Slicing Concurrent Java Programs using Indus and Kaveri*

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

September 28, 2006

Abstract

Program slicing is a program analysis and transformation technique that has been successfully applied in a wide range of applications including program comprehension, debugging, maintenance, testing, and verification. However, there are only a few full-featured implementations of program slicing that are available for industrial applications or academic research. In particular, very little tool support exists for slicing programs written in modern object-oriented languages such as Java, C#, or C++.

In this paper, we present Indus[1] — a robust framework for analysis and slicing of concurrent Java programs, and Kaveri — a feature-rich Eclipse-based GUI for Indus slicing. For Indus, we describe the underlying tool architecture, analysis components, and program dependence capabilities required for slicing. In addition, we present a collection of advanced features useful for effective slicing of Java programs including calling-context sensitive slicing, scoped slicing, control slicing, and chopping. For Kaveri, we discuss the design goals and basic capabilities of a graphical presentation of slicing information that is integrated into a Java development environment.

As this paper is an extended version of a tool demonstration paper presented at the International Conference on Fundamental Aspects of Software Engineering (FASE 2005), the exposition is tool-based and user-oriented.

1 Introduction

1.1 Slicing – Concepts and Applications

Program slicing is a well-known program analysis and transformation technique that uses program statement dependence information to identify parts of a program that influence or are influenced by an initial set of program points of interest (called the *slice criteria*). For instance, given a slicing criteria C consisting of a set of program statements, a program slicer computes a *backward slice* S_b containing all program statements that influence the statements in C by starting from C and successively adding to S_b statements upon which the C statements are (transitively) data or control dependent. A *forward slice* S_f containing all program statements that C influences is calculated in an analogous manner: the slicer successively adds to S_f all statements that are (transitively) data or control dependent on the statements in C. Upon conclusion of the slice calculation, the slicer may have the capability to (a) generate an *executable* slice — a residual program containing only

*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 and by Intel Corporation.

[1] http://indus.projects.cis.ksu.edu

the statements in the slice (perhaps with a few additional statements to guarantee well-formedness), or (b) to display the original program with nodes in the slice visually high-lighted in some way.

Slicing has been widely applied in the context of debugging, program comprehension, and testing.

- **Debugging:** When debugging software, it is often the case that a bug is detected at a state associated with single program point p_b (e.g., an assertion violation). If the software is large and complex, then it is likely that the software fault occurs at a program point p_f that is statically distant (i.e. in the source code) from the program point p_b. In such cases, the developer will need to methodically sift through the source code of the software to identify the faulting program point p_f. To expedite this process, the developer will attempt to limit the search to the parts of the software that may either directly or indirectly affect the behavior (state) of the program at the program point p_b. This process can be automated using backwards program slicing starting with p_b as the slicing criteria.

- **Program comprehension:** Software developers are frequently assigned to debug, further develop, or reverse engineer code bases that they did not author. In such cases, it is often difficult for the developer to grasp the basic architecture and relationships between code units, and this is made more difficult by the fact that the code may be poorly documented and poorly written. Both backward and forward slicing can be applied to browse the code, looking for dependences between code units, flows of data between program statements, etc.

- **Testing:** There are a number of applications of slicing in the context of testing. One particular example is *impact analysis* [28], which aims to determine the set of program statements or test cases that are impacted by a change in the program, requirements, or tests. For example, in verification and validation efforts on large code bases with huge test suites, it is often very expensive to run all the tests associated with the program. If a program statement p_b is modified (e.g., due to a bug fix), rather than re-running all tests, backwards slicing using p_b as the criteria can be used to determine the subset of the tests that actually influence the behavior of the program at the point of the bug fix, and only those relevant tests need to be re-run. In addition, a developer may want to understand the potential impact that the change at p_b can have on other statements of the program. Forward slicing with p_b as the criteria can be used to locate other statements within the program that will be impacted by the change at p_b.

1.2 Tool Support

There have been a large number of publications on slicing, but only a small number of implementations for languages such as FORTRAN, ANSI C, and Oberon.[2] Most of the implementations have been targeted to particular applications of program slicing such as program comprehension, testing, program verification, etc. Moreover, although slicing tools have been developed for programming languages like C, only a few slicing tools exist for languages like Java and C++ [21, 14].

Dealing with widely-used languages like Java, C++, C# involves a number of challenges.

- **Dealing with references and aliasing:** Calculation of data dependences (determining which definitions of a variable v reach a particular use of v) is made much more difficult by pointers/references and aliasing. It is difficult to determine statically which memory cells a variable of reference type may be pointing to, and sophisticated static analyses must be used to collect information about the memory cells that could possibly be referred to by a particular variable. For soundness, such analyses must be conservative (i.e., they must over-estimate the set of cells that could be pointed to), and this approximating effect leads to imprecision in slicing (slices are larger than actually required for correctness).

[2] Please refer to Jens Krinke's Dissertation [14] for a brief informative overview of available implementations.

- **Dealing with exceptions:** Modern languages like Java and C# support exception processing. The use of exceptions and associated exception handlers introduces implicit less-structured control flow into the program which makes it more difficult to calculate the *control dependence* information needed in slicing.

- **Dealing with concurrency:** The increasing use of multi-threading further hampers analysis since languages that emphasize a shared memory model (like Java and C#) allow accesses of a memory cell in one thread to be potentially interfering with accesses in another thread (thus, creating additional and often spurious program dependences). Reducing spurious dependences by determining that accesses do not actually interfere (e.g., as guaranteed through the use of proper locking or use of heap data that is actually not shared between threads) requires sophisticated static analyses that can detect lock states, situations where objects do not escape a particular thread context, and partial order information (e.g., detecting that actions of two different threads cannot interfere because one must definitely happen before the other).

- **Dealing with libraries:** Realistic programs make extensive use of libraries to the extent that a large majority of executable code comes from libraries as opposed to actual application code written by the developer. Slicing must be able to include program representations of relevant library code while excluding library code not actually invoked by the application code.

In summary, while the basic theory of slicing for a simple imperative language can be explained rather succinctly, building a robust tool environment for slicing realistic programs written in a language like Java requires both foundational work along a number of fronts as well as a large-scale tool engineering effort.

1.3 Motivation

Our work focuses on slicing realistic Java programs. We were originally motivated to build a slicer for Java because we were seeking ways to reduce the cost of model checking concurrent Java programs in the Bandera project [5][3]. Model checking is a verification and bug-finding technique that aims to perform an exhaustive exploration of a program's state space. In simple terms, model checking a concurrent Java program involves simulating all possible executions of the program (e.g., including all possible thread schedules) and checking the paths and states encountered in that simulation against correctness specifications phrased as assertions, automata, or temporal logic formulae. While model checking can be very effective for detecting intricate flaws that are hard to detect using conventional non-exhaustive techniques like testing, it is very expensive to apply. Thus, effective use of model checking must rely on applying different abstraction techniques, imposing bounds on the state space explored, and employing heuristics for state-space search.

The effectiveness of slicing for model reduction is based on the observation that, when trying to verify a particular specification ϕ against a program P, many parts of P do not impact whether ϕ ultimately holds for P or not. For example, it is often the case that ϕ is a simple assertion or a temporal property only mentions a few of P's features (e.g., a few variable names or program points). Thus, one can use the features mentioned in ϕ to create a slice of P that omits program statements and variables that are irrelevant to ϕ's satisfaction against P. We have shown that using slicing in this manner forms a sound and complete reduction technique for model checking [10]. Our experimental studies on small to moderate size concurrent Java programs shows that slicing almost always provides some reduction (in best cases, up to a factor of four reduction in time), and incurs very little overhead compared to the end-to-end costs of model checking [7].

As we started our work, no slicing infrastructure for Java was available, and little technical work had been done to address challenges associated with slicing realistic programs mentioned above. Thus, following the maxim *"Every good work of software starts by scratching a developer's personal*

[3]This software is available at http://bandera.projects.cis.ksu.edu.

itch."[4], the lack of a robust, flexible, and publicly available program slicer for concurrent Java motivated us to implement one ourselves.

