

VISUALIZATION TOOLS FOR OPTIMIZING COMPILERS

by

Jennifer Elizabeth Shaw

School of Computer Science
McGill University, Montreal

August, 2005

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE

Copyright © 2005 by Jennifer Elizabeth Shaw

Abstract

Optimizing compilers have traditionally had little support for visual tools which display the vast amount of information generated and which could aid in the development of analyses and teaching and provide extra information to general programmers. This thesis presents a set of visualization tools which integrate visualization support for Soot, an optimizing compiler framework, into Eclipse, a popular, extensible IDE.

In particular, this work explores making the research compiler framework more accessible to new users and general programmers. Tools for displaying data flow analysis results in intermediate representations (**IRs**) and in the original source code are discussed, with consideration for the issue of the mapping of information between the low-level **IRs** and the source using flexible and extensible mechanisms. Also described are tools for interactive control flow graphs which can be used for research and teaching and tools for displaying large graphs, such as call graphs, in a manageable way.

Additionally, the area of communicating information generated by optimizing compilers to general programmers is explored with a small case study to determine if analyses could be useful to general programmers and how the information is displayed.

This work is shown to be useful for research to find imprecision or errors in analyses, both from visualizing the intermediate results with the interactive control flow graphs and the final results at the **IR** and source code levels, and for students learning about compiler optimizations and writing their first data flow analyses.

Résumé

Les optimiseurs ont traditionnellement eu peu de support pour des outils visuels qui présentent la vaste quantité d'informations produites et qui pourraient faciliter le développement d'analyses et de l'enseignement et fournir des renseignements supplémentaires aux programmeurs généralistes. Cette thèse présente un ensemble d'outils de visualisation qui intègrent le soutien de visualisation de Soot, un cadre d'optimiseur, dans Eclipse, un environnement de développement intégré (Integrated Development Environment, IDE) populaire.

En particulier, ce travail explore des méthodes pour rendre le cadre d'optimiseur de recherche plus accessible de nouveaux utilisateurs et programmeurs généralistes. Des outils pour présenter des résultats d'analyse de flot de données dans les représentations intermédiaires (**IRs**) et dans le code source original sont discutés, en prenant en compte la difficulté d'établir la correspondance entre les **IRs** de bas niveau et le code source en utilisant des mécanismes flexibles et extensibles. De plus, des outils pour les graphes interactifs de flux de commande qui peuvent être employés pour la recherche et l'enseignement ainsi que des outils pour afficher d'une manière efficace de grands graphes, tels que des graphes d'appel, sont décrits.

En plus, le secteur de la communication de l'information produit par les optimiseurs aux programmeurs généralistes est explorée avec une petite étude de cas pour déterminer si les analyses pourraient être utiles aux programmeurs généralistes et comment l'information est exposée.

L'utilité de ce travail en recherche est démontrée en l'utilisant pour trouver l'imprécision ou les erreurs dans les analyses, en visualisant non seulement les résultats intermédiaires avec les graphes interactifs de flux de commande mais aussi les résultats finaux au

niveau IR et de la représentation intermédiaire et du code source, et pour aider des étudiants dans leur apprentissage des optimisations employées dans les compilateurs et l'écriture de leurs premières analyses de flot de données.

Acknowledgments

I am very thankful to my advisor Laurie Hendren for her encouragement and support throughout this project, for her belief that I could actually finish it and for her suggestions and enthusiasm.

This work builds upon three large software projects and I am grateful for the work that has been done before on the Soot framework, the Polyglot tool and the Eclipse platform. The Soot framework, which is the main basis for this project, was developed by the Sable Research Group which was an excellent group to work with. I would like to thank the members of the group. In particular, I would like to thank the Soot team: John Jorgenson, Patrick Lam, Ondřej Lhoták, Feng Qian and Navindra Umanee. I would also like to thank Bruno Dufour and Maxime Chevalier-Boisvert for helping me with the French version of my abstract for this thesis.

This work was mainly funded by an IBM Eclipse Innovation Grant and I am thankful to IBM for recognizing and supporting computer science research.

Finally I would like to thank my parents for their support and guidance throughout my life and my husband Ondřej, for believing in me and putting up with me throughout this very challenging time.

Contents

Abstract	i
Résumé	iii
Acknowledgments	v
Contents	vii
List of Figures	xi
List of Tables	xv
List of Algorithms	xvii
1 Introduction	1
1.1 Motivation	1
1.2 Contributions	3
1.2.1 Basic Eclipse Plugin for Soot	4
1.2.2 Analysis Results Framework	4
1.2.3 Graph Tool	5
1.3 Organization	5
2 Background	7
2.1 Tools Overview	7
2.1.1 Soot	7

2.1.2	Polyglot	11
2.1.3	Eclipse	12
3	Soot - Eclipse Plugin	13
3.1	Soot Invocation within Eclipse	13
3.1.1	Launching Soot Within Eclipse	15
3.1.2	Options Dialog	16
3.1.3	Managing Option Configuration	17
3.2	Soot Output in Eclipse	18
3.3	IR Editor	19
4	Java To Jimple	21
4.1	Motivation	21
4.2	Code Generation	22
4.2.1	Loops	24
4.2.2	Call Expressions	25
4.2.3	Try / Catch / Finally Statements	28
4.2.4	Synchronized Statements	29
4.2.5	Array Expressions	31
4.2.6	Field Expressions	31
4.2.7	Other Stmts	35
4.2.8	Nested Classes	36
4.3	Summary	45
5	Viewing Static Analysis Results	47
5.1	Motivation	47
5.2	Mapping Source and IR Position Information	49
5.2.1	Position Origins	49
5.2.2	Position Information Assignment for Source Input	50
5.3	Visual Results	67
5.3.1	StringTags and Tool-tips	68
5.3.2	ColorTags and Color Highlighting	68

5.3.3	LinkTags and Links	69
5.3.4	KeyTags and Legends	70
5.4	Collecting Tags for Output	71
5.5	Managing the Display in Eclipse	72
5.6	Summary	73
6	Applications of Tools	75
6.1	Motivation	75
6.2	Applications for Compiler Research	75
6.2.1	Analysis Results for Teaching	77
6.2.2	Analysis Results Specifically for Compiler Research	80
6.2.3	Analysis Results for Research and Advanced Users	87
6.3	Applications for Program Understanding	91
6.3.1	Unreachable Fields and Methods Analyses	94
6.3.2	Tightest Qualifiers Analysis	96
6.3.3	Loop Invariants Analysis	99
6.4	Summary	100
7	Interactive Tools	101
7.1	Motivation	101
7.2	Interactive Control Flow Graph Tool	101
7.2.1	Running	102
7.2.2	Debugging	103
7.2.3	Filtering Data Flow Sets for More Relevant Displays	106
7.3	Motivation - Displaying Large Graphs	106
7.4	Interactive Call Graph Tool	108
8	Related Work	111
8.1	Views Displaying Compiler Information	111
8.2	Correlating IR and Source Results	115
8.3	Static Analysis to Highlight Coding Problems	116

9	Conclusions and Future Work	119
9.1	Conclusions	119
9.2	Future Work	121

Appendices

A	User Guide	123
	Bibliography	125

List of Figures

2.1	Java code for Example.java	8
2.2	Corresponding Jimple code for Java source Example.java	9
2.3	Soot Overview	11
3.1	Plugin Architecture	14
3.2	Soot Invocation within Eclipse	15
3.3	Soot Options Dialog within Eclipse	17
3.4	Soot Configurations Dialog within Eclipse	18
3.5	Soot Output View within Eclipse	19
3.6	IR Editor within Eclipse	20
4.1	While Loop Code Generation	24
4.2	While Loop with Branch Statement Code Generation	25
4.3	While Loop Code Generation Before Nop Elimination	26
4.4	For Loop with Branch Statement Code Generation	28
4.5	Try/Catch/Finally with Return Statements Code Generation	30
4.6	Synchronized with Return Statements Code Generation	32
4.7	Array Code Generation	32
4.8	Sample Field Reference From Nested Class	34
4.9	Sample Field Reference From Nested Class - Generated Jimple	35
4.10	Normal Nested Classes - Naming Scheme Example	37
4.11	Anonymous Nested Classes - Naming Scheme Example	38
4.12	Local Nested Classes - Naming Scheme Example	38
4.13	Simple Final Locals Example	40

4.14	Simple Final Locals Example	40
4.15	Final Locals - Local Class Creation Example	41
4.16	Final Locals - Local Extends Example	42
4.17	Final Locals - One-level Only Example	42
5.1	Overview of Generated Position Information	49
5.2	Assert Statement Position Information Generation	52
5.3	Constructor Call Statement Position Information Generation	54
5.4	Do Statement Position Information Generation	54
5.5	For Statement Position Information Generation	55
5.6	If Statement Position Information Generation	55
5.7	Local Declaration Statement Position Information Generation	55
5.8	Switch Statement Position Information Generation	56
5.9	Synchronized Statement Position Information Generation	57
5.10	Try/Catch Statement Position Information Generation	58
5.11	While Statement Position Information Generation	58
5.12	Array Access Expression Position Information Generation	59
5.13	New Array Expression Position Information Generation	59
5.14	New Multi-Array Expression Position Information Generation	59
5.15	Array Initializer Expression Position Information Generation	60
5.16	Assignment Expression Position Information Generation	60
5.17	Assignment with Operator Expression Position Information Generation	61
5.18	Conditional And Binary Expression Position Information Generation	61
5.19	Conditional Or Binary Expression Position Information Generation	62
5.20	Call Expression Position Information Generation	62
5.21	Cast Expression Position Information Generation	63
5.22	Conditional Expression Position Information Generation	63
5.23	Field Expression Position Information Generation	64
5.24	InstanceOf Expression Position Information Generation	64
5.25	New Expression Position Information Generation	65
5.26	Simple Unary Expression Position Information Generation	66

5.27	Unary Plus Expression Position Information Generation	66
5.28	Unary Minus Expression Position Information Generation	66
5.29	Unary Bitwise Complement Expression Position Information Generation	67
5.30	Unary Logical Complement Expression Position Information Generation	67
5.31	Tool-tip with Analysis Information	68
5.32	ColorTags Representing Analysis Information	69
5.33	LinkTag with Analysis Information	70
5.34	Analysis Visualization Results Legend View	70
5.35	Tag Collection Overview	71
5.36	Analysis Visualization Results Types View	72
6.1	Process for Using Framework for Viewing Analysis Results	76
6.2	Code to Visualize Parity Analysis Results	78
6.3	Code to Add Tags to Visualize Parity Analysis Results	79
6.4	Code to Register Tagger with PackManager	79
6.5	Parity Analysis with Visualization Results	80
6.6	Code to Add LinkTags to Visualize the Call Graph	82
6.7	Call Graph Analysis with Visualization Results	83
6.8	Call Graph Analysis LinkTag with Visualization Results	83
6.9	Code to Add StringTags and ColorTags to Visualize the Live Variables	84
6.10	Liveness Analysis with Visualization Results	85
6.11	Code to Visualize Reaching Definition Analysis Results	86
6.12	Code to Visualize Cast Check Elimination Analysis Results	88
6.13	Cast Check Analysis with Visualization Results	89
6.14	Code to Visualize Array Bounds Check Analysis Results	90
6.15	Array Bounds Checks Analysis with Visualization Results	91
6.16	Code to Visualize Null Check Analysis Results	92
6.17	Code to Add StringTags and ColorTags for Null Check Analysis Results	93
6.18	Null Checks Analysis with Visualization Results	94

6.19	Unreachable Methods Analysis with Visualization Results	96
6.20	Code for Tagging Unreachable Methods with Visualization Results . .	97
6.21	Tightest Qualifiers Analysis with Visualization Results	98
6.22	Loop Invariant Analysis with Visualization Results	99
7.1	Live Variable - Interactive Control Flow Graph Example Code	103
7.2	Add <code>i</code> to Data Flow Set	104
7.3	Propagate Set	104
7.4	Add <code>x</code> to Set	104
7.5	Propagate Set	104
7.6	Partially generated flow sets on <code>cfg</code> with Liveness Analysis	105
7.7	Annotated <code>cfg</code> with filtered Parity Analysis	107
7.8	Hello World Java Program	108
7.9	Interactive Call Graph Tool Options	108
7.10	Interactive Call Graph Tool	109

List of Tables

6.1	Unreachable Methods Analysis Results	95
6.2	Unreachable Fields Analysis Results	95
6.3	Tightest Qualifiers on Methods Analysis Results	98
6.4	Tightest Qualifiers on Fields Analysis Results	99
7.1	Reachable Method Counts for HelloWorld Call Graph	106

List of Algorithms

1	Basic Soot Processing	10
2	Soot using Java source as input	22
3	While Loop Code Generation	27
4	Synchronized Statement Code Generation	33

Chapter 1

Introduction

1.1 Motivation

Optimizing compilers seek to analyze and transform the program being compiled in order to make it more efficient in terms of running time and/or space. Soot [VR00, VRGH⁺00] is a bytecode optimization framework which has successfully been used for experimentation with optimizations in many areas including pointer analysis [LH03], array bounds check elimination [QHV02], and virtual method call resolution [SHR⁺00]. These complex analyses are implemented within Soot and take advantage of its extensibility, the many available intermediate representations (IRs) and the flexible series of options. These analyses involve generating, using and understanding large amounts of information.

