

DCS 334 Computer Science Project
A Program Slicing Tool for Analysing Java Programs

Final Report
28 April 2003
Version 1

Prepared By: Kia Abdullah

Supervisor: Pasquale Malacaria

Abstract:

The Final Report presents the JSlicer: A Program Slicing Tool for Analysing Java Programs that has been developed as part of the DCS 334 Computer Science Project. This report outlines the aims of the system, details the basic concepts of program slicing and specifies the system requirements. It describes the design methodology that will be used to implement the tool and provides a critical review of the end product. A user manual and plans for future development are also presented.

Disclaimer

This report is submitted as part requirement for the degree of BSc in Computer Science at the University of London. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

Signature: _____

Acknowledgements

I would like to take this opportunity to thank my Project Supervisor Pasquale Malacaria for his support and advice throughout the course of the project.

Contents

| | |
|---|----|
| 1. Introduction | 5 |
| 2. Background | 6 |
| 2.1 Overview of Program Slicing | 6 |
| 2.2 Applications of Program Slicing | 9 |
| 2.3 Implementing Program Slicing | 10 |
| 3. Requirements Analysis | 15 |
| 3.1 Requirement Gathering | 15 |
| 3.2 Existing Solutions | 15 |
| 3.3 An Ideal Solution | 16 |
| 4. Specification | 17 |
| 5. Design and Implementation | 18 |
| 5.1 Class Diagram | 18 |
| 5.2 Program Structure | 19 |
| 5.3 Description of Components | 21 |
| 6. Testing | 29 |
| 6.1 Unit Testing | 29 |
| 6.2 Black Box Testing | 30 |
| 6.3 User/Beta Testing: | 30 |
| 6.4 Stress/Performance Testing: | 30 |
| 7. Evaluation | 32 |
| 7.1 Review of Project | 32 |
| 7.2 Future Extensions | 34 |
| 8. Conclusion | 36 |
| 9. References | 37 |
| | |
| Appendix A – User Manual | 38 |
| Appendix B – Design Document | 47 |
| Appendix C – System Manual | 49 |

1. Introduction

Aim: The aim of this project is to develop a Program Slicing tool (the JSlicer) to aid the analysis of Java programs. The slice of a program with respect to a program statement *S* is a projection of the program that includes only those program statements that may affect (either directly or indirectly) the values of the variables used at *S*. Slicing allows one to find semantically meaningful decompositions of programs, where the decompositions consist of program statements that are not textually contiguous.

The application of program slicing to the evaluation of high integrity software reduces the effort in examining software by allowing a software reviewer to focus attention on one computation at a time. Instead of examining the entire program, one needs to examine only the statements in the slice. By speeding up the process of locating relevant code for examination, a larger sample of software can be inspected with greater confidence that a relevant section of source code has not been missed. The concept of program slicing is described in detail in Section 2 – Background.

The remainder of this document has been divided into the following main sections:

Section 2: Background

This section provides detailed background information on Program Slicing. It details the basic concepts of the problem being addressed and describes the algorithms, techniques and methodology that will be used to implement the solution.

Section 3: Requirements Analysis

This section describes each phase of the requirements analysis. It provides an analysis of existing solutions and describes how an ideal solution would better meet the needs of the user.

Section 4: Specification

The Specification provides the terms of reference by which the success of the project will be measured. It specifies the functionality that should be incorporated into the completed version of the system.

Section 5: Design and Implementation

The Design section describes the high level architecture of the system. It details the key design features of the program and provides justification for the design decisions taken.

Section 6: Testing

This section details the types of testing that will be carried out on the system to ensure that it is fully functional and error-free.

Section 7: Evaluation

The Evaluation provides a critical review of the completed system and describes possible extensions of the system.

Section 8: Conclusion

Section 8 provides the final conclusion of the project. It outlines the overall success of the project and the achievements made.

2. Background

This section describes the background of the problem domain of the project. It details the basic concepts of the problem being addressed and describes the algorithms, techniques and methodology that will be used to implement the solution.

2.1 Overview of Program Slicing

The original definition of a program slice was presented by Mark Weiser in 1979 [11]. Since then, various different notions of program slices have been proposed, as well as a number of methods to compute them. In Weiser's terminology, a slicing criterion is a pair $\langle p, V \rangle$, where p is a program point and V is a subset of the program's variables.

In Weiser's work, a slice consists of all statements and predicates of the program that might affect the values of variables in V at point p . This is a more general kind of slice than is often needed. Rather than a slice taken with respect to program point p and an arbitrary variable, one is often interested in a slice taken with respect to a variable x that is *defined* or *used* at p . The criterion in the JSlicer will therefore be comprised of the line number of the statement and a set of variables defined or used at S .

- The value of a variable x *defined* at p is directly affected by the values of the variables used at p and by the loops and conditionals that enclose p .
- The value of a variable y *used* at p is directly affected by assignments to y that reach p and by the loops and conditionals that enclose p .

There are five basic concepts in Program Slicing:

- Predecessors
- Successors
- Backward Slicing
- Forward Slicing
- Chopping

Sections 2.1.1 to 2.1.5 use the code fragment given in Figure 1 to describe these five concepts in detail.

```
1  int n = 5;
2  int sum = 0;
3  int assign = 0;
4  int i = 0;

5  while(i<n)
   {
6     assign = i;
7     sum = sum+assign;
8     i++;
   }
9  System.out.println(sum);
```

Figure 1: Code Fragment

2.1.1 Predecessors

There are two types of predecessors: data predecessors and control predecessors.

Data predecessors are the assignments of values that may be used by a given statement. The data predecessors of a slicing criterion are all preceding statements that may affect the value of the variable in the criterion. Figure 2.a) shows the data predecessors of the criterion (7, sum).

Control predecessors are the control points that may affect whether a given statement gets executed. Figure 2.b) shows the control predecessors of the criterion (7, sum).

| | |
|---|--|
| <pre> 1 int n = 5; 2 int sum = 0; 3 int assign = 0; 4 int i = 0; 5 while(i<n) { 6 assign = i; 7 sum = sum+assign; 8 i++; } 9 System.out.println(sum); </pre> <p style="text-align: center;">a) Data Predecessors</p> | <pre> 1 int n = 5; 2 int sum = 0; 3 int assign = 0; 4 int i = 0; 5 while(i<n) { 6 assign = i; 7 sum = sum+assign; 8 i++; } 9 System.out.println(sum); </pre> <p style="text-align: center;">b) Control Predecessors</p> |
|---|--|

Figure 2: Predecessors of (7, sum)

2.1.2 Successors:

There are also two types of successors: data successors and control successors.

Data successors are the possible users of values assigned by a given statement. The data successors of a slice criterion are all statements that may be affected by the value of the variable in the criterion. Figure 3.a) shows the data successors of (2, sum).

Control successors are the statements whose execution depends on the control decision made at a given statement. Figure 3.b) shows the control successors of statement 5.

| | |
|---|--|
| <pre> 1 int n = 5; 2 int sum = 0; 3 int assign = 0; 4 int i = 0; 5 while(i<n) { 9 assign = i; 10 sum = sum+assign; 11 i++; } 9 System.out.println(sum); </pre> <p style="text-align: center;">a) Data Successors</p> | <pre> 1 int n = 5; 2 int sum = 0; 3 int assign = 0; 4 int i = 0; 5 while(i<n) { 6 assign = i; 7 sum = sum+assign; 8 i++; } 9 System.out.println(sum); </pre> <p style="text-align: center;">b) Control Successors</p> |
|---|--|

Figure 3: Data and Control Successors

2.1.3 Backward Slicing:

The backward slice from a program point p includes all points that may influence whether control reaches p , and all points that may influence the values of the variables used at p when control gets there. Figure 4.a) shows the backward slice from statement 5.

2.1.4 Forward Slicing:

The forward slice from a program point p includes all program points affected by the computation or conditional test at p . Figure 4.b) shows the forward slice from statement 5. Line 9 is included in this slice because line 5 affects whether line 7 is executed or not which in turn affects the value of `sum` at line 9.

| | |
|--|---|
| <pre> 1 int n = 5; 2 int sum = 0; 3 int assign = 0; 4 int i = 0; 5 while(i<n) { 6 assign = i; 7 sum = sum+assign; 8 i++; } 9 System.out.println(sum); </pre> <p style="text-align: center;">a) Backward Slice</p> | <pre> 1 int n = 5; 2 int sum = 0; 3 int assign = 0; 4 int i = 0; 5 while(i<n) { 6 assign = i; 7 sum = sum+assign; 8 i++; } 9 System.out.println(sum); </pre> <p style="text-align: center;">b) Forward Slice</p> |
|--|---|

Figure 4: Slices from Statement 5.

2.1.5 Chopping:

A chop shows the influence of one set of program points (the chop source) on another (the chop target). This query can be used to determine the information flow between two program points or to show that two parts of the program are independent. If the chop between two program points is empty then the two points are independent.

Figure 5.a) shows that the program points at lines 2 and 5 are independent; line 2 does not affect the execution of line 5 in any way. Figure 5.b) on the other hand shows that lines 1 and 6 are *not* independent as the chop between them is not empty.

| | |
|--|--|
| <pre> 1 int n = 5; 2 <u>int sum = 0;</u> 3 int assign = 0; 4 int i = 0; 5 while(i<n) { 6 assign = i; 7 sum = sum+assign; 8 i++; } 9 System.out.println(sum); </pre> <p style="text-align: center;">a) Source and Target are independent</p> | <pre> 1 <u>int n = 5;</u> 2 int sum = 0; 3 int assign = 0; 4 int i = 0; 5 while(i<n) { 6 <u>assign = i;</u> 7 sum = sum+assign; 8 i++; } 9 System.out.println(sum); </pre> <p style="text-align: center;">b) Source and Target are <i>not</i> independent</p> |
|--|--|

Figure 5: Chopping

2.2 Applications of Program Slicing

Program Slicing has applications in debugging, testing, program restructuring and program understanding. This section describes some examples of how slicing helps with these tasks:

Program Debugging

Instead of examining the entire program, one needs to examine only the statements in the slice. By speeding up the process of locating relevant code for examination, a larger sample of software can be inspected with greater confidence that a relevant section of source code has not been missed.

Testing

Suppose a proposed program modification only changes the value of variable v at program point p . If the forward slice with respect to v and p is disjoint from the coverage of regression test t , then test t does not have to be rerun. Suppose a coverage tool reveals that a use of variable v at program point p has not been tested. What input data is required in order to cover p ? The answer lies in the backward slice of v with respect to p .

Program Restructuring

Slicing isolates “computational threads”, which identifies logical components. The threads can be extracted and either replaced or used to create new programs.

Program Understanding

Slicing can be used to help programmers understand code. For example, a backward slice from a point in the program identifies all parts of the code that contribute to that point. A forward slice identifies all parts of the code that can be affected by the modification to the code at the slice point.

Program Specialization and Reuse

Executable slices can be thought of as specialized programs. Slices can be used to make code reuse more efficient. Instead of reusing an entire package, a slice can be used to identify only those parts that are really needed.

2.3 Implementing Program Slicing

This section presents the algorithms, techniques and methodology that will be used to implement the program slicing tool.

Ottenstein and Ottenstein [8] restated the problem introduced by Weiser in terms of a reachability problem in a Program Dependence Graph (PDG). When slicing a program that consists of a single procedure (Intraprocedural slicing), the slicing problem is simply a vertex reachability problem.

The work of Reps, Horwitz and Binkley [5] is concerned with the problem of Interprocedural slicing – generating a slice of an entire program where the slice crosses the boundaries of procedure calls. Their work follows the example of Ottenstein and Ottenstein by defining the slicing algorithm in terms of operations on a dependence graph. To solve the interprocedural slicing, Reps, Horwitz and Binkley use a System Dependence Graph (SDG). An SDG is a directed graph consisting of interconnected PDG's. In order to fully understand the operation of an SDG, we must first examine the construction of PDG's.

