A Program Slicing Approach for Locating Functional Concerns

Takashi Ishio[1,2], Ryusuke Niitani[2], Gail C. Murphy[1], Katsuro Inoue[2]

[1]Department of Computer Science
University of British Columbia
2366 Main Mall, Vancouver, BC
Canada V6T 1Z4
ishio@acm.org, murphy@cs.ubc.ca

[2]Graduate School of
Information Science and Technology
Osaka University
1-3 Machikaneyama, Toyonaka,
Osaka 560-8531, Japan
{rniitani, inoue}@ist.osaka-u.ac.jp

22 March 2007

Abstract

A functional concern – code that helps fulfill a functional requirement – is typically implemented by collaborative software modules. When a developer modifies or reuses the implementation of a functional concern, he must find the modules contributing to the concern and understand how the units collaborate with one another. In this paper, we describe an approach for locating the code contributing to a functional concern that is based on program slicing. Our approach uses heuristics to bound the size of the slice determined to represent the functional concern, thereby overcoming the large slice sizes that often limit the usefulness of program slicing based approaches. Our approach outputs a description of the functional concern's implementation as a concern graph, which summarizes the interactions between the program elements in the slice. We report on an evaluation in which we compared the size and content of concern graphs produced by our approach with concern graphs made by hand by two developers. We show that our method can extract concern graphs with appropriate content for a developer automatically, reducing the cost of locating functional concerns.

1 Introduction

Many software developers spend much of their time being a kind of detective, trying to locate, based on a few clues, where particular functionality is implemented in a system's source code. Consider, for example, a software developer, who as part of the team developing the Firefox web browser [1] is assigned to fix

[1]`http://www.mozilla.com/en-US/`, last verified on March 14, 2007

problem #373525, "Certificate Manager opens with wrong tab selected". To work on this problem, the developer must locate the source that implements certificates and that provides the user interface to manage certificates. We refer to the source contributing to a particular functional requirement, like certificate management, as a functional concern. Sometimes, a functional concern of interest is well-modularized and the developer can easily inspect the concern code, understand how it works, and continue with the task. More often, the functional concern code is spread across multiple collaborating modules (e.g. [20]), which complicates the location of all of the code related to the concern and the determination of how the code works together. Some estimate that locating and understanding functional concerns in source code accounts for more than half of the total cost of maintenance [7].

In this paper, we focus primarily on the problem of *locating* a functional concern of interest in a code base. Several approaches have been proposed to aid a developer with concern location. Many approaches involve a developer inspecting the results of searches across the structure of a system to determine which code is related to the functional concern of interest. For instance, the approach of Shepherd and colleagues involves a developer looking at the results of searches across a structural graph augmented with information about the use of identifiers [27]. Other approaches are more automated. For example, Robillard and Murphy introduced an approach to infer the location of concerns based on a programmer's navigation activity. While this approach automates the creation of a description of a concern, the developer must still perform the navigation work to investigate the concern.

We describe in this paper an approach called SCOLOC (Slicing-based COncern LOCation) to further automate concern location that is based on program slicing. To use SCOLOC, the developer provides a list of methods he believes are related to the concern. Using control- and data-flow from a program dependence graph, SCOLOC automatically produces a concern graph [22] describing the code related to the functional concern. The produced concern graph describes the entities—the methods and fields—involved in the concern, and also describes the relationships between those entities through which the individual entities collaborate to provide the functional concern. To ensure our approach is easy to apply, the only input required by the developer is a small list of seed methods associated with the functional concern; this list can be determined using standard techniques, such as identifying methods through keyword search [17, 27]. To ensure we produce output automatically of reasonable size, we use barriers when slicing [15]; in contrast to previous work, we determine the barriers automatically.

In this paper, we also report on an evaluation we conducted to determine if the concern graphs produced by our approach automatically are appropriate for a developer to use for a task. The evaluation compared concern graphs created by hand by two developers to concern graphs created with the SCOLOC approach. We found that we could automatically produce concern graphs of reasonable size with approximately 54% and 52% recall and precision compared to the concern graphs made by hand. This result provides evidence that our approach produces reasonable sized concern graphs with the appropriate content.

We begin by positioning our work in relation to previous work (Section 2). We then describe our approach (Section 3) and describe the implementation of SCOLOC (Section 4). We provide evidence our method is effective (Section 5) and conclude with a summary of our contributions (Section 6).

2 Related Work

The problem of concern location relates to the problem of feature location. In this section, we describe previous efforts in each of these problem areas and describe how our approach relates to previous work on program slicing.

2.1 Feature Location

Feature location techniques identify relationship between user functionality and program units [31]. Feature location techniques include two approaches, static and dynamic.

SNIAFL is a static information retrieval approach to map methods to related features [33]. This method generates a pseudo execution trace that represents the dynamic behavior of the methods for each feature. SNIAFL requires documents that describe features of a target program, such as a user manual. Our approach requires only the program code since developers do not typically write down their knowledge in design documents [18].

Closer to our approach is that of Chen and collegues who proposed a static feature location using abstract system dependence graph (ASDG) [3]. In their approach, a developer first creates a search graph including functions to be investigated. The developer then interactively finds related nodes from ASDG and adds node to the search graph. While both their technique and ours use a system dependence graph, our technique is automated.

Walkinshaw et al. proposed a dynamic approach. They proposed a program slicing based approach to restrict the call graph to contain only methods and calls that may be relevant to the execution of a particular use-case or scenario [29].

