Towards Locating a Functional Concern Based on a Program Slicing Technique

Takashi Ishio[1,2], Ryusuke Niitani[2], Katsuro Inoue[2]

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

[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

Abstract

A functional concern is often implemented by collaborative software modules. When developers modify or reuse the implementation of a concern, they need to find the module units contributing to the concern and understand how the units collaborate with one another. Although program slicing is an automatic method to extract relationship among program elements in modules, slicing often results in many program elements to understand. This position paper proposes a variant of program slicing that uses heuristics to stop visiting vertices in a program dependence graph. A preliminary case study shows that our method extracts a small subgraph of the usual program slice that seems most applicable to understand the feature. This paper also discusses other possible heuristics towards the automatic extraction of a concern.

1. Introduction

Software developers often must spread the implementation of a functional concern across several related modules. When maintaining that concern, a developer then must handle the multiple modules that contribute to the concern [10]. Some estimate that understanding concerns in source code accounts for more than half of the total cost of maintenance [3].

When a developer starts to modify or reuse a concern, developers first need to find the software entities, which includes methods and fields, related to the concern. Feature location [14, 19] and component repository [9] have been proposed as ways to find software entities related to a functional concern. These methods provide a list of software entities related to the concern of interest to a developer.

After they have found the software entities, developers have to understand how the entities collaborate to form the concern. One perspective of collaboration is control-flow and data-flow among the software entities. One way to extract appropriate software entities (in this case statements) based on control-flow and data-flow is to use program slicing [16]. However, for a large system, program slicing often results in a large number of statements since program slicing extracts all statements which may have an affect on the variables of interest [7]. This is because a functional concern is connected to other related functional and non-functional concerns. Long dependence paths through an entire software system may be needed to help complete a debugging task, but it is not suitable for a developer who needs to understand the relationships among specific software entities. For example, a developer may want to know *"Which is a parameter for business logic"*. If a slice is used to try to answer this question, software entities corresponding to how to calculate the parameter in the user-interface level will also be returned, which is not important for the developer analyzing the business logic.

In this paper, we propose the use of heuristics to help guide program slicing to return the control-flow and data-flow relationship among software elements related to a particular concern. Our heuristic rules identify vertices that terminate the traversal of the program dependence graph used to extract a program slice.

Our method takes as input a list of methods and fields of interest to a developer. The developer can identify such methods and fields by keyword search or feature location techniques [9, 14, 19]. Our method produces a subgraph of a program dependence graph that includes the vertices representing the input, other related vertices and edges representing control-flow and data-flow among the input vertices. The resultant subgraph is a union of backward and forward heuristic program slices for each vertex that corresponds to an entity in the input list.

We report on a preliminary case study we conducted to evaluate our approach. The study shows that heuristics are important to reduce the size of program slice.

We begin by describing the background of the research, which focuses on program slicing. In Section 3 we describe our methods in detail. Section 4 represents the result of preliminary case study. Section 5 discusses about our heuristics. Related works are presented in Section 6. Finally, we conclude in Section 7.

2. Background

Program slicing is a technique to extract statements that are relevant to a particular variable [16]. 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).

In the process of program slicing, a program is converted to a program dependence graph whose vertices represent statements and edges represent control and data relations [4]. A program slice is extracted by backward/forward traversal of the graph from slice criteria. A backward slice contains statements that affect on variables, and a forward slice contains statements that depend on the variables, respectively.

Chopping, which is a variant of program slicing, is proposed to support the understanding of how a statement affects to another statement [5]. Chopping extracts control-flow and data-flow paths from input variables to output variables in a program dependence graph. This approach requires developers to distinguish statements into input and output. However, such a labeling task is hard for a developer who has just found variables and methods without enough knowledge about a concern. Therefore, it is not suitable to inspect the structure of a concern.

