEEE Computer Society, November 1998.

[24] F. Tip, J.-D. Choi, J. Field, and G. Ramalingam. Slicing class hierarchies in C++. In *Proceedings of the Conference on Object-oriented Programming, Systems, and Applications*, pages 179–197. ACM, October 1996.

[25] R. J. Walker, G. C. Murphy, J. Steinbok, and M. P. Robillard. Efficient mapping of software system traces to architectural views. In *Proceedings of CASCON 2000*, pages 31–40, 2000.

[26] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[27] N. Wilde, J. A. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proceedings of the Conference on Software Maintenance*, pages 200–205. IEEE Computer Society, November 1992.