2.3.1 Program Dependence Graphs

A PDG is a directed graph with vertices corresponding to program statements and edges corresponding to data and control dependences.

- *Data dependence*
Node j is dependent on node i if there exists a variable x such that x is defined at node i ; x is referenced at node j and there exists a path from i to j with no intervening definitions of x .
- *Control dependence:*
Program point $P2$ is dependent on $P1$ if $P1$ is a condition and getting to $P2$ depends on whether $P1$ tests true or false.

The slicing criterion is identified with a vertex in the PDG and a slice corresponds to all PDG vertices from which the vertex under consideration can be reached.

To further explain the idea of a PDG, we shall use the following code fragment as an example:

```

1  n = 10
2  i = 1;
3  sum = 0;
4  product = 1;
5  while(i<=n)
   {
6   sum = sum + 1;
7   product = product * i;
8   i = i + 1;
   }
9  System.out.println(sum);
10 System.out.println(product);

```

Figure 6: Code Fragment.

Figure 7 below shows the corresponding PDG. The vertices with respect to slice criterion (10, product) are shown shaded. All statements that do not affect the variable product will be sliced away. The resulting statements in the slice are shown in Figure 8.

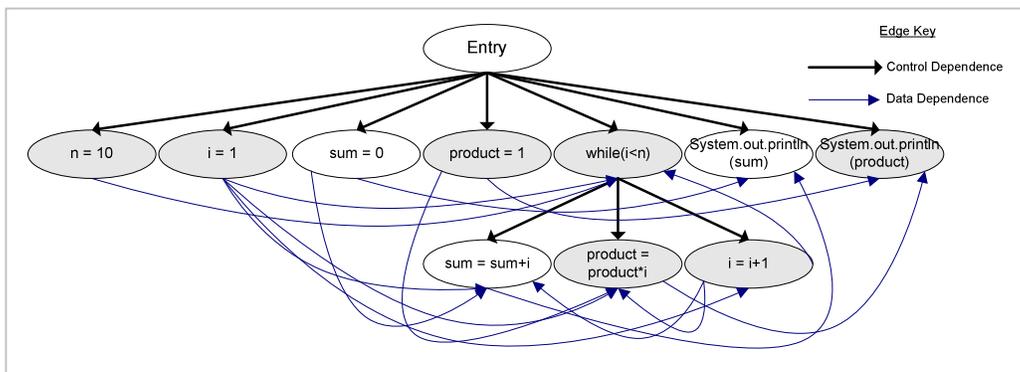


Figure 7: PDG of the program in Figure 6.

```

1  n = 10
2  i = 1;
3
4  product = 1;
5  while(i<=n)
   {
6   product = product * i;
7   i = i + 1;
   }
9
10 System.out.println(product);

```

Figure 8: Program slice of Figure 6 w.r.t (10, product).

2.3.2 System Dependence Graphs

Horwitz, Reps, and Binkley [5] introduce the notion of a System Dependence Graph (SDG) for slicing multi-procedure programs. Parameter passing between methods is modelled in Figure 9 and described below:

- At a, the calling procedure copies its actual parameters to temporary variables.
- The formal parameters of the called procedure are initialized using the corresponding temporary variables.
- Before returning, the called procedure copies the final values of the formal parameters to the temporary variables.
- After returning, the calling procedure updates the actual parameters by copying the values of the corresponding temporary variables.

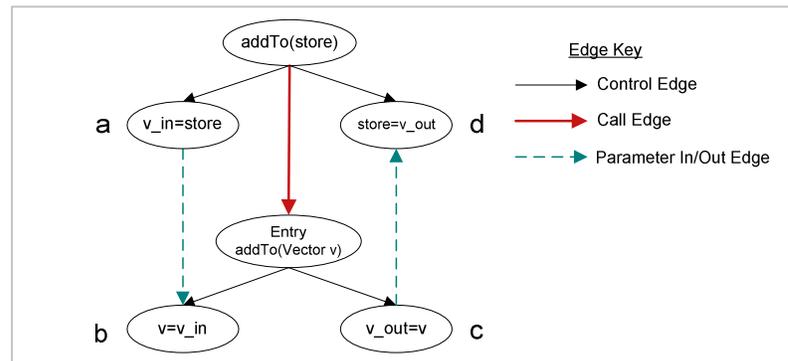


Figure 9: Parameter passing between two methods

An SDG contains a PDG for each method. There are several types of vertices and edges in an SDG which do not occur in the original PDGs:

- For each call statement, there is a call-site vertex in the SDG.
- There are actual-in and actual-out vertices which model the copying of actual parameters to/from temporary variables (a, and d in Figure 9).
- An entry vertex is added to each PDG, along with formal-in and formal-out vertices to model copying of formal parameters to/from temporary variables (b and c in Figure 9).

Actual-in and actual-out vertices are control dependent on the call-site vertex; formal-in and formal-out vertices are control dependent on the procedure's entry vertex. In addition to these intraprocedural dependence edges, an SDG contains the following interprocedural dependence edges:

- A call edge between a call-site vertex and the entry vertex of the called PDG.
- A parameter-in edge between corresponding actual-in and formal-in vertices.
- A parameter-out edge between corresponding formal-out and actual-out vertices.

The slicing is accomplished by traversing the graph in two phases.

Phase 1

Suppose that slicing starts at vertex s . The first phase determines all vertices from which s can be reached without descending into procedure calls. The transitive interprocedural dependence edges guarantee that calls can be side-stepped, without descending into them.

Phase 2

In the second phase, the algorithm descends into all previously side-stepped calls and determines the remaining vertices in the slice. Using interprocedural dataflow analysis, the sets of variables which can be referenced or modified by a procedure can be determined. This information can be used to eliminate actual-out and formal-out vertices for parameters that will never be modified, resulting in more precise slices.

The multi-procedure program shown in Figure 10 will be used to illustrate the idea how an SDG is constructed.

| | |
|--|---|
| <pre> public static void Entry() { 1 n = 10; 2 i = 1; 3 sum = 0; 4 product = 1; 5 while(i<=n) { 6 Add(sum, i); 7 Add(i, 1); 8 product = i; } 9 System.out.println(sum); 10 System.out.println(product); } </pre> | <pre> public static void Add(a,b) { 11 a = a + b; } </pre> |
|--|---|

Figure 10: Example of a multi-procedure program.

Figure 11 below shows the corresponding SDG. The vertices with respect to slice criterion (10, product) are shown shaded. Light shading indicates the vertices identified in the first phase of the algorithm, and dark shading indicates the vertices identified in the second phase.

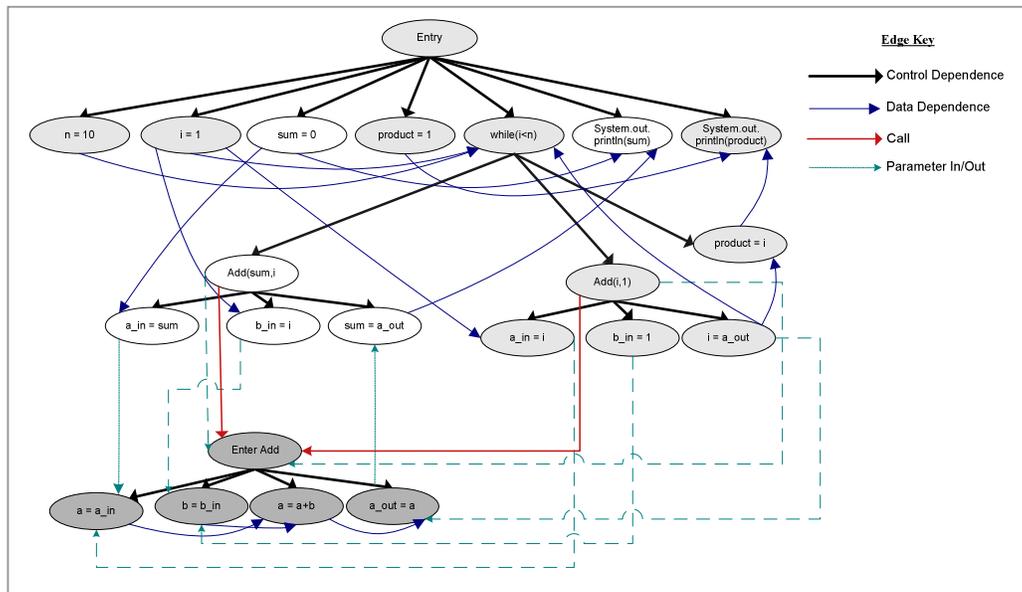


Figure 11: SDG of the program in Figure 10.

2.3.3 An Algorithm for Interprocedural Slicing

Figure 12 below shows the algorithm for interprocedural slicing given in [5].

```

procedure MarkVerticesOfSlice(G, S)
declare
  G: a system dependence graph
  S, S1: sets of vertices in G

begin
  //Phase 1: Slice without descending into called methods
  MarkReachingVertices(G, S, {def-order, parameter-in, call})

  //Phase 2: Slice called methods without ascending to call sites
  S1 = all marked vertices in G
  MarkReachingVertices(G, S, {def-order, parameter-out})
end

procedure MarkReachingVertices(G, V, Kinds)
declare
  G: a system dependence graph
  V: a set of vertices in G
  Kinds: a set of kinds of edges
  v, w: vertices in G
  WorkList: a set of vertices in G

begin
  WorkList = V
  while WorkList !=0 do
    //Select and remove a vertex v from WorkList
    Mark v
    for each unmarked vertex w such that there is an edge (w,v)
      whose kind is not in Kinds do
        Insert w into WorkList
      endfor
    endwhile
  end

```

Figure 12: Algorithm for Interprocedural Slicing

The procedure `MarkVerticesOfSlice` marks the vertices of the interprocedural slice G/S . The auxiliary procedure `MarkReachingVertices` marks all vertices in G from which there is a path to a vertex in V along edges of kinds other than those in the set $Kinds$.

The key element of the approach presented in [5] is the use of the linkage grammar's characteristic graph edges in the SDG. These edges represent transitive data dependencies from actual-in vertices to actual-out vertices due to procedure calls. The presence of such edges permits the slicing algorithm to sidestep the "calling context"; the algorithm can move "across" a call without having to descend into it.

As outlined in the previous section, the algorithm presented in Figure 12 computes the slice of a system dependence graph G with respect to vertex S in two phases. Phase 1 involves the traversal of flow edges, control edges, call edges and parameter-in edges but does *not* follow parameter-out edges. Phase two follows flow edges, control edges and parameter-out edges but does *not* follow call edges or parameter-in edges. The result of an interprocedural slice consists of the sets of vertices identified by Phase 1 and Phase 2, and the set of edges induced by this vertex set.

3. Requirements Analysis

This section describes each phase of the requirements analysis. It provides an analysis of existing solutions and describes how an ideal solution would better meet the needs of the user.

3.1 Requirement Gathering

Requirement gathering has been performed in two main stages. The first stage involved extensive research into the background material available about program slicing. Various materials such as research papers, tutorials and special-interest forums equipped me with a broad understanding of program slicing and the basic functions that should be present in a program slicing tool. The information gathered from this phase has been given in Section 2 – Background Material.

The second stage of requirements gathering involved the analysis of existing program slicing tools. It is important to evaluate existing solutions in order to ascertain which features are useful and which are ineffective. The features that meet the needs of the user well will be incorporated into the JSlicer whereas poorly designed features can be omitted or improved. The results of this stage of requirements analysis are outlined in Section 3.2 and have been used to construct the list of requirements given in the System Specification.

