Identifying High-Level Dependence Structures Using Slice-Based Dependence Analysis

Zheng Li
CREST, Department of Computer Science,
King's College London,
Strand, London, United Kingdom.
zheng.li@kcl.ac.uk

Abstract

This thesis presented a framework of the possible combination of approaches for low-level program slicing-based-dependence analysis and high-level concept assignment. Three combination techniques, concept extension, concept abbreviation and concept refinement, are presented and empirical studied to address the problem of program maintenance. The ten C subject programs are studied and more than 600 concept bindings are identified. Dependence based metrics are defined to evaluate three techniques that provides evidence of both advantages and disadvantages.

Keywords: Slicing, Dependence Analysis, Concept Assignment

1 Introduction

Program comprehension plays an important role in nearly all software related tasks. In software maintenance, it is estimated that up to 60% of the total time allotted is spent on program comprehension [7]. The main task in program comprehension is the identification of components and the reconstruction of the essential relations between them. A variety of analysis based techniques were proposed either at domain knowledge based high-level or at source code based low-level.

High-level analysis retrieves a knowledge-based view that is more abstract than the source code, such as pattern recognition [16] and concept assignment [3]. Low-level analysis represents the details of source code and analyses their relations, such as control and data flow analysis [1, 8] and dependence analysis[15].

Both high-level and low-level analysis have advantages and disadvantages [17], where each individual analysis technique at either high-level or low-level outperforms the others in certain situations. The lack of a general method triggers the demand for a reasonable combination of the single analyses between high-level and low-level which can benefit both by overcoming the weaknesses of each other. For example, high-level analysis techniques can provide a reasonable analysis scope with domain knowledge for low-level analysis techniques, then avoiding the scalability problem of low-level techniques; low-level analysis techniques can improve the accuracy of high-level techniques.

In this thesis, three combination techniques of high-level and low-level analysis techniques, Concept Extension, Concept Abbreviation and Concept Refinement, are presented and empirical investigated. Hypothesis-Based Concept Assignment (HB-CA) [12] and slicing based dependence analysis are selected as representations of high-level and low level techniques.

The three combination approaches have tackled the problem of program maintenance. An engineer can locate portions of source code (i.e., the concept bindings) related to the target using concept assignment. If the concept binding is still quite large, Concept Abbreviation that extracts key statements can help to focus attention more rapidly on the core of a concept binding, and Concept Refinement can remove the non-concept related statements within the binding. Once the code is changed as required by the task, Concept Extension can build executable concept slice with respect to the concept binding, of which the test cost would be reduced compared to testing the whole program.

Figure 1 provides a simple example of the three combination techniques. The portion of source code is the computation of cylinder area and volume. Suppose the domain model includes indicators for the concept area, indicating the computation of the *area* concept. The fifth and seventh statements include an indicator (the variable *area*) for the *area* concept. In this simplified example, Lines 5-7 are assigned to the *area* concept, which is indicated using **bold** font in Figure 1. Note that HB-CA generally assigns con-

```
1   Diameter=2*r;
2   perimeter=PI*Diameter;
3   undersurface=PI*r*r;
4   sidesurface=perimeter*h;
5   area=2*undersurface+sidesurface;
6   volume=undersurface*h;
7   printf("\nThe Area is %d\n", area);
8   printf("\nThe Volume is %d\n", volume);
```

Figure 1. An example.

cepts to regions rather than single lines.

In the example shown in Figure 1, the key statements extracted using Concept Abbreviation for the portion of source code have been `shadowed` (i.e., Line 3). This statement has contributed to the computation for both principal variables area and volume that highlighted using box (i.e., two variables in Line7 and 8). Consider the concept binding area (i.e., Lines 5-7), Concept Refinement removes the non-concept related statements (i.e., Line 6) and the refined dependence based concept binding have been `shaded` (i.e., Line 5 and Line 7). Concept Extension builds the executable concept slice with respect to the concept binding area.

The contributions of this thesis are as follows:

1. The proposal of a conceptual framework for three combination approaches between concept assignment and program slicing which extend, abbreviate and refine the concept binding using dependence analysis.

2. The proposal of a new approach for dependence based concept refinement: the Vertex Rank Model (VRM).

3. Experimental demonstration that dependence based concept refinement is better than the other two combination approaches.

4. The introduction of dependence based metrics that evaluate high-level source code extractions.

2 Experimental Design

To implement the three types of combination approaches of concept assignment and slicing based dependence analysis, the data of concept bindings and slices need to be produced first. WeSCA is an HB-CA implementation tool that identify concept bindings in the source programs of C and COBOL [9]. CodeSurfer [13], a commercial tool developed by Grammatech for C is employed as the slicer in the experiments.

The ten programs used in the studies are summarised in Table 1, which lists each program with some statistics related to the program and its System Dependence Graph (SDG). These include the size of the program in lines of code (LOC) and source lines of code (SLOC), and the size of the resulting SDG in vertices. The fifth column reports whether the program includes large dependence clusters or not and the sixth column provides the brief description for each program.

