

Kaveri: Delivering 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 full Java called Indus and an Eclipse-based user interface to the slicer called Kaveri. Indus is 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 are influenced by or do influence a given set of program points (slice criteria). It has been widely applied technique for debugging, program comprehension, and program specialization. There have been a large number of publications along with a small number of implementations for languages such as FORTRAN, ANSI C, and Oberon. Most of the implementations have been targetted to particular applications of program slicing such as program comprehension, testing, program verification, etc. Moreover, although slicers have been developed for programming languages like C, 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 vary from application to application. For example, the program slice required in model checking based program verification applications such as Bandera[1] needs to be executable. On the other hand, executability is not required in applications such as program comprehension via visualization. Similarly, in applications such as Bandera, the slice needs to be residualized whereas this need not be the case for purpose of program comprehension. Even transformations such as slice residualization can be constrained by the application. Hence, program slicers need to be modular and customizable instead of being monolithic and hard to customize.

2 Indus Java Program Slicer

Drawing from the lessons we learnt while implementing a program slicer for Bandera, we have implemented a program slicing library that can handle almost

* 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.

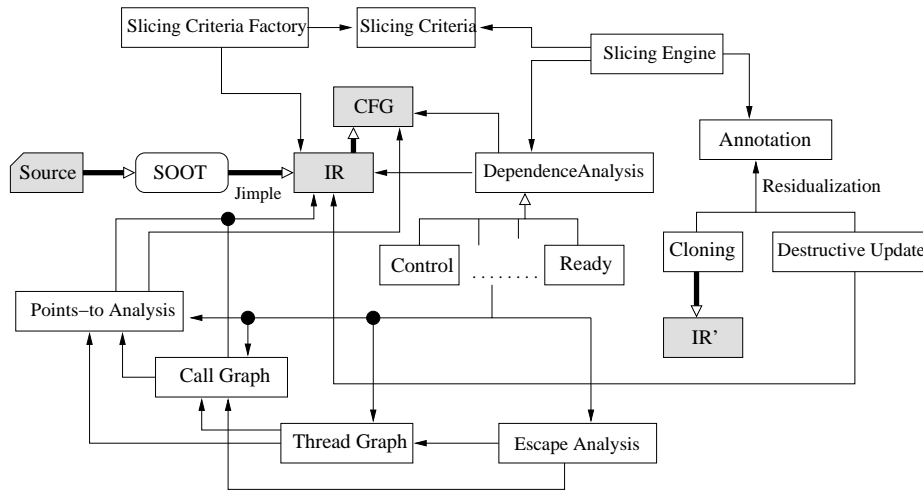


Fig. 1. Bird’s eye view of classes in Indus Java Program Slicing Library along with the relationships between classes and artifacts (the grayed boxes). The thin line with filled arrowheads indicate UML-like dependency between classes. The thin lines with empty arrowheads indicate inheritance relationship. The thick lines with arrowheads indicate input/output.

full Java¹ as part of project Indus. To the best of our knowledge, this is the first publicly available implementation of a program slicer for Java written in Java.

In Indus, Java programs are represented in Jimple via SOOT [2] library. Hence, modules in Indus libraries can be applied to source code and bytecodes provided they can be translated into Jimple.

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[3], monitor analysis, safe-lock analysis [4], and analyses to calculate and prune various dependences – method-local data dependence, inter-procedural data dependence, control dependence, data interference [5] dependence, ready dependence and synchronization dependence [4]. These high level analyses require low-level analyses and information such as object-flow information [6], call graph, and thread graph[3]. The implementation of all of these analyses and a few more are available in Indus.

Modularity Most of the above mentioned analyses are available as independent modules. Hence, a user can use an analyses independent of other analyses in Indus. Moreover, each implementation is separated from it’s interface. This enables the user to experiment with various implementations that provide the same interface. In fact, this feature is used in the slicer to vary the level of precision. This theme applies to other analyses available in Indus.

