Introduction

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. The purpose of this project is to create an effective tool for simplifying program understanding, program analysis, program debugging and testing.

Requirement

1. Eclipse: Eclipse version 3.0 or 3.0.1 is needed.
2. Indus Plug-in: Indus Plug-in version 0.5.
Name

Kaveri is one of the major rivers in India, my native land. A river has many branches through which it flows. In a similar way a program branches in it's execution. It is hoped that Kaveri makes it easier to decipher this flow and increases your understanding of the program.

Install / Uninstall

Installation

Install: Using zip files

Kaveri can be downloaded from the SAnToS sourceforge site at: http://projects.cis.ksu.edu/projects/indus/. To install, unzip the downloaded file edu.ksu.cis.indus.Kaveri-$VERSION.zip in ECLIPSE_HOME. Make sure that the folder names are preserved while decompressing. Note that Kaveri requires the Indus plug-in in order to run. The Indus plug-in can be downloaded from http://projects.cis.ksu.edu/projects/indus/.

Uninstall

To uninstall Kaveri, remove the edu.ksu.cis.indus.kaveri_$VERSION folder from your ECLIPSE_HOME/plugins/ directory.

Concepts

- **Configuration** - The Indus Java Slicer requires a configuration to specify details such as the type of slice to be performed (backward, forward or complete), the types of dependencies to track in the slice (ready, divergence, interference) [HatcliffSAS99] and the parameters for each dependency. In short, it is the configuration of the slicer. To learn more about the configuration please look at the Indus Java Slicer Documentation [http://projects.cis.ksu.edu/docman/view.php/12/71/slicer-ug.pdf]

- **Jimple** - Jimple is a typed 3-address intermediate representation of Java and Indus slicer operates on Jimple. Hence, all the Java code is converted into Jimple and then fed to the Slicer. The slicer calculates the slice of the given system in terms of Jimple.

Example 1. Example conversion of Java into Jimple

Consider `nw++`, a Java expression. Following is a Jimple representation of this expression:

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

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 information about Jimple please refer to [VH98]. For more exposition about how to pick criteria closest to your requirements, please refer to Appendix - How to pick criteria.

- **Value of the expression** - When an expression / statement is specified as a criteria, does this mean:
  - The value of the expression should be preserved (post-execution) or
  - The control reaching the expression should be preserved (pre-execution)

In the former we consider all those statements which might affect the computed value of the expression while in the latter we consider only those statements through which control can reach the selected expression or statement.

**Example 2. Example to show the influence of control or value while choosing a criteria**

Consider the following fragment of code.

```java
public static void main(final String[] args) {
    int a, b;
    a = b = 1 + randInt();
    if (a == 2) {
        a = 10;
    } else {
        b = 20;
    }
    a = a + b;
}
```

If the statement `a = a + b` is chosen as the criteria, picking control or value causes different slices to be generated. If control is picked, the statements corresponding to if conditional will be ignored in the slice as they don't affect the reachability of the chosen statement. However if the value of the expression is picked then the conditional statements have to be included as they affect the computed value of the expression.

- **Scope Specification** - A scope specification restricts the scope that the slicer considers while performing a slice. When a criteria is embedded within a scope, the slicer does not consider elements outside the scope, while calculating the slice. For example, if a backward slice if performed with a criteria $C$ which is present in a method $foo$ and the method $foo$ is specified as a scope and suppose that no other functions are called in $foo$. Then the backward slice only contains elements within the method $foo$. It should be noted that the specified criteria should be contained within the scope, to get a result from the slicer.

Kaveri allows three types of entities to be added to the scope:

- **Classes** - The class scope specification defines the class or classes that should be considering as
defining a scope, while performing the slice. The scope specification is given a Scope Name to identify it from the other scope specifications. The class scope specification also allows the scope around the class to be specified. These include:

**Just this class** - Only consider the class and not the relatives of the class. In this case, the class type specification can be entered as a regular expression. All the classes whose names match the expression are added to the scope. For example, giving a type of "foo.bar.*" will match all classes in the package "foo.bar".

