

Com Sci 417 Research Report
Program Slicing
Instructed by Dr. Simanta Mitra

Daekyoon Kim
Dept. of Computer Science
Iowa State University

Michael Fong
Dept. of Computer Engineering
Iowa State University

April, 27th, 2007

Contents

1	Lecture 1 - Program Slicing in a Nutshell	3
1.1	Introduction	3
1.1.1	Program Dependency Graph	3
1.1	Concept	5
1.0.1	Static Slicing	6
1.0.1	Dynamic Slicing	8
1.0.1	Amorphous	10
1.1	Application	11
1.1	Summary	12
2	Lecture 2 - Program Slicing Tool	13
2.1	Tool Support	13
2.1	Overview of Indus/Kaveri	15
2.2	Prerequisites	16
2.3	Setting up Indus/Kaveri Environment	16
2.4	Setting up the Criteria	18
2.5	Running Kaveri	20
2.6	Features - Toolbar Buttons	23
2.7	Appendix	25
2.7.1	How to pick criteria	25
2.7.2	Creating Contexts	26
2.7.3	Adding custom root methods	27
2.7.4	Outline of This Report	28

1 Lecture 1 - Program Slicing in a Nutshell

1.1 Introduction

As a software is developed, it is common to have a software functionality that previously worked as desired that is no longer working after changes has been made to the program. The level of difficulty on debugging increases as software application keeps growing in size and complexity. Therefore, as time goes by, a software gets harder to understand and maintain. Especially, locating the new bugs could be the hardest task to Software Testers. Once the bug is found, it can be easy to fix. One of the reason that a bug can be hard to find is that there can be a lot of lines of codes in that part of program, but not all of them will have an effect on the statements of interest.

Therefore, a concept of Program Slicing is introduced to aid developer debugging and program comprehension by reducing the complexity of the program. The idea of Program Slicing has been applied on many applications. Not only on debugging, it is also of great use in program comprehension, such as generating software metrics that calculate the measure of cohesion in a program fragment.

We first look at the concept of program slicing as well as briefly covering different slicing approaches. In the second half, we introduce an Eclipse-based Java program slicing tool, called Indus/Kaveri[9] and a commercial product, called CodeSurfer.

1.1.1 Program Dependency Graph

Before getting into details about Program Slicing, there is a technique that is essential to know: Program Dependency Graph (we will call it PDG from now on). A Program Dependency Graph of a program is a graph that has nodes assigned to each statements of the program and directed edges represented a dependence. The rule is defined that an edge from a statement s_1 to another statement s_2 exists whenever some dynamic instances, v , of s_1 shares a dependence with a later dynamic instance of s_2 [8]. Moreover, a PDG has two types of dependence edges, a data-dependence edge or a control-dependence edge. A data-dependence edge from s_1 to s_2 means that the computation performed in s_2 depends on the value computed in s_1 . A control-dependence edge from s_1 to s_2 implies that s_2 may or may not be executed depending on the boolean outcome of s_1 , for instance, a if-statement.

Consider the code of a bubble sort algorithm on array $n[]$ shown below:

```

1  for (i = (array_size - 1); i >= 0; i--){
2      for (j = 1; j <= i; j++){
3          if (numbers[j-1] > numbers[j]){
4              temp = numbers[j-1];
5              numbers[j-1] = numbers[j];
6              numbers[j] = temp;
7          }
8      }
9  }

```

Listing 1: PDG Example

On data dependent point of view, the value of i in the first for-statement (line 1) controls the boolean expression in the second for-statement (of line 2), and j controls the rest of the program from line 3-6. From the aspect of control dependency, the execution of line 4-6 depends on the boolean outcome of the if-statement in line 3. Therefore a PDG of previous program can be drawn as follow, where a solid line represents a data-dependence relation and a dotted line is the control-dependence relation.:

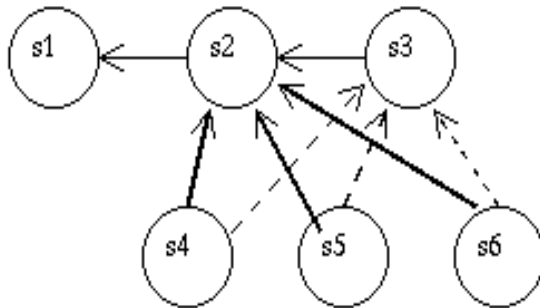


Figure 1: PDG on algorithm of Bubble Sort

Exercise

Problem 1. Illustrate a PDG of Matrix-Chain Multiplication.

Problem 2. Illustrate a PDG of the following pseudo-code of Selection Sort:

```

SELECTION-SORT(A)
1  for i ← 1 to length[A] − 1
2      do min ← i
3          for j ← i + 1 to length[A]
4              if A[j] < A[min]
5                  then do min = j
6          Exchange A[i] with A[min]

```

1.1 Concept

The essence of Program Slicing is to remove the statements that are irrelevant with respect to the selected statement. In other words, Program Slicing only preserves the statements, which have some effect with respect to the selected criterion.

Software Engineers usually analyze a program criterion from two dimensions, a semantic dimension and a syntactic dimension. The semantic dimension describes that which part of program is to be preserved. A static behavior is preserved by static slicing criteria and a dynamic behavior is preserved by dynamic criteria. In other words, static slicing works by statically examining the code without actually executing the program, whereas dynamic slicing dynamically analyzes the code by executing the program with a given input. Thus, a dynamic slice is only correct for a specific input, and conversely a static slice is correct for all inputs.

In the syntactic dimension, there are two possibilities. The slice can preserve the original syntax, removing parts of the code that has found to be no effect on the semantic of interest. This approach is called Syntax-Preserving. The other choice allows slice freely perform any syntactic transformation as long as the semantic constraints is met. This approach is known as Amorphous. Most examples in this report are considered to follow the form of syntax-preserving.