Another dynamic approach proposed by Eisenbarth et al. is a method combining formal concept analysis with dynamic analysis [6]. In this approach, a developer gives feature keywords for each execution trace. Formal concept analysis build a concept lattice for feature keywords and executed procedures. We have favoured a static approach to reduce the need to prepare enough input data to locate features.

2.2 Concern Location

Concern location techniques identify a concern spread across a target program. Concerns are usually distinguished from features by focusing on the developer's viewpoint of the code rather than the user's viewpoint of the program. A feature from a user's viewpoint is thus mapped to several functional and non-functional concerns.

Robillard and Murphy proposed to categorize entities investigated by a developer into concerns based on the developer's activity [23]. Robillard also proposed an approach based on static analysis and developers' activity to suggest software entities related to their task [24]. Both of these approaches still require investigation by the developer to create the concern.

Shepherd et al. proposed a natural language processing approach to extract crosscutting concerns from identifiers in the source code [27]. Developers can find source code fragments from a pair of a verb and a direct object. While this approach can extract scattered code related to same action, it does not help developers understand collaboration among modules.

Aspect Mining techniques focus on locating crosscutting concerns to migrate a legacy system to aspect-oriented programming. These techniques also uses heuristics including fan-in values [19], a position of a method call [16], code clone information [2] and a combination of metrics [26]. These heuristics find all candidates of crosscutting concerns in software but they do not distinguish code of a specific crosscutting concern from code of other crosscutting concerns. Our SCOLOC approach can collaborate with these techniques to exclude unrelated crosscutting concerns from a functional concern investigated by a developer.

In our previous work [12], we used heuristics based on syntactic information, such as the return type and the number of parameters for each method, obtained from a method signature to locate a functional concern. However, this syntactic information depends on the coding style of a developer and can easily change during software evolution. In this work, we introduce new heuristics, the similarity and distance metrics. The heuristics improve precision and recall and prevent small code changes from affecting the output of our SCOLOC approach.

2.3 Program Slicing

Our approach is based on program slicing, a technique to extract statements that are relevant to a particular variable [30]. Given a source program p, a *program slice* is a collection of statements possibly affecting the value of *slicing criterion* (in the pair $<s, v>$, s is a statement in p, and v is a variable defined or referred to at s).

Chopping, which is a variant of program slicing, was proposed to support the understanding of how a statement affects another statement [10]. Chopping extracts control-flow and data-flow paths from input variables to output variables in a program dependence graph. This approach requires a developer to understand what might be input and output for a functional concern in terms of data flow. To understand a functional concern, we believe it is reasonable for a developer to identify interesting methods, but not to understand deeply before finding the concern the role of two statements. Another limitation of this labeling is that a developer cannot specify interesting methods or variables in a functional concern, e.g. a variable containing an intermediate state of the functional concern. Therefore, it is not suitable to inspect the structure of a concern.

To make the output of a slicing operation more amenable for a developer to understand, the PROVIS tool in the CANTO environment allows a developer to specify the numbers of forward and backward distance from a selected node that the user wants to be displayed [1]. Krinke also proposed length-limited slicing based on the distance from a criterion to extract a small program slice [14]. These approaches stop the traversal at a vertex after k steps from the criterion vertex. However, it is hard to determine an appropriate parameter k. Krinke also proposed chopping with *barriers* [15]. A barrier is a vertex that terminates the graph traversal. In [15], only chopping criteria (source and target vertices) are identified as barriers by default. A developer adds other barriers by hand if necessary. Our approach aims to automatically identify such barriers in order to extract a small set of statements for specific software entities.

SCOLOC builds on an approach introduced by Kameda to translate a program slice into a concern graph [13]. This approach tries to provide an abstract information of a program slice to developers. Kameda's approach may still output a large concern graph for a large program slice. We combined an automatic barrier identification with Kameda's translation approach to output a small concern graph for developers to easily understand the structure of a concern.

3 The SCOLOC Approach

Our SCOLOC approach takes as input a target program and program entities that a developer has identified as contributing to a functional concern to be located. We apply heuristics to all methods in the target program to identify inter-method edges as *barriers* [15]. A barrier blocks graph traversal during program slicing to exclude methods that are likely not related to the initial set of entities specified by a developer.

The output of SCOLOC is a concern graph [22]. A concern graph contains program entities, such as methods and fields, and relationships between the entities that describe how the entities collaborate to implement the concern. A developer can use the information in a concern graph to understand the functional concern. In addition, we annotate the source code with slice information to support a developer reading a code with the concern graph.

The SCOLOC approach consists of five steps.

1. The SCOLOC tool constructs a PDG for the target program.
2. The software developer selects seed methods in the PDG that contribute to the functional concern.
3. The SCOLOC tool evaluates a heuristic function to identify barriers.
4. The SCOLOC tool calculates a program slice given the PDG, the seed methods and the heuristically-identified barriers.

5. The SCOLOC tool produces a concern graph from the computed slice.

We use the example source code shown in Figure 1 to describe the details of SCOLOC. This program outputs the name and size of files specified by arguments. We investigate a functional concern of how to calculate the size for each file. We select two seed methods `Main.main` (lines 2-8) and `FileList.getFileSize` (lines 37-42) and produce the concern graph shown in Figure 3.

3.1 Heuristic Barrier Identification

To identify barriers in the PDG automatically, we developed a heuristic based on two metrics: similarity and distance. The barriers we identify are edges between methods.

We identify barriers using a function $isBlocked(m, C)$ where m is a method in the PDG and C is the set of seed methods. SCOLOC evaluates this function for each method m in the PDG; if the function returns true, all edges connected to the method m are identified as barriers blocking graph traversal during program slicing.