On the other hand, a program slicing based approach is proposed to automatically extract a concern graph [6]. Concern graph is a graph whose vertices represent software entities such as methods and fields, and edges represent the relationship among elements such as method call and a field access, respectively [11]. This approach converts a program slice to a concern graph since a program slice is a functional unit to calculate a value of a variable. However, this approach may output a large concern graph since a program slice always includes shared concerns such as `main(String[])` method in Java and crosscutting concerns such as logging.

Krinke proposed length-limited slicing based on the distance from a criterion [7]. This approach stops 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* [8]. A barrier is a vertex that terminates the graph traversal. In [8], only chopping criteria (source and target vertices) are used as barriers. Our approach aims to automatically find such barriers in order to extract a small set of statements for specific software entities.

3. Extracting a Functional Concern with Slicing

Our method automatically extracts relationship among software entities based on methods and fields that are specified by a developer. This method supports developers to understand how software entities are contributing to a concern when they find entities that might be related to a concern.

Our method is a variant of program slicing that extracts a subgraph of a program dependence graph which includes vertices representing the entities specified by developers and edges connecting the vertices. The extracted slice shows control-flow or data-flow paths among entities to developers.

3.1. Slicing a Concern

Program dependence graph [4] is a graph whose vertices represent program statements and edges represent control and data dependence relations. In this paper, we use a program dependence graph for Java [18]. Polymorphic method calls are resolved based on points-to set analysis. One method call vertex has edges connected to one or more methods which might be executed.

We define our method based on the graph terminology.

Input: A pair of (G, V_d) is the input for the method. Graph G is a dependence graph for a program. V_d is a set of vertices in G corresponding to software entities specified by a developer. In order to allow developers to specify software entities without knowledge about graph structure, we translate a field to all vertices which refer to the field and a method to the entry point vertex of the method.

Output: A sliced graph S which is a subgraph of G includes at least vertices V_d and edges for connecting the vertices. Graph S is usually connected but may not be if a vertex v is not reachable from other vertices in V_d.

Extracting relationship among specified software entities is finding a connected subgraph that includes vertices V_d. One conservative solution is the union of backward and forward program slices for each vertex in V_d. The union slice becomes a connected graph including all vertices in V_d. However, the output will be large amount of statements.

A smaller subgraph is more suitable for developers to inspect. A program slice is calculated by traversing the dependence graph from a vertex. Therefore, we introduce heuristics to stop the traversal at vertices that satisfy certain patterns in order to find a small set of collaboration code fragments.

3.2. Heuristics for Extracting Slices

There are many possible subgraphs of a dependence graph that include vertices V_d and relationship among the vertices. We define additional conditions for understanding how particular software entities collaborate with one another.

1. The resultant graph should include information about who (an object) or where (a method call statement) starts and finishes the collaboration.
2. The resultant graph should include *reinforced* methods and fields that contribute to the same concern to which specified entities contribute. An entity *x* is reinforced by a set of interest *I* if most elements related to *x* are in *I* [13].

In this paper we discuss only the first condition. We leave the development of heuristics for the second condition to future work.

To find a subgraph that satisfies the first assumption, we search dominator and post-dominator for V_d specified by a developer. Dominator is the nearest common ancestor for V_d in a control-flow graph. The dominator is the nearest vertex reachable from all vertices in V_d via control-flow edges and method-call edges by backward traversal. For example, we show a code fragment processing files in Figure 1. In the code fragment, `execute` method (lines 01-03) sets a parameter using `setFolderList` method (lines 04-05) and calls `run` method (lines 06-18). Figure 2 represents control-flow and method calls for the code fragment. The dominator for line 05 (writing `folders` field) and line 07 (creating `FileList` object) is line 02. Similarly, post-dominator is the nearest common successor, that is the nearest vertex reachable from all vertices in V_d via control-flow edges and method return edges by forward traversal. For example, the post-dominator for line 05 and line 07 is line 03. To find nearest common ancestor, a simple iterative algorithm is defined in a part of a dominance tree analysis algorithm [1].