In this lecture, we first review static and dynamic slicing in the point of semantics, then we consider amorphous variation of program slicing.

Exercise

Problem 1. What is Program Slicing?

Problem 2. Can Program Slicing be performed on a recursion problem, and would Program Slicing simplify the problem?

1.0.1 Static Slicing

A static slicing is constructed by deleting those parts of the program that are irrelevant to the values stored in the chosen set of variables at the chosen point of interest. A point of interest in other words, a statement to be sliced, can be mathematically represented by given variable v and a point of interest at line n . This is called the slicing criterion, constructed for v at line n , and is expressed as $s(n, V)$, where s is the slice we are interested in, e.g. $s(8, x)[1, 4]$. A program slice is then computed by removing statements that have no effects to the slice criterion.

In order to choose a slicing criterion, there are two approaches to consider, a backward slicing or a forward slicing. A backward slice is defined to contain the statements of the program that are *affecting* the slicing criterion. In practice, the backward approach can be used in locating the bug by examining all previously executed statements with respect to a variable v at statement n , where output of v is found incorrect at that point. In contrast, a forward slice contains those statements of the program which are *affected* by the slicing criterion. A practical application of forward slicing can be used on keeping track of regression bugs due to a fix being made to the statement that was previously buggy and now becomes our slicing criterion.

To see how static slicing works, consider the following pseudo code example:

```
1 int x = a;  
2 int y = 25;  
3 String z = "";  
4 for(int i = 0; i < x; i++){  
5     z = z + " " + y;  
6     y = y + 2*i;           }  
7 print("Y is " + y);
```

Listing 2: Example 1

The PDG diagram of the example is shown in next page. The for-statement in s_4 depends on variable x , declared in s_1 . Value of z in s_5 relies on its initial declaration at s_3 , for-loop statement at s_4 and manipulation of y in s_6 . Finally, the output of y in s_7 is affected by all statements related to y in s_2 , s_4 and s_6

Assume we are only interested in effect of this piece of code upon variable y . We can now construct a backward slicing on y in line 9. All statements that have some affect on y are preserved in the slice. The result is shown below:

```
1 int x = a;  
2 int y = 25;  
3 for(int i = 0; i < x; i++){
```

```

4 |     y = y + 2*i;           }
5 | print("Y is " + y);

```

Listing 3: Result of Backward Slice on $(y, 9)$ of Example 1

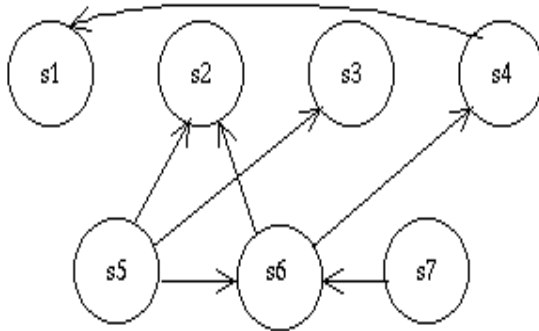


Figure 2: PDG on Example 1

Assume now we change our focus on affects upon changing variable y in line 2, and we construct a forward slicing on y . All statements that are effected by y are preserved:

```

1 | int x = a;
2 | int y = 25;
3 | String z = "";
4 | for(int i = 0; i < x; i++){
5 |     z = z + " " + y;
6 |     y = y + 2*i;           }
7 | print("Y is " + y);

```

Listing 4: Result of Forward Slice on $(y, 2)$ of Example 1

Please note that variable of x has no direct affect upon y and z . However, since x is used in the for statement, thus it must be included due to the range of influence on y and z .

Exercise

Problem 1. Perform a backward and forward Slicing over the following example code on variable k :

```

1 |     int k = 0;
2 |     int*t
3 |     for(int i = 1; i < 5; i++){
4 |         k = k + i;
5 |         g = g + k;           }

```

```
6 | Print(k);
```

Listing 5: Practice Problem

Problem 2. Please explain the potential uses of backward and forward slicing in practice.

1.0.1 Dynamic Slicing

Dynamic slicing is constructed with respect to an growing slicing criterion that additionally contains a sequence of inputs, which cause the program to go wrong for a particular execution. The slices preserve the effect of the original program on v at n , when the program is executed with the input sequence $I[2]$. Consider the following example:

```
1 scanf("%d", &n);  
2 int s = 0;  
3 int p = 0;  
4 for(int i = 1; i <= n; i++){  
5     s += i;  
6     p *= i;  
}
```

Listing 6: Example 2

The PDG diagram of the example is shown in below. User input variable n controls loop statement in s_4 . Both s and p are depends on their initial declaration and for loop statement in s_4 .

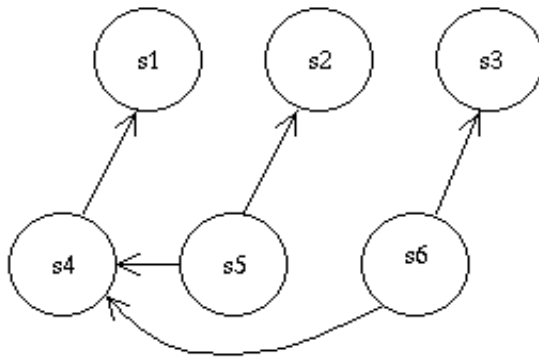


Figure 3: PDG on Example 2

The goal of this program is to calculate the sum and product from 1 to n . Apparently, the sum would yield correct result, but product would go wrong.