The ten programs used in the studies cover various types of code, including industry code and open source, without and with large dependence clusters [5]. The subject size ranges between 3KLOC and 29KLOC. Within the ten subjects, more than 600 concept bindings are identified and studied.

Dependence analysis and slices are constructed using CodeSurfer. Three algorithms are designed for the three combination approaches. In this thesis, the analysis scope for three approaches is HB-CA concept bindings, of which the size is smaller than a procedure, so the computation cost of each algorithm would not be expensive.

Informally, dependence clusters can be thought of as strongly connected statements in the dependence graph. From the standpoint of a dependence analysis, such as slicing, including any part of a dependence cluster causes the inclusion of the dependence cluster. Large dependence clusters have been defined as those that include more than 10% of the program [5]. Because they tend to affect any and all slicing based techniques, the effects of large dependence clusters need to be investigated for the combination techniques.

Statistical analysis is used to analyse the experimental data in the study. These include correlation analysis and mean (or meant) comparison analysis.

3 Concept Extension *

Concept extension uses program slicing to 'extend' a concept binding by tracing its dependencies. The slicing makes concept binding larger in size in order to make it executable, thus it is named concept extension.

Executable Concept Slicing (ECS) [9, 14] is a framework that unifies program slicing and concept assignment for higher-level executable source code extraction. Though concept assignment derives a high degree of expressibility from the domain level at which its extraction criteria are expressed, it lacks the executability of program slicing. However, slicing, though it produces executable subprograms, can only do this using extremely low level criteria. The study of concept extension presents the first experimental evidence that it is possible to combine the expressive power of these concept assignment techniques with

*This chapter is based on the work published by the author at WCRE 2006 [10] and JSS[11]

Table 1. Experimental Subjects used in the thesis.

Subjects	LoC	SLoC	Vertices	Large Cluster	Description
acct-6.3.2	3,890	2,384	9,827	No	Process monitoring tools
EPWIC-1	7,943	4,232	19,545	No	Image compressor
space	9,126	6,576	34,684	No	ESA space program
oracolo2	9,477	5,842	24,754	No	Antennae array set-up
CADP	12,930	10,246	52,591	No	Protocol toolbox
userv-1.0.1	7,301	5,320	95,056	Yes	Access control utility
indent-2.2.6	8,259	5,424	20,776	Yes	Text formatter
bc-1.06	10,449	7,342	32,381	Yes	Calculator
diffutils-2.8	10,743	6,767	26,997	Yes	File comparison utilities
findutils-4.2.25	28,887	19,216	96,073	Yes	File finding utilities
Total	109,005	73,349	421,684		

the semantic guarantees that accrue from the executability of program slicing. It defines four criteria types from the ostensibly most coherent (ECS) down to the least coherent (random) and compares the resulting slice sizes. More formally, the investigation attempts to verify the hypothesis that a slice criterion formed from a concept binding will produce smaller slices than an equivalently–sized slice criterion formed without any consideration for conceptual information.

The results from the study provide evidence to support the claim that the statements identified by concept assignment are not arbitrary: they form a coherent set of related parts of the program, because the slices constructed from them have a small degree of Coverage and Overlap between each other. This is an encouraging finding, because it indicates that there is no 'size explosion' when constructing executable concept slices.

4 Concept Abbreviation *

Concept abbreviation uses a slicing-based technique to shorten a concept binding without losing the computational impact of the concept. The resulting statements contribute more to the computation embodied by the binding than others, namely key statements. As the set of key statements is smaller in size than the original concept binding, this technique is named concept abbreviation.

The approach of concept abbreviation, parameterised Key Statement Analysis (KSA), is presented in the study of concept abbreviation.

The algorithms for KSA [14, 9] are based on the notion of Bieman and Ott's 'principal' variables, which are those which might be considered to be the result of a set of statements [2, 18]. Three kinds of principal variables are defined, i.e., global and assigned variables, call-by-reference and assigned variables, or the parameter to an output statement of a concept binding. Key statements are computed as an optimised intersection of the intraprocedural static backward slices on these principal variables.

The framework together with an improvement over previous approaches to KSA is shown. The keyness of the identified statements is measured using their Impact and Cohesion. The experiments were implemented not only for concept bindings but also for functions. The results indicate that the identified key statements have high Impact and Cohesion and thus represent the core of a function's computation. The identification of key statements can be helpful for users to focus attention more rapidly on the core of a concept binding. However, the results also reveal that only approximate one third analysis scope contains principal variables.

5 Concept Refinement

Concept refinement uses program chopping to remove non-concept-dependent statements within a concept binding and produces a more accurate dependence based concept binding. As the remaining statements are highly dependent without losing domain knowledge, this technique is named concept refinement.