3.2 Existing Solutions

There are several existing program slicing tools designed to slice programs written in ANSI C. After extensive research, I have determined that there is only one existing tool for slicing Java programs. This section evaluates this tool along with a commercial tool designed to slice programs written in C.

3.2.1 Bandera

The Bandera Model Checker is a collection of program analysis, transformation, and visualisation components for model-checking Java source code.

The slicer in Bandera is only a small part of a much larger and more complex system. The input of the slicer is a Jimple representation of Java program, which is performed by JJJC (Java to Jimple to Java Compiler [13]), and slicing points specified by users.

The current slicer in Bandera can only produce Jimple code which then has to be translated back into Java. Along with this drawback, Bandera also has the following limitations:

- The slicer cannot be used as a standalone application.
- Case statements cannot be dealt with in the process of control dependency analysis.
- The current slicer is not considering any dependence analysis in the presence of exception handling. As a result of this limitation, the residual program will keep all original exceptions and their handlers.

An ideal system would integrate flow analysis into the current slicer to make virtual method resolving more precise. It would also involve case statements in dependency analysis, as this is a common structure used in Java.

3.2.2 CodeSurfer

CodeSurfer is a commercially available program slicing tool used to analyse programs written in ANSI C. CodeSurfer is based on system dependence graphs (described in Section 2.3.2) and includes the following functionality:

- A Dependence Analyser and Program Slicer
- Multiple Query Modes:
 - Point mode lets you pose queries in terms of points in the program, but the variables that are used/defined at the points are not distinguished.
 - Variable mode lets you pose queries in terms of the variables in the program, and lets you ask separately about declarations, assignments, uses, or references.
 - Variable-point mode lets you pose queries in terms of the variables that are used or defined at particular points in the program.
 - Function mode lets you pose queries in terms of the functions in the program.
- Pointer Analysis: CodeSurfer performs pointer analysis so that complex, indirect dependency relationships can be identified and navigated as easily.

The examination of these features is a useful way of setting a standard for the JSlicer. CodeSurfer uses System Dependence Graphs to implement program slicing and can therefore be used as a basis for the JSlicer. Of course, CodeSurfer only slices programs written in C and therefore has a limited amount of usefulness when designing a program slicing tool for Java programs.

3.3 An Ideal Solution

An ideal solution would use a combination of the best features from existing solutions to provide the user with the best system possible. An ideal solution would:

- Primarily be used as a standalone application but would have all important methods available as an API that can be used with other programs.
- Incorporate a Parser that recognises and parses any Java construct.
- Perform pointer analysis so that complex, indirect dependency relationships can be identified and navigated as easily.
- Create accurate slices as efficiently as possible.

Due to time constraints, the JSlicer may not incorporate every element stated above. A critical review of the system is provided in Section 7 – Evaluation. The review compares the final completed system against the Ideal Solution and the list of requirements given in the Specification.

4. Specification

The Specification provides the terms of reference by which the success of the project will be measured. It specifies the functionality that should be incorporated into the completed version of the system. Each requirement is listed in Table 1 shown below.

| No. | Description |
|-----|--|
| 1 | Start a new Project: This should allow the user to work with a set of files separate from existing/currently open files. |
| 2 | Open a File/Project: The user must be able to open one or more Java source files whilst working on a project. |
| 3 | Build a System Dependence Graph (SDG): The system should be able to build an SDG from the current source files. |
| 4 | Allow user to choose criteria: The user must be able to choose the slice criteria from a source file. |
| 5 | Allow user to choose slicing option: The user must be able to choose a slicing option through the user interface and run the project. |
| 6 | Determine data successors and predecessors: The system must be able to determine the Data Successors or Predecessors of a slice criterion. |
| 7 | Determine control successors/predecessors: The system must be able to determine the Data Successors or Predecessors of a slice criterion. |
| 8 | Perform backward slicing: The system must be able to perform interprocedural backward slicing on a given slice criterion. |
| 13 | Perform forward slicing: The system must be able to perform interprocedural backward slicing on a given slice criterion. |
| 14 | Perform chopping: The system should be able to perform interprocedural chopping on a given slice criterion. |
| 11 | Display slice to the user: The system must display a program slice in a manner easily identifiable by user. |
| 12 | Save current project: The system should save any open source files, the SDG and any open slices in a folder. |
| 13 | Provide an on-line help system: The application should be supported with a comprehensive on-line help manual covering every aspect of the application, and application interface. |
| 14 | API: It should be possible for third party developers to access the key (non-GUI) functionality via a package with appropriate public methods and Java documentation. |

Table 1: System Requirements

5. Design and Implementation

5.1 Class Diagram

An important requirement of the JSlicer is that it should be possible for third party developers to access the key functionality via packages with appropriate public methods. The Model-View-Controller design pattern will be used to ensure the decoupling of the API from the GUI implementation. This concept will be further explained in Section 5.2 – Program Structure.

The structure of the system has therefore been divided into a model package, a View package and a Controller package. A package overview is provided in Figure 13 showing these three packages along with the IO and JSlicer packages. For detailed class diagrams of each package please refer to Appendix 2 – Design Document.

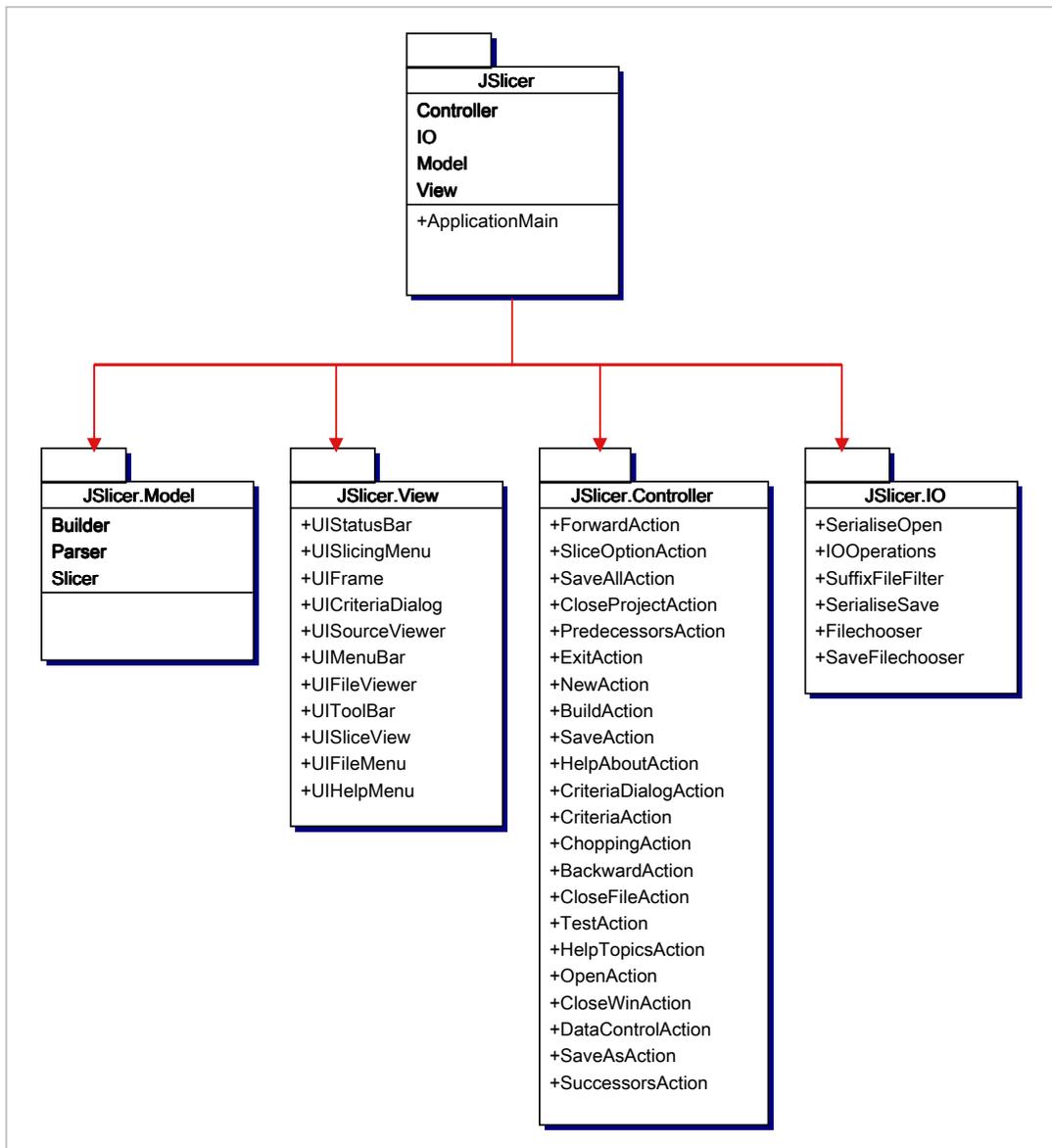


Figure 13: Package Overview

5.2 Program Structure

This section presents a detailed description of the program structure and key design features of the JSlicer.

5.2.1 Model-View-Controller

The Model-View-Controller (MVC) design pattern will be used to structure the JSlicer. The methods defined in the Parser, Builder and Slicer will be available as an API to third party programmers. In order to implement this, we must ensure that the API and GUI packages are fully decoupled. MVC incorporates several other design patterns in order to provide the programmer with a useful way of decoupling subsystems and ensuring flexibility.

In the MVC paradigm the user input, the modelling of the external world and the visual feedback to the user are explicitly separated and handled by three types of object (as shown in Figure 14), each specialized for its task.

- **Model:** The core of the application. This maintains the state and data of the application domain. It responds to requests for information about its state (usually from the view), and to instructions to change state (usually from the controller).
- **View:** The user interface which displays information about the model to the user.
- **Controller:** The user interface presented to the user to manipulate the application. It interprets the mouse and keyboard inputs from the user and commands the model and/or the view to change as appropriate.

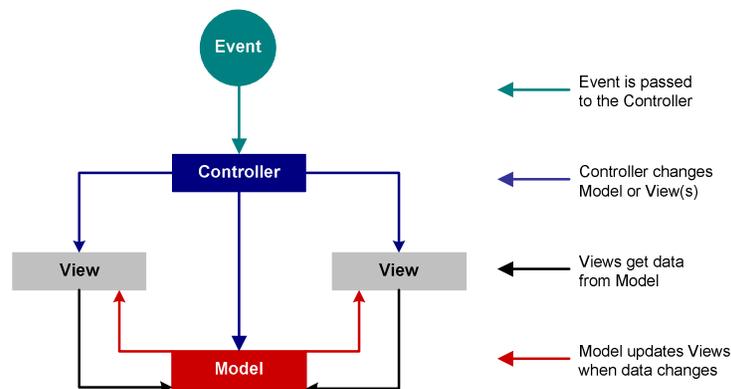


Figure 14: The Model-View-Controller Relationship

The model object knows about all the data that needs to be displayed but it knows nothing about the GUI or the manner in which the data are to be displayed. The data are accessed and manipulated through methods that are independent of the GUI. This ensures that the GUI and the API are fully decoupled.

The formal separation of these three tasks is an important notion that is particularly suited to the JSlicer where the basic behaviour can be embodied in abstract objects: Model, View and Controller. The MVC behaviour is then inherited, added to and modified as necessary to provide a flexible and powerful system. Other advantages offered by MVC are given in the next section.