1.4 Organization

In Section 2, we provide a succinct introduction to program slicing and program dependences along with the mention of relevant concepts, techniques, and efforts specific to slicing concurrent Java programs. The Indus program slicing framework is introduced in Section 3 along with its key features and an exposition of how the framework is leveraged in Bandera in the context of program verification. We provide an introduction to Kaveri along with a screenshot driven description of its features and their application in Section 4. We conclude the paper by summarizing our contributions in Section 6.

2 Program Slicing

Given a program P and a slice criteria C, one can perform two forms of slicing: static and dynamic. *Static slices* [32] are generated by leveraging only the static information about the program, i.e. program structure, possible number of threads, possible number of objects, possible sharing of objects, etc. *Dynamic slices* [1] are generated by leveraging both dynamic and static information about the program, i.e. execution histories, execution traces. Based on the distinction, static slices are ideal for program maintenance tasks that involve determining the impact of changes and understanding an existing program while dynamic slices are ideal for debugging tasks that involve determining software fault based on some field data.

Independent of whether or not a slice is calculated using static or dynamic information, we noted in Section 1 that slicing can be carried out working backward from the criteria or forward from the criteria. A *backward slice* contains parts of the program that affect the slice criteria, while the *forward slice* contains parts of the program that are affected by the slice criteria. In other words, backward slice is calculated by traversing the control/data flow paths in the program in reverse, while the forward slice is calculated by following the control/data flow paths in the program as they naturally occur.

2.1 Program Dependences

Despite the variations, all forms and types of slicing rely on a common set of definitions of program dependences that are based on the static structure of the program. Basically, a program slice can be viewed as the transitive closure of the dependence relation starting from the given slice criteria.

There are two basic forms of program dependences.

- A program point p_t *(dependent)* is *control dependent* on a program point p_e *(dependee)* if p_e can decide if the control can reach p_t during execution.
- A program point p_t *(dependent)* is *data dependent* on a program point p_e *(dependee)* if there exists a control flow path from p_e to p_t along which v is not defined at intermediate program points.

In function `bar()` in Figure 1, line 4 will be executed if `k==0` at line 3 evaluates to `true`. Similarly, line 6 will be executed if `k==0` at line 3 evaluates to `false`. Hence, the line 3 decides if either line 4 or line 6 will be executed, hence, lines 4 and 6 are *control dependent* on line 3.

In the same function, the value of the expression `v` at line 7 is determined by the definition of `v` at lines 4 and 6. Hence, the expression `v` at line 7 is *data dependent* on the lines 4 and 6.

[4]Cited from the paper titled "The Cathedral and the Bazaar" by Eric S. Raymond and available at http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/.

```
1  int bar(int k) {
2    int v;
3    if (k == 0)
4      v = 1;
5    else
6      v = 2;
7    return v;
8  }
```

```
1  int foo(int k) {
2    Pointer v, u;
3    v = new Pointer();
4    u = v;
5    if (k == 0)
6      v.o = 1;
7    else
8      v.o = 2;
9    u.o = 4;
10   return v.o;
11 }
```

Figure 1: Java examples to illustrate data and control dependence. `Pointer` is a class with an public integer field named `o`.

The initial forms of data and control dependences were introduced by Weiser [32]. Subsequently, Podgurski et.al. [23] extended the definition of control dependence to account for delayed execution due to looping and not mere non-execution. These definitions assumed that the programs have only one end node, i.e. node with zero outgoing edges. Contrary to the latter requirement, most programs (particularly the programs using exception) have multiple or zero (reactive programs) end nodes. In our recent effort [26], we identified these shortcomings and proposed alternative definitions along with algorithms to address these shortcomings.

2.2 Aliasing

In the context of languages (such as C, C++, Java) that support pointer/reference variables and heap allocated data, it is common for two variables to point to the same data/memory location, hence, lead to *aliasing*. In such cases, data dependence needs to account for the effect of aliasing. This is illustrated in the function `foo()` in Figure 1. Based merely on the identifiers of the variable `v.o`, we could naively conclude that line 10 is data dependent on lines 8 and 6. However, as `u` is an alias to `v`, only the definition of `v.o` at line 9 via *u.o* is used at line 10. Hence, line 10 is data dependent only on line 9 and not on lines 8 and 6. This issue was first identified and addressed by Horwitz et.al. [11].

2.3 Slicing Concurrent Java Programs

Most modern applications are concurrent, and it is relatively harder to comprehend concurrent applications. The main issue in comprehension is to statically determine the data flow *(interference dependence)* (due to interleavings) and control flow *(ready dependence)* (due to synchronization) between program points arising due to the scheduling choices during program execution. This issue was initially identified in the context of program dependence/slicing by Krinke [13] and Hatcliff et. al. [9]. These efforts identified the dependences required to enable sound concurrent program slicing. Subsequent efforts [16, 21] proposed approaches based on these new dependences to slice concurrent programs.

In the program in Figure 2, the *savings* `Account` object created at line 43 is shared between two threads started at lines 46 and 47. In the context of this `Account` object and threads, depending on the runtime schedule,

- the definition of `Account.amount` at line 8 (line 12) may affect the use of `Account.amount` at line 12 (line 8, line 5), hence, line 8 (line 12) is *interference dependent* on line 12 (line 8, line 5).

```
1  class Account {
2    private int amount;
3
4    public synchronized int withdraw(int a) {
5      while (amount − a < 0) {
6        wait ();
7      }
8      amount = amount − a;
9      return amount;
10   }
11   public synchronized int deposit(int a) {
12     amount = amount + a;
13     notifyAll ();
14     return amount;
15   }
16 }
17
18 class Husband implements Runnable {
19   private Account save;
20
21   public Husband(Account account) {
22     save = account;
23   }
24   public void run() {
25     save.deposit(90);
26   }
27 }
28
29 class Wife implements Runnable {
30   private Account save;
31
32   public Wife(Account account) {
33     save = account;
34   }
35   public void run() {
36     save.withdraw(10);
37     (new Account()).deposit(10);
38   }
39 }
40
41 class Home {
42   public static void main(String[] s) {
43     Account savings = new Account();
44     Runnable husband = new Husband(savings);
45     Runnable wife = new Wife(savings);
46     new Thread(husband).start();
47     new Thread(wife).start();
48   }
49 }
```

Figure 2: Example to illustrate concurrent Java program slicing. We have omitted the exception around `wait` for brevity.

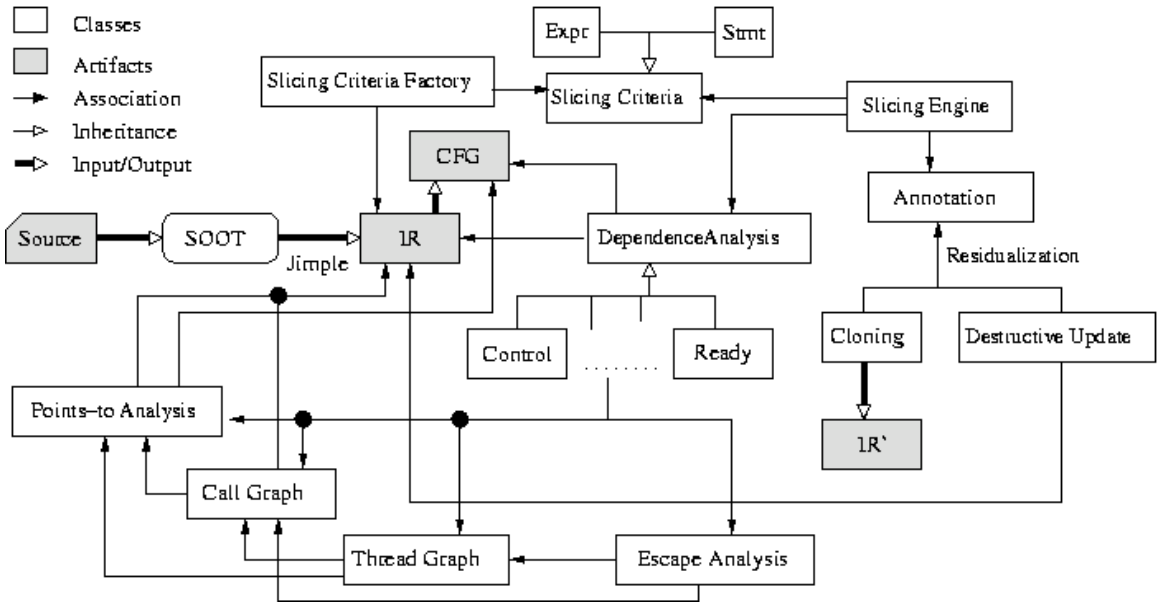

Figure 3: Bird's eye view of classes in Indus Libraries along with the relationships between classes and artifacts.