Non-SDG based Most slicing related work is based on program/system dependence graphs. In interprocedural settings, these graphs have dependence

¹ With the exception dynamic class loading, reflection, and native methods.

edges to account for various aspects of the system such as unconditional jumps, procedure calls, aliasing, synchronization, etc. This is an inconvenience if dependence information is used for purposes other than slicing. Hence in Indus, instead of maintaining such a graph, the logic to handle these various aspects is encoded in the slicing algorithm. This decreases the amount of data maintained during slicing, simplifies the access to dependence information as it is maintained in its “pure” form, and eases the maintenance of the slicing algorithm and dependence analyses due to increased cohesion and decreased coupling.

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. Transformations of any sort based on the slice are not considered as program slicing. This simplifies the slicing algorithm and enables the same slicing implementation to be used for applications with varying needs.

Inter-Procedural and Context-sensitive The implementation is capable of inter-procedural slicing. While doing so, it considers context information where possible. Hence, it is also possible for the user to specify context-sensitive slice criteria to improve the precision of slicing. Indus provides *scoping*, a feature that can be used to control the parts of the system that need to be analyzed. This can be used 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 is targeted towards concurrent programs by considering data interference and synchronization issues due to multiple threads. It also uses information from escape analysis and monitor analysis to improve the precision of slices of concurrent programs.

Highly Customizable Using Indus libraries, the user can assemble a slicer that is customized towards the end-application. For example, the user may choose cloning based residualization for differencing purposes or destructive-update based residualization in applications such as Bandera. This degree of variance is possible due to high modularity, interface-based API, and loose coupling between various modules of the library.

The library also contains Java program slicing tool that was implemented using Indus tool framework. It was successfully used it within Bandera to generate executable backward slices for the purpose of program verification via model checking. Also, it has been used to successfully slice programs (such as Java-Grande Benchmarks [7] and examples used in Bandera [8]) with 10K+ lines of application code (excluding library code).

To check if the slicing 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 have used the slicing tool inside Kaveri.

3 Kaveri: A Program Slicing plugin for Eclipse

Kaveri is a plugin that contributes program slicing as a feature to eclipse platform[9]. Kaveri utilizes 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 in Eclipse via an intuitive user interface dedicated to visualize program slice and other information from ancillary analysis.

Slice Java programs by choosing slice criteria This is a simple application of Indus Java Program Slicing library. However, Kaveri maintains a bi-directional mapping from Java to Jimple to cater the information calculated by the slicer to the user via UI. This mapping can be used by other tools as well.

View the slice in the Java editor The part of the source code included in the slice is highlighted in the editor. This simplifies the process required to reason about the application based on its slice.

Perform additive slicing During program comprehension, “what program points belong to slices starting at program points b and c ?” is a common question, and it can be answered by intersecting the slice starting from b and c . In Kaveri, the user can generate slices for b and c separately with different highlighting scheme, and view both the slices in the editor at the same time. The user can thus realize *chopping* in Kaveri.

Program comprehension through dependence tracking Understanding the dependence relations between various program points helps understand the generated program slice. In Kaveri, the user can pick program points and track various dependences in both direction to better understand the slice.

- While tracking dependences, the user can drill into the details of which program points in a statement/expression are included in the slice via the *slice comprehension view* - this view displays the Jimple statements that correspond to a Java statement indicating whether they are included in the slice.
- Eclipse has a built-in annotation-based navigation facility. As Kaveri annotates the parts of the source file in the editor, the user can use the built-in navigation support to keep track of dependence navigation. However, due to the genericity of the facility the user may lose track of the sequence of dependences that were navigated. Hence, as an aid, Kaveri remembers the sequence of dependences the user has tracked as a “path” and displays it in *dependence history view*. Using this view, the user can backtrack along the navigation path.
- 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 alike regular expressions. This simplifies the operation described in the previous item.²
- Sometimes it is required to understand the relation between various program points in a locality without considering external influences. In Kaveri, this is made possible by accepting locality specifications from the user in terms of simple concepts such as regular expressions and using it to generate a scoped slice.²

Perform context-sensitive slicing In Kaveri, the user can pick initiating call-sites from a inverted call tree of a finite depth (that is specified by the user) to identify calling contexts to be used in the generation of context-sensitive program slices.² .