The *isBlocked* function is defined using the similarity and distance metrics. The similarity metric evaluates the degree to which a method relates semantically to the seed methods. The distance metric evaluates how close a method is to the seed methods in the PDG structure. Equation (1) defines the $isBlocked(m, C)$.

$$\begin{aligned} isBlocked(m, C) &= Similarity(m, C) < threshold_{sim} \\ &\vee \; Distance(m, C) > threshold_{dist} \end{aligned} \quad (1)$$

$$(2)$$

In short, a low $Similarity(m, C)$ value indicates that the method m is not similar to any method included as a seed. A low $Distance(m, C)$ value indicates that the method m is distant from slicing criteria in the graph structure.

3.1.1 Similarity Metric

The similarity metric uses non-local names and their context referred to in a method as a proxy for likely similarity in the functionality of the methods.

We hypothesize that two methods that refer to similar classes, methods and fields contribute similar functionality.

To define the similarity value for two methods, we use a set to represent the names; $NS(m)$ is a case-insensitive token set including names referred to in method m other than the names of local variables. The following steps are used to construct $NS(m)$ for the method m:

1. Initialize $NS(m) = \phi$.

```
public class Main {
  public static void main(String[] args) {
    FileList f = new FileList();
    for (int i=0; i<args.length; ++i) {
      f.add(args[i]);
    }
    f.printFileSize();
  }
}
class FileList {
  private ArrayList files;
  public FileList() {
    files = new ArrayList();
  }
  public void add(String filename) {
    files.add(filename);
  }
  public String getFileName(int i) {
    return (String)files.get(i);
  }
  public long getTotalFileSize() {
    long total = 0;
    for (int i=0; i<files.size(); ++i) {
      total += getFileSize(getFileName(i));
    }
    return total;
  }
  public void printFileSize() {
    for (int i=0; i<files.size(); ++i) {
      System.out.print(getFileName(i));
      System.out.print("\t");
      System.out.println(
        getFileSize(getFileName(i)));
    }
    System.out.println(getTotalFileSize());
  }
  private long getFileSize(String filename) {
    File f = new File(filename);
    if (f.exists() && f.isFile())
      return f.length();
    else return 0;
  }
}
```

Figure 1: Example source code

2. Add the identifier representing m to $NS(m)$. Also add m's declaring class name, return type and parameter types to $NS(m)$.

3. For each method n called by m, add n's name, declaring class, return type, parameter types to $NS(m)$.

4. For each field f accessed by m, add f's name, declaring class, type to $NS(m)$.

5. For each class c referred to in m (used in `new`, `instanceof` or type casting expression), add c's name to $NS(m)$.

6. For each token t in $NS(m)$ extracted using any of the four previous steps, add the *decomposed* tokens of t to $NS(m)$. Decomposition involves the splitting of a long token, a compound name, into several short tokens. For example, `java.lang.StringTokenizer` is decomposed to four tokens: "java", "lang", "string" and "tokenizer". We used the decomposition rules from a component search engine [17] to handle `CamelCase`, `UPPERCase`, `under_score_name`, `Class$Innerclass` and `package.Class`.

After the construction of $NS(m)$ for each method m, we filter out common tokens such as "java" and "object" from each constructed $NS(m)$. A token t is removed from $NS(m)$ if token t appears in more than 20% of the methods in the target program. We also exclude one letter names, such as "x" and "i".

The similarity of two methods is given in equation (2). The similarity value is the probability that a token t in one method is also included in the other method. A high similarity value means the two methods share many tokens making them more likely, in our opinion, to contribute to the same functionality. If $NS(m1)$ equals $NS(m2)$, the similarity is maximum: $\mathit{Similarity}(m, m) = 1$.

$$\begin{aligned}\mathit{Similarity}(m1, m2) &= \frac{1}{2}\Big(\frac{|NS(m1) \cap NS(m2)|}{|NS(m1)|} + \\ &\qquad \frac{|NS(m1) \cap NS(m2)|}{|NS(m2)|}\Big) \qquad (3)\\ \mathit{Similarity}(m, C) &= \max_{n \in C} \mathit{Similarity}(m, n) \qquad (4)\end{aligned}$$

Equation (3) is the similarity of method m and the seed methods. We use the similarity values to filter out methods that do not contribute to the same functionality as criteria methods.

Table 1 shows the similarity values of the example code. Criteria methods `main` and `getFileSize` have maximum similarity 1.0. The similarity value of `FileList.add` is high since `main` calls `FileList.add`, and `FileList.add` calls `ArrayList.add`, respectively.

Table 1: Similarity for example slicing

Simlilarity	Method
1.000	sample.Main.main
1.000	sample.FileList.getFileSize
0.540	sample.FileList.add
0.519	sample.FileList.getTotalFileSize
0.482	sample.FileList.printFileSize
0.468	sample.FileList.init
0.404	sample.FileList.getFileName

3.1.2 Distance Metric

The similarity metric represents semantic relation. In comparison, the distance metric represents a structural relation. This metric evaluates locality because users are typically more interested in facts that are near the current point of interest than those that are further away [14]. Another motivation is that a developer typically prefers understanding how code is connected to the current focus [1].

For example, two methods `printFileSize` and `getFileSize` of `FileList` have almost the same similarity values. However, `printFileSize` is more important for interaction between criteria methods `main` and `getFileSize` since `printFileSize` connects the criteria methods. The distance metric helps include methods that contribute to connecting slicing criteria methods to one another.