- the completion of the monitor acquisition at line 4 is dependent on the release of the monitor at line 15 and at line 10, hence, line 4 is *ready dependent* on line 15 and line 10. Similarly, line 11 is *ready dependent* on line 15 and line 10.
- the completion of the wait on the monitor at line 6 is controlled by the notification on the same monitor at line 13, hence, the wait at line 6 *ready dependent* on the notification at line 13.

However, the same interference and ready dependences do not apply to the `Account` object created at line 37 as this object does not escape its thread context, i.e. this object is not shared between threads. Details about this observation and how it can be used to optimize the calculation of interference and ready dependence is presented in our earlier work [27].

3 Indus Java Program Slicing Framework

From our experience, we have found that the properties required of a slice vary across different applications of slicing. For example, the program slice required in model checking based program verification applications such as Bandera [5] needs to be executable. On the other hand, executability is not required in applications such as program comprehension via visualization. Even transformations such as slice residualization (i.e., the generation of a new program that contains only the slice) may need to be handled differently for different applications, i.e., destructively update the original program as opposed to generating a new program. Hence, program slicers need to be modular and flexible (customizable) as opposed to being monolithic and rigid.

Driven by these reasons pertaining to genericity, flexibility, and public consumption, our goal was to implement a general program slicing framework for concurrent Java that could be used to create a customized slicer for diverse applications such as model extraction and program comprehension.

3.1 Salient Features

Figure 3 presents the architecture of the Indus slicer. In the sections below, we describe the primary features of the architecture. Due to space constraints of this article, there are many interesting aspects that we do not discuss, but we focus on those features that are most relevant and/or novel.

3.1.1 Intermediate Representation

In Indus, Java programs are represented in Jimple [31], a typed three-address representation provided by the SOOT.[5] As as result, every module in Indus operates on the Jimple representation of Java programs. Hence, any program analyses that is based on Soot can leverage features from Indus with very little effort. Also, provided there is a translator from source to Jimple and/or bytecode to Jimple (as built into SOOT), each module from Indus can be applied to Java source as well as Java bytecodes. Indus leverages the bytecodes to Jimple translation built into SOOT.

3.1.2 Batteries Included

In general, program slicing depends, either directly or indirectly, on various forms of dependence analysis that capture the relation between various program points of the program with respect to certain aspects such as data flow, control flow, etc. Thus, when we provide Indus to developers interested in building slicing, analysis, or transformations on top of Indus, we aim to provide a framework where "batteries are included", i.e., all capabilities required to bring the framework to bear on interesting and realistic code bases are already included in the framework. The most common forms of dependences are intra-procedural aliasing-free data dependence, aliasing-based data dependence [25], control dependence [26], (inter-thread data) interference [13] dependence, and (inter-thread synchronization) ready dependence [9]. These analyses along with the slicing framework depend on more basic analyses such as escape analysis [27], monitor analysis, safe-lock analysis [9]. These analyses in turn depend on low-level analyses such as object-flow analysis [24], call graph analysis [2], and thread graph [27].

Instead of requiring the end-users to develop these analyses from scratch or procuring their implementation from other libraries/projects, Indus provides an implementation (more than one in many cases) of every analysis mentioned above along with other analyses (Figure 3). As Indus libraries are self-contained with respect to analysis, it makes it easier to experiment with program slicing as well as other individual analyses and transformations provided by these libraries.

3.1.3 Loose Coupling, Modularity, and Customizability

Every analysis in Indus has two parts — *an interface and an implementation.* If analysis X requires analysis Y then every implementation of analysis X is only coupled with the interface to analysis Y and not to any specific implementation of analysis Y. Hence, each analysis is *modular*, i.e. an implementation of analysis X can be used independently without relying on the implementations of the analysis Y provided by Indus or a third party. As a consequence, it is easy to combine various implementations of analyses to evaluate their combined benefits — a common situation in the application of program analysis. In Indus, this feature is extensively used withing the slicing framework and by other analyses to vary the level of accuracy of the result. For example, the slicing algorithm embedded in the `SlicingEngine` class requires a collection of implementations of `IDependencyAnalysis` interface. The sort of dependences provided by these implementations is identified by the associated enumeration value of type `DependenceSort`[6].

Based on loose coupling and modularity, it is easy to assemble a program analysis pipeline or generate a customized slicer using the components provided by Indus. This is illustrated by the

[5] Soot, a Java OptimizationFramework, is available at http://www.sable.mcgill.ca/soot/ library.

[6] This enumeration is available in the upcoming version, and it is currently represented as class constants.

sample command-line applications provided in the Indus distribution, the customized version of the Indus stock slicer used in Bandera, and the use of the analyses provided by Indus in Kaveri.

3.2 Advanced Features

3.2.1 Non-SDG based Dependence/Slicing

Based on Ottenstein and Ottenstein's result [22], most program dependence and program slicing related efforts are based on *program/system dependence graphs (PDG/SDG)* — program points are represented as nodes and the dependence between program points are represented as directed edges between nodes. Although a PDG accurately captures the dependence relations in an intra-procedural setting, they are altered in an inter-procedural setting [12] to capture dependence arising due to data flow across procedural boundaries, e.g. between formal parameters and arguments and various instances of global variables. Recent efforts pertaining to unconditional jumps [18] and interference dependence further extend SDG. These extensions enable the formulation of slicing as a simple graph reachability problem. However, the downside of these extensions is that non-slicing application will need to extract the required dependences from the dependence graph, hence, contributing to their complexity and cost.

In Indus, instead of maintaining SDGs along with the extension, the logic to handle non-dependence aspects (e.g. argument to parameter binding at call sites) is embedded in the slicing algorithm while the dependences are maintained in their native form (e.g. mappings from variable definition line to variable use line). This approach has following advantages:

- the slicing algorithm can be fine tuned independent of the representation of the dependence information,
- the dependence information can be easily accessed by non-slicing applications without any increase in complexity or cost,
- the cost of constructing and maintaining the dependence graph is eliminated, and
- the maintenance of the slicing algorithm and the dependence analyses is easier due to decreased coupling and increased cohesion.

3.2.2 Program Slicing is Program Analysis

Pragmatically, program slicing is often viewed as a transformation that deletes parts of the program P to yield a slice S. However, we have observed that the end goal for which slicing is being applied has a strong influence on whether or not one holds to this view. For example, when program slicing is used for program understanding, testing, and/or debugging, the slice is a mere projection of P that can be captured by annotating P. On the other hand, when program slicing is used to generate executable slices as in Bandera [6], the slice is another program S that is a syntactically correct projection of P.

Based on the above observation, in Indus we consider program slicing as a program analysis — program slicing only calculates the program points that belong to a slice (as in the case of program comprehension). Operations such as residualization of the slice as another program (via destructively updating the program or via cloning the slice) and transforming the slice to be executable are considered as post-slicing transformations. This approach simplifies the slicing algorithm and enables it to be used in various application by merely varying the post-slicing transformations.