A static slice on p would only simplify the program, but is not powerful enough to help us find the bug.

```
1 scanf("%d", &n);
2 int p = 0;
3 for(int i = 1; i <= n; i++){
4     p *= i;
}
```

Listing 7: Static Slicing on $s(p, 6)$ of Example 2

Next, we construct a dynamic slice upon p . Since p keeps multiplying i from itself, which is always a 0. Thus, the only value that p would get is 0. Thus, the input sequence of p is $\langle 0^1, 0^2, \dots, 0^n \rangle$. Dynamic slices will give us the result of $p = 0$:

```
1 p = 0;
```

Listing 8: Dynamic Slicing on $s(p, 6)$ of Example 2

Hence, a bug is thus detected using dynamic slicing approach.

Consequently, dynamic slicing is more useful than static slicing for testing and debugging. However, static slicing is still useful for some application, such as program comprehension, reusing, or maintenance, where the slice has to cover every possible executions. On the other hands, static slicing needs larger space to perform. This contradiction highlights the trade-off between the static and dynamic approaches. Static slicing needs more resource to perform, but will perform every possible execution of the original program. Conversely, dynamic slicing needs less space, but will execute for a particular execution.

Exercise

Problem 1. Perform a backward and forward Slicing over the following example code on variable k :

```
1     int k = 0;
2     int*t
3     for(int i = 1; i < 5; i++){
4         k = k + i;
5         g = g + k;
6     Print(k);
}
```

Listing 9: Practice Problem

Problem 2. Please describe the difference between Static and Dynamic Slicing. Using examples to identify the difference.

1.0.1 Amorphous

All approaches discussed so far have been syntax-preserving, which means the slice preserves the original syntax of the program with respect to the criterion. Conversely, amorphous slices is constructed using any possible syntax transformation, which simplifies the program and as well as preserves the effect of the program with respect to the slicing criterion.

Consider the MIN/MAX/SUM program fragment below:

```

1 for(i = 0, max = min = sum = a[0]; i < SIZE; sum = a[++i]){
2     if(a[i] > max)
3         max = a[i];
4     if(a[i] < min)
5         min = a[i];
6     printf("Max is %d\t", max);
7     printf("Min is %d\t", min);
8     printf("Sum is %d\n", sum);

```

Listing 10: Example 3

The PDG diagram is shown below. What makes this example differ from others is the if-statements. The execution of s_3 depends on the boolean outcome of if-statement in s_2 , while s_4 relies on s_5 . The output value of max and min depends on the execution in s_3 and s_5 respectively.

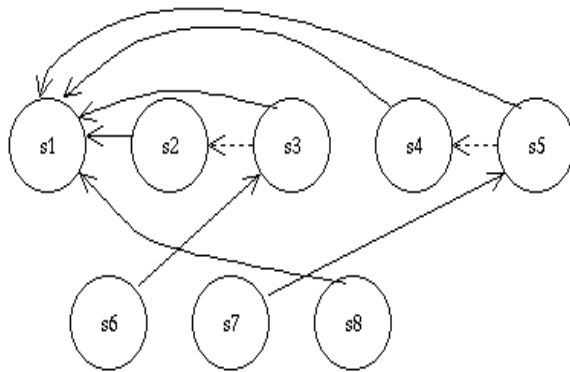


Figure 4: PDG on Example 3

This program is intended to do two jobs, one is to calculate the minimum and maximum element in the array and store them in variable min and max respectively, the other seems to calculate the sum of all elements in the array. Thus, amorphous slicing on the final value for min would transform the program to:

```

1 for(i = 0, max = min = a[0]; i < SIZE; ++i){
2     if(a[i] > max)

```

```

3 |         max = a[i];
4 |     printf("Min is %d\t", min);
   |     }

```

Listing 11: Amorphous on min of Example 3

On the other hand, the summing operation, which performs incorrectly would result in:

```

1 | sum = a[19];
2 | printf("Sum is %d\n", sum);

```

Listing 12: Amorphous on sum of Example 3

Thus, a bug is found on sum operating.

Exercise

Problem 1. Please describe the difference between Syntax-Preserving and Amorphous. Using examples to identify the difference.

Problem 2. Briefly describe how to implement a Amorphous-based Program Slicer.

1.1 Application

Program Slicing has been used as an assistance in many aspects of software development. In the early years, the idea of Program Slicing was first applied on debugging[1]. That is, if a bug occurs due to the incorrect value of a variable, then this technique can be used to find all statements that are affecting the incorrect output in the selected statement.

In addition, bug detection leads to bug correction. An experienced software engineer knows that fixing a bug may cause further bugs. There is a testing method that is used to seek for uncover regression bugs, called Regression Testing. After fix has been made, the system attempts to identify and retest those parts of the program that are affected by the changed statement[3].

Software Measurement is another application. The cohesiveness of a program indicates how well the program within a module work together to provide a specific piece of functionality. A modern development on the cohesion is the idea of encapsulation in object-oriented programming[4]. Therefore, some tools, like Understand C++ Tool, are developed to calculate a cohesion metrics of a program using Program Slicing technique[5].

In some stages of software development, a program needs to be maintained.

The maintenance phase starts with program comprehension. Unquestionably, Program Slicing can aid the programmers and speed up their process to improve productivity.

Exercise

Problem 1. Please describe how Program Slicing can be applied in a real world problem.

1.1 Summary

Program slicing is a technique to reduce the complexity of a program in order to aid testing, debugging or other computer applications. In this lecture, we have summarized the basic concept of Program Slicing, and we will introduce some powerful tools available in the next lecture. Nevertheless, something needs to be put in our mind that there is also vast scope for improvement in this area as the more modern forms of program slicing have not been implemented with more powerful tools for complete languages.

2 Lecture 2 - Program Slicing Tool

2.1 Tool Support

As a debugging aid, there are many useful tools available for the programmer. Most tools are implemented for the simplest types of slicing approach, typically static slicing and syntax-preserving approach.