Soot was originally available as only a command-line tool, which could be invoked in a shell environment. While this is a useful interface for advanced compiler developers, it was found to be very difficult to be effectively used by those unfamiliar with the tool. Therefore, it was necessary to provide access to Soot in a simpler environment, such as an IDE.

The Soot framework is manipulated by a complicated series of options. These options are often changed, and new options are frequently added. The need to keep an up-to-date interface for accessing the compiler framework is an important consideration for making the framework accessible or usable to compiler students, general

programmers and even to researchers who are unfamiliar with the framework.

Soot contains several different **IRs**, upon which analyses are performed. These **IRs** were designed specifically for easily computing analyses and are much simpler than original source code. For example, they may include fewer statements and/or express abstract ideas more concretely. This means, however, that they are unfamiliar to the majority of users and are less compact than traditional source code. However, despite these issues, there was no type of editor support for these **IRs**, and yet compiler students are required to learn about them and compiler researchers need to understand and manipulate them.

The Soot compiler framework, like many other optimizing compilers, includes numerous standard compiler analyses, such as liveness and reachability analyses and is extensible to provide support for developing new analyses. Unfortunately, there was limited support for generating and visually displaying the analysis results in relation to the **IR**. In some optimizing compilers, visual displays are available for specific analyses¹, but there are few tools or formats for generating associated visual information regarding the results of the analysis, in an extensible way, such that it may be applicable to new analyses, and displaying that data. Viewing the results of the analyses is useful for debugging purposes and as time goes on, more advanced analyses are extensions of basic analyses and thus it is vital to be able to understand the fundamental analyses and to ensure that they are correct.

Research compiler frameworks generate analysis information which could be useful for general programmers. Usually, in an integrated development environment (**IDE**), general programmers have access to extra program information, such as the type hierarchy view in the Eclipse **IDE** [ecl03], which are based on structural analyses. Compiler frameworks, like Soot, can generate much more precise information, based on data flow and control flow analyses, that could aid in development of large software projects. However, there was a lack of mechanisms to communicate the information generated by an optimizing compilers to the end-users. These programmers are unlikely to be familiar with the available **IRs** and thus there was a need to provide tools

¹See Related Work in Chapter 8

to convey the analysis results in a way that is related to the original source.

Soot provides an extensible analysis framework which can be used to compute intra-procedural analyses until a fixed-point has been reached. This computation framework usually performs many iterations, generating data for each point in the program during each iteration. This information changes over time during the analysis and there were no interactive debugging type tools available to capture the partial results or the changes being made. It is useful to be able to view this information as it is being generated before it is hidden by the data generated in the next iteration to determine where the analysis is broken, or where the analysis loses precision. These partial results enable the analysis developer to determine which kinds of statements are handled incorrectly. Additionally, compiler students must learn how standard analyses work and how to construct their own analyses and it is useful if they can see the analyses in a step by step procedure.

Often very large graphs, such as call graphs, are created in optimizing compiler frameworks. These large graphs may then be used to compute more complicated analyses. Traditional tools to display large graphs have had problems with layout and size issues. Thus tools are needed to view these large graphs in effective ways, where the amount of data shown at one time is limited and adequate control over which data should be displayed at a given time is provided.

Thus, the main goal in this project is to expose the inner workings of a research compiler framework in a visual way. Much information is generated by compilers and there were a lack of tools to communicate this information to compiler writers, students and general programmers. This project seeks to address the lack of tools and find ways to communicate the wealth of information, generated by optimizing compilers, to users, taking into consideration methods that may be used for providing tools in other areas as well.

1.2 Contributions

To address these issues, a generic set of extensible tools has been developed, which can be used for aiding researchers, students and end-users in working with compilers. These tools are based in a plugin which links the Soot optimizing compiler framework with Eclipse a popular IDE.

1.2.1 Basic Eclipse Plugin for Soot

The first contribution of this project is the basic plugin used for invoking the research compiler in an IDE. This plugin integrates the optimizing research compiler Soot into the Eclipse IDE, providing menu support, dialogs and views for general use of the Soot framework. This allows students and general users to easily invoke the compiler framework in a familiar environment. In particular this plugin provides an extensible, re-generatable dialog to manage the many options found in the research compiler framework.

As part of the basic plugin, an IR editor which provides convenient support of the different IRs generated by the Soot compiler, is available. This editor provides keyword highlighting to help students and researchers to better understand the IRs and a content outline which is useful for manipulating the IRs, which tend to be much longer than the original source code.

1.2.2 Analysis Results Framework

The second main contribution is a series of mechanisms for displaying, in a visual way, the results of analyses computed by the compiler. These mechanisms are generic enough to handle many different types of analysis results, including new analyses which have not yet been considered. These mechanisms are designed to handle different types of analysis results that may be generated, instead of specific types of analyses, which makes them suitable for a wide range of analyses. In addition to the mechanisms to generate the required visual data, a framework for displaying the data, again in ways that depend not on the data but on the format of the data, has

been developed.

A key contribution to this framework is Java to Jimple, a code generation project, which provides the necessary information for displaying the results of analyses at the source code level.

As this framework for displaying analysis results has been found to be successful in helping to debug compiler analyses, easily identifying areas where imprecise or incorrect results are generated, some analyses which generate information for general programmers: an unreachable fields and methods analysis, a tightest qualifiers analysis and a loop invariant analysis, have been developed and their results displayed in this framework.

At the intra-procedural level results are computed on control flow graphs of the method being analyzed. An interactive control flow graph tool, which allows researchers to develop analyses while being able to easily view intermediate results, as the fixed point is being computed, is the third contribution. This tool displays the method as a graph and updates the information generated for each statement in an interactive way. This allows researchers to debug their analysis and students to understand data flow analysis.

1.2.3 Graph Tool

Finally, to handle the issue of large generated graphs, a second plugin, a graph tool has been created, which displays graphs and can be extended to display compiler generated graphs. This generic graph tool has been extended with an interactive call graph tool which allows researchers to view and manipulate a precise call graph limiting the size of the partial graph shown at one time. This generic graph tool could also be used for other compiler generated graphs such as points-to graphs.

1.3 Organization

The rest of this thesis is organized as follows. Chapter 2 reviews the background tools, including Soot and Eclipse, upon which this work is based. Chapter 3 presents

a technical overview of the basic Soot - Eclipse plugin, covering the menus, dialogs and views, as well as the **IR** editor. Chapter 4 discusses the Java to Jimple project which is used to relate analysis results to the original source code. Chapter 5 explains the framework for generating and displaying visual analysis results. Chapter 6 introduces applications of the visualization framework and discusses new analyses which are useful for general programmers. Chapter 7 introduces the interactive framework for viewing generated analysis information as it is being generated and the set of tools used for viewing large amounts of graph data generated by compilers and discusses the example of a partial call graph tool. Chapter 8 discusses related work. Finally, in chapter 9 conclusions are given and areas of future work are discussed.

Chapter 2

Background

2.1 Tools Overview

This work is based upon three large software projects and in a way integrates the three tools together. The tools are Soot, Polyglot and Eclipse and in this chapter, we give an overview of each and describe how they are used in this work.

2.1.1 Soot

Soot¹ [VR00, VRGH⁺00] is a Java bytecode optimization and analysis framework developed in the Sable Research Lab over the past several years. It has several intermediate representations (IRs): Jimple, Baf, Grimp, Shimple and Dava, which are used for analyses and transformations. The main IR is Jimple: a three-address, typed, non-stack based representation. Jimple is used for analyses because it contains far fewer kinds of statements and expressions than Java source, and is also much simpler to work with than Java bytecode, as it abstracts away the stack and has type information available for locals. Baf is a bytecode-like representation, Grimp is similar to Jimple but without the three-address per statement restriction, Shimple is a static single assignment (SSA) version of Jimple and Dava [Mie03, MH02, MH01] is a structured abstract syntax tree (AST) representation used for decompiling. Figure

¹<http://www.sable.mcgill.ca/soot>

2.1 lists a short Java program and Figure 2.2 displays the corresponding Jimple code.

```
public class Example {  
    public void foo(){  
        int [] arr = new int [10];  
        for(int i = 0; i < arr.length; i++){  
            arr[i] = i;  
            System.out.println(i);  
        }  
    }  
}
```

Figure 2.1: Java code for Example.java

There are several key features of Jimple to note in Figure 2.2. First, all of the variables have declared types. For example, the variable `arr` is declared to be an integer array. Second, the `for` loop has been translated to a set of `if` and `goto` statements starting at `label10` and ending at `label11`. Finally, the three-address code limit is shown in the translation of the `if` statement condition. This condition requires four bytecodes: one to load the variable `i`, one to load the variable `arr`, one to determine the array length and one to perform the comparison. To represent these four bytecodes in three-address code statements, two statements are needed. The first statement, `$i0 = lengthof arr`, handles accessing the variable `arr` and determining the array length, which is stored in the intermediate variable `$i0`. In the next statement, `if i >= $i0 goto label11`, the variable `i` is accessed, the intermediate variable `$i0` is accessed and the comparison is made.

The Soot framework provides tools for developing inter- and intra- procedural analyses. In particular, it includes a data flow analysis framework which facilitates writing intra-procedural analyses and computing the fixed-point. Additionally, Soot can compute a precise call graph which can be used for additional analyses, including whole program analyses. The general functionality of Soot is given in Algorithm 1. First Soot loads all classes required to process the class being analyzed, this includes

```
public class Example extends java.lang.Object
{
    public void foo ()
    {
        Example this;
        int[] arr;
        int i, $i0;
        java.io.PrintStream $r0;
        this := @this: Example;
        arr = newarray (int)[10];
        i = 0;
    label0:
        $i0 = lengthof arr;
        if i >= $i0 goto label1;
        arr[i] = i;
        $r0 = <java.lang.System: java.io.PrintStream out>;
        virtualinvoke $r0.<java.io.PrintStream: void println(int)>(i);
        i = i + 1;
        goto label0;
    label1:
        return;
    }

    public void <init>()
    {
        Example this;
        this := @this Example;
        specialinvoke this.<java.lang.Object: void <init>()>();
        return;
    }
}
```

Figure 2.2: Corresponding Jimple code for Java source Example.java

all referenced classes. Upon loading the classes Soot creates a skeleton **SootClass**, for each, consisting of **SootFields** and **SootMethods**, but not method bodies. Soot may optionally perform whole program analyses or may proceed directly to performing intra-procedural analyses. Method bodies are generated during analyses as they are needed. This implies that unreachable method bodies of classes referenced from the class library are never generated. Finally, any generated results are output.