We define the distance metric based on a method-level PDG, whose vertices and edges represent methods and inter-method edges in PDG, respectively. A method-level PDG is constructed from PDG using a simple set of rules:

- Create method vertices for each method m in the original PDG.
- Add a *call* edge $m1$ to $m2$ if a call edge from a vertex in $m1$ to $m2$ is in the original PDG.
- Add a *return* edge $m2$ to $m1$ if a parameter-out edge from a vertex in $m2$ to $m1$ is in the original PDG.
- Add a *field* data-flow edge $m1$ to $m2$ if a field data-flow edge from a vertex in $m2$ to $m1$ is in the original PDG.

The forward distance $fd(src, dest)$ between two methods src and $dest$ is the length of the shortest realizable path by forward traversal in the method-level PDG from src to $dest$. Using the $fd(src, dest)$, we define the distance metric $Distance(m, C)$ for method m and slicing criteria methods C as follows.

$$Distance(m, C) \quad = \quad \min_{(src,dest)\in path(C)} d(m, src, dest) \qquad (5)$$

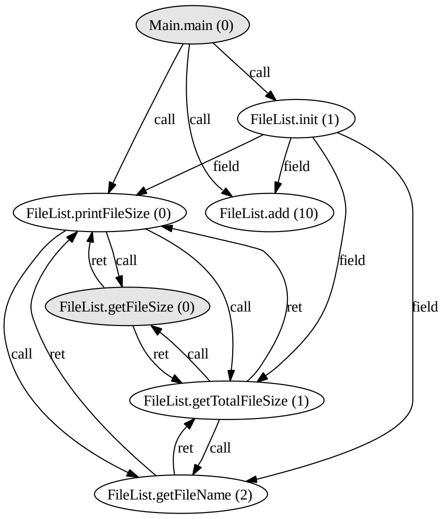


Figure 2: A method-level PDG with the distance metric (gray nodes are the seed methods)

$$d(m, src, dest) \quad = \quad fd(src, m) + fd(m, dest) - fd(src, dest) \tag{6}$$

In equation (4), $path(C)$ enumerates all reachable paths from src to $dest$ among C ($src \neq dest$). If all methods in C are reachable one another and $|C| = n$, $path(C)$ enumerates $n(n-1)$ pairs. In equation (5), $d(m, src, dest)$ is zero if the method m is on the shortest path from src to $dest$. A method on a shorter path has a lower distance value. If method m is not involved in paths for all pair $(src, dest) \in path(C)$, m does not contribute to interaction among methods C; we assign $Distance(m, C) = \infty$ for the method m.

Figure 2 shows the method-level PDG of the example code excluding library methods. The seed methods are gray nodes. An integer annotating a method name indicates the distance metric. The distance value of `printFileSize` method is zero since the method is on the shortest path from `main` to `getFileSize`. `FileList.add` has a high distance value 10 since the `add` is on a longer data-flow path from `main` to `getFileSize` through `ArrayList`.

3.2 Slicing with Barriers

We calculate a program slice as the union of both forward and backward slice from given criteria. Slicing criteria is all vertices in the seed methods. We use two-phase slicing with summary edges [9, 21]. The extension of program slicing to handle barriers is straightforward; the algorithm simply skips barrier edges in the traversal process. For summary edge calculations, an algorithm to identify

Table 2: Rules translating a program slice to a concern graph

Edge type	An edge is generated if:
(calls *m1* *m2*)	The slice includes a call edge from method *m1* to method *m2*.
(reads *m* *f*)	The slice includes a vertex of method *m* that has a data-flow from field *f*.
(writes *m* *f*)	The slice includes a vertex of method *m* that has a data-flow to field *f*.
(declares *c* *f*|*m*)	The slice includes field *f* or method *m* of class *c*.
(creates *m* *c*)	The slice includes a vertex of method *m* that creates an instance of class *c*.
(checks *m* emphc)	The slice includes type checking (`instanceof` operation) or type casting operation.
(superclass *c1* *c2*)	The concern graph includes both class *c1* and class *c2*.

summary edges blocked by barriers is used that was defined previously [15]. As the original algorithm handled only vertex barriers, we extended and applied the algorithm for edge barriers.

3.3 Translating a Slice into a Concern Graph

To ensure the information produced in our approach can be understood easily by developers, we translate the subgraph of the PDG produced by our approach into a concern graph [22]. This translation involves two steps: filtering library vertices and applying translation rules.

First, we remove vertices in the sub-graph of the PDG that correspond to libraries. We perform this filtering to ensure the concern graph focuses on the application-specific entities. The filter we applied removes JDK standard packages including "java.", "javax.", "com.sun." and so on. If a developer would like to analyze in a functional concern, the developer can customize the list.

Next, we apply the set of rules translating a program slice into a concern graph. We used the rules proposed by Kameda et al [13], shown in Table 2.

Figure 3 is an example of the generated concern graph from the example code using $threshold_{sim} = 0.45, threshold_{dist} = 2$ and the seed methods stated earlier. This concern graph includes only methods to calculate the size of files, which is the concern of interest, and excludes other concerns; for example not

```
Main.main(java.lang.String[]): void
  creates FileList
  calls FileList.printFileSize(): void
  calls FileList.<init>(): void

FileList.<init>(): void
  writes files: java.util.ArrayList

FileList.printFileSize(): void
  calls getFileSize(java.lang.String): long
  calls getTotalFileSize(): long
  reads files: java.util.ArrayList

FileList.getTotalFileSize(): long
  calls getFileSize(java.lang.String): long
  reads files: java.util.ArrayList

FileList.getFileSize(java.lang.String): long
```

Figure 3: A generated concern graph

including code to list up target files using `FileList.add` and `FileList.getMethodName`.