3.2.3 Calling Context Sensitive Slicing

Consider the example in in Figure 4, suppose the expression `v` in line 5 is provided as the slice criteria to calculate the backward slice of the program. For sake of simplicity, all of line 5 will be included in the slice. As the value of `v` depends on the definition in line 3, all of line 3 needs to be

```
1  class CCSS {
2    public static void main(String[] s) {
3      int v = foo();
4      int u = bar();
5      System.out.println(Integer.toString(v));
6      int k = foo();
7    }
8
9    static int foo() {
10     return bar();
11   }
12
13   static int bar() {
14     Long l = new Long(System.currentTimeMillis());
15     return l.intValue();
16   }
17 }
```

Figure 4: Example to illustrate calling context sensitive.

included in the slice. At this point the slicing algorithm needs to decide which parts of `foo()` needs to be included in the slice. Hence, it descends into `foo()` and decides to include the invocation of `bar()` at line 10. Similarly, upon descending into `bar()` and including line 15, the slicing algorithm has two options to exit `bar()`:

- process every calling contexts leading up to `bar()`, i.e. resume processing at the call sites at line 4 and at line 10.
- process only *realizable calling contexts*, i.e. resume processing at the call site at line 10.

The former option is cheap but inaccurate as it calculates the slice based on a semantically unrealizable control flow path (`main()->foo()->bar()->main()`). The latter option requires the slicing algorithm to record the call chain to track realizable paths in order to calculate an accurate slice.

Slicing algorithms that support the first option are said to be *calling context insensitive*. The algorithms that support the second option by keeping track of calling contexts while descending into call sites and tracing back the recorded calling contexts are said to be *calling context sensitive*. Horwitz et.al. [12] proposed the first SDG based calling context sensitive algorithm for sequential programs.

Indus supports calling context insensitive and calling context sensitive slicing of sequential programs. Further, despite calling context sensitive slicing of concurrent programs being exponential, Indus provides a restrictive form of calling context sensitive slicing of concurrent programs that is both efficient and relatively accurate. For more details, please refer to Ranganath's dissertation [25].

3.2.4 Context-restricted Slicing

In the program in Figure 4, suppose an user is interested in determining the parts of the program that are affected by the invocation at line 14 as a result of its execution due to the invocation at line 3. In such a case, specifying line 15 as a criteria would result in an inaccurate slice containing lines 3, 4 and 6. The reason being that the slicing algorithm will consider every calling context leading out from (into) the method containing the slice criteria (line 14) and include every method occurring in these calling contexts in the slice.

We addressed this shortcoming in Indus by enriching the slice criteria with a calling context that can be specified by the user. For example, in the scenario described above, the sequence

`[main():*, foo():3, bar():10]` will be supplied as the calling context with the criteria. This restricts the slicing algorithm ascent into invocation sites (e.g. `bar():10`) to only those mentioned in the sequence from right to left. By adopting this approach, the inter-procedural slicing algorithm can trivially leverage auxiliary contextual information to provide accurate call chain specific slices.

This form of slicing is referred to as *context-restricted slicing* and it was introduced by Binkley [3] (as *calling context slicing*) and later refined by Krinke [17]. Unlike tailoring the slicing algorithm as in Binkley's and Krinke's approaches, our approach leverages calling context sensitive and non-graph based slicing along with support for calling context enriched slice criterion to realize context-restriction.

This form of slicing is useful for debugging an application based on an exception stack trace, i.e. an user would like to calculate the slice that affects only the parts of the program occurring on an exception stack trace. In security-related applications, this feature is useful to accurately identify the parts of the programs that affect the data/control flow path between two modules, hence, easily identify any insecure parts of the program.

3.2.5 Scoped Slicing

In some applications/situations, it is known that some parts of the program may not contribute interestingly to the slice, e.g. the classes corresponding to the AST nodes of a compiler infrastructure, or the user may want to perform incremental slices to expedite slicing of large programs. In such cases, analyses and slicing can be made more efficient by not considering such parts of the program. For this purpose, the user can limit the analyses and slicing in Indus. This *scoping* feature of Indus can be used to specify a scope of the analysis, i.e. only parts of the system that are within the scope are analyzed. The scope is usually defined in terms of classes, methods, and fields, e.g. $appl.*|Appl$ will limit the analysis to consider only parts of the program belonging to the class `Appl` or the classes with fully qualified name beginning with `appl.`, i.e. belonging to package `appl`).

The slicing framework in Indus supports scoping, and we refer to this form of slicing as *scoped analysis*.[7] Similarly, we refer to any analysis that is *scope-sensitive* as a *scoped analysis*. Currently[8], every analysis implementation in Indus can be executed in a scope sensitive manner by leveraging the general scoping framework.

Scoping	Class	Method	Fields	Size (KB)	Time (sec)	Mem (MB)
none	1198	6136	1973	972	117/539	64/607
auto-slicing	688	2881	879	485	67/462	31/568
auto-analysis	478	1902	597	334	33/142	21/164
manual	436	1856	590	318	30/129	20/150

Table 1: Data from generating sequential executable slices of JReversePro. The data was collected on a Linux box (2GHz/2GB) running Java 1.5.0 with maximum heap space of 1700MB. In the data of the form X/Y, X represents the data for slicing only and Y represents the (overall) data for slicing and the dependent analyses (not transformations). The classes, methods, fields, and bytecode count is inclusive of code pertaining to the application and the required libraries.

Scoping is particularly useful for removing parts of the runtime library that are used during boot strapping and/or for user interface, hence, contribute unnecessarily to slices pertaining to core functionality of the applications. Such a situation occurs in JReversePro[9], a Java Decompiler/ Disassembler consisting of 90 Java classes that amount to 264KB of bytecodes. JReversePro can be used in two modes: command line mode and GUI mode, and the usage mode does not affect its core functionality. However, this separation of functionalities (interface v/s core) cannot be identified in

[7] This notion of restrictive slicing (not analysis) was introduced as *Barrier Slicing* by Krinke [15].
[8] In the latest development version of the libraries.
[9] http://jrevpro.sourceforge.net

the web of program dependences as they occur in the program. Hence, to illustrate the benefits of scoping, we performed an experiment of sequential slicing on JReversePro in four different settings: 1) *none*, 2) *manual* by eliminating code that connects the GUI part to the core functionality, 3) *auto-slicing* by leveraging the scoping support in Indus only during slicing, and 4) *auto analysis* by leveraging the scoping support in all analyses. In each of these settings, we selected an arbitrary statement that contributes to the core functionality as the slice criterion.

The data from the experiment (as given in Table 1) indicates that both automatic scoped analysis and automatic scoped slicing provide interesting improvements in terms of execution time and memory footprint. Hence, scoping can be used as to scale slicing and other analysis to be applicable to large real-world software.

Interestingly, automatic scoped slicing performs 30% worse than automatic scoped analysis both in terms of time, memory, and slice size. As scoped slicing does not depend on the information pertaining to program points outside the scope, this data indicates that it would be efficient to perform scoped analysis instead of merely performing scoped slicing.

Further, automatic scoped analysis provides slices that are comparable in terms of time, memory, and slice size to those generated via manual scoping. Hence, in situations requiring scoping, automatic scoping can be used for all analysis without much loss of accuracy.

Scoped slicing is also useful in checking for data confinement in the realm of security. For example, one could define a secure scope and calculate a forward slice w.r.t. this scope; if the slice contains any part of the boundary of the scope then it indicates a breach of security.

Although scoped slicing is usually fast and cheap but it may be unsound. In particular, when two program points within the scope may be related by a chain of dependences that involves program points outside the scope. However, such cases are trivially exposed by the inclusion of program points belonging to scope boundary, hence, the user can amend the scope appropriately and incrementally obtain accurate and sound slices.

3.2.6 Concurrent Java Program Slicing

Indus provides various implementations of all dependences mentioned in Section 2 to perform both sequential and concurrent slicing of Java programs.

The implementations of interference and ready dependence analysis in Indus rely on a novel, aggressive, and relatively accurate approach [27] to calculate interference and ready dependence based on object flow analysis, and an equivalence class based escape analysis [19, 29]. The approach leverages the escape analysis to rule out cases where the receiver object involved in the dependence relation is not visible outside the creator thread and the object flow analysis to rule out cases where the receiver variables participating in the dependence can never be aliases. Hence, based on this optimization, a slice of the program (in Figure 2) that contains every monitor related operations will contain every invocation to `Account.withdraw()` and `Account.deposit()` except the invocation to `Account.deposit()` at line 37 — the receiver object in this invocation is does not escape the scope of the creating thread.