This technique first builds and optimises dependence graphs for HB-CA concept bindings, where the vertices are the program points of the binding and the edges are the data dependence and control dependencies between vertices. It then uses program chopping to extract the refined dependence-based concept bindings. The Vertex Rank Model (VRM) is proposed to rank all vertices in a dependence graph by assigning a weight value to each vertex based on the dependencies between them. A vertex with higher dependence (directly or indirectly) upon others is assigned a higher value. Essentially, the VRM is the PageRank Model [6] used by Google to rank web pages. In this

*This chapter is based on the work published by the author at SCAM 2008 [4]

study of concept refinement, the VRM is used to identify the source and the sink for chopping.

The results from the experiments and the statistical analysis indicate that the VRM is an efficient technique for computing the source and sink vertices and more than 80% of the refined the dependence based concept bindings have the same impact as the original HB-CA concept bindings. The significance of size reduction confirms the approach has the ability to rule out the non-concept-dependent statements. Therefore, the resulting refined concept bindings are more accurate than the original concept bindings, which reduces cost and effort of analysing.

6 Summary and Future Work

This thesis is geared toward providing a whole framework of the possible combination of approaches for low-level program slicing-based-dependence analysis and high-level concept assignment. Three combination techniques are empirically studied using ten subject programs. Apart from a knowledge base for concept assignment with human involvement, the process can be implemented automatically. More than 600 concept bindings were identified and studied to provide evidence of both advantages and disadvantages of these three approaches. Within the three combination techniques, concept refinement may be more suitable for program comprehension.

In the future, further approaches for dependence based concept could combine HB-CA concept assignment with both syntactic and dependence analysis. Applying syntactic analysis could help the segmentation of the HB-CA concepts to give more precise concept boundaries. Further work on the effect of large dependence clusters could be implemented by investigating the causes of large dependence clusters and removing large dependence clusters.

Acknowledgements

The author is funded by EPSRC Grant GR/T22872/01 and GR/S93684/01. The author wishes to thank his supervisor Mark Harman and second supervisor Nicolas Gold for providing the support and motivation to do research throughout the duration of this work.

References

[1] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137, 1976.

[2] J. M. Bieman and L. M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, Aug. 1994.

[3] T. J. Biggerstaff, B. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *15th International Conference on Software Engineering*, pages 482–498, Los Alamitos, California, USA, May 1993. IEEE Computer Society Press.

[4] D. Binkley, N. Gold, M. Harman, Z. Li, and K. Mahdavi. Evaluating key statements analysis. In 8^{th} *International Workshop on Source Code Analysis and Manipulation (SCAM 08)*, pages 121–130, Beijing, China, Sept. 2008. IEEE Computer Society Press.

[5] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In 21^{st} *IEEE International Conference on Software Maintenance*, pages 177–186, Los Alamitos, California, USA, 2005. IEEE Computer Society Press.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.

[7] G. Canfora, L. Mancini, and M. Tortorella. A workbench for program comprehension during software maintenance. In *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*, page 30, Washington, DC, USA, 1996. IEEE Computer Society.

[8] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.

[9] N. Gold, M. Harman, D. W. Binkley, and R. M. Hierons. Unifying program slicing and concept assignment for higher-level executable source code extraction. *Software Practice and Experience*, 35(10):977–1006, 2005.

[10] N. Gold, M. Harman, Z. Li, and K. Mahdavi. An empirical study of executable concept slice size. In 13^{th} *Working Conference on Reverse Engineering (WCRE 06)*, pages 103–114, Benevento, Italy, Oct. 2006. IEEE Computer Society Press.

[11] N. Gold, M. Harman, Z. Li, and K. Mahdavi. An empirical study of the relationship between the concepts expressed in source code and dependence. *Journal of Systems and Software*, 81:2287–2298, 2008.

[12] N. E. Gold. *Hypothesis-Based Concept Assignment to Support Software Maintenance*. PhD Thesis, Department of Computer Science, University of Durham, 2000.

[13] Grammatech Inc. The codesurfer slicing system, 2002.

[14] M. Harman, N. Gold, R. M. Hierons, and D. W. Binkley. Code extraction algorithms which unify slicing and concept assignment. In *IEEE Working Conference on Reverse Engineering (WCRE 2002)*, pages 11 – 21, Los Alamitos, California, USA, Oct. 2002. IEEE Computer Society Press.

[15] S. Horwitz, T. Reps, and D. W. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[16] R. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-based reverse-engineering of design components. *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 226–235, 1999.

[17] W. Löwe, M. Ericsson, J. Lundberg, and T. Panas. Software comprehension - integrating program analysis and software visualization. In *Software Engineering Research and Practice - SERPS*, 2002.

[18] L. M. Ott. Using slice profiles and metrics during software maintenance. In *Proceedings of the* 10^{th} *Annual Software Reliability Symposium*, pages 16–23, 1992.