4 Implementation

We have implemented our method to support functional concern location from Java programs. Our implementation uses the Soot framework [28].

4.1 PDG

We use the System Dependence Graph for Java defined by Zhao [32]. A vertex in the Java SDG corresponds to Java byte-code (a Jimple code from Soot specifically). Within a sub-graph of the SDG representing a method implementation, the edges in a method represent control dependence relations and data dependence relations. Inter-method edges include method call, parameter-in, parameter-out and field data-flow edges. Method call and parameter edges for a method call are connected to methods that might be invoked. Field data-flow edges are connected for all possible data-flow via field.

To construct a Java SDG, we obtain control-flow, data-flow, a call graph and points-to set information from the Soot framework. We have chosen context-insensitive points-to set provided by Spark tool in the framework. We extended the framework with control-dependence analysis based on iterative dominator analysis algorithm [4]. We also added a source-to-vertices map generator that analyze line number attribute in Java bytecode and creates a map from a pair of class name and line number to SDG vertices.

4.2 Input Format

Our tool accepts a list of classes, methods and lines of code. A developer can list software entities of interest using various tools including a natural language based search [27], a component search [17], or a keyword search tool such as `grep`. A class name is mapped into all methods in the class, and a line is mapped into a method contains the line, respectively.

4.3 Output Format

A resultant program slice is saved in three formats: a raw slice, a concern graph in a textual form and in a graphical form. The graphical form can be visualized by Graphviz [11]. In the graphical form, we adopted a class diagram style that replaces "declares" edges with a class node containing its methods and fields and omits intra-class edges to output a smaller graph.

4.4 Thresholds for Heuristics

Our heuristic function has two parameters, $threshold_{sim}$ and $threshold_{dist}$. Currently, a developer must specify these thresholds and can easily vary them to get a concern graph of reasonable size, possibly starting with tight thresholds that produce a small concern graph and relaxing them to include more detail as needed. Eventually, we may be able to suggest or preset the thresholds once we have gained more experience with the approach.

5 Evaluation

To be helpful for developers, our technique needs to generate concern graphs that describe a functional concern adequately without overwhelming the developer in detail. As a first step in evaluating whether the concern graphs produced are helpful to a developer, we conducted an evaluation in which we compare concern graphs generated by the SCOLOC tool with concern graphs made for the concerns by hand by different developers.

5.1 Evaluation Method

Our evaluation involved choosing three functional concerns for each of two software systems. We had two developers create concern graphs manually for the six chosen concerns. We then compared handmade graphs with concern graphs generated by our method.

We used two software system, jEdit and our SCOLOC tool. jEdit version is 4.2 final. jEdit comprises 890 classes and 140,665 lines of code including comments. For jEdit, we chose two functional concerns, `Autosave` and `Marker`, investigated by Robillard [23, 25] as well as a `History` concern, which is a part of bug recorded in jEdit bug repository. Our SCOLOC tool comprises 234 classes and 19,895 lines of code including comments. We selected three different kinds

Table 3: The target concerns

Concern	Task Description
jEdit 4.2:	
Autosave	Autosave option saves a file being edited in the background. This concern is a change request to backup all autosaved versions in a directory.
Marker	A user can put markers in a file. This concern is a request to allow a user to add a short description for a marker, and show the description in the `Marker` menu.
History	jEdit has a problem of IndexOutOfBoundsException caused when a property "history" size is zero. This concern is a change request to fix the bug.
SCOLOC:	
Batch	The SCOLOC tool creates a new algorithm object (using Factory and Strategy pattern [8]) for each slicing. This concern is a change request to reuse the objects if the same algorithm is specified.
Input	The SCOLOC tool accepts a pair of class name and line number. This concern is a change request to modify the input parser to accept a class name, a method name, and a wild card selecting classes and methods.
MethodId	The SCOLOC has a problem of saving a list of method id into a file: the same file is saved twice during one execution. This concern is a change request to fix a problem of MethodIdList file.

Table 4: The size of handmade concern graphs

Concern	Participant 1			Participant 2		
	C	F	M	C	F	M
Autosave	7	8	18	12	19	48
Marker	8	11	30	12	8	33
History	3	3	6	12	9	21
Batch	20	58	32	7	11	14
Input	9	5	11	4	1	7
MethodID	5	8	13	7	1	9

of functional concerns from our to-do list for the tool: improving performance, adding a particular piece of new functionality and fixing a bug. Table 3 lists the six concerns. Each functional concern is defined as a description of an evolution task since this is how we envision developers defining functional concerns and using our approach.

We had two participants create concern graphs as documents for a new developer who does not know the target software to help maintenance tasks. Participant 1 is a doctoral student. He read jEdit code for his research and he also knows program slicing implementations. Participant 2 is a master student. She also read jEdit code for her research but does not know a program slicing system. Both participants used IDE to investigate the target software.