3.2.7 Complete Slicing, Chopping, and Control Slicing

Besides the support to generate forward and backward slices, the Indus slicing framework, also supports the generation of *complete slices* — a slice that contains parts of the program that affect and are affected by the slice criteria and *every program point in the slice*. The latter aspect distinguishes a complete slice from the mere union of the backward and forward slices w.r.t. the slice criteria. Hence, a complete slice can be perceived as a *software carving* technique that extracts a coherent and syntactically and semantically complete projection of the software w.r.t. the slice criteria.

A common application of slicing/dependences is to determine the programs points that propagate the effect from one given program point to another program point. This set of program points is referred to as the *chop* of the program w.r.t.to the two given program points. As the slicing the algorithm in Indus merely identifies the slice by annotating the program, a chop can be trivially

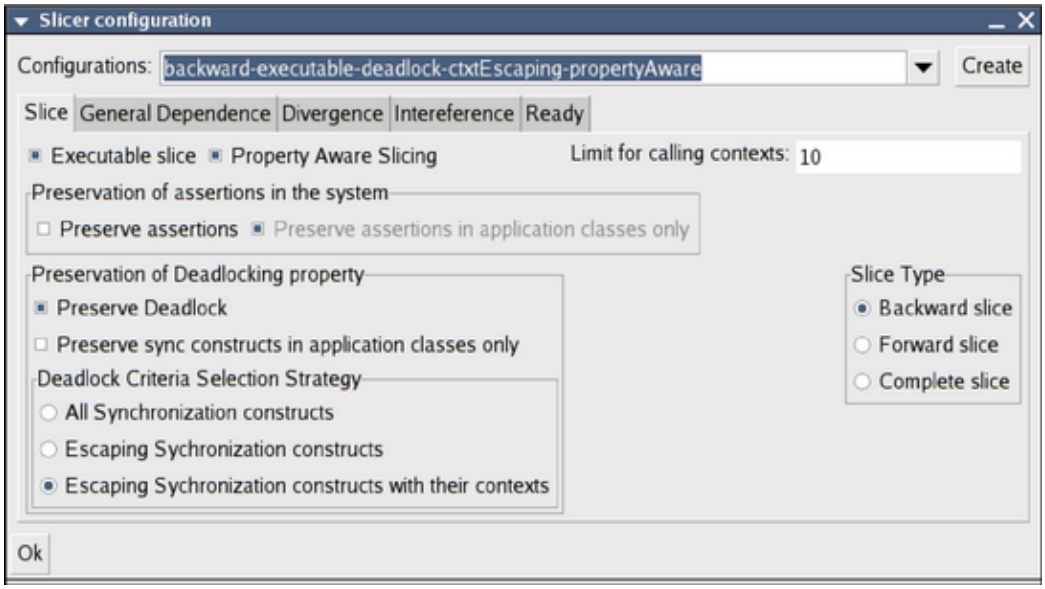

Figure 5: The pre-packaged graphical user interface to configure the Indus slicer tool.

calculated by union-ing the forward slice w.r.t. the first program point with the backward slice w.r.t. the second program point and identifying the parts of the program that contain both the forward and backward annotations.

In many applications of program verification, properties are expressed as "if/when the control reaches this program point". For such applications, the slice should contain parts of the program that are required to ensure that control flow will reach the program point but not to reproduce the behavior of the program when the program point is executed. Such slices are referred to as *control slices*. The Indus program slicing framework supports the generation of control slices by merely enriching the slice criteria with the information that indicates if the slice should preserve the behavior of the program before or after the execution of the slice criteria.

3.3 Applying the Framework in Program Verification

As mentioned earlier, program slicing is used in Bandera as a model reduction technique to optimize program verification via model checking. The approach is to identify the program points at which the property will be checked or the program points that are required to preserve a property. These program points are provided as the slice criteria to the slicer to generate an executable backward slice. The slice is residualized by deleting parts of the original program that are not in the slice and by adding necessary parts required to ensure executability. The model generated from the residual sliced program is verified to ensure the program satisfies the property. The details of this approach is documented in earlier efforts [10, 26].

Due to our interests in using Indus for model checking reductions and the observation that deadlock and assertion violations are commonly checked faults, the Indus slicing tool contains pre-packaged features to generate slice criteria required to preserve the deadlocking behavior of the program and/or the assertions in the program with varying levels of precision. The tool also supports a rich configuration that could be used to vary the dependence analyses that need to be used to construct the slice, to control the accuracy of the used dependence analyses, and to select the type of slice. A pre-packaged graphical user interface (shown in Figure 5 can be used to configure the tool as described above.

This tool has been adapted to fit into the tool framework used in Bandera. Similarly, other

analyses from Indus have been exposed as tools in the Bandera tool framework. We have successfully applied these tools to various programs consisting of ∼144 KB (∼10 KLOC) of application bytecodes (∼1197 KB of application + library bytecodes).

Recently, we conducted experiments to empirically evaluate the benefits of slicing as a model reduction technique in the context of program verification [7]. Most of the programs (*alarm clock, bounded buffer, disk scheduler, pipeline, sleeping barbers*) considered in the experiments were common examples used to illustrate concurrency in academia. Of the uncommon programs, *RAX* was the distillation of a bug in the NASA remote experiment platform [4], *Moldyn* and *Ray Tracer* were examples from JavaGrande benchmarks[10], and *Siena-Server* and *Siena-Client-Server* were programs built on top of *Siena*, an Internet-scale publish-subscribe infrastructure.[11] For this publication, we generated calling-context sensitive concurrent backward slices of the above mentioned programs. The deadlocking behavior of the programs were preserved by selecting the synchronization commands in the programs as the slicing criteria. The data from these experiments are summarized in Table 2.

Application	**Classes**	**Fields**	**Methods**	**Statements**	**Time** (seconds)	**Memory** (MB)	**App Size** (bytecodes)
Alarm clock	349/24	1292/11	3434/42	40426/252	7	8	5922
Bounded Buffer	351/25	1297/14	3432/39	40348/191	6	1	5914
Disk Scheduler	348/23	1298/12	3428/38	40500/282	7	8	5697
Pipeline	347/25	1283/5	3421/29	40225/94	0	44	2586
Readers & Writers	348/22	1290/12	3438/43	40476/267	15	39	5919
Sleeping Barbers	347/21	1290/11	3421/29	40322/184	9	41	3139
RAX	347/21	1286/8	3421/29	40229/98	5	44	2501
Moldyn	355/51	1388/99	3478/139	42266/3169	40	52	30598
Ray Tracer	363/70	1359/100	3529/194	42128/3035	34	39	44418
Siena-Server	489/160	1689/259	4929/783	57701/10664	139	111	215366
Siena-Client-Server	489/197	1691/310	4929/927	57722/13677	149	143	210266

Table 2: Summary of data pertaining to slicing concurrent programs in the context of program verification. The data was collected on a 1.4GHz Linux Box running Java 1.5.0_06 with maximum heap space of 512MB. The data in the form X/Y represents the count of the corresponding entity (classes, methods, fields, (Jimple) statements, and bytecodes) before slicing (X) and after slicing (Y). The memory data does include the storage (which was at most 30MB) required by Jimple. The data is inclusive of code pertaining to the application and required libraries.

The data in Table 2 suggests that the Indus slicing tool (framework) yields a reduction of at least 60%, 82%, 82%, and 77% in terms of number of classes, fields, methods, and statements, respectively, in the program (application + libraries) used in our recent experiments; hence, it indicates a high level of accuracy of the embedded slicing algorithm. Similarly, the data also suggests the slicing tool (framework) is efficient both in terms of time and memory as, in the experiments, it could analyze an infrastructure-based application such as Siena-Client-Server within two minutes while requiring less than 256MB of memory on a desktop Linux box.

4 Kaveri: A Program Slicing plugin for Eclipse

To the best of our knowledge, to this date there has been only one feature rich slicing tool with a sophisticated UI — the commercial tool named CodeSurfer[12] that is targeted towards C programs. Hence, as part of developing the program slicing framework, we also decided to develop an user

[10] Java Grande Benchmarking Project is available at http://www.epcc.ed.ac.uk/javagrande/.