Algorithm 1 Basic Soot Processing

```
load all required classes
generate SootClass skeletons
<perform whole-program analyses
    generating method bodies as needed>
perform intra-procedural analysis
    generating method bodies as needed
output results
```

Soot has been extended with an annotation framework [PQVR⁺01], which allows **Tags** to be attached to **Hosts**. **Tags** are any piece of information such as the result of an analysis. For example, a **Tag** could be added to each array access to indicate whether it is potentially out of the array boundaries or if it is definitely safely within the boundaries. **Hosts** are structures that may need related information attached to them, such as classes, fields, methods, statements and expressions. These **Tags** are propagated throughout the different **IRs** and updated as required.

As shown in Figure 2.3, Soot takes as input Java bytecode, Jimple and now Java source code², creates Jimple, performs analyses, adding **Tags** where necessary and outputs Java bytecode or any of the Soot **IRs**.

The tools presented in this thesis build upon and extend Soot in several different ways. First, we integrate the basic functionality of Soot into Eclipse. Second, we extend Soot to take Java source as input. Third, we provide extensions to the control flow analysis framework and call graph to enable visualization. Fourth, we add

²Support for Java source code input to Soot is presented in Chapter 4 of this thesis.

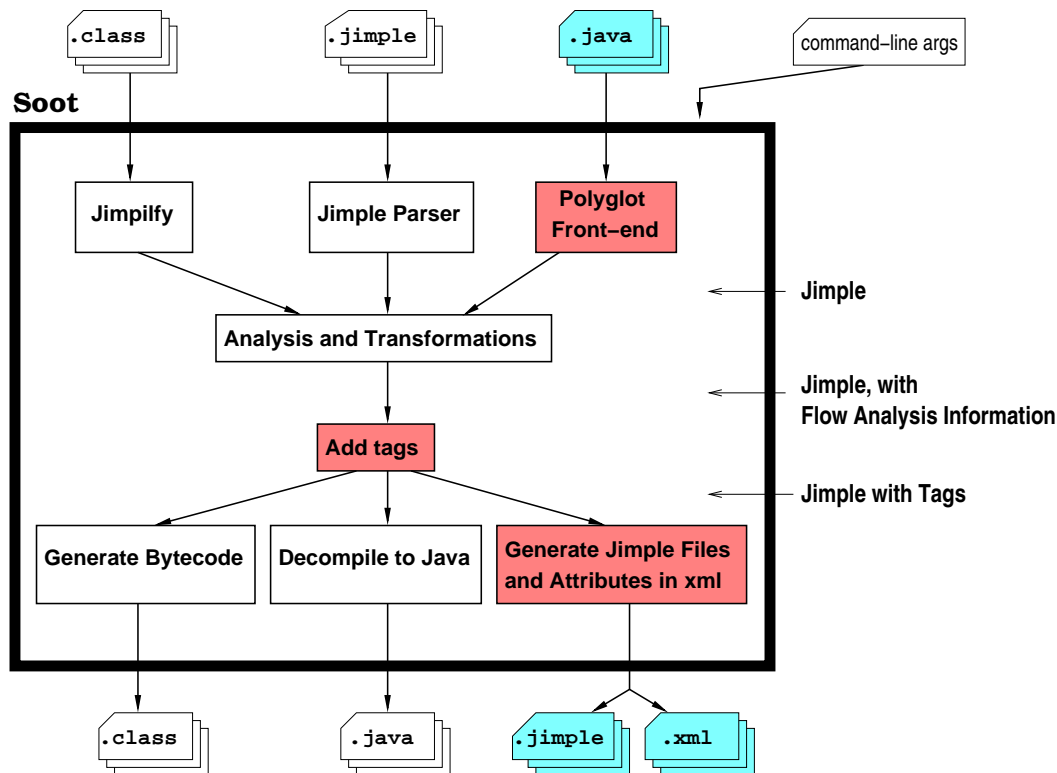


Figure 2.3: Soot Overview

graphical result **Tags** for encoding and displaying analysis results. Fifth, we utilize the framework to build new analyses.

2.1.2 Polyglot

Polyglot [NCM03] is a front-end Java source to Java source compiler. Its main purpose is to allow researchers to easily extend the Java language, providing an extensible framework for doing so. We do not use it in this capacity, instead, we extract the **AST** generated by the Polyglot front-end and from it generate Jimple code. Polyglot provides all the information required by a front-end compiler such as position information, error checking and type checking and we take advantage of this information to generate Jimple and in the development of our tools. Polyglot covers the entire

Java language making it suitable for integration with Soot.

2.1.3 Eclipse

Eclipse [ecl03] is an open-source, extensible integrated development environment (IDE). Eclipse is a framework with multiple graphical views, editors, dialogs and menus each of which may be extended or customized. The underlying graphical system is the standard widget toolkit (SWT), an alternative to Swing, the standard Java graphical system. We integrate our work into Eclipse as a plugin, as Eclipse was designed as a plugin framework where one can easily add new functionality. We take advantage of the many features of Eclipse in order to avoid duplicating user interface and graph layout work. Our basic integration extends menus, editors and views and builds upon SWT. Our interactive control flow graph and call graph tools build upon the graphical editing framework (GEF) [GEF], which is itself an extension of Eclipse and provides the basics for graphical editors and graph layout.

Chapter 3

Soot - Eclipse Plugin

3.1 Soot Invocation within Eclipse

Eclipse is a multi-purpose development framework which includes tools for Java development in an extensible graphical environment. The Soot bytecode analysis framework is a powerful compiler framework which has traditionally been available only as a command-line tool. Integration of Soot into Eclipse has many benefits to users and researchers alike. The main benefits include: enabling new programmers to easily use the framework without any complicated set-up usually associated with research frameworks, enabling students to become familiar with the Soot framework and all the options available, and allowing researchers unfamiliar with the framework to develop new compiler analyses within the environment of an IDE.

The plugin is comprised of the basic plugin, an analysis results visualization framework and an extended set of interactive tools as shown in Figure 3.1. In Figure 3.1 the grey box at the top represents Eclipse, and the big white plug shaped box represents the Soot - Eclipse plugin. All of Soot is contained within the plugin as shown in the smaller white box. The four boxes along the bottom of the figure represent the four modules making up the plugin. The basic plugin consists of the Soot Launcher module and the IR Editor module. The Visualization Framework and Interactive Tools modules make up the rest of the plugin. The plugin modules are designed to interact with Soot in such a way that Soot is kept completely independent from the

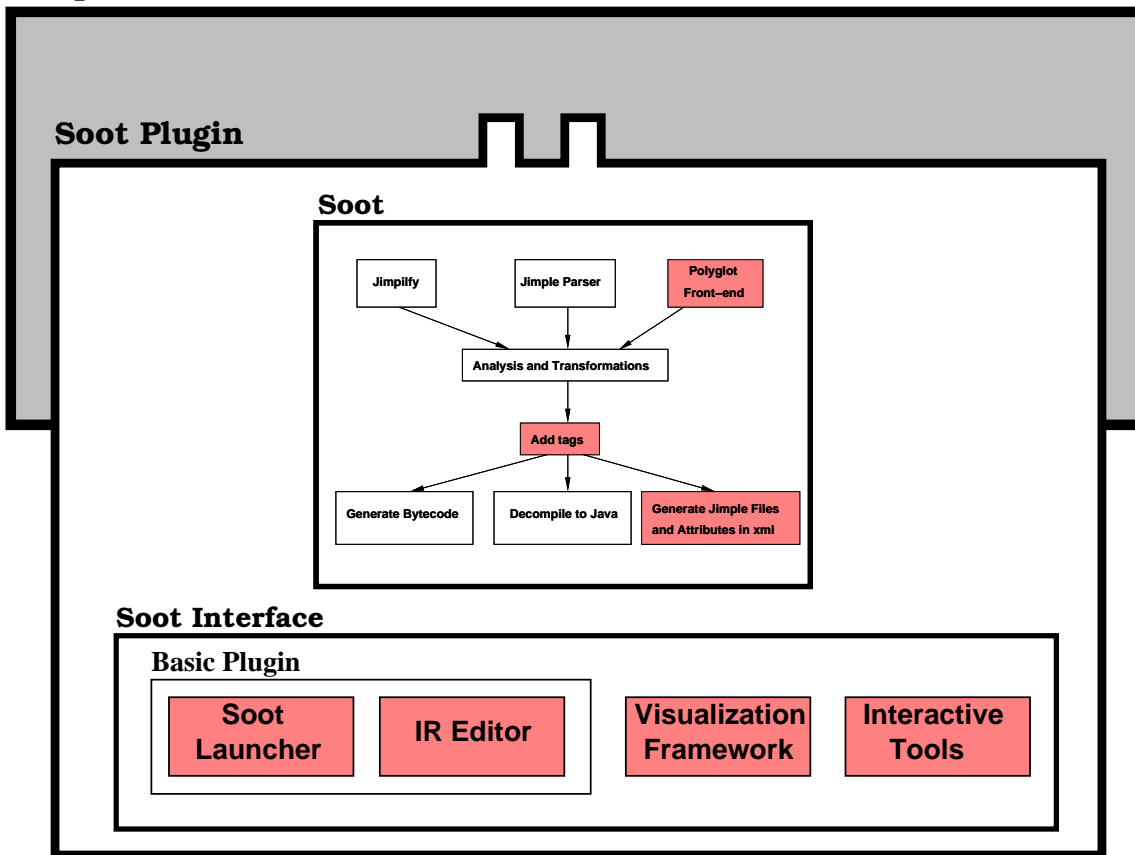
Eclipse

Figure 3.1: Plugin Architecture

modules. This design allows Soot to continue to be available as a command line tool, with no dependencies on the Eclipse project, for those who prefer it in that format. This chapter discuss the basic plugin, chapters 4, 5 and 6 look at the visualization framework and chapter 7 describes the interactive tools.

The main contributions to the basic plugin are: menu items, which can be used to invoke simple Soot operations, a programmatically generated options dialog which can be used to invoke all of the functionality available in Soot, an IR editor and an output view which displays the standard Soot output. The extended features which include the attribute visualization framework and interactive control flow and call

graphs are discussed in later chapters 5, 7.

3.1.1 Launching Soot Within Eclipse

The Soot launcher module is used for launching Soot within Eclipse. This module handles many basic components for the invocation of the Soot framework. It handles the selection of files to be processed determining the source precedence, whether Soot takes as input class files, Jimple files or Java source files. The Soot output folder is setup and refreshed from the file system within Eclipse after Soot has run, so that all generated output files are available within Eclipse. This module handles sending the required options as arguments to Soot, including the classpath required for the files being processed. Soot is invoked on a separate thread and the Soot launcher provides a mechanism for handling Soot output.