5.2.2 Advantages of Model-View-Controller

Extensive research was carried out in order to determine the best program structure for the JSlicer. The MVC architecture was chosen for a number of reasons. The most significant advantages offered by MVC over other structural design patterns are given below:

Clarity of design

The public methods in the model stand as an API for all the commands available to manipulate its data and state. By glancing at the model's public method list, it should be easy to understand how to control the model's behaviour. This trait makes the system being developed easier to implement and maintain.

Efficient modularity

Efficient modularity of the design allows any of the components to be swapped in and out as the user or programmer desires. The components of the system are decoupled therefore changes to one aspect of the program do not affect other aspects. Also, development of the various components can progress in parallel, once the interface between the components is clearly defined.

Powerful user interfaces

Using the model's API, the user interface can combine the method calls when presenting commands to the user. Macros can be seen as a series of "standard" commands sent to the model, all triggered by a single user action. This allows the program to present the user with a cleaner, friendlier interface.

5.2.3 Interfaces

Another key feature used to implement the system is the use of interfaces. Interfaces are a powerful programming tool because they allow one to separate the definition of objects from their implementation, allowing objects to evolve without risk of breaking existing code.

Interfaces have been created for the classes in the Model package. These classes contain methods which will be available as an API to third party programmers. Interfaces provide programmers with the flexibility to change the way methods are implemented safely. Interfaces eliminate a major problem of class inheritance: the likelihood of breaking code when you make post-implementation changes to the design.

Class inheritance forces one to make the majority of design decisions when the class is first created. For example, if we define a method that expects an Integer argument, and a third party programmer wishes to change the data type to long, they cannot safely change the original class, because applications designed for classes derived from the original may not compile correctly. This problem can be magnified because a single base class can affect hundreds of subclasses.

One solution is to define a new method that overloads the original and that takes an argument of type Long. However, this might not be satisfactory because a derived class may need to override the method that takes the integer, and may not function properly if the method that takes a Long is not overridden also. Interfaces solve this problem by allowing one to publish an updated interface that accepts the new data type hence providing users with the flexibility they need to change the application

5.4 Description of Components

This section presents a detailed description of each software component contained within the architecture.

5.4.1 Model.Parser

The Parser package is responsible for parsing Java source code into a tree form representing its internal structure. It does this in two phases using a combination of an automatic parser and a hand crafted parser. Figure 15 shows an overview of the processing detail of the Parser which is further explained in the subsequent sections.



Figure 15: Processing detail of the Parser

Phase 1: The first pass of the java file is made by an automatic parser generated by a parser generator. ANLTR (Another Tool for Language Recognition) [12] generates parsers based on a grammar defined by the user. The parser in Phase 1 creates an Abstract Syntax Tree for each java file based on tokens defined in the grammar.

An automatic parser is used to increase the efficiency of the parsing. One drawback however is that it does not provide us with the flexibility we need; the output trees need to be changed from a token based format into a statement based format. This is carried out by Phase 2 of the parsing process.

Phase 2: The second phase of tree construction uses a hand crafted parser to transform the token based tree into a statement based tree. Each statement in the java file corresponds to a node in the output tree. This tree is then passed to the Builder.

Figure 16. a) shows the graph fragment produced by Phase 1 after parsing the statement: `double x1 =xVect*v.x`. Node 2 in Figure 16.b) shows the output of Phase 2 after parsing the output produced by Phase 1.

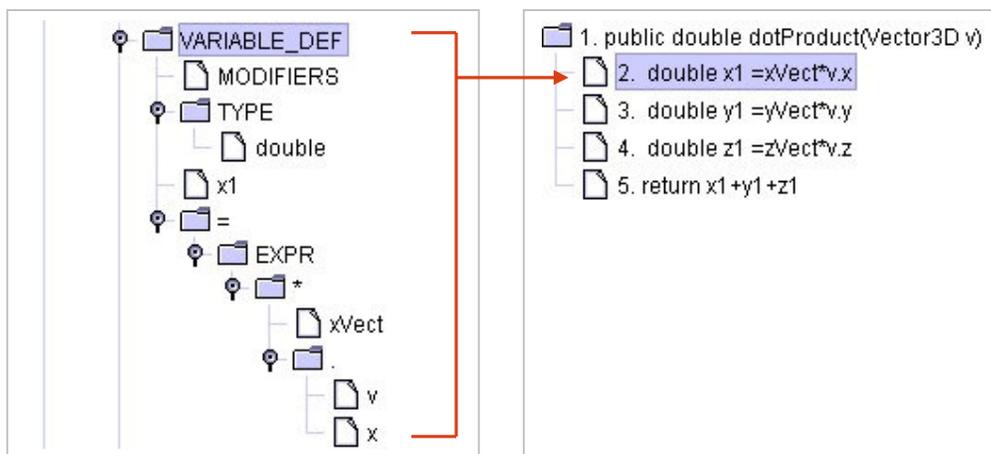


Figure 16 a) Output of Phase 1

Figure 16 b) Output of Phase 2

5.4.2 Model.Builder

The Builder package takes one or more trees created by the Parser as input. These trees consist of a set of Nodes. The information contained within a Node is vital to the construction of the Program Dependence Graphs and the System Dependence Graph.

A Node object is specified below:

```
public Node(int lineNo, String statement, String defines,
            Vector uses, Vector calls, String PDGParent)
```

lineNo: A unique identifier for each node.
statement: The Java statement that this node represents.
defines: The variable that this node defines.
uses: All variables that this node uses.
calls: The names of all methods that this node calls.
PDGParent: The name of the method that this node belongs to.

The Builder performs dependence analysis on the trees created by the Parser and adds data and control dependence edges between the nodes in order to create a set of Program Dependence Graphs (PDGs) which are then used to build the System Dependence Graph. Figure 17 overleaf shows the algorithm used to build a PDG.

Figure 17 shows the PDG Building Algorithm.

```

procedure PDG(TreeList)
declare
  TreeList: set of trees produced by the Parser
  Pos: A counter that is used to iterate through TreeList

begin
  while TreeList !=0 do
    //Select and remove a tree from TreeList
    Tree: the tree in TreeList at position pos
    NodeList: The nodes contained in Tree
    while NodeList !=0 do
      Select and remove a node n from NodeList
      Defines: A string specifying the value defined at n
      addEdges(NodeList, n, Defines, pos);
    endwhile
    pos++
  endwhile
end

procedure addEdges(NodeList, n, define, position)
declare
  NodeList: A list of all nodes in a tree
  n: The node being tested for dependencies
  defines: The variable that is defined at n
  position: The position in NodeList where n resides
  counter: position+1

begin
  while NodeList !=0 do
    //Select and remove a node n2 to compare with n
    n2: The node in NodeList at counter
    usesList: A list of all variable used at node n2
    parentList: A list of all parent nodes of node n2
    while useList !=0 do
      //Select and remove a use variable use from useList
      if use = defines
        Add a data edge between n and n2
      endif
    endwhile

    while parentList !=0 do
      //Select and remove a parentNode from parentList
      if parentNode = n
        Add a control edge between n and n2
      endif
    endwhile
    counter++
  endwhile
end

```

Figure 17: PDG Building Algorithm

The set of PDGs created by the algorithm given in Figure 15 is used to construct the System Dependence Graph. During the creation of the PDG's, if the Builder comes across a node that calls another method, it creates a Call object and adds it to a Vector callRecord.

A Call object is specified below:

```
Call(String callingPDG, int lineNumber, calledPDG)
```

calledPDG The PDG that contains the node that is making the call.

callingPDG The PDG that is being called.

lineNumber The line number of the node that is making the call.

The SDG takes the callRecord and the PDGList in as parameters. It then uses the algorithm given in Figure 18 to add the appropriate edges between the PDGs to create the SDG.

```
procedure SDG(PDGList, callRecord)
declare
  PDGList: set of PDGs created for the current project
  CallRecord: all the calls made between the various PDGs

begin
  while callRecord !=0 do
    //Select and remove a Call c from CallRecord
    CalledPDG: The PDG that is making the call
    CallingPDG: The PDG that is making the call
    CallingPDG: The PDG that is making the call
    lineNumber: The line number of the node making the call
    n: The node in callingPDG with line Number=lineNumber
    n2: The parent node of callingPDG

    Add Call Edge from n to n2
    //Create parameter in nodes for values passed from n to n2
    Add control edges from n to act_in and act_out nodes

    //Create parameter out nodes for parameters changed by n2
    Add control edges from n2 to formal_in and formal_out nodes

    Add parameter_in edges from act_in to formal_in nodes
    Add parameter_out edges from formal_out to act_out nodes
    Reset any dependencies
  endwhile
end
```

Figure 18: SDG Building Algorithm

5.4.3 Model.Slicer

The Slicer is responsible for performing slicing operations on the System Dependence Graph. The Slice class creates a program slice using the algorithm presented in Figure 12 in Section 2- Background.

A Slice object is specified below.

```
public Slice(SDG sdg, Vector criteria, boolean isBackward,
boolean isForward, boolean isData, boolean isControl)
```

| | |
|------------|---|
| sdg | The current build of the SDG |
| criteria | A record of all the criteria that needs to be sliced upon. |
| isBackward | Set to true if the slice is a <i>backward</i> slice. |
| isForward | Set to true if the slice is a <i>forward</i> slice. |
| isData | True if the slice is determining <i>data</i> successors or predecessors. |
| isControl | True if the slice is determining <i>control</i> successors or predecessors. |

The criteria are used to slice upon the SDG. Much of the same processing is needed for each type of slice therefore we use a generalised “Slice” object with boolean flags to specify what type of slice is needed. This prevents the repetition of the same code in four different classes.

Java is an object-oriented language and therefore we are interested in statements that *define* and *use* variables representing objects. If we are slicing upon a variable *v* that refers to an object, the JSlicer currently obtains the statements that either assign to *v* or use *v*. The JSlicer allows the user to choose whether they want to slice with respect to “Definitions Only” or “Definitions and Uses”. The code fragment given in Figure 19 is used to describe these two options.

```
1. JFrame frame = new JFrame("JFrame");
2. Frame.setSize(300, 300);
3. String str = frame paramString();
4. System.out.println(frame.getTitle());
```

Figure 19: Code Fragment

Option 1: Definitions Only

When performing a slice on a variable *v* at program point *p*, the slice will include only *assignments* of values that *v* may affect or may be affected by. It will *not* include any *uses* of *v* prior to *p*. For example, using the code fragment given in Figure 19, if we perform a backward slice on Line 4, we will obtain only Line 1 as this line defines *frame*.

Option 2: Definitions and Uses

When performing a slice on a variable *v* at program point *p*, the slice will include both *assignments* and *uses* of values that *v* may affect or may be affected by. For example, using the code fragment given in Figure 19, if we perform a backward slice on Line 4, we obtain Lines 1, 2 and 3. Line 1 assigns to *frame* and Lines 2 and 3 use *frame*.

This option is particularly useful for program understanding. It helps a programmer see the flow of control through a program and allows them to examine every line that uses the variable *v* in the criteria.

It would be useful to further separate *uses* of *v* into two categories:

1. Uses that may change the object in some way.
2. Uses that cannot change the object.

For example, the use of *frame* on Line 2 *changes* *frame* in some way whereas the use on Line 3 does not. This separation would present the user with an option similar to Option 2 but would yield a more precise slice. The process of separation would require examination of the object's constructor and whether the statement calls a method that affects one of the data members of the object.

This third option has not been implemented in the JSlicer. Determining whether a use of an object actually changes it is a long and complex process. After discussion with the Project Advisor, it was decided that this level of complexity is beyond the scope of the project. Even if extra time and effort was spent to implement this option, the trade-off between heightened accuracy and slower slicing may not be worth it.

The separation of the two types of uses is a possible further extension of the JSlicer and is discussed in more detail in Section 7.2.1 - Future Extensions on Page 34.