We compared the handmade concern graphs with concern graphs generated by our tool. For tasks `Autosave` through `MethodId`, we selected seeds for each concern using Java search in Eclipse [5], a keyword search system. We searched concern names "autosave", "marker", "methodid" for corresponding functional concerns. We also searched keyword "create" for `Batch` concern since the keyword is used to indicate our Factory pattern. For `Input` concern, we use keyword "criterion". `History` seeds is different; in this case, we used a stacktrace data (a list of line number) as concern seeds since a stacktrace data is a starting point for developer to locate a functional concern related to an error. The criteria selection task took 10-15 minutes for each concern. This task was completed before participants created concern graphs. We compared the results of applying SCOLOC for various threshold values: We varied $threshold_{sim}$ from 0 to 1 by 0.01. We varied $threshold_{dist}$ from 0 to 20 by 5.

5.2 Size of Concern Graphs

Table 4 shows the size of each concern graph produced by the participants. Column `C`, `M` and `F` respectively indicates the number of classes, methods and fields in the concern graph. The minimum graph has 12 vertices, the maximum one has 110 vertices. The average is 40.8.

Figure 4 plots the number of vertices in generated concern graphs for various similarity thresholds. The distance threshold is fixed to zero in this graph. This

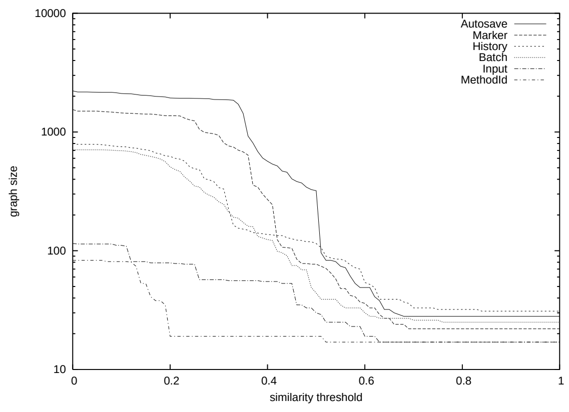


Figure 4: The size of generated concern graphs

figure shows that high similarity threshold values result in concern graphs whose vertices are less than 100. We believe these graphs are likely small enough for developers to understand.

5.3 Recall and Precision

We use recall, precision and F-value for each pair of a generated graph and a handmade graph to assess the content of the generated concer graphs. These values are calculated by the following expressions:

$$\begin{aligned} Recall &= \frac{|GeneratedGraph \cap HandmadeGraph|}{|HandmadeGraph|} \\ Precision &= \frac{|GeneratedGraph \cap HandmadeGraph|}{|GeneratedGraph|} \\ F-value &= \frac{2 \cdot Recall \cdot Precision}{Recall + Precision} \end{aligned}$$

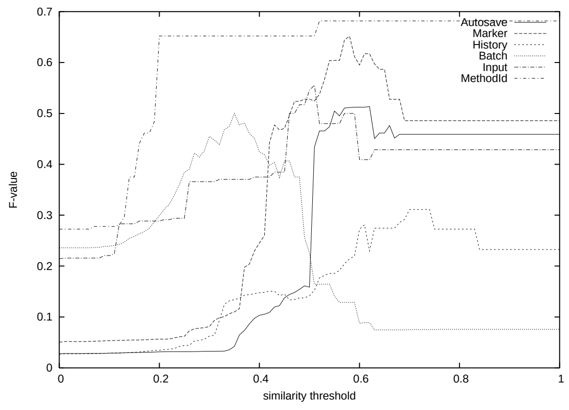
An F-value represents the trade-off between recall and precision. Table 5 shows the best pairs of recall and precision according to F-value. Table 5 shows the threshold values in the columns *sim* and *dist* to indicate $threshold_{sim}$ and $threshold_{dist}$ respectively. If there are two or more highest (tied) pairs, we chose the maximum similarity threshold and the minimum distance threshold since the pair of thresholds produces the smallest graph amongst the tied pairs.

Figure 5 shows the F-values of the generated graphs for each participant; For all of these figures we set $threshold_{dist} = 0$ as it produced good F-values as shown in Table 5. We found two trends in F-value distribution:

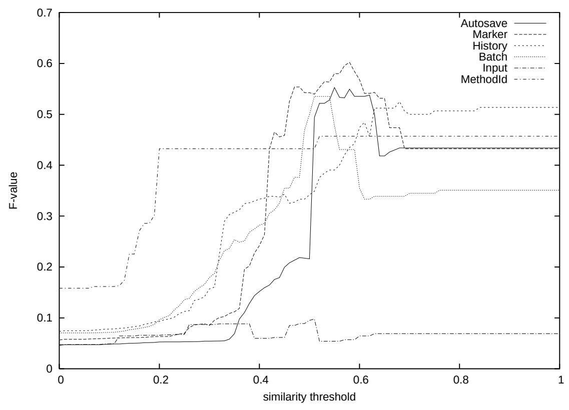
1. In most cases, the best F-value appears when $threshold_{sim}$ ranges 0.5-0.6. Higher threshold values also result in good F-values.

Table 5: The best recall and precision

System	Concern	Participant 1				Participant 2			
		sim	dist	recall	precision	sim	dist	recall	precision
jEdit	Autosave	0.62	0	0.5758	0.4634	0.58	0	0.4615	0.6792
	Marker	0.58	0	0.6042	0.7073	0.58	0	0.5385	0.6829
	History	0.74	0	0.5833	0.2121	1.00	0	0.4419	0.6129
SCOLOC	Batch	0.35	0	0.6542	0.4046	0.54	0	0.5938	0.4872
	Input	0.51	0	0.6000	0.5172	0.38	0	0.2500	0.0536
	MethodId	0.77	10	0.6667	0.9000	1.00	0	0.4444	0.4706



(a) F-value for Participant 1



(b) F-value for Participant 2

Figure 5: F-value for each participant

2. Similarity threshold values less than 0.2 result in low F-values because a concern graph generated with a lower similarity value is generally much larger than a handmade concern graph.