For beginning users, several basic menu items are provided to run Soot with common options on a single file or a project of files as shown in Figure 3.2, which shows

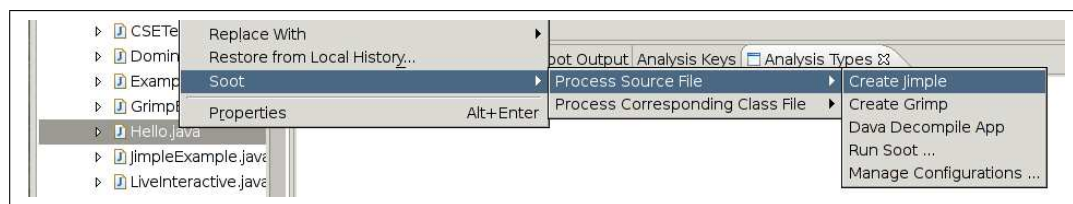


Figure 3.2: Soot Invocation within Eclipse

the menu item for invoking Soot to create Jimple, using a source file `Hello.java` as input. For example, one of the first activities that new users of Soot try is to produce Jimple, one of the intermediate representations (IRs) in Soot. Menu items are provided, for new users, for commonly performed actions such as producing Jimple files or decompiling. For more advanced users, an options dialog, with all Soot options available to be set, is provided and also provided is a second dialog for managing Soot configurations. These are discussed below in sections 3.1.2 and 3.1.3.

3.1.2 Options Dialog

Soot has approximately 180 options and new options are added often. In order to keep the plugin synchronized with Soot all options and related documentation are stored in an extensible markup language (XML) file and the option parsing code and the options dialog are generated programmatically. Using this method also allows the documentation to be used as tool-tips for the widgets in the dialog.

In order to create the options dialog, a different visual widget is used to represent each type of option available. A boolean widget with a check-box is used for boolean options. These are options that can either be selected or not selected and take no parameters. A string widget with a text box is used for options requiring a single string parameter. A list widget with a text box which has multiple lines is used for options requiring a list of parameters. Finally, a multi widget with a set of radio buttons is used for options which take a single parameter from a designated set.

Generating the options dialog in this way allows the dialog to always stay up to date and synchronized with Soot, with no extra programming required. This classification of option types ensures that as long as new options are formed as one of the specified types, then no user interface work is required. As well, updating the look of the user interface is simply a matter of updating the four widgets without having to change code for approximately 200 options. Additionally, organizing and classifying the options with an associated visual widget simplifies and streamlines the dialog. As shown in Figure 3.3, the options are classified into different groups represented in the tree on the left. These groups can be selected to reveal the options to be specified. In this figure the **Output Options** group is selected. Several boolean options are listed at the top right, shown as check boxes. In the middle right a set of radio buttons are shown, which represent the **Output Format** to be produced and at the bottom a tool-tip is shown, giving a description of the **Jimple File - Output Format** option.

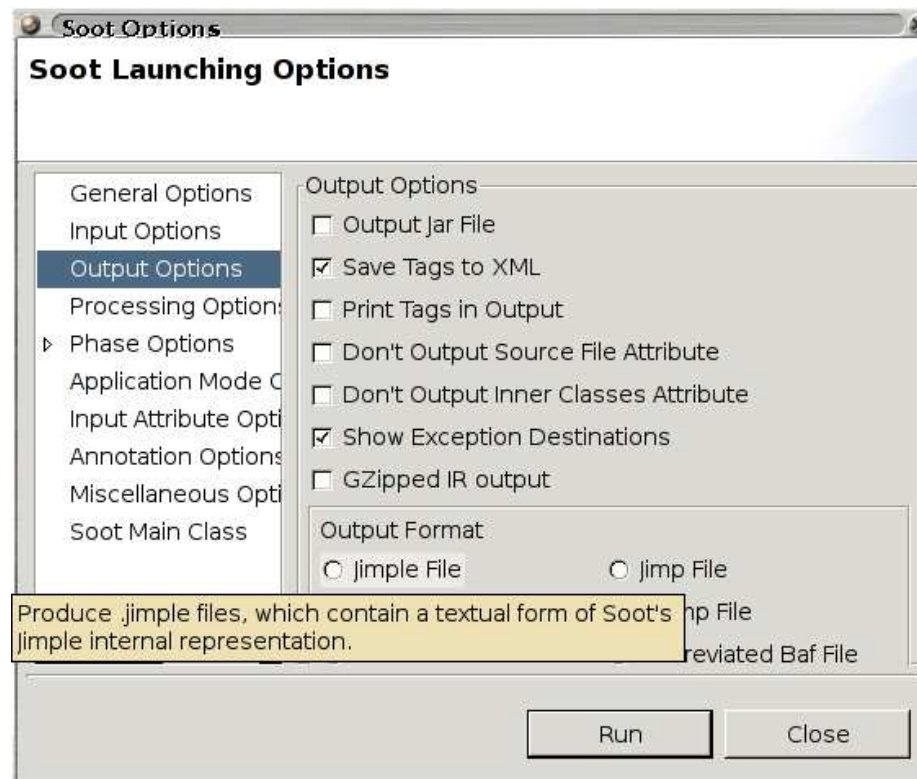


Figure 3.3: Soot Options Dialog within Eclipse

3.1.3 Managing Option Configuration

Often Soot is run with the same set of options many times. Thus, the basic plugin provides a dialog to configure a set of options and save them with a unique name. This dialog provides several options; **new**, **edit**, **delete**, **rename**, **clone** and **run** and a list of all configured option sets, as shown in Figure 3.4. The **new** option produces a dialog asking for a unique name, it then displays the options dialog and allows the user to select the options required for the configuration and then to save them. It then displays the name of the configuration in the list on the left. After configurations have been created, they may be selected and manipulated. Selecting the **edit** option displays the options dialog with the configuration's settings selected, these can then be changed and saved. Selecting **delete**, removes the configuration

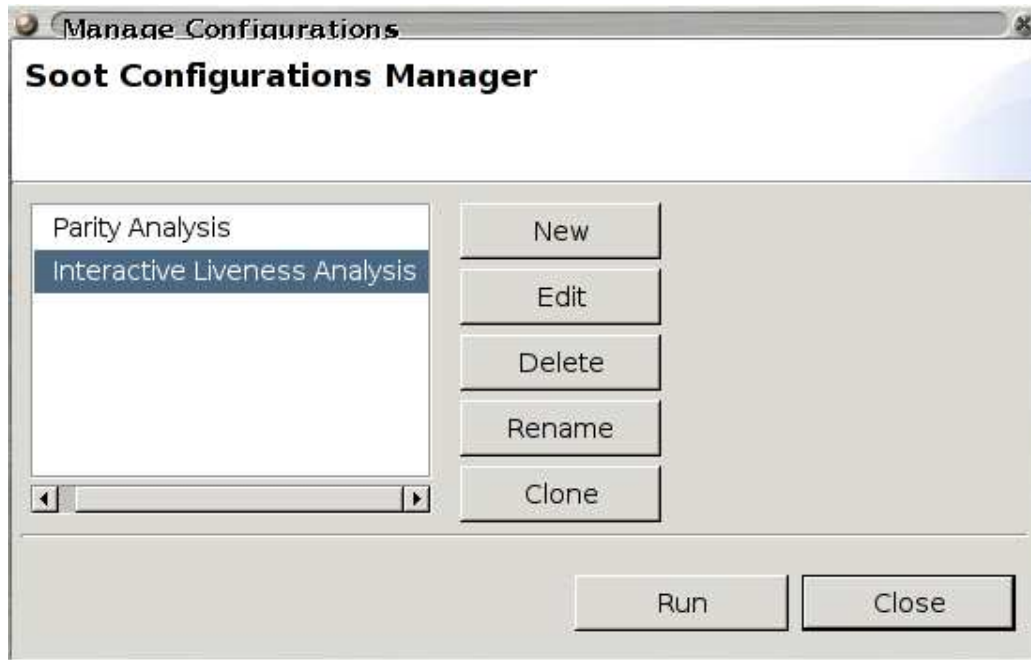


Figure 3.4: Soot Configurations Dialog within Eclipse

from the list. Selecting **rename** produces a dialog where the user can set a new name for the configuration. Selecting **clone** creates a copy of the set of options, which can then be slightly modified and saved under a new name.

Configurations are persistent and thus are available on subsequent invocations of Eclipse. Having this kind of feature is imperative in a system with so many different combinations of options. This dialog allows researchers, who may need to run the same set of options several times, to configure their system as they like.

3.2 Soot Output in Eclipse

Output from Soot is caught by a stream gobble and sent line by line to a small output view as Soot is running as shown in Figure 3.5. The output scrolls automatically. This view allows selection and copying of the text generated. This simple approach simulates the normal output one would see if running Soot as a command-line tool

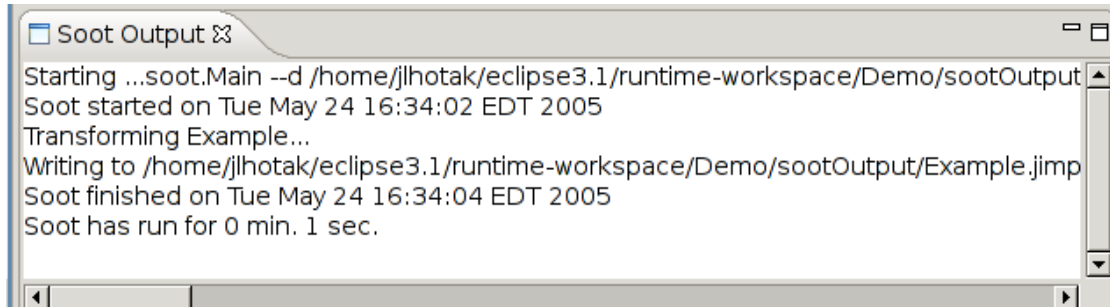


Figure 3.5: Soot Output View within Eclipse

in a shell and is a good way to bridge the gap between the shell and the graphical based application.

3.3 IR Editor

The IR editor shown in Figure 3.6 provides syntax highlighting and a content outline view for several of the Soot IRs. The editor part is shown on the left with a sample Jimple file and the content outline view, which lists the fields and methods in the class, is on the right. The content outliner is useful as the IRs are often much longer than original source code. When a selection is made in the outline the cursor selects the appropriate place in the text editor making navigating simple. Similarly, selecting text in the editor updates the selection in the outline. The outline is updated automatically upon saving the IR text after editing it. Editing the IRs by hand is sometimes necessary when debugging, especially when working with the Jimple IR. Finally, for the Jimple IR, the editor provides attribute annotations described later in chapter 5.

As most textual editors use many of the same features, it makes sense to simply extend a basic editor and make slight modifications for special features of a particular IR. This is the approach used for the IR editor, as a basic text editor is provided in Eclipse, allowing reuse of common editor functionality.