5.4.4 View

This package contains all the user interface classes. It is responsible for displaying information about the model to the user. The GUI methods of the system are separate from those in the Model in order to keep the API methods fully decoupled from the GUI methods. This facilitates the re-use of API methods.

5.4.5 Controller

The Controller is the interface presented to the user to manipulate the application. It interprets the mouse and keyboard inputs from the user and commands the model and/or the view to change as appropriate. Each toolbar and menu item has an associated Action class in the Controller package. Each Action class has an `actionPerformed()` method which carries out the required task.

For example, if the user presses “Open” on the toolbar or menu, the Controller will take this input command and respond by creating an instance of the `OpenAction` class. This class will invoke its `actionPerformed()` method which calls the appropriate methods in the IO package.

It is important to note that the *same* action class is called whether a command is made through the toolbar or its corresponding menu item. This eliminates the need to repeat the same code twice. This concept is illustrated by Figure 20.

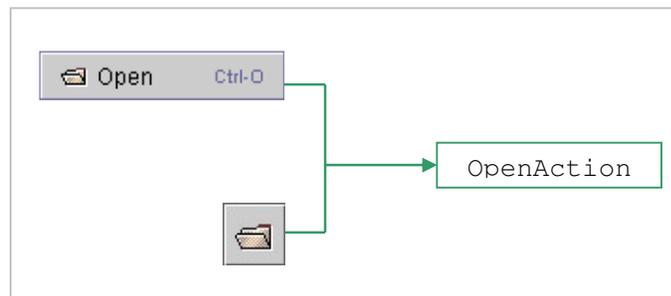


Figure 20: Menu Items and Toolbar Icons call the same action

5.4.6 IO

The IO Package is responsible for all input and output operations. It receives commands from the Controller package and sends commands to the View package.

Input

The JSlicer can be used to analyse any java source file as long as it adheres to one convention: any variable that is not declared or initialised in a method must be declared at the *bottom* of the file.

When a java file is opened in the JSlicer, every statement is given a unique number in order to identify it within the System Dependence Graph. This allows us to map the nodes in a particular slice back onto the source file. This obviously means that the number given to a statement in a source file must match the number given to its corresponding node in the SDG.

The automatic parser generated by ANTLR parses the original java source file, not the numbered one. It parses all variables that do not belong to a method *after* it has parsed all other statements in the java source file. Thus, the nodes created for these statements will be created *after* all other nodes. This means that in order to ensure that the number given to a statement and its corresponding node are consistent; we must declare the global variables at the *bottom* of the file after all other statements.

Saving and Loading

The system saves the currently open java files; all slice files and the current build of the SDG in a folder under the name chosen by the user. All java and slice files are saved in text format. The SDG is saved in serialised format.

The system also creates a .root file for each project. For example, if the user called the current project "myProject", the system would create a myProject.root file. Subsequently, if the user wants to open the project, they can simply open the root file which opens all files related to the project.

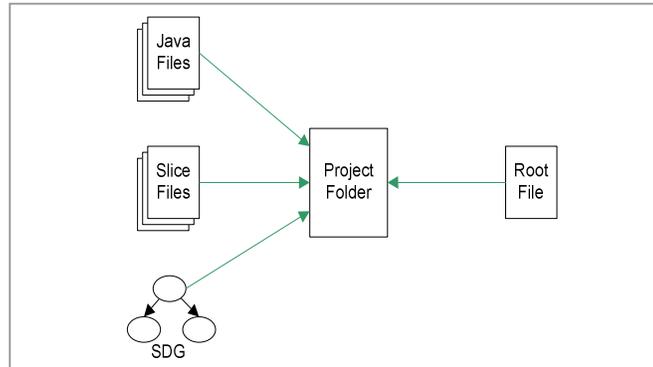


Figure 21: The Save operation performed by the IO package

6. Testing

This stage of the project development cycle involves four types of testing. *White box testing* was conducted using JUnit. *Black box testing* was conducted with test cases derived from the Specification. *User/Beta Testing* was carried out by external users. The fourth type of testing is *Stress Testing* which was conducted to test the performance of the system under extreme circumstances.

6.1 Unit Testing

JUnit testing was carried out on the system to ensure that all the API methods worked correctly. The following types of testing were carried out:

- Correctness Testing – Ensuring each method performs the required operation.
- Error Testing – Testing the effects of invalid arguments being sent to the each method.

JUnit provides a graphical output of the test results. The following is a screen shot of the result obtained from testing the Add Criteria feature.

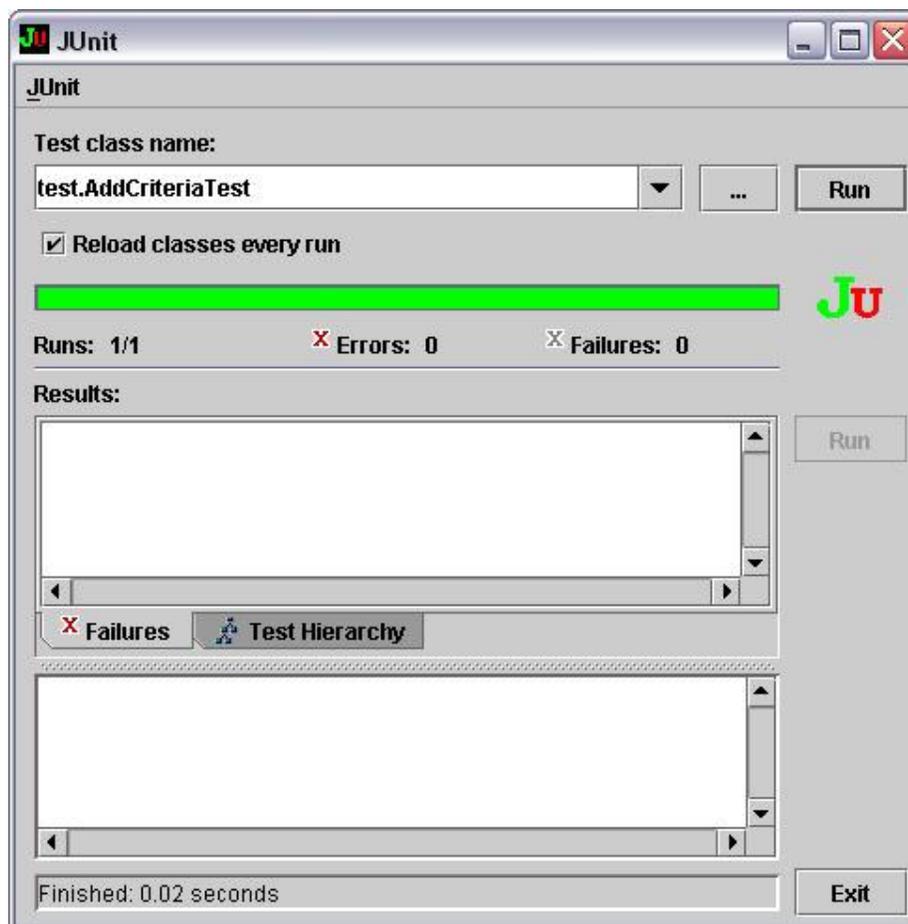


Figure 22: The JUnit results of the “Add criteria” test.

6.2 Black Box Testing

Black box testing was focused on the Input-Output behaviour of the system. Test cases were drawn from the Specification. Each test was conducted with regards to a particular requirement. If for any given input, we can predict the output, then the module passes the test. Black box testing helped uncover a number of bugs. The series of tests were repeated after each error was fixed.

6.3 User/Beta Testing:

User testing was carried out by giving the jar file to external users along with a list of the functional requirements and asking them to assess the usability of the system. The 'external users' included other computer science students as well as a number of friends and relatives. Feedback from the users helped improve the quality of the non-functional requirements of the system but did not uncover any bugs.

6.4 Stress/Performance Testing:

The API was stress tested against different numbers of lines of code (LOC) and increasing depth of method calls.

6.4.1 Performance against LOC

The LOC affects the system as it has to analyse every line of code in order to check if it should be included in the slice. The graph in Figure 23 shows the LOC against the execution time of a backward slice operation. The maximum number of LOC that could be included was two thousand before there was a considerable deterioration in performance. Please note that these tests were conducted on a set of source files which did not contain any interprocedural method calls. The performance of the system with respect to method calls are discussed in the next section.

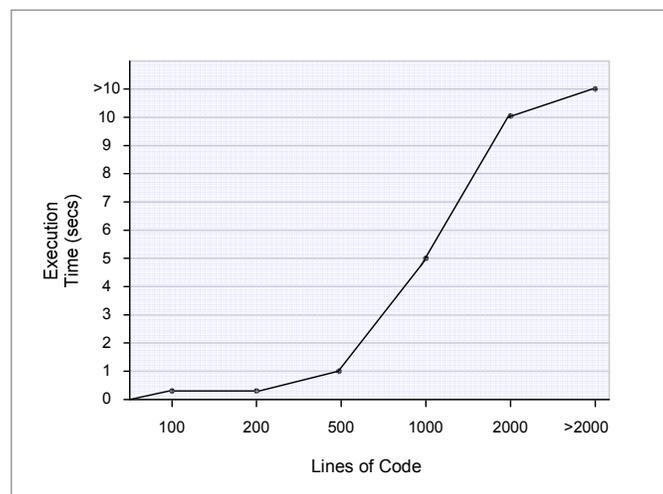


Figure 23: Execution time of a Backward Slice against LOC

6.4.2 Performance against Depth of Method Calls

By the “depth of method calls”, we mean the number of transitive dependencies between methods in the slice. That is, if we know line A is in the slice and it calls method B which calls method C which calls method D, the depth of method calls is three and we must analyse each method.

The graph in Figure 24 shows the depth of method calls against the execution time of a backward slice operation. The tests corresponding to each “depth” value were run across a different number of files with the number of LOC varying from 100 to 1000. The execution times were recorded and averaged to produce the time values shown in Figure 24. The maximum depth of method calls that could be handled was ten before there was a considerable deterioration in performance.

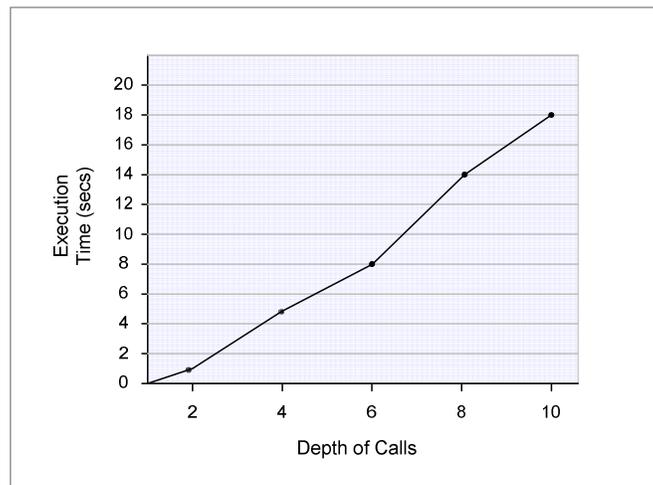


Figure 24: Execution time of a Backward Slice against Depth of Calls

7. Evaluation

The Evaluation provides an overall review of the system. It measures the success of the project against the requirements given in the Specification and describes possible future extensions to the completed system.

7.1 Review of Project

7.1.1 Final System versus Requirements Specification

The requirements specification (Section 4, Page 17) lists fourteen functional requirements that the completed system was expected to meet. All fourteen requirements have been successfully incorporated into the final system. The only requirement that has not been fulfilled to my satisfaction is Requirement 10:

Perform chopping: The system should be able to perform interprocedural chopping on a given slice criterion.