The most widely known program slicing tool is the Wisconsin Program Slicer. This toolkit can perform backward and forward slicing on C/C++ program. However, this project is no longer being distributed, and has evolved into a commercial product, named CodeSurfer[12].

Unravel[10] is another widely known tool. This tool can only perform simple static backward slicing on C program, and is very easy to use.

Indus[9] is a project at Kansas State University, which provides collections of program analysis and transformation on Java programs. Indus contains many useful module, including Program Static Analysis and Java Program Slicer. Kaveri is an Eclipse plug-in front-end for the Indus Java slicer. It utilizes the Indus program slicer to calculate slices of Java programs and then displays the results visually in the editor. This application provides static forward and backward slicing only on a Java-written program. It also can handle object orientation and concurrent program.

Exercise

Problem 1. What is the current issues in the field of Program Slicing?

Problem 2. The following code is Buggle Sort that contains many uncovered bugs, use Program Slicing Tool to locate the bug:

```
1 boolean swappedOnPrevRun = true;
2 long temp=0;
3 while(swappedOnPrevRun){
4     // reset flag
5     swappedOnPrevRun = false;
6
7     // Loop over array bubbling element to top
8     for(int i = 0; i < array.length-1; i++){
9         if(array[i] > array[i + 1]){
10            // set flag
11            swappedOnPrevRun = true;
12
13            // swap the two elements
14            temp =(long) array[i];
15            array[i] = array[i+1];
```

```
16         array[i+1] = temp;
17     } // end if
18 } // loop over elements
19 } // end of while
```

Listing 13: Practice on Locating Bugs

2.1 Overview of Indus/Kaveri

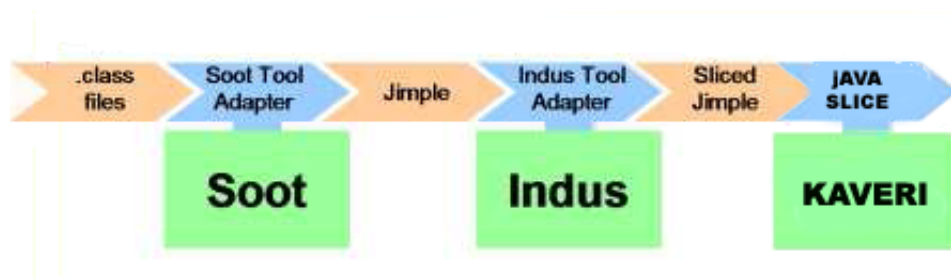


Figure 5: Indus/Kaveri Framework Architecture

Under Indus environment, Java programs (.class files) are represented in Jimple, a typed three-address representation, provided by SOOT. SOOT[14] is a Java Optimization Framework library to provides four intermediate representations for analyzing and transforming Java bytecode, and Jimple is one of those four representations. Hence, all the Java code is converted into Jimple and then fed to the Slicer. Next, the slicer calculates the slice of the given system in terms of Jimple, and displays resulting slice on the code screen.

Example on Converting Java into Jimple

Consider `j++`, a Java expression, to be translated into Jimple representation as following:

```
1 $i2 = r0.<myPackage.Monitor: -> int j>
2 $i3 = $i2 + 1
3 r0.<myPackage.Monitor: -> int j> = $i3
```

Listing 14: Simple Jimple Example

In the first statement, the reference from monitor field is assigned to a local variable `i2`. In the next statement, the local variable `i3` is assigned the incremented value of `i2` and finally the original field is updated. As the Indus Java Slicer works on Jimple, the criteria for slicing is specified as Jimple chunks. From the above illustration, a Java statement can map to many Jimple statements. Therefore, when a Java statement/expression needs to be used as a slice criteria in Kaveri, the user has to pick one or more of the Jimple statements to which the Java statement/expression is mapped to. For more exposition about how to pick criteria closest to your requirements, please refer to Appendix - How to pick criteria.

2.2 Prerequisites

There are several package required to install on your local machine. Please note that the recommended version of the packages are those we have tested on our machine and run properly.

1. Java 1.4.2

The recommended Java runtime environment from Indus/Kaveri officials is 1.4¹. If you are using newer version of Java, we do not guarantee its compatibility with current release of Indus/Kaveri.

2. Eclipse 3.2

The newer version of Indus/Kaveri has adapted Eclipse 3.2². If you are using older version of Eclipse SDK[13], we do not guarantee its compatibility with current release of Indus/Kaveri.

1. Indus

Download edu.ksu.cis.indus-0.8.3.6.zip file from the Indus Download Site³. Note that this file is for Indus plug-in for ECLIPSE. To install, unzip the downloaded file and place it under ECLIPSE plug-in folder.

2. Kaveri

Download edu.ksu.cis.indus.kaveri-0.8.3.6.zip file from the Kaveri Download Site⁴. Note that this file is for Kaveri plug-in for ECLIPSE. Kaveri is the GUI interface for Indus Java slicer. To install, unzip the file and place it under ECLIPSE plug-in folder.

3. Groovy Monkey

Download Groovy Monkey Script ECLIPSE plug-in from the Groovy Monkey Site⁵. Note that this file is required in order to make Indus / Kaveri work properly.

2.3 Setting up Indus/Kaveri Environment

1. Configure the slicer preferences

- a. In ECLIPSE, click Window→Preferences→Indus Preferences
- b. Set of default configurations are provided. (See Fig. 6)
- c. From the Configuration tab you can manage the set of slicer configurations.