The dominator and the post dominator in control-flow represent the start and the end of control-paths related to vertices V_d. Therefore, we apply usual program slice in the paths from the dominator to the post-dominator via V_d. Heuristic rules block traversal process at a vertex out of the paths.

The modified program slicing process is as follows.

```
    class  Console {
01:  public void execute() {
          :
02:    checker.setFolderList(folders);
03:    checker.run();
     } }
    class Checker {
04:  public void setFolderList(List folders) {
05:    this.folders = folders;
     }
06:  public void run() {
07:    FileList filelist = new FileList();
       :
08:    for (Iterator it = folders.iterator(); it.hasNext(); ) {
09:      String folder = (String)it.next();
10:      filelist.parseFile(new File(folder), is_recursive);
      }
       :
11:    int count = 0;
12:    for (Iterator it = filelist.iterator(); it.hasNext(); ) {
13:      Collection files = (Collection)it.next();
14:      if (files.size() > 1) {
15:        makeDuplicationGroups(files);
         }
16:      count += files.size();
17:      notifyProgressValue(count);
       }
       :
18:    notifyFinished();
     }
19:  private void makeDuplicationGroups(Collection l) {
20:    for (Iterator it = l.iterator(); it.hasNext(); ) {
21:      File f = (File)it.next();
22:      String signature = FileMD5.makeMD5(f);
23:      if (signature.startsWith(Settings.SIGNATURE_ERROR)) {
24:        notifyErrorFile(signature);
         } else {  /* process a file */ }  }  }
```

Figure 1. An example code fragment

1. We get the dominator and the post-dominator for vertices V_d in control-flow. We refer to the dominator as $dom(V_d)$, and the post-dominator as $pdom(V_d)$ in the following steps.
2. The vertices in V_d, $dom(V_d)$ and $pdom(V_d)$ bounds a region based on control-flow. All vertices in backward paths from the vertices in V_d to $dom(V_d)$ and in forward paths from the vertices in V_d to $pdom(V_d)$ are marked as "primary" vertices.
3. We apply the usual program slice algorithm to the primary vertices. A backward/forward search from each vertex in V_d is blocked at a non-"primary" vertex if:
   - the vertex is a field vertex, or
   - the vertex is a method entry point which has no arguments, or
   - the vertex is a method entry point for an abstract method, or

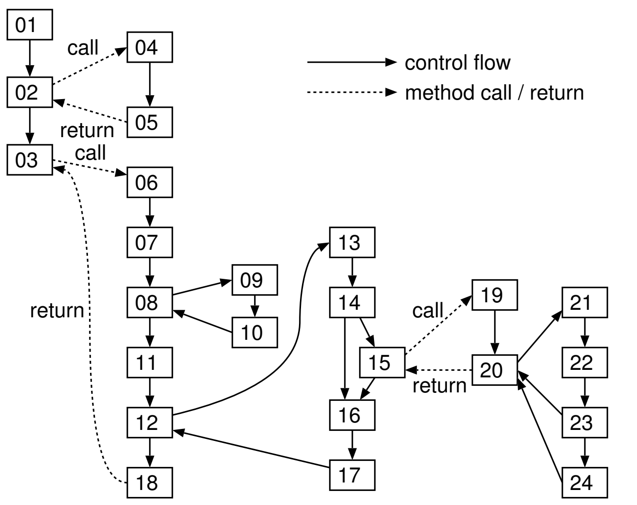


Figure 2. Control flow graph of the example code

- the vertex is in library classes (e.g. `java` and `javax` packages).

We set the condition for a method entry point without arguments since the behavior of such a method is independent of the caller method but depends on only the fields of the invoked object. The fields and the methods that have no arguments represent the boundary of the intermethod data-flow for the vertices in V_d. For example, line 05 writes field `folders` and line 08 reads the field. This intermethod data-dependence edge is not traversed unless both lines are primary vertices.