The final system chops between two program points that belong to the same method but does not perform interprocedural chopping (i.e. it does not chop across methods). This is due to time constraints and the fact that sometimes, even the best laid plans go astray. Though the implementation of this requirement is not entirely satisfactory on a personal level, it is acceptable as the main aim of the project was to implement the slicing correctly. All slicing operations and other requirements have withstood thorough testing and work fully and correctly.

7.1.2 Final System versus the Ideal Solution

The Ideal Solution (Section 3.3, page 16) specifies four key features of the ideal system. Each key feature is discussed below.

1. Primarily be used as a standalone application but would have all important methods available as an API that can be used with other programs.

The final system meets this criterion. It is designed to be a standalone application but all important methods are available as part of an API. The effective use of the MVC architecture makes reusing the methods easier for third-party programmers.

2. Incorporate a Parser that recognises and parses any Java construct.

The final system uses a combination of a generated parser and a hand crafted parser. Testing a system like JSlicer is difficult. Every single Java file cannot be tested as there is a multitude of different ways one can construct a Java statement. Thorough testing has been carried out with a variety of Java source files including those created by myself, those taken from the Java sun site as well files created by fellow students. Though we cannot say with 100% certainty that the Parser can deal with every single Java construct, it has dealt with every test file correctly and therefore I can confidently say that it recognises and parses any Java construct.

3. Perform pointer analysis so that complex, indirect dependency relationships can be identified and navigated easily.

The final system does perform pointer analysis in order to identify complex, indirect dependency relationships. It takes aliasing into consideration so that changes to an object through two or more different pointers are detected. It analyses all method calls in a procedure so that indirect dependencies can be detected and included in the slice.

4. Create accurate slices as efficiently as possible.

The final system does create accurate slices with respect to the two slicing options (Definitions Only, Definitions and Uses) but could be made more precise with a third option that includes only those uses of a variable that can change it. Please refer to Section 4.4.4, Page 25 for details on this third slicing option. Implementing this third option is a possible future extension of the project and is discussed in detail in Section 7.2 – Future Extensions.

Another possible extension is to speed up the slicing. The slicing is currently fairly slow and the performance of the system begins to deteriorate with increasing depth of method calls. This is also discussed in Section 7.2 – Future Extensions.

7.2 Future Extensions

Future extensions to the project could include the following:

7.2.1 Implementing a more precise Slice

Section 4.4.4 on Page 25 describes a third slicing option in which *uses* of a variable *v* which refers to an object can be separated into two categories:

3. Uses that may change the object in some way.
4. Uses that cannot change the object.

This option was not implemented in the final system as it is a long and complex process which would have slowed down the slicing. This section discusses how one would begin implementing this option.

In order to distinguish between the two types of use, the System Dependence Graph would have to be extended to include data members of every object in the files being sliced. Each use of a variable referring to an object would have to be examined in order to determine whether it calls a method that changes one of the data members of the object. Consider a variable *frame* that refers to a *JFrame* object. The statement *frame.setSize()* would require the examination of the *setSize()* method in the *JFrame* class in order to determine if it changes any data members of the *JFrame* object.

Liang and Harrold [7] present an SDG for object oriented software that is more precise than the original SDG. The Liang and Harrold SDG contains the following vertices:

- The call site contains *actual-in data member* vertices to represent data members that are referenced.
- The call site contains *actual-out data member* vertices to represent data members that are modified, by the called method.
- The *method entry* vertex contains *formal-in data member* vertices to represent those data members referenced in the method
- The *method entry* vertex contains *formal-out data member* vertices to represent those data members modified in the method.

Like other parameter vertices, the *actual-in/-out data member* vertices are control dependent on the method call vertex, and the *formal-in/-out data member* vertices are control dependent on the method entry vertex.

Formal-in data member vertices are labelled with assignments that copy the value of data members from an object into the scope of the method at the beginning of the method, and *formal-out data member* vertices are labelled with assignments that copy the value of data members from the scope of the method into the object at the end of the method.

Under this approach, the data dependences related to data members of the class in a method are computed in the same way as the data dependences for a procedure. This is a complex process that can be implemented in the future in order to obtain more precise slices. The JSlicer is flexible due to its modular program structure and can be modified in order to implement this feature.

7.2.2 Increasing the Efficiency of the slicing algorithm

Slicing in essence is a slow process so what can one do to speed it up? There has been some research into this area carried out by Sagiv and Rosay [9]. They have developed an algorithm which they claim is asymptotically faster than the Horwitz-Reps-Binkley (HRB) algorithm which was used to implement the JSlicer. The values listed below are used to express the respective costs of the HRB and Sagiv and Rosay algorithms

P: The number of procedures in the program.
Sites: The maximum number of call sites in any procedure.
TotalSites: The total number of call sites in the program.
E: The maximum number of control and flow edges in any procedure's PDG.
Params: The maximum number of formal-in vertices in any procedure's PDG.

The cost of interprocedural slicing using the HRB algorithm is denoted by:
 $O((TotalSites \times E \times Params) + (TotalSites \times Sites^2 \times Params^4)).$

The cost of the algorithm developed in [9] is bounded by:
 $O((P \times E \times Params) + (TotalSites \times Params^3)).$

Under the assumption that the total number of call sites in a program is much greater than the number of procedures, each term of the cost of the new algorithm is asymptotically smaller than the corresponding term of the cost of the HRB-summary algorithm. Implementing the Sagiv and Rosay algorithm rather than the more established HRB algorithm could serve to make the slicing faster. Implementing this algorithm is a possible future extension of the JSlicer.

8. Conclusion

The final version of the system successfully meets the aims and objectives of the project outlined at the beginning of the report. The JSlicer has been implemented successfully and adheres to the Specification set out in Section 4. It closely follows the criteria set out for an ideal solution and has proved to be a genuinely useful tool as I have used working versions to locate semantic errors in my own programming.

The project has proved to be an extremely challenging endeavour but at the same time, it has been immensely rewarding. It has given me an opportunity to improve my programming and design skills. It has enhanced my confidence and capability to investigate, analyse and design software systems. I have researched into and implemented design patterns and now know the importance of good program structure when developing a stable but flexible application.

During the course of the project, I have developed a genuine interest in program slicing and the wider field of program analysis. Whilst researching this field, I came across various program analysis tools but was surprised to see that there were not any program slicing tools for java programs. There is plenty of research into the field and therefore I had access to a multitude of research material but there were no applications which actually applied all this research. I was initially daunted by this fact but it also motivated me as it was a challenge. As a result, I have produced a tool that is both unique and useful and I am genuinely proud of my achievements.

9. References

Research Papers:

- [1] Anderson, P., Teitelbaum, T. Dependence Graphs and Program Slicing. *CodeSurfer Technology Overview*, 2001.
- [2] Anderson, P., Teitelbaum, T. Software Inspection Using CodeSurfer. *Proceeding of the First Workshop of Inspection in Software Engineering*, 2001.
- [3] Chen, Z., Baowen, X. Slicing Object Oriented Java Programs. *ACM SIGPLAN Notices*, 2001.
- [4] Dwyer, M. Extracting Finite-state Models from Java Source Code. *International Conference on Software Engineering*. 2000.
- [5] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems* 12, 1990,
- [6] Horwitz, S., Reps, T. The Use Program Dependence Graphs in Software Engineering, *Proceedings of the 14th International Conference on Software Engineering*, 1992.
- [7] Liang, D., Harrold, M. Slicing Objects Using System Dependence Graph. *Proceedings of the International Conference on Software Maintenance*, 1998.
- [8] Ottenstein, K., Ottenstein, L. The program dependence graph in a software development environment. *Proceedings of the conference on Programming language design and implementation*, 1990.
- [9] Sagiv, M., Rosay, G. Speeding up Slicing. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 1994.
- [10] Tip, F. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121-89, 1995.
- [11] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering*, 1984.

Web-sites:

- [12] ANTLR: www.antlr.org
- [13] Artima: www.artima.com/designtechniques
- [14] Bandera: www.cis.ksu.edu/santos/bandera/papers/slicing.html
- [15] CodeSurfer: www.grammatech.com
- [16] Pattern Depot: www.patterndepot.com/put/8/JavaPatterns.htm
- [17] Research Index: <http://citeseer.nj.nec.com>
- [18] T. Reps: www.cs.wisc.edu/~reps/

Appendix A – User Manual

Contents

| | |
|--|----|
| 1. Getting Started | 39 |
| 1.1 Toolbar | 40 |
| 1.2 Menus | 40 |
| 1.3 File Viewer | 41 |
| 1.4 Source Viewer | 41 |
| 1.5 Status Bar | 42 |
| 2. New: Building a Project | 43 |
| 3. Open and Browse | 43 |
| 3.1 Open | 43 |
| 3.2 Browse | 43 |
| 4. Saving | 44 |
| 5. Build a System Dependence Graph | 44 |
| 6. Criteria | 44 |
| 6.1 Choose Criteria | 44 |
| 6.2 Clear Criteria | 44 |
| 7. Slicing Operations | 45 |
| 7.1 Predecessors | 45 |
| 7.1 Successors | 45 |
| 7.3 Backward Slicing | 45 |
| 7.4 Forward Slicing | 46 |
| 7.5 Chopping | 46 |

A1. Getting Started

This User Manual accompanies the JSlicer Program Slicing Tool. It can be used to find out about the various features of the JSlicer. To get you started, we shall go through the basic components of the user interface. Figure A1 below shows a labelled screenshot of the first screen you will see when you run the JSlicer. Each subsection of Section A1 describes each of the labelled components shown below.

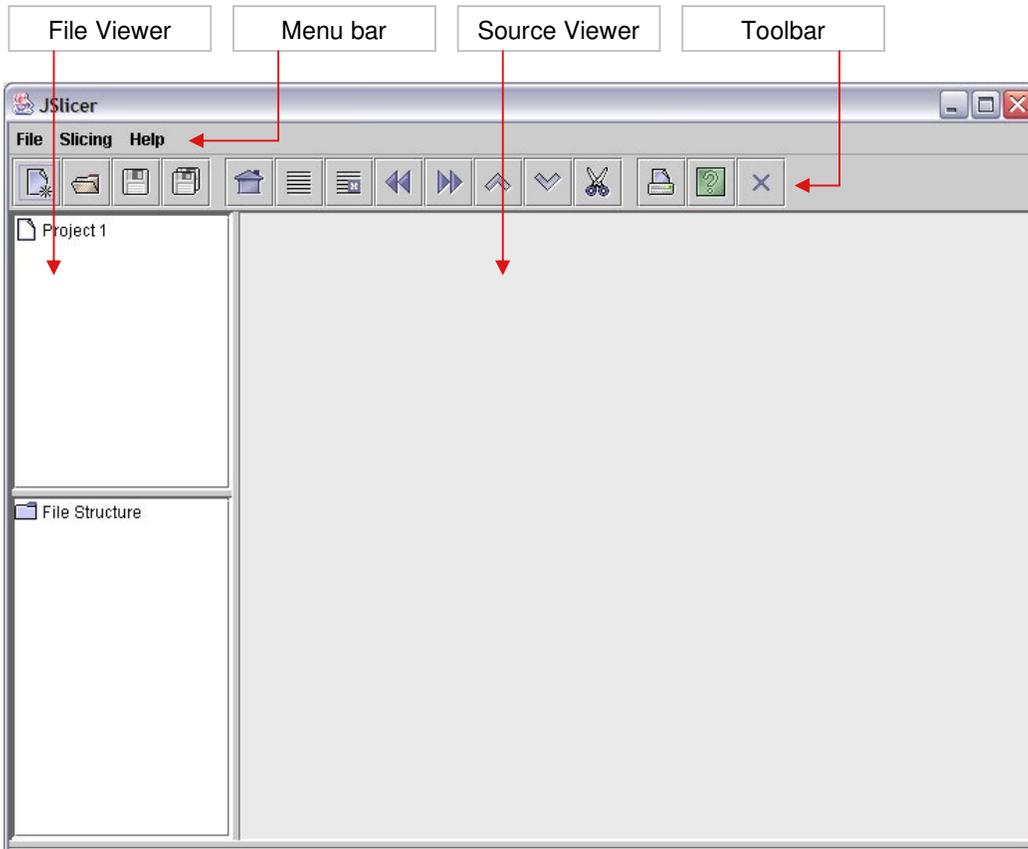


Figure A1: The First Screen

A1.1 Toolbar

The toolbar is a collection of buttons that you use to carry out commands. Each button on the toolbar shown in Figure A2 is described briefly in this section. For a more detailed explanation of each command, please see the respective sections.