¹<http://java.sun.com>

²<http://download.eclipse.org/eclipse/downloads/drops/R-3.2.2-200702121330/>

³<https://projects.cis.ksu.edu/projects/indus/>

⁴<https://projects.cis.ksu.edu/projects/indus/>

⁵<http://groovy.codehaus.org/Groovy+Monkey>

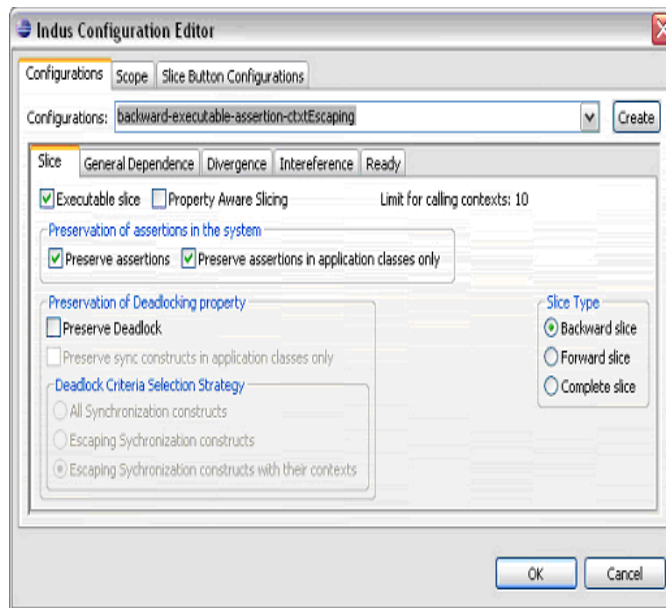


Figure 6: Slicer Configuration Preferences

- d. Be sure to press Apply button after any changes to preserve the settings.
2. Scope Preferences The Scope tab shows the set of current scope entities that have been already defined. The set of displayed scope entities are available to all the projects during slicing. The scope tab allows the user to create a new class scope specification and delete existing scope specifications. (See Fig. 7 Above)
3. Slice Button Configuration Preferences The Slice Button Configurations tab allows the user to change the slice configurations used internally by the Backward and Forward slice action buttons. This enables the user to associate a more useful slice configuration with the slice buttons (for example, a "backward slice without deadlock" configuration to the Backward slice action button). The set configuration is then used to drive the slicer, when the corresponding button is pressed. (See Fig. 7 Below)

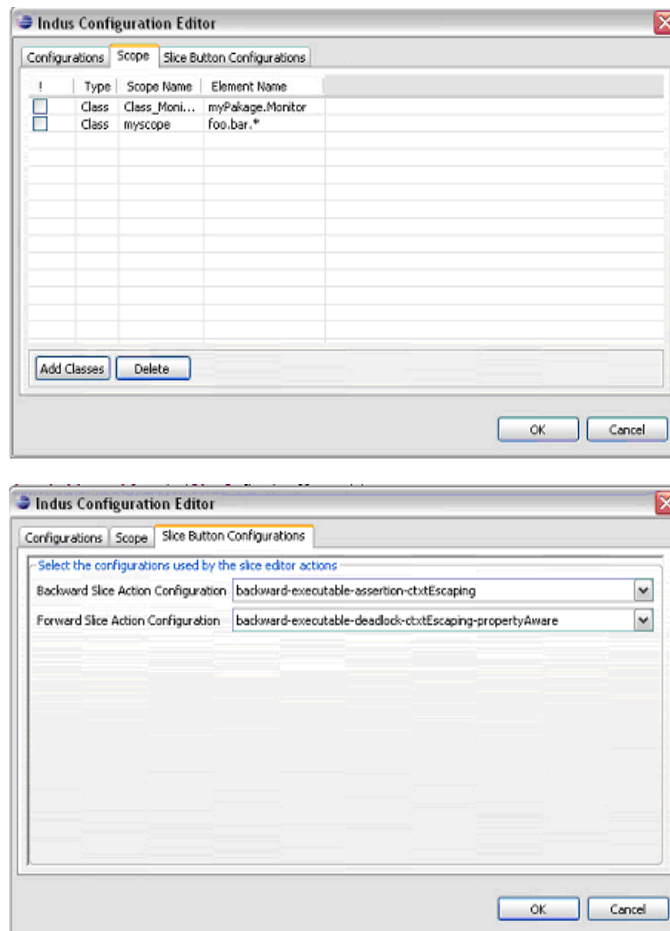


Figure 7: Scope Preference (Above), Slice Button Configuration Preferences (Below)

2.4 Setting up the Criteria

1. Enable Jimple View

- a. In ECLIPSE, click Window → Show View → Other → Kaveri → Jimple View
- b. Once the Jimple View has been enabled, the view is activated by pressing the Track Java Statements toolbar button.
- c. Upon pressing the button, the button changes its color to green to indicate that the view is active.
- d. Pressing it again will toggle the view to the disabled mode. When the view is active, navigate to a Java statement from the Java

editor.

- e. The view's contents automatically change to reflect the Jimple statements that correspond to the chosen Java statement. The view also indicates if the statements are associated with the slice, if a slice has already been performed. (See Fig. 8)