Our method might stop at vertices before the result includes enough information for developers. A developer can add a new criterion vertex to find further vertices.

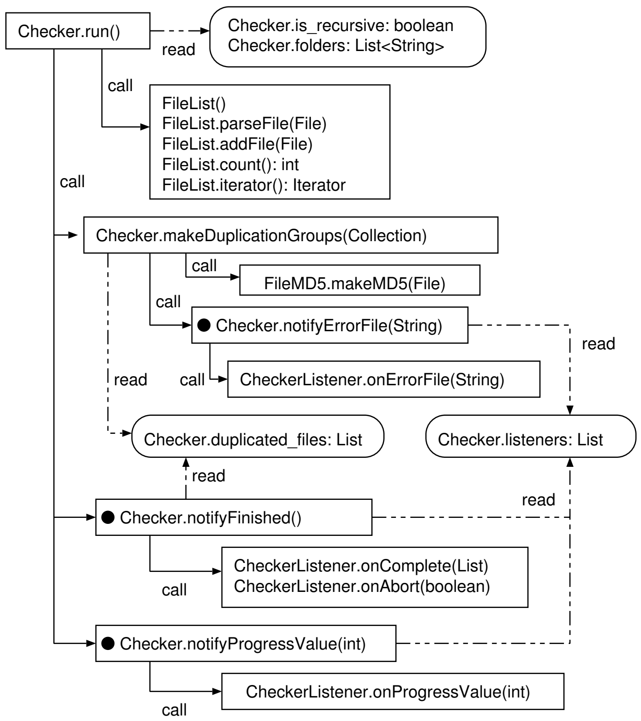
4. Preliminary Case Study

In order to estimate the effectiveness of our approach, we have conducted a preliminary case study. A target program is a Duplicated File Checker, a program listing files which has the same MD5 hash code in the selected directories. The Checker program consists of 14 Java source files. The total size is 1331 lines of source code including comments. We have built a dependence graph for a part of the system based on Java byte-code in partially automated way. To get points-to set information, we have used context-sensitive pointer analysis provided by `bddbddb` project [17].

This case study compared a conservative solution with our heuristic approach. We have calculated a program slice in both approaches, mapped the result on source code and translated the result into concern graphs.

	#lines	#methods	#files
Heuristic slice	43	15	4
Non-heuristic slice	198	46	8
Program size	1331	110	14

Table 1. The size of slices



Checker.run()
read
Checker.is_recursive: boolean
Checker.folders: List<String>
call
FileList()
FileList.parseFile(File)
FileList.addFile(File)
FileList.count(): int
FileList.iterator(): Iterator
call
Checker.makeDuplicationGroups(Collection)
call
FileMD5.makeMD5(File)
call
● Checker.notifyErrorFile(String)
read
read
call
CheckerListener.onErrorFile(String)
Checker.duplicated_files: List
Checker.listeners: List
read
read
● Checker.notifyFinished()
call
CheckerListener.onComplete(List)
CheckerListener.onAbort(boolean)
● Checker.notifyProgressValue(int)
call
CheckerListener.onProgressValue(int)

Figure 3. A concern graph for "notify" methods

In this case study, we want to extract the concern for notification. We identify particular methods involved in notification, namely `notifyProgressValue`, `notifyErrorFile` and `notifyFinished`. Applying our method to this program with the identified methods results in a slice with about 100 vertices. The slice is mapped on 43 lines in 4 files. Table 1 shows the size of our slice and conservative program slice. Our resultant slice is a subset of the conservative slice.

We present the slice as a concern graph in Figure 3. A vertex in the graph represent a method or a field. The translation rules we have used (defined in [6]) are following:

- A `call` edge from *m1* to *m2* is generated if the slice includes a call edge from method *m1* to method *m2*.
- A `read` edge from *m* to *f* is generated if the slice in-

cludes a vertex of method *m* which has a data-flow from field *f*.