Figure A2: The JSlicer Toolbar

-  **New:** Starts a new JSlicer project.
-  **Open:** Can be used to open existing Java files, slice files or projects
-  **Save:** Can be used to save an individual slice file in a chosen directory.
-  **Save All:** Saves the current project in a chosen directory.
-  **Build SDG:** Builds the System Dependence Graph.
-  **Add Criteria:** Adds the variables at a selected statement to the criteria.
-  **Clear Criteria:** Can be used to clear all criteria that have been set so far.
-  **Backward Slicing:** Backward slices the currently open Java files.
-  **Forward Slicing:** Forward slices the currently open Java files.
-  **Chopping:** Performs chopping on the currently open Java files.
-  **Predecessors:** Determines the Successors of the set criteria using the SDG.
-  **Successors:** Determines the Successors of the set criteria using the SDG.
-  **Print:** Prints the current Java file or Slice file.
-  **Help:** Starts up the Help system but of course, you already knew that!
-  **Close Project:** Closes the current project.

A1.2 Menus

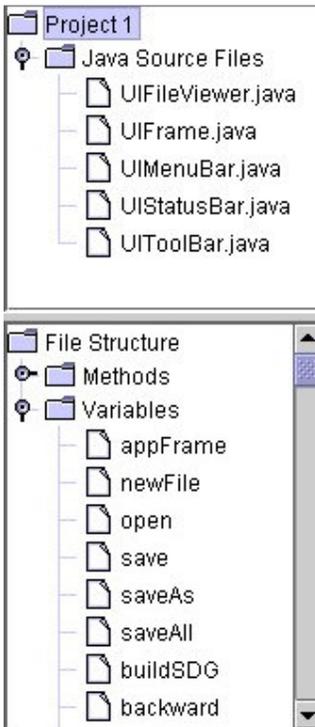
There are three main menus of the JSlicer.

The '**File**' menu can be used to perform all basic File operations such as open and save.

The '**Slicing**' menu can be used to perform all the main Slicing operations. For details on what each of the slicing commands do, please see Section 7 - Slicing Operations.

The '**Help**' menu can be used to start up the Help System.

A1.3 File Viewer



The top component of the File Viewer summarises every open Java file and every Slice file contained within the current project. Clicking on a file name will open the corresponding file in the Source Viewer.

The bottom component of the File Viewer shows every method and global variable contained within the file that is currently open in the Source Viewer.

Figure A3: The File Viewer

A1.4 Source Viewer

The Source Viewer is used to view the contents of a Java source file or a Slice file.



Figure A4: The Source Viewer

A1.5 Status Bar

Figure A5 shows a screenshot of the JSlicer after a Backward Slice has been performed on the file UIFrame.java. It is the Status Bar's job to provide you with a summary of the most recent slice that has been performed. The Status Bar is situated at the bottom of the application as shown in Figure A5.

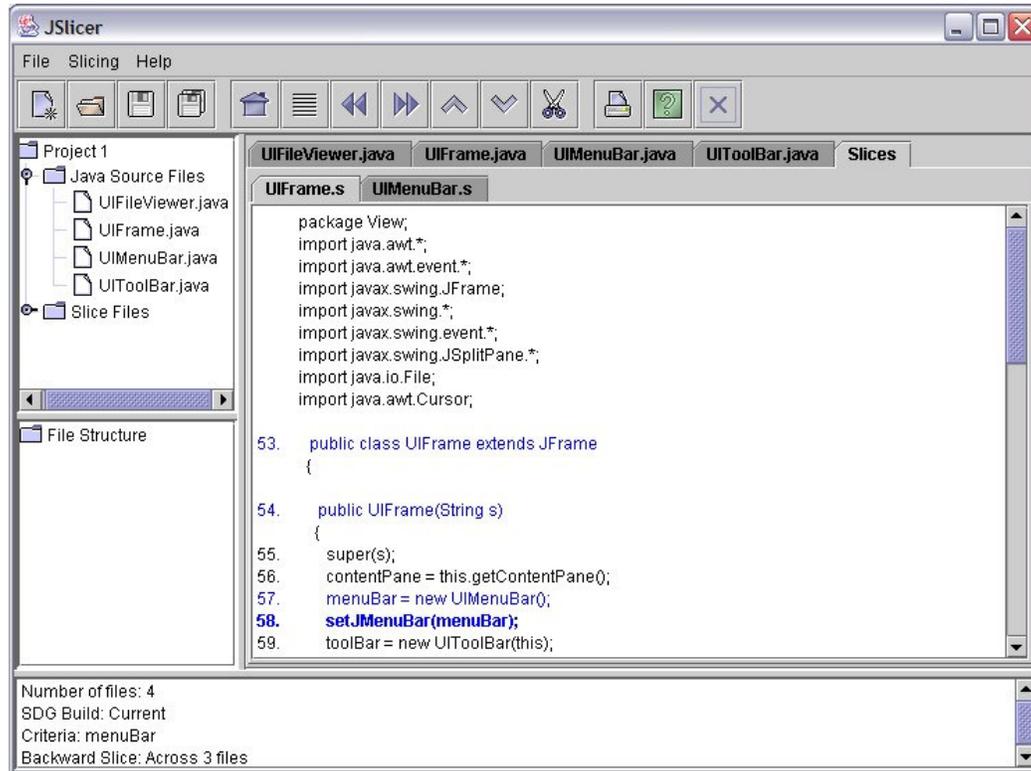


Figure A5: The JSlicer after a Backward Slice has been performed

As you can see from Figure A6, the Status Bar provides the following information:

- The number of java files open in the current project.
- Whether the SDG Build is up to date or not.
- The current Criteria if it has been set.
- A summary of a slice including the type of slice that was performed and the number of files that the slice cuts across.



Figure A6: The Status Bar

A2. New: Building a Project

A JSlicer “project” consists of four types of files. These are described below:

Java Source Files: You can open existing Java source files in the JSlicer which can then be used to build the SDG and perform slicing operations. In order to save a new project, it must consist of at least one Java source file.

An SDG: This is built from the Java source files contained within the current project. The most recent SDG built is saved as part of the project.

Slice Files: A Slice File is created by performing a Slice Operation on the SDG. Any generated Slice Files are saved as part of a project.

Root File: This is a file generated automatically by the JSlicer each time you save a new project. Instead of opening the different files in a project separately, the root file can be used to open the project and all files contained within it.

You can set up a new project by following these simple steps:

1. Press the  “New” button on the toolbar. (Or “New” in the “File” menu).
2. The JSlicer will close any current projects and an empty canvas will appear in the Source Viewer.
3. You are now ready to Open Java source files. After opening at least one Java file, you can Build the SDG, perform Slicing Operations and/or Save the project.

A3. Open and Browse

A3.1 Open

You can follow the steps described below to open a set of Java source files for a new project *or* to open an existing project.

1. Press the  “Open” button on the toolbar. (Or “Open” in the “File” menu). The system will display a filechooser.
2. If you wish to open Java source files for a new project, select all the files you want to include in the project and press “OK”. The files will be displayed in the Source Viewer.
3. If you wish to open an existing project, use the filechooser to navigate to the correct directory, select the .root file and press “Ok.” This will open the project and will allow you to view all the java files and slices contained within the project.

A3.2 Browse

You can browse a project using the tree in the File Viewer. This tree lists every Java file and Slice contained within the current project. Selecting a file name in the tree will open the file in the Source Viewer. This makes browsing the different files easier.

A4. Saving

You can follow the steps described below to save a project.

1. Press  "Save All" on the toolbar (Or "Save All" in the "File" menu).
2. This action will cause the system to display a Save Filechooser.
3. Simply type in the name which you wish to save the project under and press 'OK'. This will save all open Java source files, the most recent SDG Build and any Slice files to a folder with the chosen name.

To save an individual Java source file or slice file, press the  "Save" button on the toolbar or the "Save File" option on the "File" Menu.

A5. Build a System Dependence Graph (SDG)

To build the System Dependence Graph, press the  "Build SDG" button on the toolbar or select the "Build SDG" option from the "Slicing" menu. The system will process all the Java files contained within the current project. Once it has been completed, the Status Bar will display the following message: "SDG Build: Current".

Adding Java files to the project will require an SDG Rebuild therefore the Status Bar message will change to "SDG Build: Rebuild Required."

A6. Choose Criteria

A6.1 Choose Criteria

To set the Criteria, follow these simple steps:

1. Make sure that the SDG is up to date by reading the message displayed in the Status Bar. This will read either "SDG Build: Current" or "SDG Build: Rebuild Required."
2. Select the line that contains the variable(s) which you want to slice on.
3. Press the  "Add Criteria" button on the toolbar. (Or the "Add Criteria" option in the "Slicing" menu).
4. The "Choose Criteria" dialog box will be displayed. This dialog lists all the variables that can be sliced upon in the selected statement.
5. Select the variable(s) you want to slice on and press "Ok". The system will set this as the criteria. Notice that the Status Bar message will show you what the current Criteria are.
6. You are now ready to perform Slicing using the SDG and criteria.

A6.2 Clear Criteria

To clear the current Criteria, simply press the  "Clear Criteria" on the toolbar or press the "Clear Criteria" option in the "Slicing" menu. You can confirm that the Criteria have been cleared by checking the Status Bar message.

A7. Slicing Operations

There are five main Slicing Operations that can be performed using the JSlicer. This section explains how to carry out each of these five operations.

A7.1 Predecessors

To determine the Predecessors of a variable, follow these simple steps:

1. Make sure that the variable has been set as the criteria.
2. Go to the “Slicing” menu and then to the “Predecessors” sub-menu OR press the  “Predecessors” button on the toolbar.
3. You then have three options. You can choose to view the “Data Predecessors”, “Control Predecessors” or “All Predecessors”.
4. After choosing one of these options, the Predecessors of the criteria will be highlighted in any Java source file that contains a Predecessor. The Status Bar will provide information on which files contain Predecessors.

A7.2 Successors

To determine the Successors of a variable, you can use the following two options.

1. Make sure that the variable has been set as the criteria.
2. Go to the “Slicing” menu and then to the “Successors” sub-menu OR press the  “Successors” button on the toolbar.
3. You then have three options. You can choose to view the “Data Successors”, “Control Successors” or “All Successors”.
4. After choosing one of these options, the Successors of the criteria will be highlighted in any Java source file that contains a Successor. The Status Bar will provide information on which files contain Successors.

A7.3 Backward Slicing:

To perform Backward Slicing, you must complete the following steps:

1. Make sure that the SDG is up to date.
2. Set the Criteria.
3. Press the  “Backward Slicing” button on the toolbar or the “Backward Slicing” option in the “Slicing” menu.
4. The system will calculate which Java files contain code that should be included in the slice. A “Slice File” corresponding to each of these Java files will be made. A slice file basically consists of all the original Java source code but simply highlights all the code that is in the Slice.
5. A new window for each Slice File will be displayed in the Source Viewer. The Status bar will provide information on how many statements are in the slice and how many java files were involved (i.e. how many Slice Files were produced).