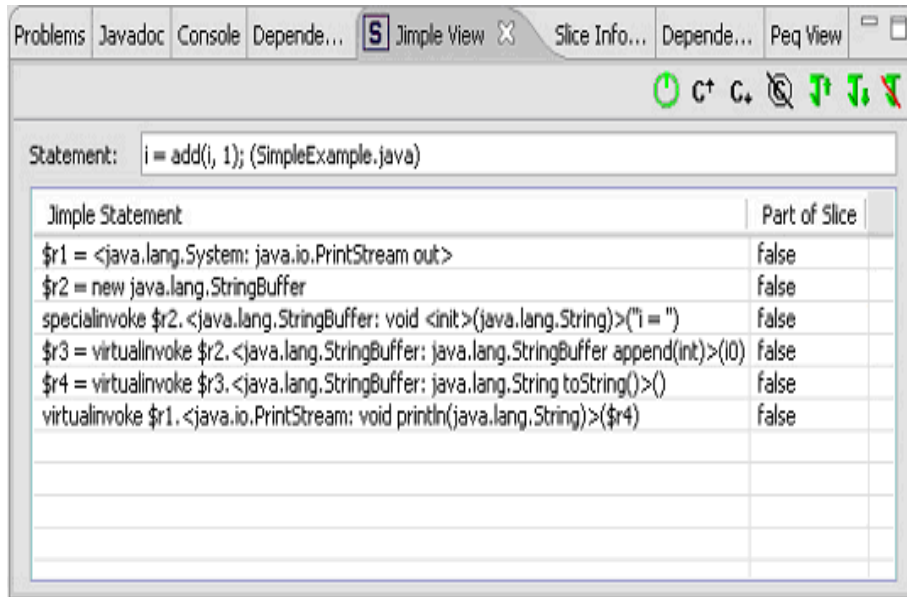


Figure 8: Enable Jimple View

2.5 Running Kaveri

1. Right click the Java file or Java project in the navigator or package explorer window in ECLIPSE and choose either Indus → Slice Java file or Indus → Slice project from the context menu. The difference is that in the former only the specified file is sliced while in the latter all the files present in the Java project are sliced.
2. In the slice configuration dialog that appears, pick the slice configuration and the criteria for slicing. The Additive slice display check box indicates that the new slice should be added onto the display of the previous slice. This displays the slice as the union of the previous and the new slice. (See Fig. 9)

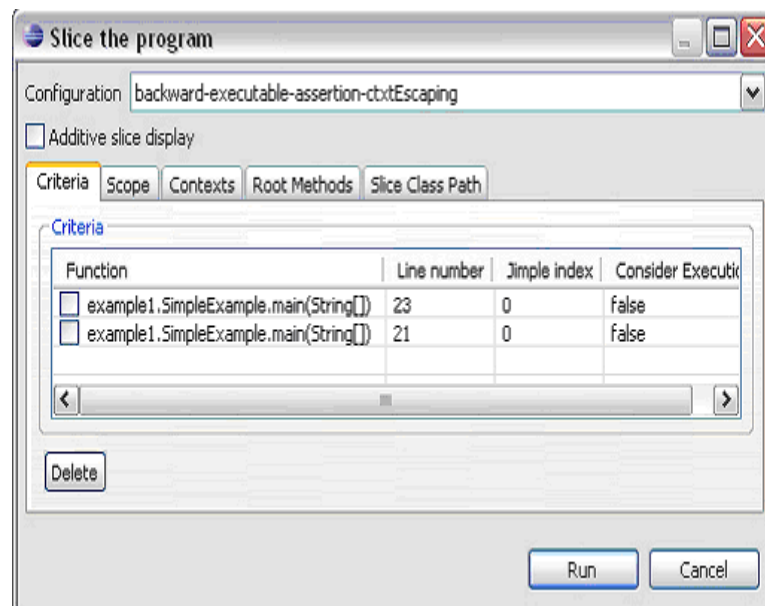


Figure 9: Slice configuration dialog

3. Pick the scope that needs to be considered during the slice. The Scope tab in the slice configuration dialog enables the user to choose the scopes to be considered from the set of scope specifications created previously.
4. Pick the context for the slice, if required. The Contexts tab in the slice configuration dialog allows the user to specify the context for the slice. (See Appendix - Creating Contexts)

5. Manage the set of additional root methods. The Root Methods tab in the slice configuration dialog allows the user to delete any user-defined root methods before performing the slice. (See Appendix - Root Methods for details about adding custom root methods.)
6. Change the class path used for slicing, if required. The Slice Class Path tab in the slice configuration dialog allows the user to change the classpath used by soot. The user can change the classpath to instrument alternative libraries in order to increase the speed of slicing.
7. Press the Run button to start the slicing. The slice is automatically highlighted. There are two types of highlighting that are displayed. Java statements for which all the corresponding Jimple statements have the slice tag are highlighted in once color while those in which only a part of the Jimple statements have the slice tag are highlighted in a different color. (See Fig. 10)

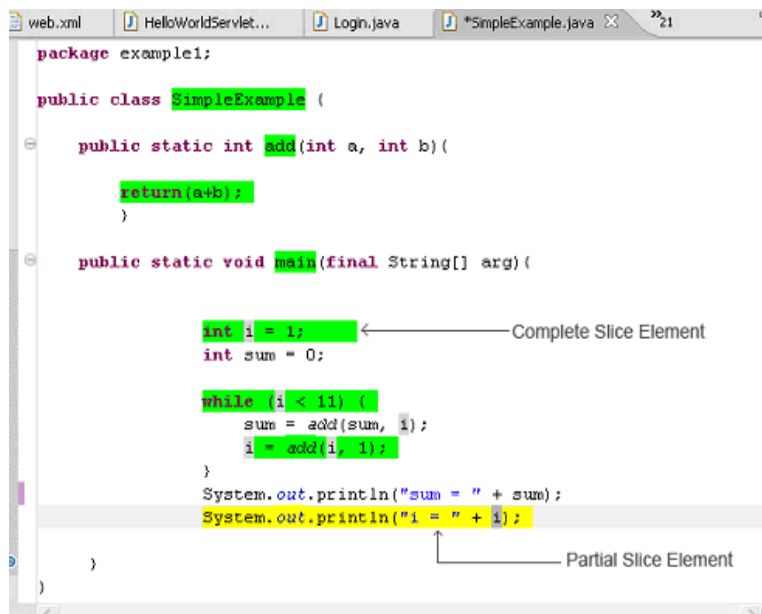


Figure 10: Slicing result view

8. Dependence Tracking View

Kaveri allows the user to view and follow the program dependence information for a Java statement. To perform this action