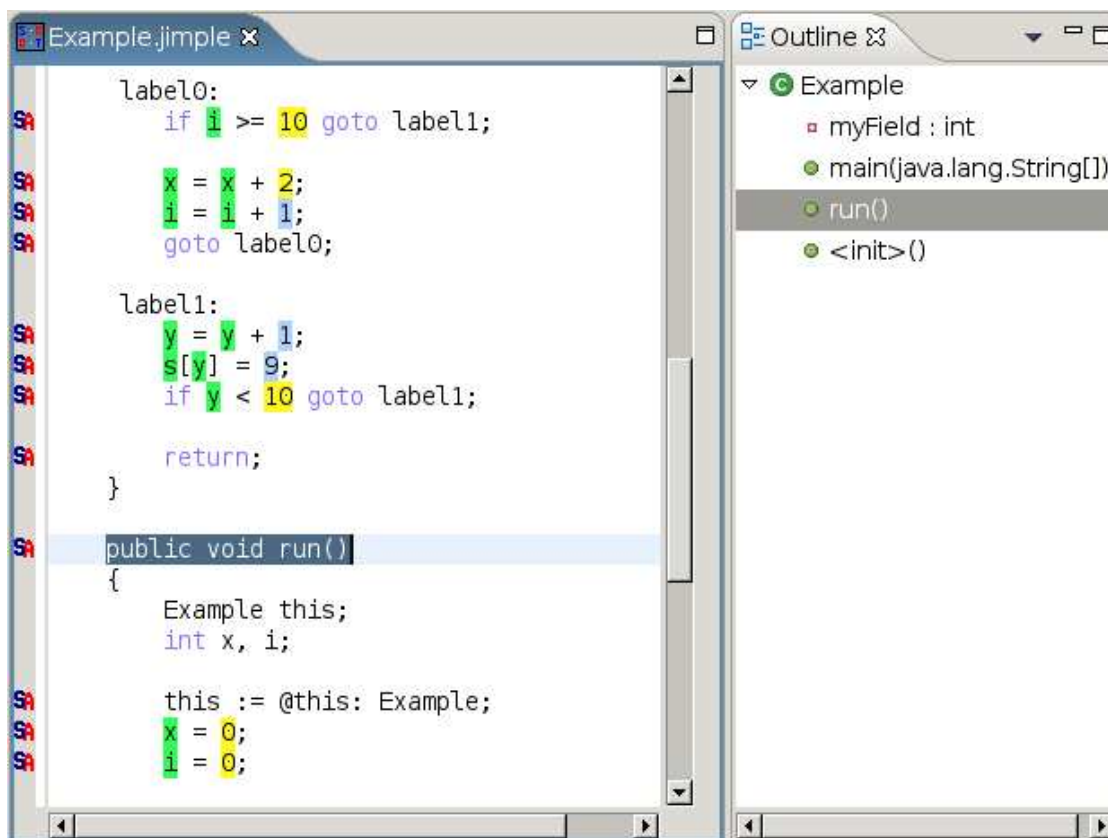


Figure 3.6: IR Editor within Eclipse

Chapter 4

Java To Jimple

4.1 Motivation

One of the main goals in this work is to provide mechanisms for viewing analysis results at the source code level. Analyses, within Soot, are usually performed on the Jimple IR and we therefore need a way to compute a clear translation between Jimple and the original Java source code on a line and column position granularity, in order that we may visualize the results of analyses at the source level. Previously, Jimple could only be generated from Java bytecode. Line number information mapping each bytecode to its original source code statement line is included in bytecode but column position information for statements and expressions and position information for methods and fields is unavailable. In order to obtain this extra position information about the original source code we use Polyglot to generate Jimple directly from the Java source code. Polyglot, a front-end compiler that converts Java source code to Java source code stores the start and end line number and the start and end column position information in each node in the generated AST, including method and field nodes. When we generate Jimple from the Polyglot AST we assign all of the position information to Jimple constructs. When the analysis results are generated, the source position information is available to provide mechanisms for easily viewing the results at the source level. This chapter describes the Java source to Jimple code generation in detail. The next chapter describes the propagation of position information.

4.2 Code Generation

We invoke Polyglot from within Soot to generate an **AST** corresponding to the source file that we want to process. There may be several classes represented in the **AST** as one can include several classes in a single source file, including both other top-level classes (other than the single public top-level class) and nested classes. We first process the entire **AST** to find all of the class declarations. We then map each Java class to a **SootClass** and proceed to process the classes one at a time.

Algorithm 2 Soot using Java source as input

```
for all classes required by Soot do  
    invoke Polyglot to build AST  
    create map of pointers to each class in AST  
    generate SootClass skeleton  
    for all method bodies needed during analyses in Soot do  
        generate body from saved pointers to Polyglot AST  
    end for  
end for
```

Each **SootClass** is initially empty with only a name with which it may be identified. We then build it up incrementally by adding modifiers, setting the super class and setting the implements clauses. These map one-to-one directly with the original source except in some special cases of nested classes to be discussed in Section 4.2.8. If the original Java class is an interface we proceed in a similar way, as the Soot representation of classes, like bytecode, does not differentiate between classes and interfaces.

Once the **SootClass** is initialized we build the outline by adding fields and methods. Fields are added in the **SootClass** as **SootFields** and a map of fields referring to their corresponding initial values is saved so the initialization code can be later created inside the initializer method (or class initializer method in the case of static fields). Methods and constructors are added to the **SootClass** as **SootMethods**. The

Jimple IR does not have a special constructor construct, but represents the constructors in methods named `<init>`. At this point parameters and exception lists are added that correspond to the original source. In the case of nested classes that need extra parameters, these extra parameters are added later on and described below in Section 4.2.8. Also, at this point initializer blocks are also stored in a map so they can be later created in the initializer method (or class initializer method in the case of static initializer blocks). This processing algorithm is given in Algorithm 2.

Once this skeleton of the `SootClass` is built with its fields and methods we build the method bodies as they are needed for analyses in Soot. The strategy is to create Jimple statements for each corresponding Java statement, inserting `nop` statements for handling control flow where needed. These `nop` statements are later removed but they are useful, enabling us to generate code from top to bottom without the need to patch up the generated code. Expressions are created completely and assigned to a single `local` that can be used when creating more complex expressions or within statements. This gives us a general mechanism for generating complicated expressions without worrying about their context allowing us to generate code in a straight-forward and elegant fashion.

Jimple has only 15 kinds of statements compared to the 24 kinds of statements in Java. Assignment statements from Java source are mapped directly to Jimple assignment statements. `If`, `while`, `for`, `do`, `try/catch/finally`, `break`, `continue` and `assert` statements are all generated in Jimple using the Jimple `if` and `goto` statements. The `synchronized` Java statement is created with `entermonitor` and `exitmonitor` Jimple statements along with `if` and `goto`. `Return` statements from the source are created using the Jimple `return` and `return void` statements. Java `switch` statements are generated using the Jimple `lookupswitch` and `tableswitch` statements. `Call` and `new` expressions and constructor call statements are created using Jimple `invoke` statements.

As most code is generated in a straightforward way we will discuss only the interesting parts in the sections below.

4.2.1 Loops

There are two important requirements while generating code for loops: first to minimize the number of jumps required and second to correctly and easily handle branches. It is important to minimize the number of jumps required for the execution of the loop in the event that it is invoked many times. For example, consider the **while** loop shown in Figure 4.1 part a). In parts b) and c) we show two ways to generate the **while** loop as Jimple. In part b) we see that we need 2 jumps to start executing the loop and 1 jump for every iteration. In part c) we do not need any jumps to start the loop, we need one for every iteration and 1 to end the execution of the loop. Therefore the code generated in part c), which is in the style of the bytecode produced by `javac`, is always more efficient than the code generated in part b).

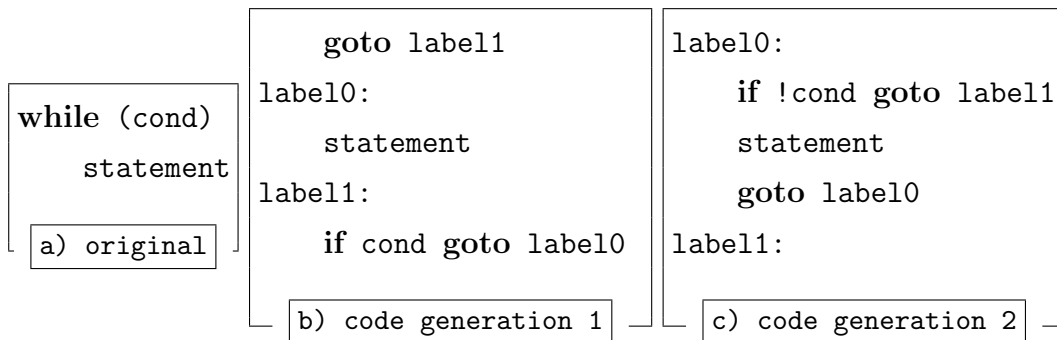


Figure 4.1: While Loop Code Generation

The more interesting part of code generation for loops is when the loop body contains one or more **break** or **continue** statements. These are represented in Jimple as **gotos**. We need to keep track of **goto** targets for each **break** and **continue** statements coming from the original source, as we often generate the target before generating the **gotos**. To accomplish this we store a **nop**, corresponding to each target, in each loop, in two stacks, one for **continue** statement targets and one for **break** statement targets. A **continue** statement target is the condition statement for **while** and **do** loops and the iterator statements for **for** loops as shown in the example code generated for a **for** loop containing a **continue** branch statement in Figure 4.4. A **break** target is the statement after the loop in all cases as shown

in Figure 4.2, showing sample code generated for a **while** loop containing a **break** branch statement. These **nops** are popped off the stack at the proper location during the loop creation. Any **break** or **continue** statement without a label is created as **goto** with the target being the corresponding **nop**. This method, shown in Algorithm 3, allows us to correctly handle multiple **break** and **continue** statements in nested loops. We also create a map storing labels of labelled statements to the corresponding **goto** target for use in the cases when the branch statement has a label target. Figure 4.3 shows the considerably longer code generated, for the example in Figure 4.2, before the **nop** statements and extra labelled blocks are eliminated.

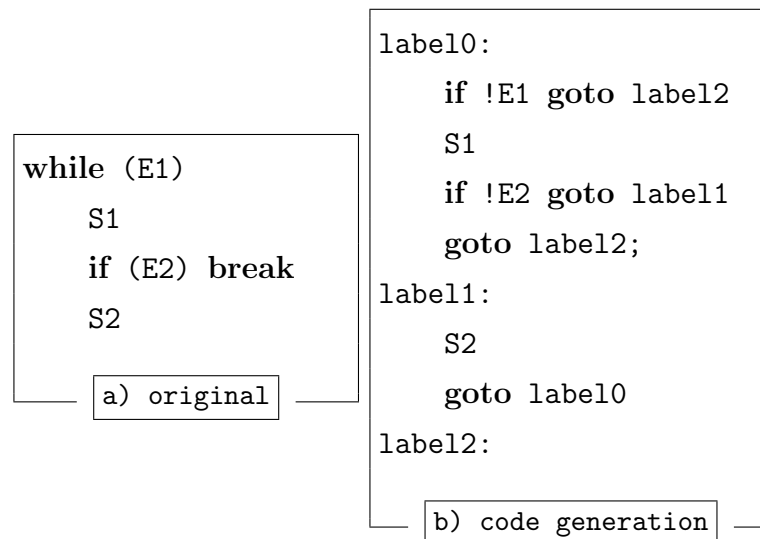


Figure 4.2: While Loop with Branch Statement Code Generation

4.2.2 Call Expressions

When creating a call expression we use the following scheme to determine which type of **invoke** to generate. If the class containing the method to invoke is an interface and the method is **abstract**, then we make an **interfaceinvoke** expression. If the method to invoke is **static**, we make a **staticinvoke** expression. If the method to invoke is **private**, we make a **specialinvoke** expression. If the call receiver is **this** or **super**, then we create a **specialinvoke** expression. Otherwise we create a

```
label0:
    nop
    nop
    if !E1 goto label3
    nop
    S1
    if !E2 goto label1
    nop
    nop
    goto label2;
label1:
    nop
    nop
    S2
    goto label0
label2:
    nop
label3:
    nop
    nop
```

c) code generation with nops

Figure 4.3: While Loop Code Generation Before Nop Elimination

Algorithm 3 While Loop Code Generation

```
cond_true_nop_stack.push ( new nop )
cond_false_nop_stack.push ( new nop )
break_nop_stack.push ( new nop )
continue_nop_stack.push ( new nop )
begin_loop_nop  $\leftarrow$  new nop
end_loop_nop  $\leftarrow$  new nop
emit begin_loop_nop
continue_nop  $\leftarrow$  continue_nop_stack.pop ()
emit continue_nop
continue_nop_stack.push ( continue_nop )
condition_expression  $\leftarrow$  generate_expression ( while_statement.expression )
cond_true_nop  $\leftarrow$  cond_true_nop_stack.pop ()
cond_false_nop  $\leftarrow$  cond_false_nop_stack.pop ()
condition_expression  $\leftarrow$  not ( condition_expression )
if condition_expression not ( constant and true ) then
    emit new if statement ( condition_expression, end_loop_nop )
end if
emit cond_true_nop
generate_statement ( while_statement.body )
emit new goto statement ( begin_loop_nop )
emit break_nop_stack.pop ()
emit end_loop_nop
emit cond_false_nop
continue_nop_stack.pop ()
```

