Kaveri: Delivering the Indus Java Program Slicer to Eclipse*

Ganeshan Jayaraman, Venkatesh Prasad Ranganath, and John Hatcliff

Department of Computing and Information Sciences,
Kansas State University,
234 Nichols Hall, Manhattan KS, 66506, USA
{ganeshan, rvprasad, hatcliff}@cis.ksu.edu

Abstract. This tool paper describes a modular program slicer for Java built using the Indus program analysis framework along with it's Eclipse-based user interface called Kaveri. Indus provides a library of classes that enables users to quickly assemble a highly customized non-system dependence graph based inter-procedural program slicer capable of slicing concurrent Java programs. Kaveri is an Eclipse plugin that relies on the above library to deliver program slicing to the eclipse platform. Apart from the basic feature for generating program slices from within eclipse along with an intuitive UI to view the slice, the plugin also provides the capability for chasing various dependences in the application to understand the slice.

1 Introduction

Program slicing is a well known analysis that can be used to identify parts of the program that influence or are influenced by a given set of program points (slice criteria). There have been a large number of publications along with a small number of implementations for languages such as FORTRAN, ANSI C, and Oberon. [1] Most of the implementations have been targeted to particular applications of program slicing such as program comprehension, testing, program verification, etc. Moreover, only few robust slicing tools exist for languages like Java and C++.

From our experience we have found that the properties required of a slice depend on the application. For example, the program slice needs to be executable for program verification applications such as Bandera[2] but not for program comprehension purposes. Similarly, the slice needs to be "residualizable" for some applications and such transformations can again be constrained by the application. Hence, program slicers need to be modular and flexible (customizable) as opposed to being monolithic and rigid.

* This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF (CCR-0306607) by Lockheed Martin, and by Intel Corporation.

[1] Please refer to Jens Krinke's Dissertation[1] for a brief informative overview of available implementations.

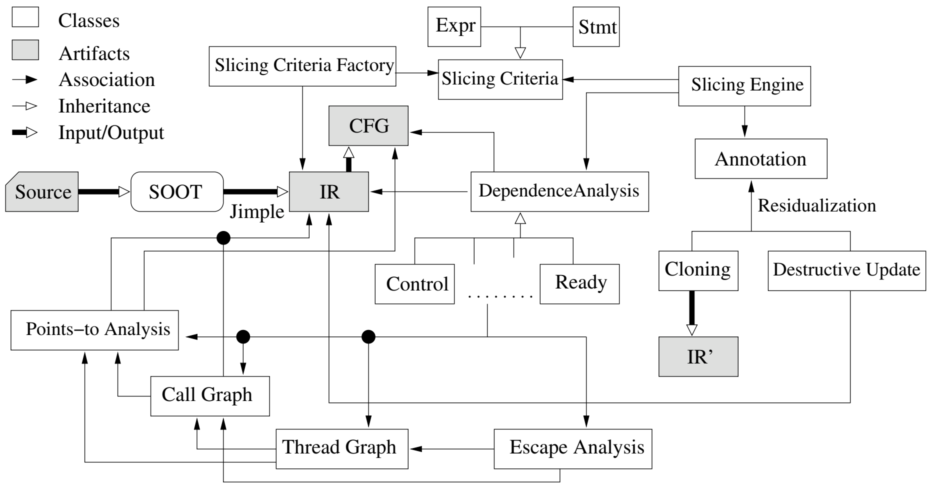

Fig. 1. Bird's eye view of classes and artifacts in Indus Java Program Slicing Library

2 Indus Java Program Slicer

Drawing from the our experience with Bandera slicer, we have implemented a program slicing library that can handle almost full Java[2]. To the best of our knowledge, this is the first publicly available Java implementation of a program slicer for Java.

Indus modules work on Jimple (SOOT [3]) representation of Java programs and bytecode.

The key features of Indus Java Program Slicing library apart from generating backward and forward slices are as follows.

Batteries Included. The program slicing library, directly or indirectly, requires various high level analyses such as escape analysis [4], monitor analysis, safe-lock analysis [5], and analyses to calculate and prune various dependences – intra- and inter-procedural data dependence, control [6] dependence, interference [7] dependence, ready dependence and synchronization dependence [5]. These high level analyses rely on low-level information such as object-flow information [8], call graph, and thread graph [4]. All of the above analyses and other related analyses are available in Indus.

Modularity. Most of the above mentioned analyses are available as independent modules. Hence, the user can use only the required analyses. Each analysis implementation is decoupled from it's interface to enable easy experimentation with various implementations. This is a recurring theme in Indus which is leveraged in the slicer.

Non-SDG Based. Most slicing related work is based on program/system dependence graphs (PDG/SDG) that contain dependence edges to account for various aspects of the language such as unconditional jumps, procedure calls, aliasing, etc. This can be an obstacle for reusability. Instead, in Indus, the logic to handle such aspects is encoded

[2] With the exception of dynamic class loading, reflection, and native methods.

in the slicing algorithm to decrease coupling and increase cohesion. As a result, dependence information is readily reusable, fine-tuning of slicing algorithm is simplified, and maintenance becomes easy.