[11] For more details about these programs, please refer to [7].

[12] CodeSurfer, an analysis and inspection tool for C, is available at http://www.grammatech.com/products/codesurfer/.

interface to an instance of our framework. Given that Eclipse[13] was a open platform that is well accepted by the Java community as a development platform, we choose to develop the slicing UI as a plugin to Eclipse to leverage the extensive Java development support available in Eclipse and to appeal to the Java community.

Kaveri is a plugin that contributes program slicing via an intuitive user interface as a feature to Eclipse platform. **The plugin was implemented as a graduate project by Ganeshan Jayaraman under our supervision in our group.**

4.1 Architectural Overview

Every aspect of presenting the information via the UI is handled by Kaveri. Trying to be a true source level analysis plugin, Kaveri handles the mapping of between Java and Jimple representations of the programs. This is done by compiling the Java source via the Eclipse Java compiler, generating the Jimple representation from the generated bytecodes, and then constructing a mapping between the Java source and the Jimple representation. This Jimple representation is fed to the Indus program slicing tool (described in Section 3.3) to perform slicing.

During slicing, Kaveri leverages the "program slicing is an analysis" feature of Indus to represent the generated slice by annotating/tagging parts of the Jimple representation that belongs to the slice. By combining these annotations with the mapping between Java and Jimple, Kaveri displays the slice in the Eclipse Java Editor view (as shown in the top part of Figure 6). Beyond this, it provides five views that present various dependences existing in the program, hence, aiding the understanding of the generated slice or the program.

4.2 Features

The various features and views[14] provided by Kaveri are illustratively described in the following subsections along with a brief outline of how they can be leveraged within Eclipse.

4.2.1 Slice Java Programs

As advertised, Kaveri enables the user to perform slicing and view the results at the Java source level. The user can select an arbitrary line in the Java Editor and click on either the backward slice or the forward slice button on the toolbar to perform slicing. The result of slicing is displayed by highlighting the lines of source file included in the slice in green within the Eclipse Java Editor (as illustrated in the top part of Figure 6). Also, the name of any file that contains some part of the slice is decorated in the package explorer[15].

Usually, there will be a few Java statements such that not all parts of these statements are in the slice, e.g., some subexpressions of a statement are omitted from the slice. Such instances are distinguished by highlighting the line in yellow (indicating that only parts of the statement are in the slice) as opposed to green (indicating that all parts of the statement are in the slice). Further, more information about such instances can be retrieved via the Jimple View described next.

4.2.2 Java to Jimple Mapping

The mapping between Java to Jimple generated by Kaveri can be viewed in the *Jimple View*. As shown in Figure 6, this view displays the Jimple statements that collectively represent the Java statement that occurs on the current line in the Java Editor. This view primarily serves four purposes.

[13] Eclipse, an open extensible IDE and tool platform written in Java, is available at http://www.eclipse.org.

[14] The term used in Eclipse to refer to a non-context sensitive display.

[15] A Java specific view in Eclipse that presents project artifacts in hierarchical terms of packages, source files, classes, fields, and methods.

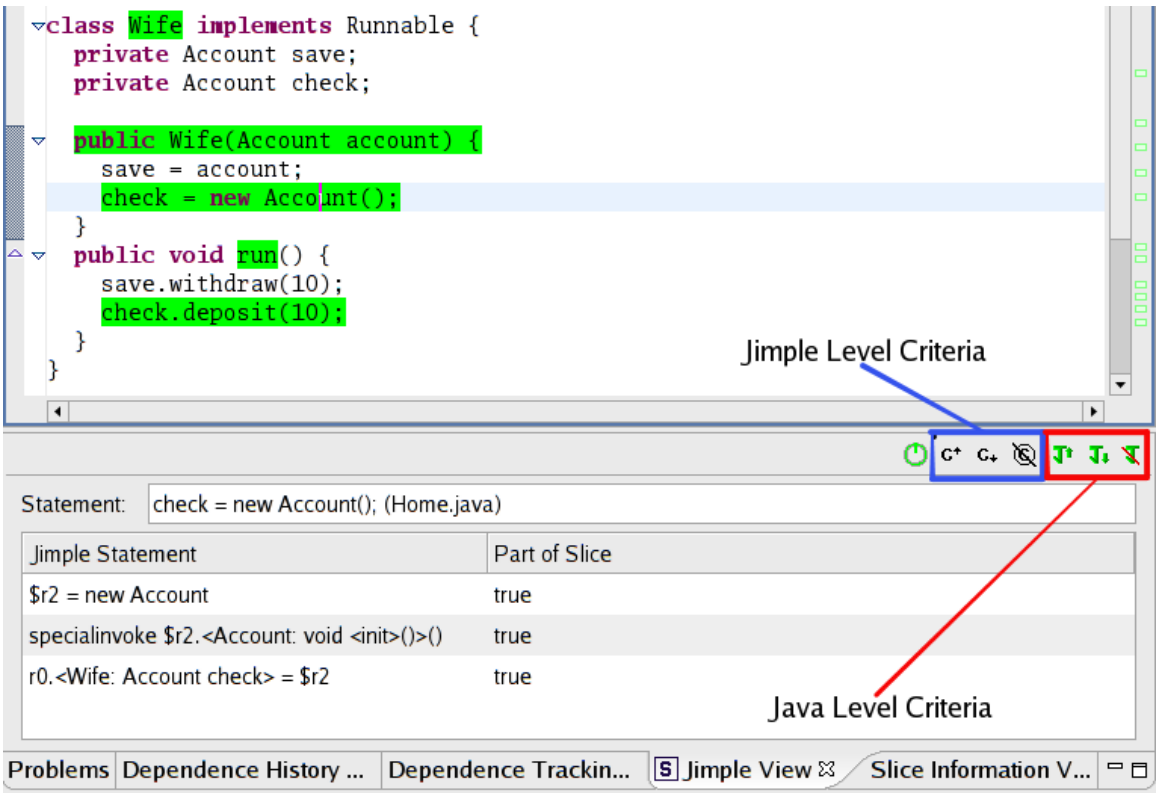

Figure 6: A program slice is displayed in the Eclipse Java Editor and the mapping between Java and Jimple representation is displayed the Jimple View.

1. It provides a simple approach to understand how a Java statement is represented as a set of Jimple statements, hence, serving as a Jimple learning aid.

2. It displays the slice at the level of Jimple statements; this information is indicated by displaying *true* or *false* in the "Part of Slice" column against each Jimple statement occurring in the "Jimple Statement" column. This feature is useful in situations when not all parts of a Java statement are included in the slice. In such cases, by clicking on the line highlighted in yellow in the Java editor, the user can view the parts of the statement that are (not) part of the slice.

3. It enables the user to select/unselect slice criteria at the granularity of Java statements. The user can position the cursor at a particular line in Java editor and use the buttons marked as "Java Level Criteria" to select and unselect slice criteria in terms of Java statements (lines). Further, it also provides the fine grained control to select slice criteria to generated control slices (described in Section 3.2.7).

4. An advanced user can select and unselect slice criteria at the granularity of Jimple statements (bytecodes) by selecting the appropriate Jimple statement in view and clicking the buttons marked as "Jimple Level Criteria". This feature is suitable for users who are interested in slicing Java bytecodes.

Kaveri generates the mapping on demand when either slicing is performed or when the user turns on the Jimple View (by clicking on the "power-on" button in the toolbar). As the mapping generation can be a costly operation, the feature to control mapping generation enables the user to use Kaveri in a performance non-intrusive manner.

4.2.3 Dependence Tracking

Beyond slicing, users may use Kaveri to view the dependences between various program points in a Java program. This is possible via the *Dependence Tracking View*. The program dependences for the Java statement occurring on the current line in the Java Editor is displayed in this view as show in Figure 7. The dependence information is hierarchically displayed at the granularity of both Java and Jimple statements and in both directions (forward and backward). The user can traverse the dependences by clicking on the dependents/dependees displayed in the "Dependence" column of this view.