We have successfully applied Indus/Kaveri to code bases of up to 10,000 lines of Java application code (< 80K bytecodes) (this does not include size of library code). All software released from project Indus along with user guides, publications, API documentation, and support forums are available at [10].

² Currently, this feature is not available in the released version of the software. It will be available shortly

References

1. 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
2. Sable Group: Soot, a Java Optimization Framework. (This software is available at <http://www.sable.mcgill.ca/soot/>)
3. Ranganath, V.P., Hatcliff, J.: Pruning interference and ready dependences for slicing concurrent java programs. In Duesterwald, E., ed.: Proceedings of Compiler Construction (CC'04). Volume 2985 of Lecture Notes in Computer Science., Springer-Verlag (2004) 39–56
4. 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 1999 International Symposium on Static Analysis (SAS'99). Lecture Notes in Computer Science (1999)
5. Krinke, J.: Static slicing of threaded programs. In: Proceedings ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98), Montreal, Canada (1998) 35–42 ACM SIGPLAN Notices 33(7).
6. Ranganath, V.P.: Object-flow analysis for optimizing finite-state models of java software. Master's thesis, Kansas State University (2002)
7. Java Grande Benchmarking Project: Java Grande Forum Benchmark Suite - Thread Version 1.0. (This software is available at http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/)
8. SAnToS Laboratory: Bandera. (This software is available at <http://bandera.projects.cis.ksu.edu>)
9. OTI: Eclipse, an open extensible IDE and tool platform written in Java. (This software is available at <http://www.eclipse.org>)
10. SAnToS Laboratory: Indus, a toolkit to customize and adapt Java programs. (This software is available at <http://indus.projects.cis.ksu.edu>)

A Features that will be demonstrated

- Slicing Java programs.
Kaveri will be used to slice a multithreaded application and the resulting slice will be explained.
Example screenshots:
 - Adding criteria. (Refer to figure 1)
 - Running the slice. (Refer to figure 2)
 - Viewing the slice. (Refer to figure 3)
 - Understand the slice. (Refer to figure 4)
- Dependence chasing.
Kaveri will be used to explore a chain of dependencies. The aim is to get a better understanding of the program through dependence chasing. The dependence history will be used to demonstrate the ease of exploring complex dependence paths using Kaveri.
Example screenshots:
 - Chasing Control and Interference Dependence. (Refer to figure 5)
 - Dependence History View. (Refer to figure 6)
- Parametric Regular Path Queries. A parametric regular path query will be used to understand a particular feature of the program in a simple manner. This feature is currently under development and we plan to release it soon.
- Scoped and Context-Sensitive slicing. These features are not part of the released version of Kaveri. These will be rolled out in the next version of Kaveri which will be released soon.

B Hardware Requirements

Kaveri has been tested to work well with the following configuration. As always higher or equivalent configurations (particularly a higher memory subsystem) would be beneficial:

- Pentium IV (2.66 GHz)
- 512MB RAM

C Software Requirements

- Eclipse – Version 3.0
- Indus Eclipse plugin – Version 0.4

D Licensing Requirements

- SAnToS laboratory license – Kansas State University.

