An Overview of the Indus Framework for Analysis and Slicing of Concurrent Java Software (Keynote Talk – Extended Abstract)

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

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++.

This talk presents an overview of 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. We will also briefly overview the Indus scripting framework that allows developers easy access to a variety of information collected by the underlying Indus program analysis framework.

Motivation

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 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 visualizally 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

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

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*, 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.

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++ – the work of Nanda [7] and Krinke [6] being notable efforts for Java.

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, 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 overestimate 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).

- **Dealing with exceptions:** Modern languages like Java and C# rely extensively on 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 libary 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 in a language like Java requires both foundational work along a number of fronts as well as a large-scale tool engineering effort.

Our work – Perspective

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 [1][3]. Model checking is a verification and bug-finding technique

[2] Please refer to Jens Krinke's Dissertation[6] for a brief informative overview of available implementations.

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

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 [4]. 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 [2].

Indus and Kaveri

Drawing from the our experience with Bandera slicer, we have implemented a program slicing library that can handle almost full Java[4]. Indus modules work on Jimple (SOOT [12]) 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.

Analysis Library The program slicing library, directly or indirectly, requires various high level analyses such as escape analysis [11], monitor analysis, safe-lock analysis [3], and analyses to calculate and prune various dependences – intra- and inter-procedural data dependence, control [10] dependence, interference [5] dependence, ready dependence and synchronization dependence [3]. These high level analyses rely on low-level information such as object-flow information [9], call graph, and thread graph [11]. 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 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.

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

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. Kaveri is a plugin that contributes program slicing as a feature to Eclipse [8]. 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 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 *b* and *c*?" is a common question during program comprehension. It can be answered by generating a *chop*, the intersection of the slices based on criteria *b* and *c*. In Kaveri, the user can associate different highlighting schemes to slices based on *b* and *c*, and view both the slices in the editor at the same time to realize a *chop*.

Program comprehension through dependence tracking 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 [13].

References

[1] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting Finite-state Models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 439–448, June 2000.

[2] M. B. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, Robby, and T. Wallentine. 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.

[3] J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng. 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, pages 1–18, Sept 1999.

[4] J. Hatcliff, M. B. Dwyer, and H. Zheng. 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.

[5] J. Krinke. Static Slicing of Threaded Programs. In *Proceedings ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, Montreal, Canada, June 1998. ACM SIGPLAN Notices 33(7).

[6] J. Krinke. *Advanced Slicing of Sequntial and Concurrent Programs*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2003.

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

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

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

[10] V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, and J. Hatcliff. 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*, April 2005. Extended version is available at http://projects.cis.ksu.edu/docman/?group_id=12.

[11] V. P. Ranganath and J. Hatcliff. Pruning Interference and Ready Dependences for Slicing Concurrent Java Programs. In E. Duesterwald, editor, *Proceedings of Compiler Construction (CC'04)*, volume 2985 of *Lecture Notes in Computer Science*, pages 39–56, March 2004.

[12] Sable Group. Soot, a Java Optimization Framework. This software is available at http://www.sable.mcgill.ca/soot/.

[13] SAnToS Laboratory. Indus – a program slicing and analysis framework for java. This software is available at http://bandera.projects.cis.ksu.edu.