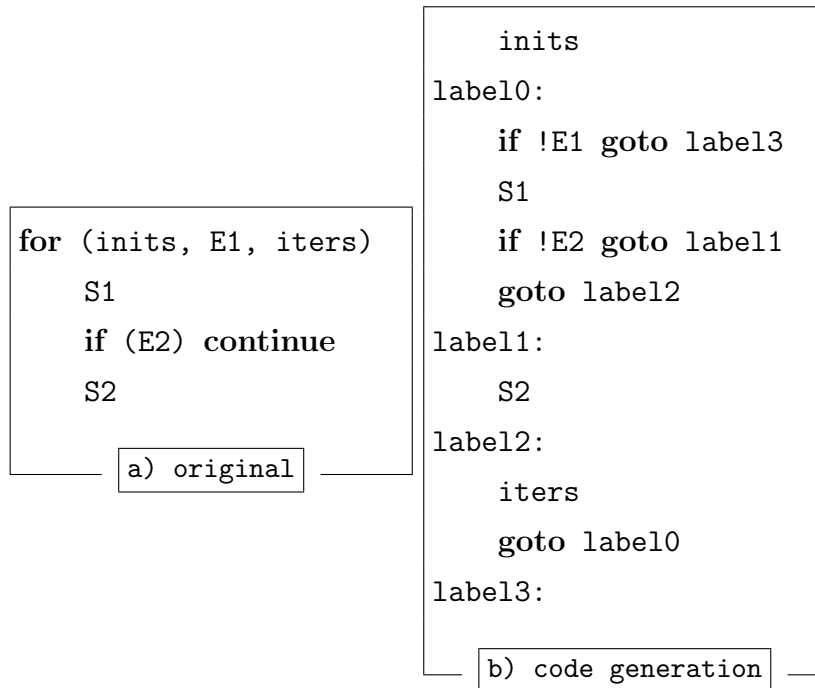


Figure 4.4: For Loop with Branch Statement Code Generation

`virtualinvoke`. When a call is in a nested class and it invokes a **private** method of an enclosing class or a **protected** method of a super class of an enclosing class, it must call a special access method and we discuss this below in Section 4.2.8. Finally, if the return type of the method to invoke is `void` we turn the `invoke` expression into an `invoke` statement directly. Otherwise, we create an assignment statement assigning a `local` to the `invoke` expression that can then be used in more complex expressions or statements.

4.2.3 Try / Catch / Finally Statements

A `try` statement with any number of `catch` statements is generated in quite a straight forward way. The `try` block statements are created with an additional `goto` statement that has a target of the first statement after the `try/catch` block. This is an example of a place where we simply insert a `nop` statement for the target. Then the first `catch` block is created again with a `goto` statement with a target of the first statement after

the `try/catch` block. Subsequent `catch` blocks are created in a similar way. Even when the `try` or `catch` statements contain a `return` statement the code generation is quite straightforward with `return` statements replacing the added `goto` statements.

The interesting part for code generation occurs when `finally` statements are introduced. The `finally` block must always be executed on every path through a `try/catch` sequence. In the general case we create the statements for the `finally` block at the end of each `try` and `catch` block just before the `goto` statement with a target of the first statement after the `try/catch` block or the `return`. This results in some duplication of code but avoids the use of Java subroutines. In practice, it appears that `finally` blocks are used so rarely that this code duplication does not cause any problems and additionally, the code generated by `javac` for the new Java 1.5 compiler follows this same approach.

If the `finally` block has a `return` statement, the `return` statement must replace the `return` statement that may have been generated from the `try` or `catch`. To do this, we push a reference to each `try` statement onto a stack just before the `try` block statements are created. Then, if we are creating a `return` statement that is within the `try` block we check to see if there is an `finally` block associated with the `try` statement. If there is a `finally` block we create it and then create the `try` block `return` statement. We do not worry about creating multiple return statements, for example one from the `try` block and one from the `finally` block, because the unreachable code eliminator phase, a standard phase available in Soot, will eliminate the unnecessary `try` block `return` statement if required. Figure 4.5 shows the generated code, where only the `return` from the `finally` block is used and where the `finally` block is always executed even in the case of an extra un-caught exception. Using a stack allows us to properly handle nested `try` statements. We follow a similar method for handling nested `catch` blocks.

4.2.4 Synchronized Statements

In general, a `synchronized` statement is generated quite easily. An `entermonitor` on the expression is created and added, then the statements from the `synchronized`

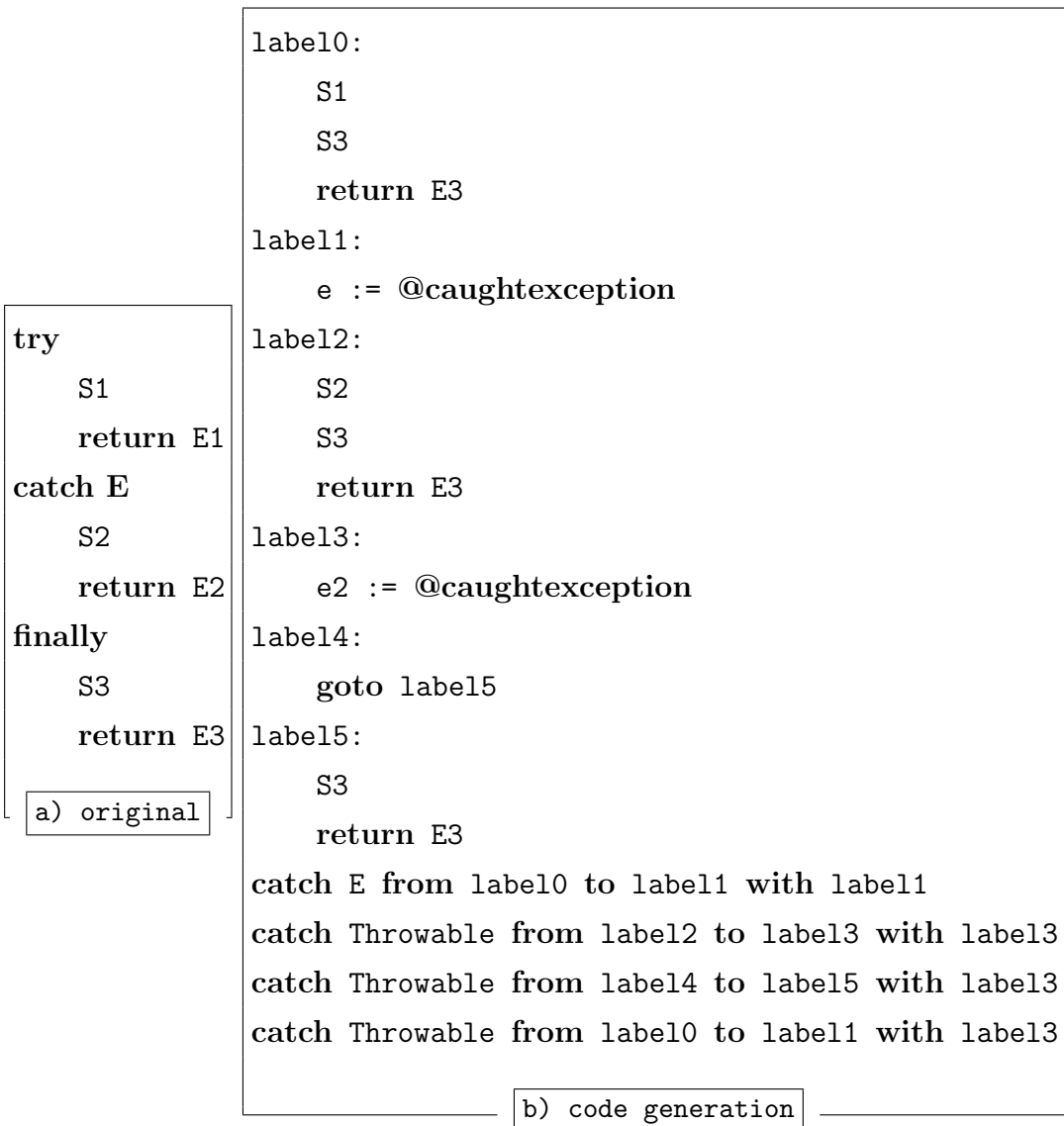


Figure 4.5: Try/Catch/Finally with Return Statements Code Generation

block, then an `exitmonitor` on the expression, and then a catch clause for any exceptions that may have been thrown from within the `synchronized` block. These exceptions need to be caught because the monitor must exit. So we generate code to catch the exceptions, exit the monitor and re-throw the exceptions. `Synchronized` statements can be generated to nest neatly inside each other.

The interesting part for generating `synchronized` statements is when `return` statements occur within the `synchronized` block as the monitor must exit before the return. In the case of nested `synchronized` blocks, the monitors must be exited in the correct order, so we save a stack of all the `entermonitor` statements and upon encountering a `return` we pop off the `entermonitor` statements one by one so that we can generate matching `exitmonitor` statements and then we push them back on to be able to generate the code to properly generate the `exitmonitor` statements as if there were no `return` statements as shown in Figure 4.6. The complete algorithm is given in Algorithm 4.

4.2.5 Array Expressions

When creating an array access expression that is on the left hand side of an assignment expression or as part of a unary expression that needs to be set, it is important to only generate the array access index once. For example, when generating an expression such as `arr[i++]++`, the `i++` needs to be generated once and used twice or the wrong array value will be set see Figure 4.7. Normally, an increment expression can be represented as an assignment statement with a binary addition expression on the right hand side, hence `arr[x]++` would be equivalent to `arr[x] = arr[x] + 1`. However, this is too complicated for Jimple and the `arr[x]` needs to be generated in an intermediate step on both sides. This is fine unless `x` is a complicated expression such as `i++` when we need to ensure it is only generated once.