Because of space limitations, several methods are reduced to one vertex in the concern graph, and fields in `FileList` class are omitted. We marked the criterion methods with a dot.

The concern graph includes the identified notification methods, and it includes the methods `run` and `makeDuplicationGroups` that implement the process detecting duplicated files. The notification methods have method calls to methods in `CheckerListener` class. This concern graph shows the structure of the notification concern: the methods for processing files use the notification methods to notify the progress to listener objects.

The concern graph also includes `FileList` class and the fields `is_recursive` and `folders`. These entities represent the list of files to be processed in `run` method. The `duplicated_files` field in the concern graph contains the detected duplicated files. Therefore, the concern graph includes entities needed to understand the notification concern and the related process that detects the duplicated files.

In contrast, the conservative (non-heuristic) program slicing results in many methods for user interface that process input parameters and output the result of `run` method. These methods excluded from our heuristic slice are useless to understand the notification concern.

5. Discussion

The preliminary case study shows that heuristic approach is promising to extract a small set of software entities related to a concern. Applying our method to various concerns in other applications is our next work.

Our heuristics for preliminary study is based on control-flow. However, the dominator for input vertices may be too far from the vertices and it results a large program slice. To handle such kind of concerns, we will have to evaluate other possible heuristics for example:

Data-flow pattern We have noticed two patterns in data-flow. One is an object returned by a method that is followed in the next statement by an invocation. This kind of data path might indicate a series of action contributing to one concern. The other is long data-flow path to share one parameter among several modules. Such a shared parameter might have an important role in a concern.

Name similarity Developers often have a consistent rule to name methods and fields. When a method is determined as a vertex terminating slicing, another method which has similar name may be a termination vertex.

The size of dependence graph is proportional to the number of methods in the graph. We should handle a concern even if a concern is crosscutting many modules. We plan to develop a source code viewer which is same as SeeSoft [2] for large-scale software.

6. Related Work

Chopping, which is a variant of program slicing, is proposed to support the understanding of how a statement affects to another statement [5]. Chopping extracts control-flow and data-flow paths from input variables to output variables in a program dependence graph. This approach requires developers to distinguish statements into input and output. However, such a labeling task is hard for a developer who has just found variables and methods without enough knowledge about a concern. Therefore, it is not suitable to inspect the structure of a concern.

SNIAFL is an information retrieval approach to extract methods related to the feature and a pseudo execution trace that represents the dynamic behavior of the methods [19]. Our method is useful for developers to analyze static structure of entities related to a feature found by this method.

Shepherd et al. proposed a natural language processing approach to extract functional concerns from identifiers in the source codes [14]. 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 is not suitable to support understanding collaboration among modules since this approach is based on identifiers in source code.

Walkinshaw et al. proposed a method based on program slicing 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 [15]. Their approach extracts a call path including a set of "landmark methods" specified by a user. A scenario usually consists of several concerns including user-interface, business logic and other non-functional concerns. Our approach aims to find a set of entities which contribute a concern excluding other concern in a scenario.

Krinke proposed a filtering approach to slicing and chopping with *barriers* [8]. Barriers are vertices which terminate graph traversal. They regard source and target criteria as barriers to extract only statements involved in the paths from the source to the target. Our approach is an automatic method to find appropriate barriers.

Robillard et al. proposed to categorize entities investigated by a developer into concerns based on the developer's activity [12]. This method provides a list of entities for a concern, but not the relations among entities. Our method provides the additional information for understanding the concern.

Robillard also proposed an approach based on static analysis and developers' activity to suggest software entities related to their task [13]. This approach calculates a value for an entity indicating its degree of relevance to the set of entities investigated by developers. The value might be effective to filter the resultant slice of our approach.

7. Summary and Future Work

Understanding how software entities collaborate to implement a functional concern is an important process in the software maintenance task.