**Ancestors excluding this class** - Consider all the ancestors of this class for scoping, but ignore the chosen class. In this case, the class type should be the fully qualified name of the class.

**Ancestors including this class** - Consider all the ancestors of this class for scoping, including the chosen class. In this case, the class type should be the fully qualified name of the class.

**Descendants excluding this class** - Consider all the descendants of this class for scoping, but ignore the chosen class. In this case, the class type should be the fully qualified name of the class.

**Descendants including this class** - Consider all the descendants of this class for scoping, including the chosen class. In this case, the class type should be the fully qualified name of the class.

**Methods**

The method scope specification defines the method that should be considered as a scope while performing the slice. The scope specification is given a Scope Name to identify it from the other scope specifications.

**Fields**

The field scope specification defines the field that should be considered as a scope while performing the slice. The scope specification is given a Scope Name to identify it from the other scope specifications.

- **Context Sensitive Slicing** - Context sensitive slicing allows the user to prune the call chain used by the slicer when calculating a backward slice. For example, if method m contains the criteria C, and there are two call chains z -> y -> x -> m and a -> b -> c -> m (z,y,x,a,b,c are also methods) coming into m. The call chain a -> b -> c -> m might be more interesting with respect to a backward slice from m with C as the criteria than the other call chain. The user can specify the call chain a -> b -> c -> m as the Context for the backward slice. The slicer then ignores the call chain z -> y -> x -> m. Note that beyond a, the call chain is unspecified, so all the call chains beyond that point are considered for slicing.

*Note that to perform a context sensitive slice, the call chain should terminate with a method which contains a criteria for the slice.*

## Setup

### Setting up Kaveri

### Preferences

**Indus Preferences**
Before using Kaveri the user needs to define a set of configurations to use with the slicer. To do so open **Window -> Preferences -> Indus Preferences** from the Eclipse menu. A set of default configurations are provided. (Refer to Figure 1). From the Configuration tab you can manage the set of slicer configurations. Be sure to press the **Apply** button after any changes to preserve the settings.

**Figure 1. Slicer Configuration Preferences**

![Indus Preferences]

Note that installing Kaveri version 0.4 will delete the previous slice configurations. The reason being that Indus version 0.5 uses a different configuration scheme than Indus 0.3.

**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.

**Figure 2. Scope Preference**
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.

**Figure 3. Slice Button Configuration**

Exception Ignore List

Exceptions usually inject control (decision) points. However, most of these points are superflous. Moreover, it is hard to trim these. For the purposes of program comprehension, it is useful if the effects of such exceptions can be ignored. For example, a set containing `java.lang.Exception` would be useful to analyze concurrent programs. Similarly, a set of thread related exceptions (`java.lang.InterruptedIOException`) would be useful to analyze sequential programs.
The Exceptions Tab allows the user to add a set of exceptions that are ignored when calculating the slice. The Add Exception button adds the given fully qualified exception name into the set of ignored exceptions. The Delete button can be used to delete an exception from this list.

Figure 4. Exception Ignore List

![Exception Ignore List](image)

Setting up the criteria

Slice View

A Java project can be associated with a set of slicing criteria pertaining to the Java files included in the project. Kaveri currently allows you to add Java statements as criteria for slicing. To add a Java statement as a criteria, enable the Slice View by going to Window -> Show View -> Other -> Kaveri -> Slice View. Once the Slice View has been enabled, the view is activated by pressing the Track Java Statements toolbar button. Upon pressing the button, the button changes its color to green to indicate that the view is active. 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. 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.

Jimple statements that have been already chosen as criteria as displayed with a blue or red foreground color in the view. A Jimple statement with a blue foreground color indicates that the statement was added as a criteria with value of expression turned on. A Jimple statement with a red foreground color indicates that the statement was added as a criteria with Control reaching the statement turned on.

Toolbar Buttons:

<table>
<thead>
<tr>
<th>Toolbar Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Track Java Statements</td>
<td>Toggles the active state of the view. The Jimple statements for Java statements chosen in the editor are only displayed when the view is switched on</td>
</tr>
<tr>
<td>Add as criteria (Pre-Execution)</td>
<td>Adds the selected Jimple statement to the set of criteria associated</td>
</tr>
</tbody>
</table>
with the project. The criteria is added with the *control reaching the statement* property turned on.