4.2.6 Field Expressions

A field expression, in general, is created as a field reference in Jimple, using the field target as the field reference base if it is an instance field reference. If the field

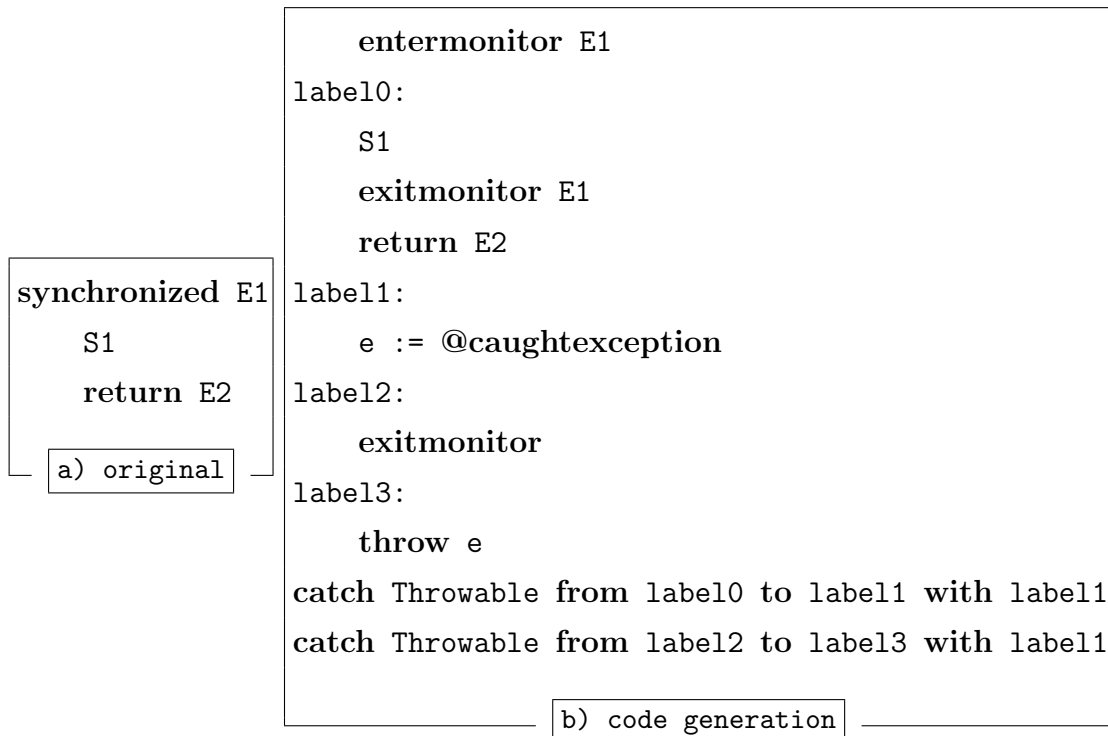


Figure 4.6: Synchronized with Return Statements Code Generation

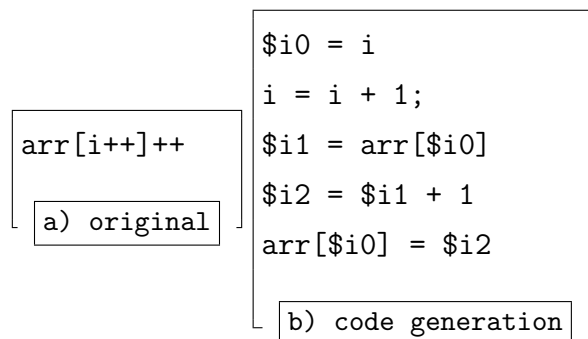


Figure 4.7: Array Code Generation

Algorithm 4 Synchronized Statement Code Generation

```
expression ← generate_expression ( synchronized_statement.expression )
emit new entermonitor statement ( expression )
monitor_stack.push ( expression )
start_nop ← new nop
emit start_nop
generate_block ( synchronized_statement.body )
emit new exitmonitor statement ( expression )
monitor_stack.pop ()
end_synchronized_nop ← new nop
goto_end_statement ← new goto statement ( end_synchronized_nop )
end_nop ← new nop
emit end_nop
emit goto_end_statement
catch_all_before_nop ← new nop
emit catch_all_before_nop
catch_all_local ← generate_local_of_type.throwable ()
emit new identity statement ( new caught exception reference, catch_all_local )
catch_before_nop ← new nop
emit catch_before_nop
catch_all_local ← generate_local_of_type.throwable ()
emit new assign statement ( catch_all_before_local, catch_all_local )
emit new exitmonitor statement ( expression )
catch_after_nop ← new nop
emit catch_after_nop
emit new throw statement ( catch_all_local )
emit end_synchronized_nop
generate_exception_regions ( start_nop, end_nop, catch_all_before_nop,
    catch_before_nop, catch_after_nop )
```

expression is representing the length of an array, instead of a field reference, a length expression is created. If the field expression is a **private** field of an enclosing class or a **protected** field of a super class of an outer class, a special access method is used to get the field. We discuss in detail access methods below in Section 4.2.8. Here we consider the special case where the field expression is the left hand side of an assignment expression or part of a unary expression that needs to be set, where we must be careful to ensure that the receiver is generated only once and then used multiple times as needed. For example, consider the example shown in Figure 4.8. When we are creating the method **meth()** in the class **Inner** we need to use an access method to get the **x** using the **foo()** receiver as the parameter but we also need to use an access method to set **x** using the **foo()** receiver again as a parameter. Thus we must create an **invoke** to method **foo()** only once or else we would actually be getting and setting **x** on different instances of **MyClass** which is generated within method **foo()**.

```
public class MyClass {  
    private int x = 9;  
    public MyClass foo(){  
        return new MyClass();  
    }  
    class Inner {  
        public void meth(){  
            foo().x += 1;  
        }  
    }  
}
```

Figure 4.8: Sample Field Reference From Nested Class

The generated Jimple for the access of **foo().x** is shown in Figure 4.9, where we see the variable **\$r1** stores the invoke of method **meth** and that it is invoked only one time.

```
...
public void meth(){
    ...
    this := @this: MyClass$Inner;
    $r0 = this.<MyClass$Inner: MyClass this$0>;
    $r1 = virtualinvoke $r0.<MyClass: MyClass foo()>();
    $i0 = staticinvoke <MyClass: int access$000(MyClass)>($r1);
    $i1 = $i0 + 1;
    staticinvoke <MyClass: int access$100(MyClass,int)>($r1, $i1);
    return;
}
...
```

Figure 4.9: Sample Field Reference From Nested Class - Generated Jimple

4.2.7 Other Stmts

Condition Expressions:

When creating conditional binary expressions with double, float and long operands we create special Jimple comparison expressions beyond the regular equal, not equal, etc. expressions used for integers. If the operands of the expression are floats or doubles and the operator is greater than or greater than or equals then we generate code for a `cmpg` expression, otherwise we generate a `cmpl` expression. If the operands are longs then we generate `cmp` expressions.

Class Literal Expressions:

If there is at least one `class` literal declared in the class then we must generate and add an extra method named `class$` which takes a single string argument. This method contains code to find and load the desired class, as well as code for handling an error in finding or loading the class. This method is added one single time per class. Additionally, for each `class` literal referenced for an object or array type, a

special field is added. Field names are made up to correspond to the standard field descriptor naming scheme.

Assert Statements:

If there is at least one **assert** statement in the class a method named **class\$** is added to the class. This method takes a single string parameter and contains code to determine if the class given by the argument passed in exists, and to throw a **NoClassDefFoundException** otherwise. A field named **class\$ClassName** is also added to the same class as this method. A field named **assertionsDisabled\$** is added to the class containing an **assert** even if that class is a nested class. Finally, if the class containing the **assert** statement does not have a class initializer method, one is added and the field **class\$** is initialized with the result of the invocation of the method **class\$**.

For processing both **assert** statements and **class** literal expressions, if we are processing a nested class, then the method named **class\$** is added to the outer class. If the outer class is an interface, then a special anonymous nested class is created to contain this method.

4.2.8 Nested Classes

Nested classes present an interesting challenge for us, as they must be created independently of their enclosing class. Jimple has no concept of nested classes. A nested class is any class declared within the body of some other class. If it is declared to be **static**, or declared in a static context, then it is like any normal top-level class and has no permission to access any members of its enclosing class. If it is non-static then it can access members of its enclosing class, including any **private** members. A local class is a named class declared within a block and an anonymous class is an un-named class declared within a block. Similarly to **static** nested classes, local and anonymous classes that are declared in a static context have no access to their enclosing class.

When we encounter a nested class we must make up a name for it, following the code generation strategy of `javac` [INN96]. In general the names are composed of the enclosing class name, a \$, and the nested class name, where there could be several levels of nested classes. When the class is a local or anonymous nested class, we invent names according to the following the scheme. For anonymous classes, we keep a counter and assign the class a name composed of the very outer most enclosing class name, a \$, and the next available number. For local classes the name is composed of the very outer most enclosing class name, a \$, the next local class number for classes with the same simple name, and the local class simple name. The number is needed for local classes as there could be many local classes with the same simple name in one enclosing class and they need to be distinguishable.

Consider Figure 4.10 where the first nested class `Inner1` is named with a composition of the enclosing class and the second nested class `Inner2` named with a chain of all enclosing classes. For anonymous classes consider the example in Figure 4.11,

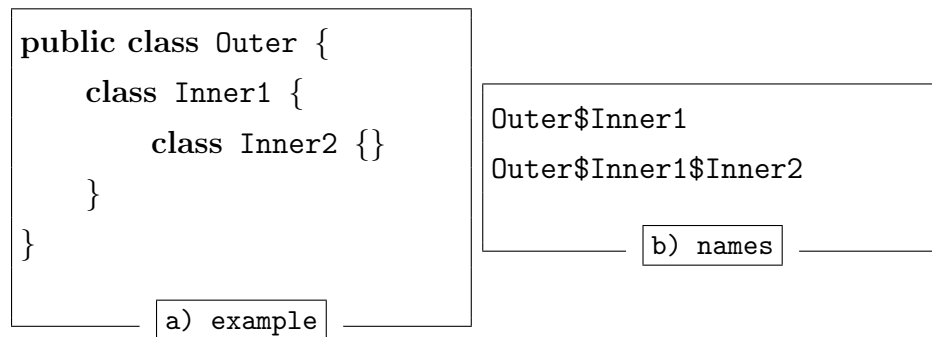


Figure 4.10: Normal Nested Classes - Naming Scheme Example

where the three anonymous class names are generated as a composition of the enclosing class and the next available number. For local classes consider the example in Figure 4.12, where both `method` and `method2` contain a local class declaration of type `Inner1`. The class names are generated as a composition of the enclosing class, the next available number and the simple local class name.

Nested classes that are not implicitly (declared in a static context) or explicitly (declared to be static) `static` can access all of the members of their enclosing class.

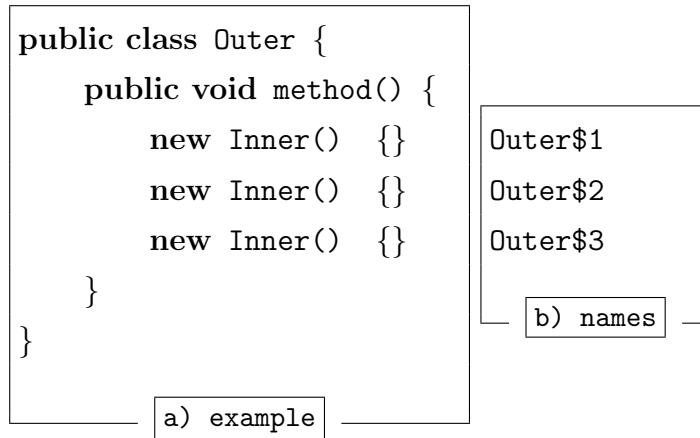


Figure 4.11: Anonymous Nested Classes - Naming Scheme Example

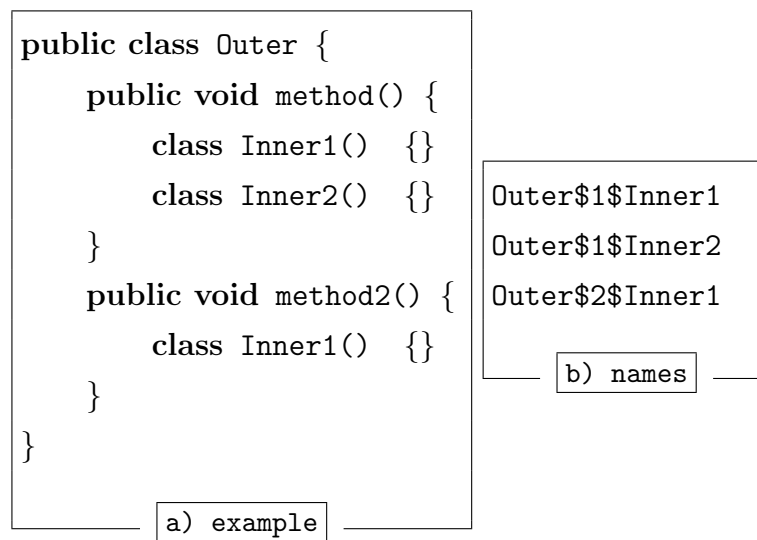


Figure 4.12: Local Nested Classes - Naming Scheme Example

To enable this functionality, some special parameters are created in the initializers, for use when invoking these nested classes. At the beginning of the parameter list for initializers of nested classes, we add a parameter corresponding to the type of the enclosing class. This is used for accessing **private** members of the enclosing class. If the class is anonymous and has a qualifier then we also add a parameter corresponding to the type of the qualifier. This is used for invoking the initializer of the super class of the anonymous class which is a nested class of the qualifying class. Local and anonymous classes can, additionally, access all of the **final** variables from their enclosing blocks. In order for this to be possible we add the types of variables needed to the end of the parameter list of the initializer of the local or anonymous class. When we are creating methods, fields and initializer blocks, we need to determine which **final** local declarations and **final** formals are available for use by contained anonymous and local classes. In methods, we find all **final** locals declared and all **final** formals not including ones declared in enclosed classes (ie: we only look one level deep). In initializer blocks we find all **final** locals. In field initializers, which can declare anonymous classes but not local classes, there are no **final** variables declared. We take all these **final** locals and formals and create a map from enclosed class type to the list of **final** locals available. At this phase of creating the method, field and initializer blocks we do not add the **final** locals as parameters to the initializer methods of the local or anonymous classes though, instead we wait until we can determine which ones are actually used, which we determine when creating the local or anonymous class.