A7.4 Forward Slicing:

To perform Forward Slicing, you must complete the following steps:

1. Make sure that the SDG is up to date.
2. Set the Criteria.
3. Press the  “Forward Slicing” button on the toolbar or the “Forward Slicing” option in the “Slicing” menu.
4. The system will calculate which Java files contain code that should be included in the slice. A “Slice File” corresponding to each of these Java files will be made. A slice file basically consists of all the original Java source code but simply highlights all the code that is in the Slice.
5. A new window for each Slice File will be displayed in the Source Viewer. The Status bar will provide information on how many statements are in the slice and how many java files were involved (i.e. how many Slice Files were produced).

A7.5 Chopping:

To perform Chopping, you must complete the following steps:

1. Make sure that the SDG is up to date.
2. Set the Criteria.
3. Press the  “Chopping” button on the toolbar or the “Chopping” option in the “Slicing” menu.
4. The system will calculate which Java files contain code that should be included in the slice. A “Slice File” corresponding to each of these Java files will be made. A slice file basically consists of all the original Java source code but simply highlights all the code that is in the Slice.
5. A new window for each Slice File will be displayed in the Source Viewer. The Status bar will provide information on how many statements are in the slice and how many java files were involved (i.e. how many Slice Files were produced).

Appendix B – Design Document

The Design Document contains the class diagrams for the Model.Builder and Model.Slicer packages as they are the most important components of the system.

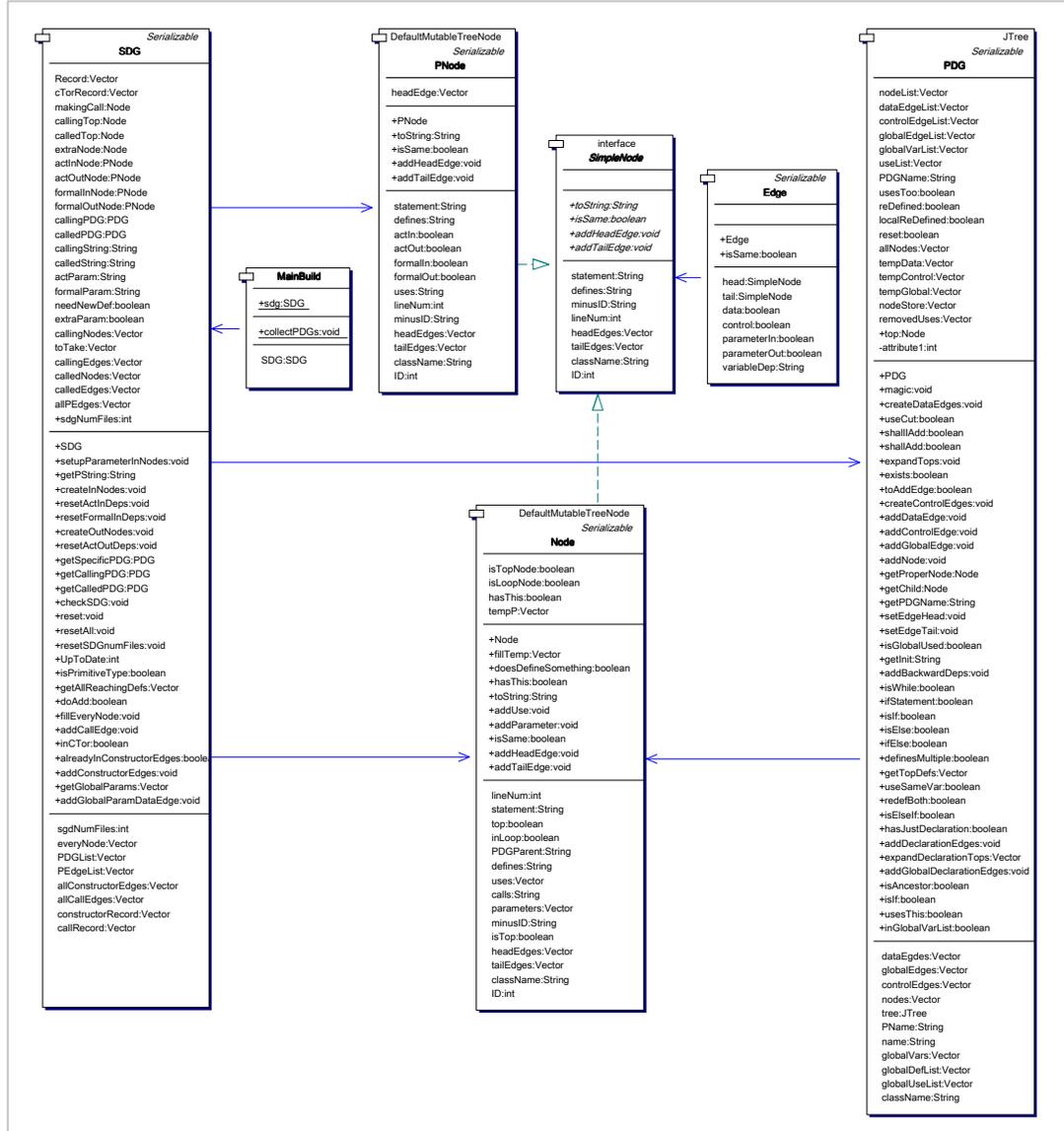


Figure B1: The Model.Builder Class Diagram

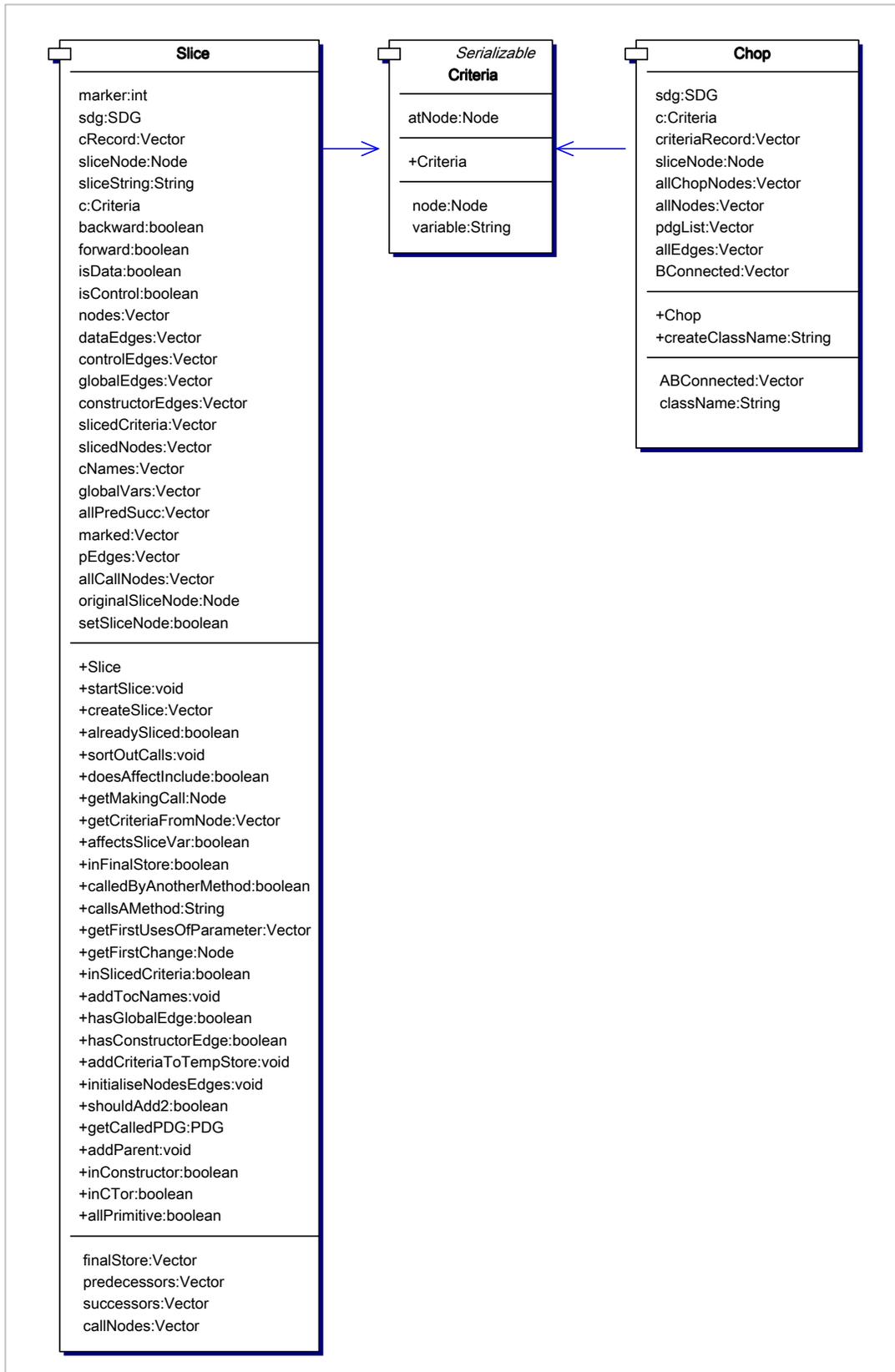


Figure B2: The Model.Slicer Class Diagram

Appendix C – System Manual

An API containing information about all the packages, classes and methods in the JSlicer has been created. A screenshot of this is shown in Figure C1. The complete API can be found in the “API” folder on the CD that has been handed in as part of the final deliverable.

The screenshot shows a web browser displaying the JSlicer API documentation. The browser window title is "Generated Documentation (Untitled) - Microsoft Internet Explorer provided by BTopenworld". The address bar shows the file path: "C:\Documents and Settings\Rukia\My Documents\Rua1\myProject\myDocs\5. Final\Appendices\JSlicerDocs\BDOC\index.html".

The main content area is divided into two sections, both with navigation tabs: "Overview", "Package", "Class", "Tree", "Deprecated", "Index", and "Help".

The first section, titled "Overview Package Class Tree Deprecated Index Help", contains a table of packages:

| Package | Description |
|---------------------------------------|---|
| JSlicer | The top level package of the system. It contains four sub-packages. |
| JSlicer.Controller | The Controller classes carry out the actions associated with each toolbar icon and menu item. |
| JSlicer.IO | The IO package handles all input/output operations such as saving and loading. |
| JSlicer.Model.Builder | The Builder classes are responsible for building the PDGs and the system SDG. |
| JSlicer.Model.Parser | The Parser classes parse Java source files into a tree format. |
| JSlicer.Model.Slicer | The Slicer package carries out all the slice operations using the SDG and criteria. |
| JSlicer.View | The View package contains all the user interface classes. |

The second section, also titled "Overview Package Class Tree Deprecated Index Help", is currently empty.

The left sidebar contains two lists:

- Packages:** [JSlicer](#), [JSlicer.Controller](#), [JSlicer.IO](#), [JSlicer.Model.Builder](#), [JSlicer.Model.Parser](#), [JSlicer.Model.Slicer](#), [JSlicer.View](#)
- All Classes:** [ApplicationMain](#), [Assignment](#), [BackwardAction](#), [BuildAction](#), [Call](#), [Chop](#), [ChoppingAction](#), [CloseFileAction](#), [CloseProjectAction](#), [CloseWinAction](#), [Criteria](#), [CriteriaAction](#), [DataControlAction](#), [DataControlDialogAction](#), [Edge](#), [ExitAction](#), [Filechooser](#), [ForwardAction](#), [Graph](#)

Figure C1: Screenshot of the JSlicer API