We also found two exceptional cases.

1. For the `MethodId` concern, $threshold_{sim} = 1.0$ results in the highest F-value if $threshold_{dist} = 0$. We think this is because the seed methods for the concern are well localized in four classes; most of methods are filtered out by the distance metric.

2. Even when we generate concern graphs with SCOLOC with a high similarity threshold for the Batch and Input concerns, we see low F-values compared to participant 1 and participant 2 respectively. In the case of participant 1, his `Batch` concern graph is too large to compare with generated concern graphs; the handmade concern graph has 107 vertices but the SCOLOC-generated concern graph has only 17 vertices. In the case of participant 2, her `Input` concern graph included only methods that prepare parameters for the input functionality but excluded the methods of the input functionality. Concern graphs generated by the SCOLOC included few methods selected by participant 2.

Overall, since our approach extracts smaller graphs for higher similarity threshold values, a developer can start to read graphs from a smallest one and proceed to larger graphs with lower similarity values to get additional detail.

5.4 Difference between Concern Graphs

To get a better sense of where SCOLOC might be including too much or not enough in a concern graph, we calculated differences between SCOLOC-generated graphs and handmade graphs and analyzed the differences with participants. We found several reasons for the differences.

There were six kinds of situations in which SCOLOC included some entities the participants excluded.

1. Participants excluded fields used only in accessor methods. For example, `flags` field is only accessed through `getFlags` and `setFlags` methods in jEdit. A participant included these methods in a concern graph, but excluded the field.

2. Participants excluded fields initialized by a constant value. A participant included a constructor, but omitted the fields.

3. Participants omitted several polymorphic methods. Typically, they included only a superclass method.

4. Participants missed implicit constructor calls for a superclass. It is hard to recognize a superclass without explicit `super` constructor call.

5. Participants forgot to include several methods after they read the methods. This is a simple human error.

6. Generated graphs included methods that have similar name but implement a different functionality. These methods are false positive of our approach. An example is a pair of `autosave()` and `save()` methods in `BufferIORequest` class. These methods have high similarity each other but only `autosave()` method is related to the concern.

These differences do not prevent understanding a functional concern since most of entities are related to concern code.

There were three situations in which a participant included entities that were excluded by SCOLOC.

1. In several cases, SCOLOC left out control-flow (method call) relations because the distance metric extracted other shorter flow paths amongst the entities. One example is an option dialog in jEdit. A dialog calls a `properyChanged` method to update other classes using options after saving the options. Data-flow paths from the dialog to classes loading options via a manager class are shorter than notification method call path from the dialog to option users. Therefore, our tool extracted only data-flow. Participants included both paths in concern graphs.

2. Participants included fields written by a method but never read in a concern. Our approach excluded these fields since program slicing uses data-flow information. We found two cases. One case involved fields that store the output data of a concern. The other case contains fields accidentally added to a concern graph by a participant. The former case should be included in concern graphs, but the latter case should be excluded. Distinguishing these two cases automatically is a future work.

3. Participants included several polymorphic methods. In this case, participants used a type hierarchy analysis tool and added all of them into their concern graph. The SCOLOC tool excluded methods of subclasses that are unreachable from a call site.

Many of these cases involve a participant accidentally including an entity of less importance that happens to be collocated with more important entities. SCOLOC found the latter but not the former; however not including the accidental entity affects precision and recall values.

There were several situations in which the technology did not allow a participant to capture desired information in a concern graph.

- Our tool does not support interface handling. Participants wanted an *implements* edge connecting a class to a event listener interface since event listeners have important roles in a GUI application such as jEdit. Our tool also does not support a call edge to an abstract method since PDG includes only concrete methods.

- Our tool does not support constant fields that are removed from bytecode during compilation. Since many flags are manipulated by `setFlag(int)` method with constant values in jEdit, participants wanted to include constant values in concern graphs.

- A concern graph cannot represent instance-aware method calls, such as `Class.fieldName.getX()`. This is represented as a pair of a field access and a method call.

- A concern graph does not support specialized parameter/return type, such as `String List.get(int)` instead of `Object List.get(int)`. This kind of return type checking is represented as a pair of a method call edge and a type check edge.

The results of the evaluaiton gave us ideas for improving our approach. One direction for the future work is extending the concern graph representation with additional information, such as an instance-awareness and data-flow amongst edges in a concern graph.

6 Conclusion

To complete a maintenance task thoroughly, a software developer needs to understand how the software entities related to the functional concerns of interest collaborate. To aid a developer in locating and understanding a functional concern, we have proposed, implemented and evaluated an approach based on program slicing.

We introduced the similarity and distance metrics into program slicing to stop graph traversal during program slicing to extract a small set of entities closely related to the entities of interest to a developer.

We compared concern graphs generated by the SCOLOC tool with concern graphs made by hand. The result shows the SCOLOC effectively reduces the size of concern graphs and keeps interesting entities in the concern graphs.

In the future work, we will have developers use the SCOLOC tool for maintenance tasks. We are planning to generate a more expressive concern graph by detecting patterns in the concern graph. We also interested in application of aspect mining techniques to exclude crosscutting concerns from a functional concern graph.

Acknowledgement

This work was supported by MEXT.Grant-in-Aid for JSPS Fellows (No.17-9539).