For example, in Figure 7, the dependence tracking view indicates that the statement `amount = amount + a;` in `Account.deposit()` is data dependent on the statement `amount = amount - a;` in `Account.withdraw()` as well on the assignments to `this` and `a` in `Account.deposit()`. The latter is a mere identifier based intra-procedural data dependence whereas the former is a reference-based inter-procedural data dependence. Similarly, the view indicates that the same statement is interference dependent on the assignment `amount = amount - a;` in `Account.withdraw()`. Although the interference dependence is valid, the reference-based data dependence is invalid; this is due to the current accuracy limitation of data dependence analysis.

During debugging, it is a common task to traverse through the source code based on the structure of the code. It is also common to backtrack during such traversal and continue forward towards new (or old) program points. For this purpose, tools such as *ctags*[16] and *etags*[17] generate tagging information to expedite such traversal and editors such as Vim[18] and Emacs[19] support such traversals by leveraging tagging information. Even Eclipse has built-in non-Java specific support for annotation traversal.

[16] Exuberant Ctags: A multilanguage implementation of Ctags, is available at http://ctags.sourceforge.net/.
[17] This is a tagging software available with Emacs.
[18] Vim, an extensible editor, is available at http://www.vim.org/.
[19] Emacs, an extensible editor, is available at http://www.gnu.org/software/emacs/.

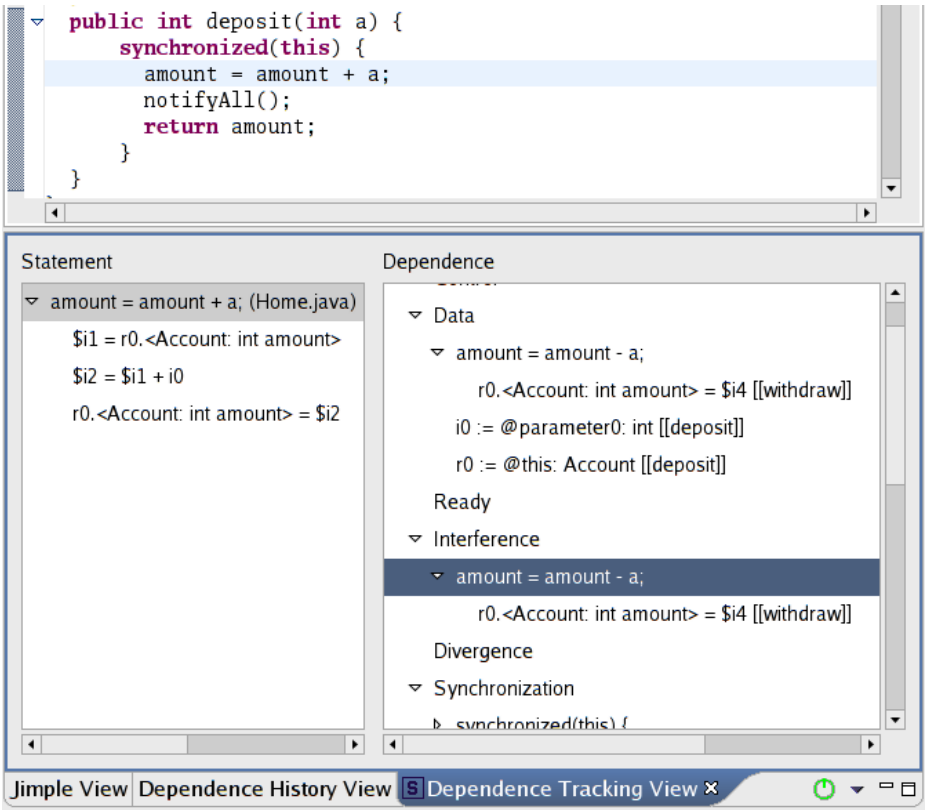

Figure 7: The dependence information for the highlighted statement in the editor is displayed in the Dependence Tracking View by Kaveri.

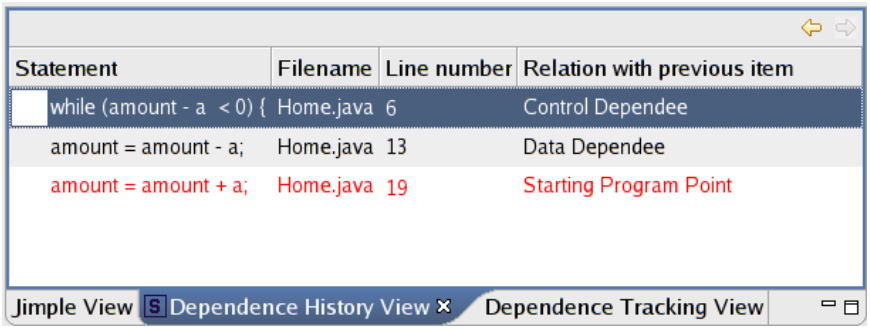

Figure 8: The history of the dependences "chased" by the user is displayed in the Dependence History View by Kaveri.

Using the dependence tracking view, the user can perform a traversal based on a chain of program dependences. Inspired by the above mentioned support for various forms of code traversal, we provided a similar support to traverse dependences. As part of this support, Kaveri maintains a stack of dependences traversed by the user and this stack can be accessed via the *Dependence History View* as shown in Figure 8. At any point during the traversal, the user can jump to a point in the chain and continue traversing.

In the figure Figure 8, the user has started at line 19 in Home.java and traversed a data dependence to line 13 followed by a control dependence to line 6 in the same file.

4.2.4 Beyond Basics

Apart from the basic (select-the-line-and-click-button) mode of slicing, Kaveri provides the following features. These features can be leveraged via the dialog (shown in Figure 9) dedicated to perform slicing in the advanced mode.

- *Perform fine grained slicing*: The user can select Jimple level slice criteria via the Jimple View and then use advanced slicing dialog to perform Jimple statement (bytecodes) level slicing.
- *Perform scoped slicing*: The user can defines various scopes via regular expressions on the fully qualified names of packages, classes, and methods of the software. The regular expressions can be combined with inheritance relation as well. These scopes are managed in a workspace specific manner by Kaveri. As for their usage, the user may choose to enable any of these scopes while performing advanced slicing via the dialog shown in Figure 9.
- *Control the classes used for slicing*: In many situations, it may be required to understand a Java program in the context of a particular version of a library. In such situations, the user can switch to advanced mode of slicing and plug in the required library into the "Slice Class Path" before performing slicing.
- *Perform calling context driven slicing*: Kaveri provides support to choose calling contexts via the Eclipse *call hierarchy view*[20]. In the advanced mode of slicing, the user can select the required calling contexts. Before slicing, Kaveri will combine the calling contexts appropriately with the slicing criteria to achieve the effect of calling context driven slicing.

[20] An view that hierarchically describes the call hierarchy for the selected method.

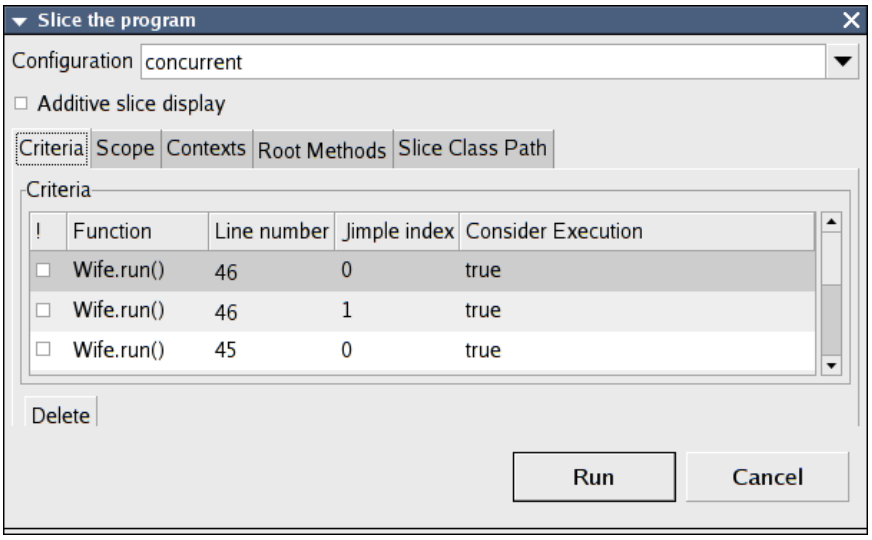

Figure 9: The Kaveri dialog that enables users to select criteria, scoping, and starting calling contexts before slicing.