Program Slicing = Analysis. In Indus, program slicing is considered to be pure program analysis – program slicing only calculates the program points that belong to a slice. This simplifies the slicing algorithm and enables the same slicing algorithm to be used with different transformations as required by the applications.

Inter-Procedural and Context-Sensitive. The slicer considers calling contexts (where possible) to generate precise inter-procedural slices. The user can generate context-sensitive slice criteria to further improve precision. *Scoping*, a feature that can be used to control the parts of the system that need to be analyzed, can be used to to restrict the scope of slicing to a single method, a collection of methods, a collection of methods belonging to a collection of classes, etc.

Concurrent Programs. This implementation can slice concurrent programs by considering data interference and other synchronization related aspects that are inherent to concurrent programs. Information from escape analysis and monitor analysis is used to improve the precision of concurrent program slices.

Highly Customizable. Using Indus libraries, the user can assemble a slicer that is customized for the end-application. For example, the user may choose cloning based residualization for differencing purposes or destructive-update based residualization for program verification purposes.

To verify that our library is indeed customizable to multiple application domains and also to realize a long term goal of having an UI to visualize program slices, we developed Kaveri.

3 Kaveri: A Program Slicing Plugin for Eclipse

Kaveri is a plugin that contributes program slicing as a feature to Eclipse [9]. Kaveri utilizes the Indus program slicing library to perform slicing, thereby, hiding the details of assembling a slicer customized for the purpose of program comprehension. As a program comprehension aid, Kaveri contributes the following features to Eclipse.

Slice Java Programs by Choosing Slice Criteria. The user can pick the criteria, generate the program slice, and view the slice all using the Java source editor. The plugin handles the intricacies such as mapping from Java to Jimple and driving the slicer.

View the Slice in the Java Editor. The part of the source code included in the slice is highlighted in the editor. This aids slice-based program comprehension.

Perform Additive Slicing. "What program points are common to slices based on criteria *b* and *c*?" is a common question during program comprehension. It can be answered by intersecting the slices based on criteria *b* and *c*. In Kaveri, the user can achieve this by associating different highlighting schemes to slices based on *b* and *c*, and viewing both the slices in the editor at the same time. Similarly, *Chop*s can be realised by intersecting backward and forward slices.

Support for Program Comprehension. Understanding dependence relations between various program points helps understand the generated program slice. In Kaveri, this is achieved by "chasing" dependences.

– The user can view which program points in a Java statement/expression are included in the slice via *slice comprehension view*, an eclipse view displays the Java-to-Jimple mapping for a Java statement/expression along with Jimple level slice annotations.
– As Kaveri annotates the parts of the source file in the editor, the user can use the built-in annotation navigation facility in Eclipse to keep track of dependence navigation. However, to compensate for the genericity of this facility, Kaveri maintains the dependence-based path taken by the user. The user can navigate this path and backtrack on it via a *dependence history view*.
– Kaveri also supports *path queries* that can be used to find sequences of program points that are related via a pattern of dependences and other relations specified by a language such as regular expressions.

The user can also generate a scoped slice based on scope specifications to understand the relation between certain program points independent of external influences.

Perform Context-Sensitive Slicing. In Kaveri, the user can identify calling contexts (from a inverted call tree of a finite depth) to be used in the generation of context-sensitive program slices.

We have successfully used Kaveri with code bases of $\leq$ 10K lines of Java application code (< 80K bytecodes) (excluding library code). All software and related artifacts pertaining to Indus and Kaveri are available at [10].

References

1. Krinke, J.: Advanced Slicing of Sequntial and Concurrent Programs. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau (2003)
2. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby, Zheng, H.: Bandera: Extracting finite-state models from Java source code. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE'00). (2000) 439–448
3. Sable Group: Soot, a Java Optimization Framework. (http://www.sable.mcgill.ca/soot/)
4. Ranganath, V.P., Hatcliff, J.: Pruning interference and ready dependences for slicing concurrent java programs. In: Proceedings of Compiler Construction (CC'04). Volume 2985., Springer-Verlag (2004) 39–56
5. Hatcliff, J., Corbett, J.C., Dwyer, M.B., Sokolowski, S., Zheng, H.: A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In: Proceedings on the International Symposium on Static Analysis (SAS'99). (1999)
6. Ranganath, V.P., Amtoft, T., Banerjee, A., B.Dwyer, M., Hatcliff, J.: A new foundation for control-dependence and slicing for modern program structures, Springer-Verlag (2005) *To appear in the* Proceedings of European Symposium On Programming (ESOP'05).
7. Krinke, J.: Static slicing of threaded programs. In: Proceedings ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98). (1998) 35–42 ACM SIGPLAN Notices 33(7).
8. Ranganath, V.P.: Object-Flow Analysis for Optimizing Finite-State Models of Java Software. Master's thesis, Kansas State University (2002)
9. OTI: Eclipse, an open extensible IDE written in Java. (http://www.eclipse.org)
10. SAnToS Laboratory: Indus. (http://indus.projects.cis.ksu.edu)