Java Program Analysis Projects in Osaka University: Aspect-Based Slicing System ADAS and Ranked-Component Search System SPARS-J

Reishi Yokomori [†], Takashi Ishio [†], Tetsuo Yamamoto [††],
Makoto Matsushita [†], Shinji Kusumoto [†] and Katsuro Inoue [†]
[†] Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
[††] Japan Science and Technology Corporation, 4-1-8, Honmachi,
Kawaguchi, Saitama 332-8531, Japan
{yokomori, t-isio, t-yamamt, matusita, kusumoto, inoue}@ist.osaka-u.ac.jp

Abstract

In our research demonstration, we show two development support systems for Java programs. One is an Aspect-oriented Dynamic Analysis and Slice calculation system named ADAS, and another is a Software Product archiving, Analyzing, and Retrieving System for Java named SPARS-J.

1. Introduction

In our research demonstration, we will show two types of development support systems for Java programs. The one is a slicing system for Java programs named ADAS(Aspect-oriented Dynamic Analysis and Slice calculation system), and another is a retrieval system for Java components named SPARS-J(Software Product archiving, Analyzing, and Retrieving System for Java). ADAS supports debugging tasks for Java programs based on program slicing technique, and SPARS-J provides a retrieval result with the evaluation value based on actual use relations among components.

2 Aspect-oriented Dynamic Analysis and Slice Calculation System: ADAS

ADAS is a tool, which aids debugging tasks based on program slicing technique.

Program slicing has been proposed to localize faults efficiently in the program[7]. By definition, slicing is a technique which extracts all statements that possibly affect some set of variables in the program. The set of all extracted statements is called a slice.

We have been extended the program slicing to DC slicing, using dynamic data dependence information to calculate accurate slices with lightweight costs [6].

In process of DC slice calculation, it is an important issue how to analyze dynamic data dependence. In the past research, such function have not been encapsulated in a single module. Actually, the function was implemented as a pre-processor which inserts analysis operations in the target program code, or as a customized Java Virtual Machine (JVM). But these approaches are hard to implement and to maintain. We have applied Aspect-Oriented Programming (AOP) to collect dynamic information. Since collection of dynamic information affects over all target program, this functionality becomes a typical crosscutting concern, which is modularized as an aspect in AOP [4].

ADAS is a DC slicing tool for Java. This system consists of three subsystems, the Dynamic Information Analyzer subsystem, the PDG (*Program Dependence Graph*) Constructor subsystem and the Slice Calculation subsystem.

The Dynamic Information Analyzer subsystem is implemented as an aspect. A programmer debugging a Java program adds this aspect to the target program and compiles all sources using AspectJ[1]. When the programmer executes the target program with a test case input, the added aspect collects runtime information, such as field data dependence, method polymorphism resolution and exception handling, and outputs the result to a file. AOP approach is independent of JVM features, and we prefer this approach for usability and adaptability. Since the module of the dynamic analysis is written as an aspect, a programmer can easily extend the analysis aspect using inheritance mechanism of AspectJ and add the analysis aspect to the file list to be compiled. Recent IDEs, support to manage configuration of the files.

The PDG Constructor subsystem reads information from the file and analyze Java source files to construct a PDG, a directed graph whose nodes represent statements in the source program, and whose edges denote dependence relations (data dependence or control dependence) between statements[5].

The Slice Calculation subsystem provides a graphical user interface. A source code viewer shows source code and slice criterions contained in the file. When a programmer selects a slice criterion, the system calculates the slice and indicates it in the viewer.

3. SPARS-J : Software Product Archiving, Analyzing, and Retrieving System for Java

SPARS-J is a retrieval system for Java components. It would be easily imagined that similar programs have been developed independently in different locations of the world or in different times in the history, without sharing knowledge of other programs. It is considered that a well-organized collection of programs or program components will improve productivity of the development and quality of the developed software products.

SPARS-J might be considered as a Google-like[2] system for software engineers. In this system, various Java source programs are collected, and they are stored in a component archive. Those components are ranked by the evaluation values (called Component Ranks[3]), which are determined by their use relations. A component searcher who wants to know about a definition or a usage of a component will give queries by keywords to SPARS-J. These queries are analyzed and the matched results are listed by the Component Ranks. By the Component Ranks, we consider that components with high reusability can be found effectively.

SPARS-J consists of two subsystems, Constructing Databases subsystem and Searching Components subsystem.

In Constructing Databases subsystem, a database for component search is built from Java source code files. Collected various Java source programs are analyzed syntactically and stored in the archive with information, such as all the appearance words, use class names and the metrics in each component. The appearance words are indexed to make a reverse dictionary for retrieval. The use class names are analyzed to determine use-relations between components. The metrics are used for measurement of a similarity between components. In software development process, we can imagine that a component may be reused with minor changes, by not simple copying. In order to calculate these components as one group, this subsystem measures similarity between components. As an example of a metric value, we use LOC, cyclomatic complexity, and so on. Similar components are packed into one group, and use relations associated to those components are also merged. All components(groups) are ranked by the evaluation values called Component Ranks, which are determined by their use relations, and the Component Rank of each component is also stored in the archive.

On the other hand, Searching Components subsystem provides a retrieval function for user. Basically, this subsystem performs component retrieval from all the appearing words including comments in source code files. However, a user can request to perform an advanced search according to the purpose of the user. For example, this subsystem can remove the components with which keywords appear only in its comment from a result, or extract the only components with which keywords appears in the definition of a class or a method, or extract only the use example.

A query specified by a user is analyzed and decomposed into a set of keywords. For each keyword, this subsystem checks the component with which the keyword appears by referring to reverse dictionary. A result is ordered by the Component Ranks, the user receives the result through the browser with the information of each components and a part of its code. Furthermore, the user may click the results to get detailed information on the search components, or may add keywords to perform further retrieval. In this way, the user can effectively find a component that is used most frequently and has high reusability.

References

[1] AspectJ Team, “The AspectJ Programming Guide”, http://aspectj.org/doc/dist/progguide/

[2] google, http://www.google.com/

[3] K. Inoue et al.: “Component Rank: Relative Significance Rank for Software Component Search”, to be appeared in Proceedings of ICSE 2003, Portland, Oregon, 2003.

[4] G. Kiczales et al.: “Aspect Oriented Programming”, Proceedings of ECOOP, vol.1241 of LNCS, pp.220-242(1997).

[5] K. J. Ottenstein and L. M. Ottenstein: “The program dependence graph in a software development environment”, Proceedings of SESPSDE, pp.177–184, Pittsburgh, Pennsylvania, April (1984).

[6] T. Takada et al.:“Dependence-Cache Slicing: A Program Slicing Method Using Lightweight Dynamic Information”, Proceedings of IWPC2002, pp.169-177, Paris, France, June (2002).

[7] M. Weiser: “Program slicing”, IEEE Transactions on Software Engineering, SE-10(4):352-357(1984).