References

[1] Antoniol, G., Fiutem, R., Lutteri,, G., Tonella, P., Zanfei, S. and Merlo, E.: Program Understanding and Maintenance with the CANTO Environment. In Proc. of ICSM, pp.72-81, 1997.

[2] Bruntink, M., van Deursen, A., van Engelen, R. and Tourwe, T.: On the Use of Clone Detection for Identifying Crosscutting Concern Code. IEEE TSE, Vol.31, No.10, pp.804-818, 2005.

[3] Chen, K. and Rajlich, V.: Case Study of Feature Location Using Dependence Graph. In Proc. of IWPC, pp.241-247, 2000.

[4] Cooper, K. D., Harvey, T. J. and Kennedy, K.: A Simple, Fast Dominance Algorithm. `http://www.cs.rice.edu/~keith/EMBED/dom.pdf`

[5] Eclipse Project. `http://eclipse.org/`

[6] Eisenbarth, T., Koschke, R. and Simon, D.: Locating Features in Source Code. IEEE Computer, Vol.29, No.3, pp.210-224, 2003.

[7] Fjeldstad, R. K. and Hamlen, W. T.: Application Program Maintenance Study: Report to Our Respondents. In Proc. of GUIDE 48, 1983.

[8] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: Design Patterns. Addison-Wesley Pub Co., 1995.

[9] Horwitz, S., Reps, T. and Binkley, D.: Interprocedural Slicing Using Dependence Graphs. ACM TOPLAS, Vol.12, No.1, pp.26-60, 1990.

[10] Jackson, D. and Rollins, E. J.: A New Model of Program Dependences for Reverse Engineering. In Proc. of FSE, pp.2-10, 1994.

[11] Graphviz Project. `http://www.graphviz.org/`

[12] Ishio, T., Niitani, R., Inoue, K.: Towards Locating a Functional Concern Based on a Program Slicing Technique. Selected for Presentation in AOAsia2, `http://sel.ics.es.osaka-u.ac.jp/~lab-db/betuzuri/contents.en/602.html`, 2006.

[13] Kameda, D. and Takimoto, M.: Building Concern Graph Based on Program Slicing. IPSJ Transactions in Programming Language, Vol.46, No.SIG11, pp.45-56, 2005. In Japanese.

[14] Krinke, J.: Visualization of Program Dependence and Slices. In Proc. of ICSM, pp.168-177, 2004.

[15] Krinke, J.: Slicing, Chopping, and Path Conditions with Barriers. Software Quality Journal, Vol.12, No.4, pp.339-360, 2004.

[16] Krinke, J.: Mining Control Flow Graphs for Crosscutting Concerns. In Proc. of WCRE, pp.334-342, 2006.

[17] Inoue, K., Yokomori, R., Yamamoto, T., Matsushita, M. and Kusumoto, S.: Ranking Significance of Software Components Based on Use Relations. IEEE TSE, Vol.31, No.3, pp.213-225, March 2005.

[18] LaToza, T. D., Venolia, G. and DeLine, R.: Maintaining Mental Models: A Study of Developer Work Habits. In Proc. of ICSE, pp.492-501, 2006.

[19] Marin, M., van Deursen, A. and Moonen, L.: Identifying Aspects using Fan-In Analysis. In Proc. of WCRE, pp.132-141, 2004.

[20] Murphy, G. C., Kersten, M., Robillard, M. P. and Čubranić D. C.: The Emergent Structure of Development Tools. In Proc. of ECOOP, pp.33-48, 2005.

[21] Reps, T., Horwitz, S., Sagiv, M. and Rosay, G.: Speeding Up Slicing. In Proc. of FSE, pp.11-20, 1994.

[22] Robillard, M. P. and Murphy, G. C.: Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In Proc. of ICSE, pp.406-416, 2002.

[23] Robillard, M. P., Murphy, G. C.: Automatically Inferring Concern Code From Program Investigation Activities. In Proc. of ASE, pp.225-234, 2003.

[24] Robillard, M. P.: Automatic Generation of Suggestions for Program Investigation. In Proc. of FSE, pp.11-20, 2005.

[25] Robillard, M. P.: Tracking Concerns in Evolving Source Code: An Empirical Study. In Proc. of ICSM, pp.479-482, 2006.

[26] Shepherd, D., Palm, J., Pollock, L. and Chu-Carroll, M.: Timna: a Framework for Automatically Combining Aspect Mining Analyses. In Proc. of ASE, pp.184-193, 2005.

[27] Shepherd, D., Pollock, L. and Vijay-Shanker, K.: Towards Supporting On-Demand Virtual Remodularization Using Program Graphs. In Proc. of AOSD, pp.3-14, 2006.

[28] Soot: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`

[29] Walkinshaw, N., Roper, M. and Wood, M.: Understanding Object-Oriented Source Code from the Behavioural Perspective. In Proc. of IWPC, pp.215-224, 2005.

[30] Weiser, M.: Program Slicing. IEEE TSE, Vol.10, No.4, pp.352-357, 1984.

[31] Wilde, N., Gomez, J. A., Gust, T. and Strasburg, D.: Locating User Functionality in Old Code. In Proc. of ICSM, pp.200-205, 1992.

[32] Zhao, J.: Applying Program Dependence Analysis to Java Software. In Proc. of Workshop on Software Engineering and Database Systems, pp.162-169, 1998.

[33] Zhao, W., Zhang, L., Liu, Y., Sun, J. and Yang, F.: SNIAFL: Towards a Static Noninteractive Approach to Feature Location. ACM TOSEM, Vol.15, No.2, pp.195-226, 2006.