E Screenshots

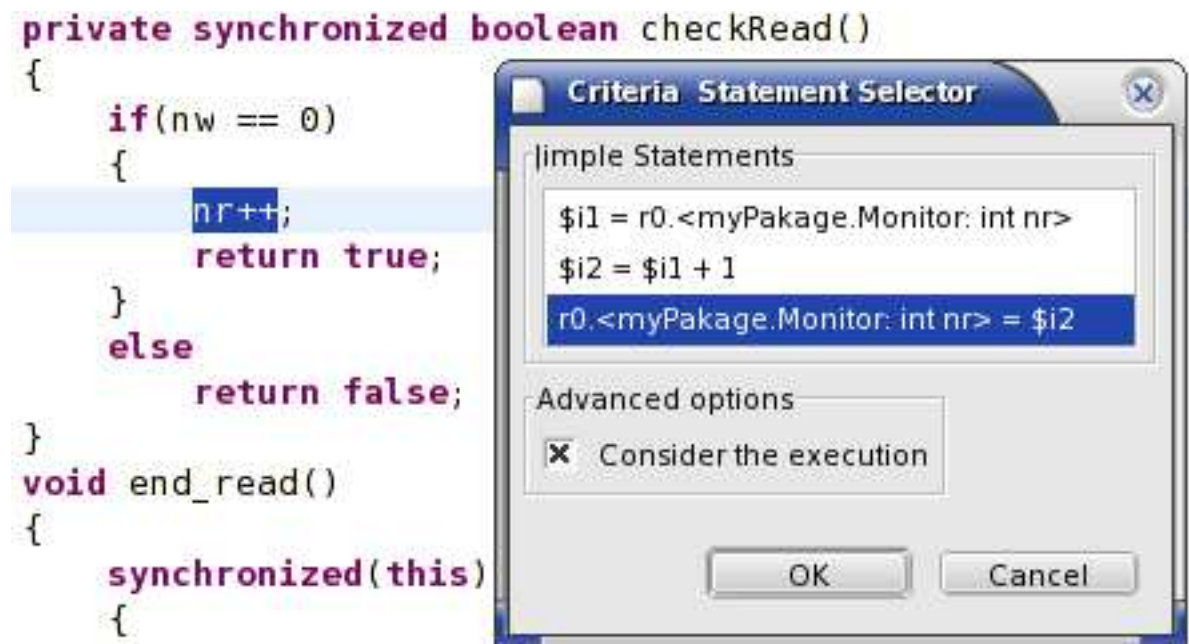


Fig. 2. Adding a statement as criteria. As the translation between Java and Jimple is not one – one, the user is allowed to pick the Jimple statements that are closest to the program point of interest. The user can also choose if the effect of executing the criterion be preserved in the slice.