5 Related Work

There have been numerous efforts pertaining to static slicing of sequential programs over the last two decades, and we refer the reader to Frank Tip's survey [30] to learn more about these efforts. In our effort, we have realized various contributions from these efforts along with numerous new advances in Indus, the first publicly available Java program slicing framework.

Our earlier work provides a more formal perspective on several aspects of Indus including novel forms of control dependence using for languages like Java [26], foundations of dependences for concurrent Java [9], and using escape analysis to reduce spurious inter-thread dependences [27].

In Indus, we have realized context-restrictive (calling-context) slicing and barrier slicing. Further, we have extended the concept of barrier/scope to various program analyses to further improve the scalability of scoped slicing without much loss of accuracy.

In the context of static slicing of concurrent programs, almost all previous efforts [16, 20] have been based on PDGs. In contrast, we have proposed the first non-SDG based concurrent slicing algorithm [25].

Bandera [5] was the first effort to apply slicing to Java programs in the context of program verification. Our effort supplements this effort with a robust and accurate Java slicing implementation that can be readily used for the purpose of model reduction in model checking Java programs [7].

To the best of our knowledge, the Java slicer from Univeristy of Passau [8] is the only other program slicer that can handle almost all features of Java. As this slicer is not publicly available, we are unable to compare the catered features of the slicers. Currently, Kaveri is the first and the only publicly available graphical environment for Java program slicing.

6 Conclusion

En route to addressing a very local and specific problem in the realm of program verification, we have developed the Indus Java program slicing framework. This framework is the first and only publicly available general purpose Java slicing implementation that is robust, flexible, and capable of handling almost all features of Java programming language. During this effort, we have found that a non-graph based approach to slicing is feasible and correct, contributed optimizations for interference and

ready dependence analysis, proposed alternative definitions for control dependences, and developed new and useful forms of slicing — complete slicing, context-restricted slicing, and scoped slicing. As a result of this effort, we have successfully applied program slicing as a model reduction technique in the context of program verification.

In addition, we have also developed Kaveri, the first and only GUI-based and publicly available Java program slicing tool, with the set of features that are targeted to enable program comprehension via program slicing and program dependences. This tool can serve as a good starting point to introduce program slicing as a program comprehension technique in academia.

Please refer to http://indus.projects.cis.ksu.edu for various publications, software artifacts, and documentation pertaining to Indus and Kaveri.

References

[1] Agarwal, H., Horgan, J.H.: Dynamic program slicing. In: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation, pp. 246–256 (1990)

[2] Bacon, D.F., Sweeney, P.F.: Fast Static Analysis of C++ Virtual Function Calls. In: OOPSLA, pp. 324–341 (1996)

[3] Binkley, D.: Semantics Guided Regression Test Cost Reduction. IEEE Transactions on Software Engineering **23**(8), 489–516 (1997)

[4] Brat, G., Havelund, K., Park, S., Visser, W.: Java PathFinder – A second generation of a Java model-checker. In: Proceedings of the Workshop on Advances in Verification (2000)

[5] 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), pp. 439–448 (2000)

[6] Dwyer, M.B., Hatcliff, J.: Slicing Software for Model Construction. In: Proceedings of Partial Evaluation and Semantic-Based Program Manipulation (PEPM'99), pp. 105–118 (1999)

[7] Dwyer, M.B., Hatcliff, J., Hoosier, M., Ranganath, V., Robby, Wallentine, T.: Evaluating the Effectiveness of Slicing for Model Reduction of Concurrent Object-Oriented Programs. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2006) (2006)

[8] Hammer, C., Snelting, G.: An Improved Slicer for Java. In: Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE'04), pp. 17–22 (2004)

[9] 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, pp. 1–18 (1999)

[10] Hatcliff, J., Dwyer, M.B., Zheng, H.: Slicing Software for Model Construction. Journal of Higher-order and Symbolic Computation **13**(4), 315–353 (2000). A special issue containing selected papers from the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation.

[11] Horwitz, S., Pfeiffer, P., Reps, T.W.: Dependence Analysis for Pointer Variables. In: Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI'89), pp. 28–40 (1989)

[12] Horwitz, S., Reps, T., Binkley, D.: Interprocedural Slicing Using Dependence Graphs. ACM Transactions on Programming Language and Systems **12**(1), 26–60 (1990)

[13] Krinke, J.: Static Slicing of Threaded Programs. In: Proceedings ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98), pp. 35–42 (1998)

[14] Krinke, J.: Advanced Slicing of Sequntial and Concurrent Programs. Ph.D. thesis, Fakultät für Mathematik und Informatik, Universität Passau (2003)

[15] Krinke, J.: Barrier Slicing and Chopping. In: Proceedings of Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003), pp. 81–87 (2003)

[16] Krinke, J.: Context-Sensitive Slicing of Concurrent Programs. In: Proceedings of ESEC/SIGSOFT FSE'03, pp. 178–187 (2003)

[17] Krinke, J.: Context-Sensitivity Matters, But Context Does Not. In: Proceedings of Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004), pp. 29–35 (2004)

[18] Kumar, S., Horwitz, S.: Better Slicing of Programs with Jumps and Switches. In: Proceedings of Fundamental Approaches of Software Engineering (FASE'02), pp. 96–112 (2002)

[19] Liang, D., Harrold, M.J.: Efficient Computation of Parameterized Pointer Information for Interprocedural Analyses. In: Proceedings 8th International Static Analysis Symposium (SAS 2001), vol. 2126, pp. 279–298 (2001)

[20] Nanda, M.G.: Slicing Concurrent Java Programs: Issues and Solutions. Ph.D. thesis, Indian Institute of Technology, Bombay (2001)

[21] Nanda, M.G., Ramesh, S.: Slicing concurrent programs. In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA'00), pp. 180–190 (2000)

[22] Ottenstein, K., Ottenstein, L.: The Program Dependence Graph in a Software Development Environment. In: Proceedings of ACM SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments (PPDE'84), pp. 177–184 (1984)

[23] Podgurski, A., Clarke, L.: A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Transactions on Software Engineering **16**(8), 965–979 (1990)

[24] Ranganath, V.P.: Object-Flow Analysis for Optimizing Finite-State Models of Java Software. Master's thesis, Department of Computing and Information Science, Kansas State University (2002)

[25] Ranganath, V.P.: Scalable and Accurate Approaches to Program Dependence Analysis, Slicing, and Verification of Concurrent Object Oriented Programs. Ph.D. thesis, Department of Computing and Information Science, Kansas State University (2006). Work In Progress.

[26] Ranganath, V.P., Amtoft, T., Banerjee, A., Dwyer, M.B., Hatcliff, J.: A New Foundation For Control-Dependence and Slicing for Modern Program Structures. In: Programming Languages and Systems, Proceedings of 14th European Symposium on Programming, ESOP 2005 (2005). Extended version is available at http://projects.cis.ksu.edu/docman/?group_id=12

[27] Ranganath, V.P., Hatcliff, J.: Pruning Interference and Ready Dependences for Slicing Concurrent Java Programs. In: Proceedings of Compiler Construction (CC'04), *Lecture Notes in Computer Science*, vol. 2985, pp. 39–56 (2004)

[28] Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: a tool for change impact analysis of java programs. In: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 432–448 (2004)

[29] Ruf, E.: Effective Synchronization Removal for Java. In: Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI'00), pp. 208–218 (2000)

[30] Tip, F.: A survey of program slicing techniques. Journal of programming languages **3**, 121–189 (1995)

[31] Vallée-Rai, R.: SOOT: A Java Bytecode Optimization Framework. Master's thesis, School of Computer Science, McGill University (2000)

[32] Weiser, M.: Program Slicing. IEEE Transactions on Software Engineering **10**(4), 352–357 (1984)