We proposed a novel approach to find a subgraph of a program dependence graph that represents collaboration of entities specified by developers. We introduced heuristics into program slicing to stop graph traversal in slice calculation to get a small set of entities closely related to the specified entities.

In the future work, we will develop a fully automated analysis tool for Java and apply various heuristic patterns to large-scale software. We are planning to evaluate the effectiveness by comparing with other feature location and slicing techniques.

Acknowledgement

This work was supported by MEXT.Grant-in-Aid for JSPS Fellows (No.17-9539). The authors are grateful to Gail Murphy for her valuable comments.

References

[1] Cooper, K. D., Harvey, T. J. and Kennedy, K.: A Simple, Fast Dominance Algorithm. Avilable online at http://www.cs.rice.edu/k̃eith/EMBED/dom.pdf.

[2] Eick, S. G., Steffen, J. L. and Sumner Jr., E. E.: Seesoft - a Tool for Visualizing Line Oriented Software Statistics. IEEE Transactions on Software Engineering, Vol.18, No.11, pp.957-968, November 1992.

[3] Fjeldstad, R. K. and Hamlen, W. T.: Application Program Maintenance Study: Report to Our Respondents. Proceedings of GUIDE 48, Philadelphia, Pennsylvania, April 1983.

[4] Horwitz, S., Reps, T. and Binkley, D.: Interprocedural Slicing Using Dependence Graphs. ACM Transactions on Programming Languages and Systems, Vol.12, No.1, pp.26-60, January 1990.

[5] Jackson, D. and Rollins, E. J.: A New Model of Program Dependences for Reverse Engineering. Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE 1994), pp.2-10, New Orleans, Louisiana, USA, December 1994.

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

[7] Krinke, J.: Visualization of Program Dependence and Slices. Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004), pp.168-177, Chicago, Illinois, USA, September 2004.

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

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

[10] Murphy, G. C., Kersten, M., Robillard, M. P. and Čubranić D. C.: The Emergent Structure of Development Tools. Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005), pp.33-48, Glasgow, UK, July 2005.

[11] Robillard, M. P. and Murphy, G. C.: Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), pp.406-416, Orland, Florida, USA, May 2002.

[12] Robillard, M. P., Murphy, G. C.: Automatically Inferring Concern Code From Program Investigation Activities. Proceedings of the 18th International Conference on Automated Software Engineering (ASE 2003), pp.225-234, Montreal, Canada, October 2003.

[13] Robillard, M. P.: Automatic Generation of Suggestions for Program Investigation. Proceedings of the 13th SIGSOFT Symposium on Foundations on Software Engineering (FSE 2005), pp.11-20, Lisbon, Portugal, September 2005.

[14] Shepherd, D., Pollock, L. and Vijay-Shanker, K.: Towards Supporting On-Demand Virtual Remodularization Using Program Graphs. Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006), pp.3-14, Bonn, Germany, March 2006.

[15] Walkinshaw, N., Roper, M. and Wood, M.: Understanding Object-Oriented Source Code from the Behavioural Perspective. Proceedings of the 13th International Workshop on Program Comprehension (IWPC 2005), pp.215-224, St. Louis, Missouri, USA, May 2005.

[16] Weiser, M.: Program Slicing. IEEE Transactions on Software Engineering, Vol.10, No.4, pp.352-357, July 1984.

[17] Whaley, J. and Lam, M. S.: Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2004), pp.131-144, Washington, DC, USA, June 2004.

[18] Zhao, J.: Applying Program Dependence Analysis to Java Software. Proceedings of Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium, pp.162-169, December 1998.

[19] Zhao, W., Zhang, L., Liu, Y., Sun, J. and Yang, F.: SNIAFL: Towards a Static Noninteractive Approach to Feature Location. ACM Transactions on Software Engineering and Methodology, Vol.15, No.2, pp.195-226, April 2006.