**Add as criteria (Post-Execution)**

Adds the selected Jimple statement to the set of criteria associated with the project. The criteria is added with the *value of the expression* property turned on.

**Remove Criteria**

Removes the selected Jimple statement from the set of criteria associated with the project.

**Add as Java criteria (Pre-Execution)**

Adds all the Jimple statements shown in the view as criteria with control reaching the statements property turned on. This is a shortcut to manually adding all the statements as criteria.

**Add as Java criteria (Post-Execution)**

Adds all the Jimple statements shown in the view as criteria with value of the expression property turned on. This is a shortcut to manually adding all the statements as criteria.

**Remove all Criteria**

Removes all the Jimple statement in the view from the set of criteria associated with the project.

---

**Figure 5. Slice View**

---

**Running Kaveri**

**Steps**

**Step 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.

**Step 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.
Step 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.

Step 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. Please refer to **Appendix - Contexts** for details about picking contexts.

Step 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. Please refer to **Appendix - Root Methods** for details about adding custom root methods.

Step 6 Change the classpath 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.

Step 7 Press the **Run** button to start the slicing. A progress dialog indicates the progress of the slicing. After the slice is completed, open the Java file if it is not currently open. If the *Kaveri decorator* is enabled (Refer to Figure 10) the slice is automatically highlighted. Else press the **Slice Toggle** button (Refer to Figure 9) to highlight the slice in the editor. 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. To change the color for the slice highlighting go to **Window -> Preferences -> Workbench -> Editors -> Annotations** from the Eclipse menu and change the color for the `IndusSliceAnnotation` for the complete slice element highlight annotation and `IndusPartialSliceAnnotation` for
the partial slice element highlight annotation. You may have to reopen the file in the editor and toggle the slice highlight for the change in color to be updated.

Figure 7. Viewing the slice

Please note that in this version of Kaveri, the soot environment is not reset for performance reasons. This means that when a slice has been performed on a project, the contents of the project should not be changed until Eclipse is closed. Doing so, may lead to incorrect slice displays. For example, if among two consecutive statements 'a' and 'b', only 'a' has the slice highlight and the statements are interchanged; then, on performing the same slice operation, 'b' will be tagged with the slice highlight meant for 'a'. Thus, it is necessary that the project contents remain unchanged after using the slice on a project. If any changes are made, please press the reset toolbar button and then perform a new slice on the changed code.

Dependence Chasing

Dependence Tracking View

Kaveri allows the user to view and follow the program dependence information for a Java statement. To perform this action, the Dependence Tracking View needs to be activated. The view can be enabled from Window -> Show Views -> Kaveri -> Dependence Tracking View. 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. The toolbar button changes its color to green to indicate the active state.

Once the view has been activated, selecting a Java statement in the editor will display the selected statement in the left pane of the view. The dependence information for that statement is presented in the right pane. The information displayed is as follows: for any tree node displaying Java statements, child nodes display the equivalent Jimple statements. If the Java source for one of the Jimple statements in the right pane is not available, only the Jimple node is displayed. To follow a dependence link, perform a double
click on the corresponding node in the right pane. This displays the selected node in the editor and the left, and right panes are updated to show the dependence information corresponding to the selected node. When a double click is performed on a node in the right pane, the plugin records this information in the Dependence History View, so that the user has a visual record of the dependence information that was followed. The history information can be navigated so that the user can return to a previous point in dependence tracking history and follow a different dependence path.

The Dependence Tracking View has a set of filters that can be used to control the dependence information that is displayed in the right pane. For example, disabling the Dependee -> Control removes the control dependee information from the right pane. This allows the user to concentrate on the more relevant dependence information. The filters can be activated from the pull down menu in the toolbar.

Please note that the view currently does not allow navigation to points for which no Java source is available. These nodes are identified by the fact that the node displays the Jimple content and there is no parent node with a Java source text. Also, in Kaveri 0.4 Beta 1, the dependence tracking view is only operational when a slice has already been performed on a project. This requirement will be removed in the final release of Kaveri 0.4.