- a. In ECLIPSE, click Window → Show Views → Kaveri → Dependence Tracking View

- b. The view has a Track Java Statements toolbar button similar to the Slice View that is used to toggle the active state of the view. Set of default configurations are provided.
- c. The toolbar button changes its color to green to indicate the active state. (See Fig. 11)

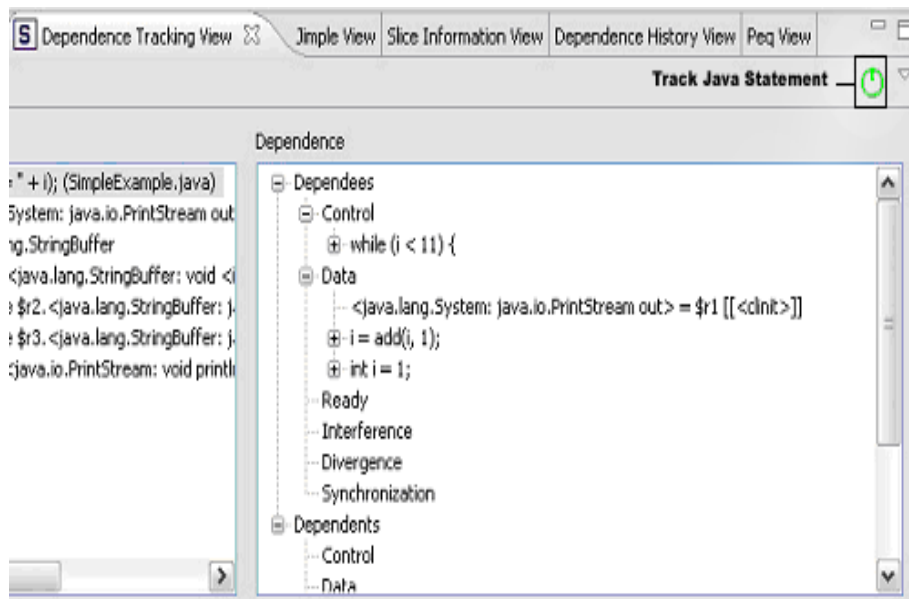


Figure 11: Dependence Tracking View

2.6 Features - Toolbar Buttons

The Indus Java Slicer requires a configuration to specify details such as the type of slice to be performed (backward, forward or complete). The following toolbar buttons are activated when a Java file is open in the editor. (See Fig. 12)

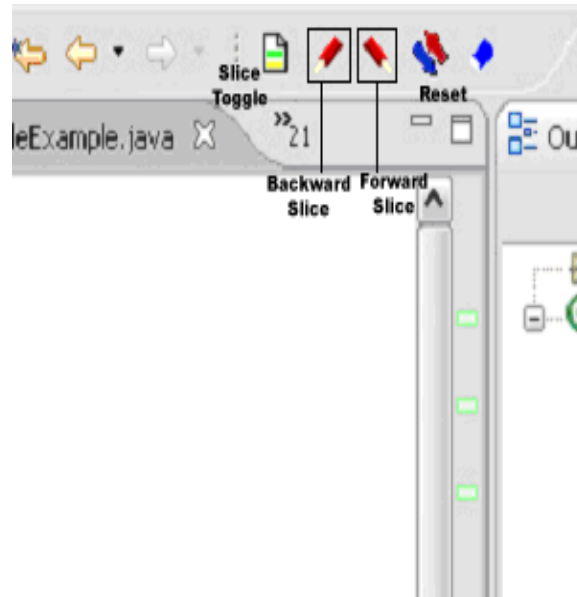


Figure 12: Toolbar Buttons

1. Slice highlight toggle

The slice toggle menu button is used to toggle the slice highlighting on / off in the Java editor. After a file is sliced, open it and press this button to show the slice in the editor.

2. Backward Slice

To run a backward slice without going through the criteria and configuration selection phases, you can select a Java statement and press the backward slice menu button to run a backward slice with the selected statement as the criteria. After the slice is performed the relevant statements are highlighted automatically.

3. Forward Slice

To run the forward slice on a given Java statement selects it and press the Forward Slice menu button. Note that Indus currently supports only non-executable forward slicing.

4. **Reset**

The reset button resets the slice, removes the annotations and clears the views. If any changes are made to the program, it is advisable to press this button as doing so resets soot, allowing the changed code to be sliced.

2.7 Appendix

2.7.1 How to pick criteria

As mentioned before, the criteria for slicing is a Jimple statement. When a Java statement is picked, the user is allowed to select one of the corresponding Jimple statements as the criteria. The following examples show how to choose the correct Jimple statement depending on the requirement. Consider the simple statement: `j++`, where `j` is an integer. The equivalent Jimple expression for this statement consists of the following:

```
1 $i2 = r0.<myPackage.Monitor: -> int j>
2 $i3 = $i2 + 1
3 r0.<myPackage.Monitor: -> int j> = $i3
```

Listing 15: Simple Jimp Example

2.7.2 Creating Contexts

As specified earlier, the user can specify a call-chain as a context for the slice. To do so:

1. Navigate to the method containing the criteria to be used for the slice in the editor and open the Call Hierarchy view by choosing Open call hierarchy from the context menu.
2. Switch to Caller mode by pressing the Show Caller Hierarchy toolbar button. (See Fig. 13)