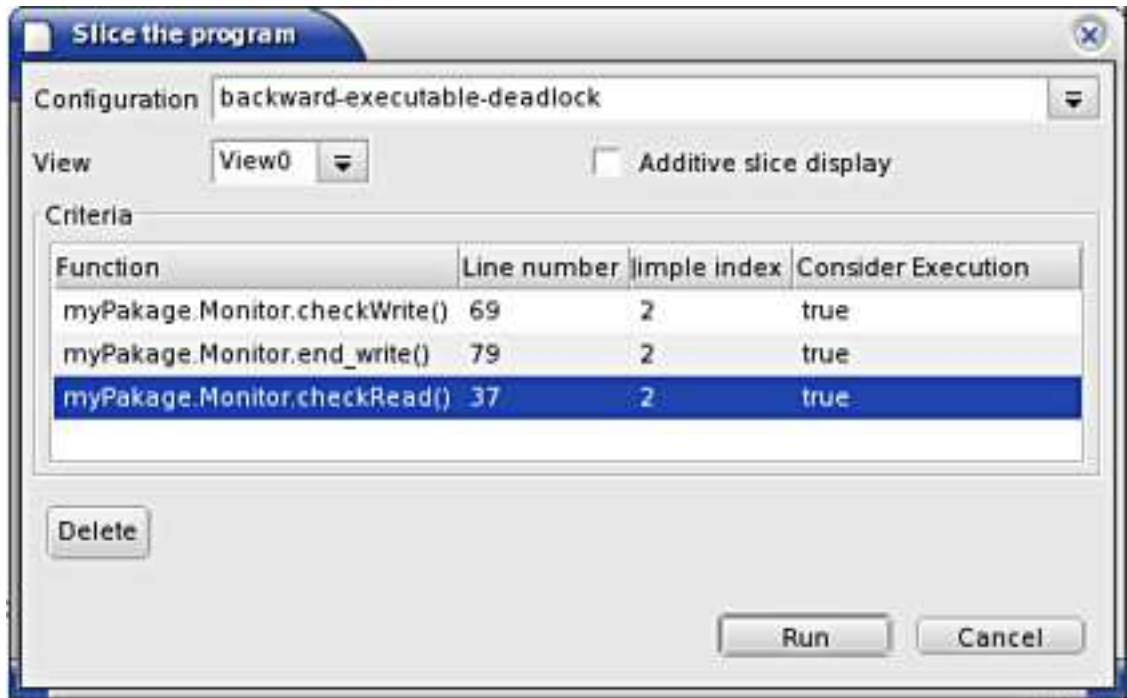


Fig. 3. When running the slice, the configuration and the criteria can be chosen. The configuration captures the required properties of the slice along with the options for slicing. In the example, the chosen configuration indicates that *a backward executable slice that preserves any deadlocks in the system* should be generated along with added slice criteria `nr++` as the criteria. The user can choose if the previous slice should be remembered for the purpose of *chopping*.

```
private synchronized boolean checkRead()
{
    if(nw == 0)
    {
        nr++;
        return true;
    }
    else
        return false;
}

void end_read()
{
    synchronized(this)
    {
        nr--;
    }
    synchronized(objectW) {objectW.notify();}
}

```

Annotation indicating that the complete statement is included in the slice

Annotation indicating that only part of the statement is included in the slice

Fig. 4. The slice is viewed as highlight annotations on the source text. The green highlight indicates that the statement is completely present in the slice (all the Jimple statements mapped to the corresponding Java statement are included in the slice) while the yellow highlight indicates only a subset of the Jimple statements mapped to the corresponding Java statement are included in the slice.

```

75 void end_write()
76 {
77     synchronized(this)
78     {
79         nw--;
80     }
81     synchronized(objectR) { objectR.notifyAll();}
82     synchronized(objectW) { objectW.notify();}
83 }
84 };
85

```

problems javadoc Console **S** Slice View Dependence History View

Statement: synchronized(objectR) { objectR.notifyAll();} line: 81

Simple Statement	Part of Slice
pr = r0.<myPackage.Monitor: java.lang.Object objectR>	true
virtualinvoke \$r7.<java.lang.Object: void notifyAll()>()	true
exitmonitor r6	true
goto [?= \$r9 = r0.<myPackage.Monitor: java.lang.Object objectW>]	true
\$r8 := @caughtexception	false
exitmonitor r6	false
throw \$r8	false

Fig. 5. The user can select a statement and view the underlying slice result. The “Slice View” shows the Jimple statements mapped to the chosen statement. It also indicates which of the Jimple statements are included in the slice. In the figure, the “Slice View” shows the underlying slice for the statement `synchronized(objectR) { objectR.notifyAll();}` Notice that some of the Jimple statements are not part of the slice as indicated by **false** in the “Part of Slice” column in the view. Because of these, the Java statement is given a partial slice annotation

```

33 private synchronized boolean checkRead()
34 {
35     if(nw == 0)
36     {
37         nr++;
38         return true;
39     }
40     else
41         return false;
42 }
43 void end_read()
44 {
45     synchronized(this)
46     {
47         nr--;
48     }
49     synchronized(objectW) {objectW.notify();}
50 }

```

Marker indicating that multiple dependencies are present (points to line 35)
 Control Dependence (points to line 38)
 Interference Dependence (points to line 47)

Fig. 6. This figure indicates the effect of viewing the control successor of the if (nw == 0) statement and the interference successor of the nr++ statement. Each dependence is highlighted with different color. The marker in the lefthand ruler line indicates that the corresponding statement participates in two different dependencies that are currently being viewed.

Tasks **S** Dependence History View

Filename	Statement	Line number	Relation with previous item
ReadersWriters.java	nr++;	37	Control Dependent
ReadersWriters.java	if(nw == 0)	35	

Fig. 7. The dependence history view shows the chain of dependences navigated by the user in the form of a stack. The user can backtrack to an earlier point in the dependence chain and follow other dependences. The view indicates the statements that were selected and the dependence that was followed to reach that statement. If the statement nr++ is present in the body of the if (nw == 0) statement and we follow the control successor of if (nw ==0), the view presented above is reached.