Figure 8. Dependence Tracking View

Dependence History View

The Dependence History View records the dependence path followed by the user using the Dependence Tracking View. This view records the information in the form of a stack with the first row displaying the current program point under observation. The dependence history can be navigated with the help of the navigation buttons present in the toolbar. The navigation is similar to the way a web browser’s back and forward navigation buttons operate. To navigate to a specific point in the history, either the navigation buttons can be used or the user can select the point and click the Goto Source menu item from the con-
Each entry in the Dependence History View displays the Java statement that was tracked, the file in which the statement is present and the relation with the item below it. The relation can either be the dependence relation connecting the two statements that was followed from the Dependence Tracking View or “Starting Program Point” to indicate that the statements were not followed as a link from the Dependence Tracking View.

### Figure 9. Dependence History View

<table>
<thead>
<tr>
<th>Statement</th>
<th>Filename</th>
<th>Line number</th>
<th>Relation with previous item</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>objectW = new Object();</code></td>
<td>ReadersWriters.java</td>
<td>14</td>
<td>Data Dependent</td>
</tr>
<tr>
<td><code>objectW = new Object();</code></td>
<td>ReadersWriters.java</td>
<td>14</td>
<td>Interference Dependeen</td>
</tr>
<tr>
<td><code>synchronized(objectW)</code></td>
<td>ReadersWriters.java</td>
<td>54</td>
<td>Starting Program Point</td>
</tr>
<tr>
<td><code>synchronized(this)</code></td>
<td>ReadersWriters.java</td>
<td>45</td>
<td>Synchronization Dependeen</td>
</tr>
<tr>
<td><code>nr--;</code></td>
<td>ReadersWriters.java</td>
<td>47</td>
<td>Interference Dependeen</td>
</tr>
<tr>
<td><code>nr++;</code></td>
<td>ReadersWriters.java</td>
<td>37</td>
<td>Interference Dependeen</td>
</tr>
<tr>
<td><code>nr--;</code></td>
<td>ReadersWriters.java</td>
<td>47</td>
<td>Data Dependeen</td>
</tr>
<tr>
<td><code>nr++;</code></td>
<td>ReadersWriters.java</td>
<td>37</td>
<td>Control Dependeen</td>
</tr>
<tr>
<td><code>if(nw == 0)</code></td>
<td>ReadersWriters.java</td>
<td>35</td>
<td>Starting Program Point</td>
</tr>
</tbody>
</table>

### Features

#### Toolbar buttons

The following toolbar buttons are activated when a Java file is open in the editor.

### Figure 10. Toolbars
**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.

**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.

**Forward Slice**

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

**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.

**Slice label decoration**

Kaveri has a *label decorator* that annotates the list of Java files shown in the package or navigator window with a marker to indicate that the file has a slice associated with it. To enable or disable it go to Window -> Preferences -> Workbench -> Label Decorations from the Eclipse menu and toggle the checkbox with the Kaveri.Decorator label.

**Figure 11. Slice label decorator**
Criteria View

The criteria view allows the user to view the criteria specified in a Java file. The view can be activated from Window -> Show Views -> Kaveri -> Criteria View. Once the view is open, loading any Java file in the editor displays the list of criteria specified in that file, in the criteria view. This is helpful for the user in determining the list of criterias specified for a given Java file.

Figure 12. Criteria View

<table>
<thead>
<tr>
<th>Function</th>
<th>Line number</th>
<th>Jimple index</th>
<th>Consider Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>myPakage.Monitor.checkRead()</td>
<td>37</td>
<td>0</td>
<td>true</td>
</tr>
<tr>
<td>myPakage.Monitor.checkRead()</td>
<td>37</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>myPakage.Monitor.checkRead()</td>
<td>37</td>
<td>2</td>
<td>true</td>
</tr>
</tbody>
</table>

Indus Menu

The Indus Menu visible in the main eclipse menu bar allows the user to quickly access Indus features. The following items are included at the moment:
• **Slicer Configuration:** This is a shortcut to the Indus preferences option. It allows you to change the slicer configurations, manage the scope entities and add elements to the exception ignore list.