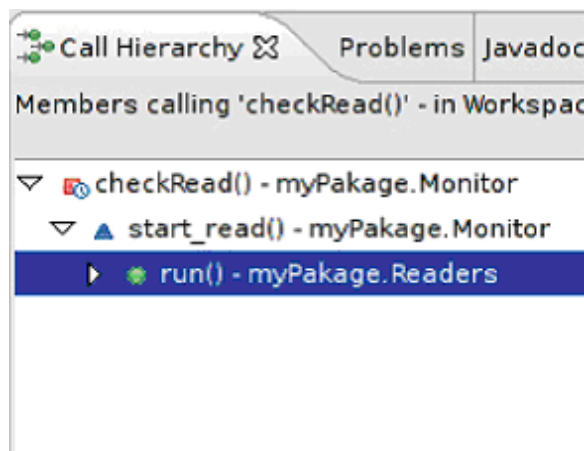


Figure 13: Caller Hierarchy view

3. Expand the chain of method calls until a desired method is reached.
4. Select the method starting the call chain and choose Indus → Add to context from the context menu
5. This adds the call chain to the list of call-chain contexts available for slicing.
6. This can then be picked from the Slice Configuration dialog before performing a slice.

2.7.3 Adding custom root methods

Kaveri now allows the user to slice any Java program including applets, servlets and Etc. To do so, the user needs to define a set of root methods for the project. These act as entry points to be used while slicing. In projects containing the main method, there is no need to specify the root methods as these are picked up automatically. (See Fig. 14) To add a method as a

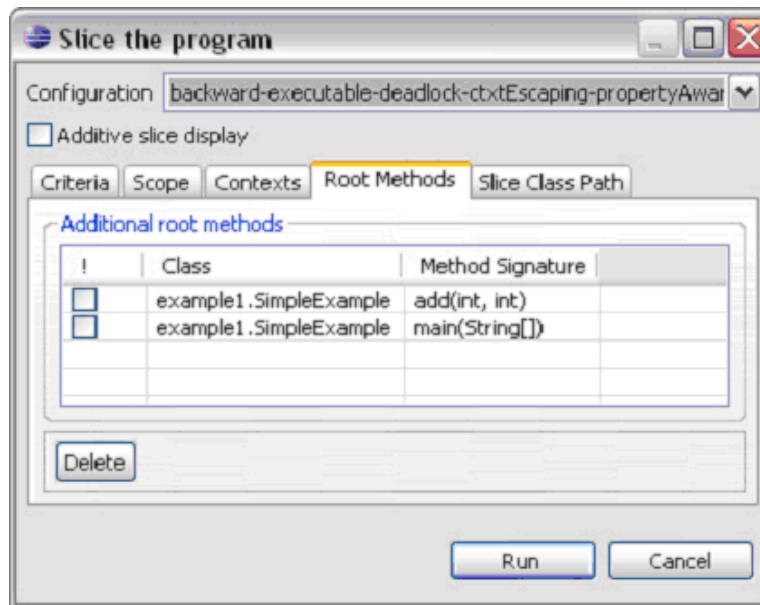


Figure 14: Root Methods Tab

root method:

1. Select the method in the Java outline view and select Indus → Mark as root method from the context menu.
2. A marker is placed on the method, which is visible from the Java editor indicating that this is a root method.
3. Once this has been done, the root methods are added to any slice that is performed on the file. This means that the Slice Actions can also be used to perform the slice. Note that there is no need to explicitly pick the root methods for slicing, as they are automatically used once they are defined.
4. The user can delete the unnecessary root methods from the Slice Configuration dialog.

2.7.4 Outline of This Report

Lecture 1

1. Introduction to Program Slicing (5 min)
2. Program Dependence Graph (10 min)
3. Static Slicing (15 min)
4. Dynamic Slicing (15 min)
5. Syntax-Preserving vs. Amorphous (15 min)
6. Application (10 min)
7. Conclusion (5 min)

Lecture 2

1. Overview of Tools (5 min)
2. Overview of Indus/Kaveri (5 min)
3. Prerequisite of Indus/Kaveri (5 min)
4. Setting up Indus/Kaveri Environment (15 min)
5. Setting up Criteria (15 min)
6. Running Kaveri (15 min)
7. Features - Toolbar Buttons (15 min)
8. (Appendix) How to create a slice (5 min)
9. (Appendix) Creating Contexts (5 min)
10. (Appendix) Adding custom root methods (5 min)

References

- [1] Mark Weiser. *Program Slicing*, 1984
- [2] Hiralal Agrawal and Joseph R. Horgan. *Dynamic Program Slicing*, 1990.
- [3] Rajiv Gupta, Mary J. Harrold and Mary L. Soffa. *An Approach to Regression Testing using Slicing*, 1992
- [4] Mark Harman, and Robert H. Hierons. *An Overview of Program Slicing*, 2001
- [5] Kai Pan, Sunghun Kim and James Whitehead, Jr. *Bug Classification Using Program Slicing Metrics*, 2006
- [6] Ganeshan Jayaraman, Venkatesh Prasad Ranganath and John Hatcliff *Kaveri:Delivering the Indus Java Program Slicer to Eclipse*, 2005
- [7] Venkatesh Prasad Ranganath, and John Hatcliff. *Slicing Concurrent Java Program using Indus and Kaveri*, 2006
- [8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*, Page 854-857, 2006
- [9] Indus Project. <http://indus.projects.cis.ksu.edu/>
- [10] Unravel Project. <http://hissa.nist.gov/unravel/>
- [11] Wisconsin Program Slicing Project. <http://www.cs.wisc.edu/wpis/html/>
- [12] CodeSufer Product. <http://www.grammatech.com/products/codesurfer/overview.html>
- [13] IBM Eclipse Project. <http://www.eclipse.org>
- [14] SOOT Project <http://www.sable.mcgill.ca/soot/>