• **Slice Statistics:** This action presents statistics for the previously performed slice.

More items will be included in this menu to increase the usability of the tool.

**Appendix - 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.

**Examples of picking Jimple criteria**

**Example 3. Setting up criteria for an assignment statement**

Consider the simple statement: `nw++`, where `nw` is an integer. The equivalent Jimple for this statement consists of the following:

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

Here `r0.<myPackage.Monitor: int nw>` corresponds to the variable `nw`. So to pick `nw = nw + 1` as a criteria you would pick the last statement `r0.<myPackage.Monitor: int nw> = i3` as in that statement the final value of `nw` is assigned.

**Example 4. Setting up criteria for a function call**

Consider the expression `monitor.start_write()`. The Jimple for this consists of two statements:

- `r1 = r0.<myPackage.Writers: myPackage.Monitor mon>`
- `virtualinvoke r1.<myPackage.Monitor: void start_write>()()`

To pick the function call, the second statement `virtualinvoke r1.<myPackage.Monitor: void start_write>()()` is to be selected as it involves the call to the function `start_write()` in the `Monitor` class.

**Example 5. Setting up criteria for a conditional**

Consider the expression: `if (!empty(Clist))`. The equivalent Jimple for this statement is:

- `$r4 = r0.<temp.DiskScheduler: java.util.LinkedList Clist>`
• $z0 = \text{virtualinvoke } r0.<\text{temp.DiskScheduler: boolean empty(java.util.LinkedList)}>(r4)

• if $z0 \neq 0 \text{ goto } r7 = r0.<\text{temp.DiskScheduler:java.util.LinkedList NList}>

To pick the function call $\text{empty(Clist)}$ you would choose the second statement $z0 = \text{virtualinvoke } r0.<\text{temp.DiskScheduler: boolean empty(java.util.LinkedList)}>\text{ as it invokes the } empty() \text{ method. On the other hand if you wanted to choose the if conditional you would choose the last line if } z0 \neq 0 \text{ goto } r7 = r0.<\text{temp.DiskScheduler:java.util.LinkedList NList}>

### Appendix - Creating Scope Specifications

Kaveri allows classes, methods and class fields to be entered into the scope. To add these elements to the scope, select the required Class, Method or Field from the Outline View of the Java text editor and choose Indus -> Add to Scope from the context menu. For the Method and Field Specifications, the user is required to enter the name for the scope specification. In case of a class, the user can enter the scope name, change the class name specification to a regular expression matching several classes and pick the scope around the specified type, as indicated earlier. Upon choosing the scope name and the parameters, the scope specification for the chosen element is available to be included in the slice via the Slice the Program dialog box.

**Figure 13. Adding a class specification**

A screenshot showing how to add a class specification in the Indus-Kaveri tool.

### Appendix - Creating Contexts
As specified earlier, the user can specify a call-chain as a context for the slice. To do so, 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. Switch to Caller mode by pressing the Show Caller Hierarchy toolbar button. Expand the chain of method calls until a desired method is reached. Select the method starting the call chain and choose Indus -> Add to context from the context menu. This adds the call chain to the list of call-chain contexts available for slicing. This can then be picked from the Slice Configuration dialog before performing a slice.

Figure 14. Adding a call chain as context

Note that the created call chain contexts are not serialized and are lost when the session is closed.

Appendix - Adding custom root methods

Kaveri now allows the user to slice any Java program including applets, servlets, 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.

To add a method as a root method, select the method in the Java outline view and select Indus -> Mark as root method from the context menu. A marker is placed on the method, which is visible from the Java editor indicating that this is a root method. 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. The user can delete the unnecessary root methods from the Slice Configuration dialog.

Figure 15. Viewing the root methods for a project
Appendix - Performance

Kaveri has a JVM memory requirement for a smooth operation. Typically a memory specification of 200 MB for the Java virtual machine suffices for small projects. For large projects it is advisable to allocate more memory. To supply more memory to Eclipse pass the string "-vmargs -Xmx200m" as a parameter to the Eclipse launch application.

Bibliography