When creating a local or anonymous class we look in the list of **final** locals available and look in the class body to determine which ones are used. These include locals used directly, locals needed for invoking some other local or anonymous class and locals needed for invoking the initializer of a local superclass (anonymous classes cannot be extended). We also look for any **new** expressions in the class body that are declaring a local or anonymous class that might need a local to be available. The following provides examples outlining the different scenarios which may arise.

For example, consider the method `myMethod`, which could be declared in a class `FinalLocals`, shown in Figure 4.13. The **final** variables available are `i`, `j` and `k`, but

```
public void myMethod(final int i, String x){
    final String j = x;
    final String k = "Hello";
    new Object() {
        public void foo(){
            System.out.println(i+" and "+j);
        }
    }
}
```

Figure 4.13: Simple Final Locals Example

we would only make extra parameters for `i` and `j` for invoking the initializer method of the anonymous local class shown in Figure 4.14.

```
public void myMethod(int, java.lang.String){
    ...
    $r0 = new FinalLocals$1;
    specialinvoke $r0.<FinalLocals$1: void
        <init>(FinalLocals,int,java.lang.String)>(this, i, j);
    return;
}
```

Figure 4.14: Simple Final Locals Example

Now consider the method `MyMethod` in Figure 4.15. In this case the `final` variable available is `i`. In `Class1` it would seem that we don't use `i`, and therefore would not create an extra parameter in the initializer method, but in fact `i` is needed for use in the initializer method for `Class2` where it is used.

Another interesting case where a local appears to not be used but is actually needed is when a local class extends another local class that uses the final, it is

```
public void myMethod(final int i){
    class Class2 {
        public void foo(){
            System.out.println(i);
        }
    }
    class Class1 {
        public void foo(){
            System.out.println("Hi");
            new Class2();
        }
    }
}
```

Figure 4.15: Final Locals - Local Class Creation Example

necessary here in Figure 4.16 because of the constructor call. In this case **Class2** needs the **final** local **i** as a parameter because it will invoke the initializer method of **Class1**, its superclass, during its initializer method and will need to pass the **i** to **Class1**.

So we modify the parameter lists for initializer methods of these local and anonymous nested classes to include these **final** variables used, as the last parameters.

Note that we only pass **final** locals into the immediately enclosed local class declared in a given method for the final locals available, even if the locals are actually used in a deeply nested class. For example, in Figure 4.17 the **i** is used in **Class2** but declared in the method immediately enclosing **Class1**. Therefore an extra parameter is added to the initializer method of **Class1** to receive **i**. When **Class2** needs to use the local **i** an access method is added to **Class1** that returns the value of the field where **i** is stored, a field named **val\$i** and **Class2** invokes the access method to get the value of **i**.

```
public void myMethod(final int i){
    class Class1 {
        public void do(){
            System.out.println("Hi: "+i);
        }
    }
    class Class2 extends Class1 {
        public void do(){
            System.out.println("hi");
        }
    }
}
```

Figure 4.16: Final Locals - Local Extends Example

```
public void MyMethod(final int i){
    class Class1 {
        public void run(){
            class Class2 {
                public void run(){
                    System.out.println(i);
                }
            }
            new Class2().run();
        }
        new Class1().run();
    }
}
```

Figure 4.17: Final Locals - One-level Only Example

New Expressions and Constructor Call Statements

When creating the code for a **new** expression or a constructor call statement it is necessary to generate and use any extra parameters that may not have been in the original source and hence need to be made up. There are two types of extra parameters; outer class references and **final** variables from the enclosing method. If a **static** inner class is being invoked, then no extra parameters are required. Otherwise an outer class reference parameter is required. It can be acquired in one of two ways: from the field named **this\$0**, that was added to the class, or from a qualifier. If there is a qualifier it is always used, otherwise the field reference is used. In the case of invoking an anonymous nested class it may be necessary to send both a reference to the outer class field named **this\$0** and the qualifier. When invoking a local or anonymous nested class from a **static** method no outer class field named **this\$0** will be available and so only the qualifier is sent if one exists. The second type of extra parameters are necessary when creating new instances of local or anonymous classes when **final** local variables from the current method must be made available to nested class. The **final** local variables which are needed are pre-determined at the time of creating the nested class and are stored in a map available at the time of instance creation.

Inner Class Attribute Tags

When generating inner classes, we must add inner class **Tags**, which are later turned into inner class attributes. A class adds an inner class **Tag** about itself if it is an inner class, a **Tag** about any of its immediately enclosed classes, **Tags** about all of its enclosing classes, a **Tag** about any inner class that it makes a new instance of, and a **Tag** for any inner class that it extends. In addition, when **assert** statements or **class** literals are created in a nested class of an interface and a special anonymous class is created to handle the method named **class\$** an inner class **Tag** is added to the interface.

Accessing Outer Class This

When processing inner classes it is often necessary to get a reference to an enclosing class sometimes many levels away. We provide a general mechanism to easily acquire this reference for a given type. First we check to see if the type we need is the current class then we can return the current `this` local, then we check to see if we need a local representing the immediately enclosing outer class in which case we can return a local equal to a field reference of the field named `this$0`. Otherwise, we add a `static` method to the outer class that takes as an argument a reference to outer class field named `this$0` and returns a reference to the outer class' outer class field named `this$0`, continuing up the chain of outer classes as necessary. Accessing the outer class is primarily used when adding the outer class field reference as an argument for `new` expressions and constructor call statements, and when generating `this` or `super` expressions. For `new` expressions and constructor call statements the outer class field reference argument is needed if there is no qualifier, or always for anonymous classes, in cases where an argument is required. For special expressions it is needed when accessing super classes of enclosing classes. It is also used when a `final` local is needed from an enclosing class, that is possibly several levels away. In general to find `final` locals first we look to see if the local is declared in the current body and if it is then we use it, then we look to see if there is a field named `val$local_name` and use that if it exists, otherwise we get a reference to the `this` for the type of the field named `this$0` to get the next enclosing class and then look for a field named `val$local_name` there and then continue until it is found.

Accessing Methods

When an nested class accesses a `private` member of an enclosing class, or a `protected` member of a super class of an enclosing class, it must do so via a special access method. We generate three types of access methods: one for calling methods, one for getting fields and one for setting fields as they are required, in otherwords we do not generate access methods unless an enclosed class needs to access a particular enclosing class' member. For calling `private` methods from an enclosing class or for calling

`protected` methods of a super class of an enclosing class, we add the access method to the enclosing class. This access method is `static` and takes one argument that can be used as the receiver if the method is an instance method in the enclosing class. The return type is the return type of the method being invoked. For accessing `private` or `protected` fields, we again add the access method to the enclosing class. This access method is `static` and takes one argument that can be used as the receiver if the field is an instance field in the enclosing class. The return type is the field type. For setting `private` or `protected` fields, we add an access method to the enclosing class that takes two arguments; one that can be used as a receiver for an instance field and a value that should be assigned to the field. As these access methods are created we put them in a map, so they are not re-created but are available for re-use.

4.3 Summary

This chapter explained the challenging parts of generating Java source code into Jimple, the first step in achieving our goal of providing visual attribute information at the source code level. The next chapter discusses, in detail, how the required position information is mapped from the source to the Jimple, the schemes for encoding analysis information and the mechanisms for displaying the analysis results.

Chapter 5

Viewing Static Analysis Results

5.1 Motivation

Research compilers are often used when experimenting with new compiler analyses or for learning about program analysis. They may even be used by general programmers as specialized tools, for example for information about the code for refactoring. When compiler analyses are performed, their results are used for transforming the code or are encoded in the output for use in other tools and thus it was difficult to examine them. There was no general format for viewing the results generated by compiler analyses.

In this work, we present a general framework for graphically displaying the results of compiler analyses. We describe how information needed for displaying the results is calculated. We describe a general way to encode the generated information, using three types of **Tags**, which were created for visualization purposes. We then describe how this information encoded in **Tags** is presented to the user.

This framework enables researchers and students to visualize the results of their analyses. The mechanisms provide a standard way to view many different types of analysis results, not only for analyses already in Soot, but also for many new analyses not yet discovered.

In versions of Soot prior to version 2.0, the results of analyses were generated in the IR in an embedded textual format. This format was difficult to read as the

results were intermingled with the code and unformatted in any way. As well, the results were not available at all in the source code. The framework presented in this chapter provides visualizations of analysis results in the Jimple IR and in the original source code in a sophisticated, yet easy to understand graphical format. Displaying the results in the IR provides a way to view the results in an uncluttered way, so they may be used for debugging and for verifying the correctness of the analyses. Providing the visual results of the analyses in the original source code is useful for people who are unfamiliar with the IR, such as new students. They may use the results displayed in the source code for understanding the analysis, performed on the IR, in relation to the original code. It is also useful to see the effects of the analysis on the original code.

Once generated, these visualizations of results can be used by students writing their first analysis to easily see if and where they are making mistakes. For example, in writing a simple parity analysis that assigns the colour blue to odd locals and yellow to even locals, it is easy to see which are correct and which are incorrect simply by glancing at sample test programs either in the generated Jimple IR or the original source. For researchers writing more complicated analysis, these visualizations provide a way to quickly see where the analysis is incorrect or imprecise.

The general framework for displaying analysis results in a visual way works as follows. When creating Jimple, position information is assigned to each Jimple construct. The analyses are performed and the results are encoded in special visualization **Tags**. All **Tags** encoding analysis results and position information are collected and formatted in a way that enables communication between Soot and the plugin which can display the information. When the source or IR code is viewed within the plugin in Eclipse, the analysis results are displayed automatically in a visual format. In the following sections we discuss in detail, the mapping of position information between the source code and the IR, the **Tags** for encoding analysis results and the mechanisms used to display the different types of analysis information, how the **Tags** are collected and formatted for use within Eclipse and how the plugin handles this generated information and manages its display.

5.2 Mapping Source and IR Position Information

In order to generate visual information at a high level of precision, access to detailed line and column position information about the original source code and the IR is necessary. This section explains the details of collecting the position information and associating it with Jimple constructs during the Jimple creation phase.

5.2.1 Position Origins

When using bytecode as the input to Soot, the line numbers of statements in the original source code are stored in the line number table attribute [LY99] as shown in boxes 1 and 2 of Figure 5.1. When the internal Jimple representation is generated

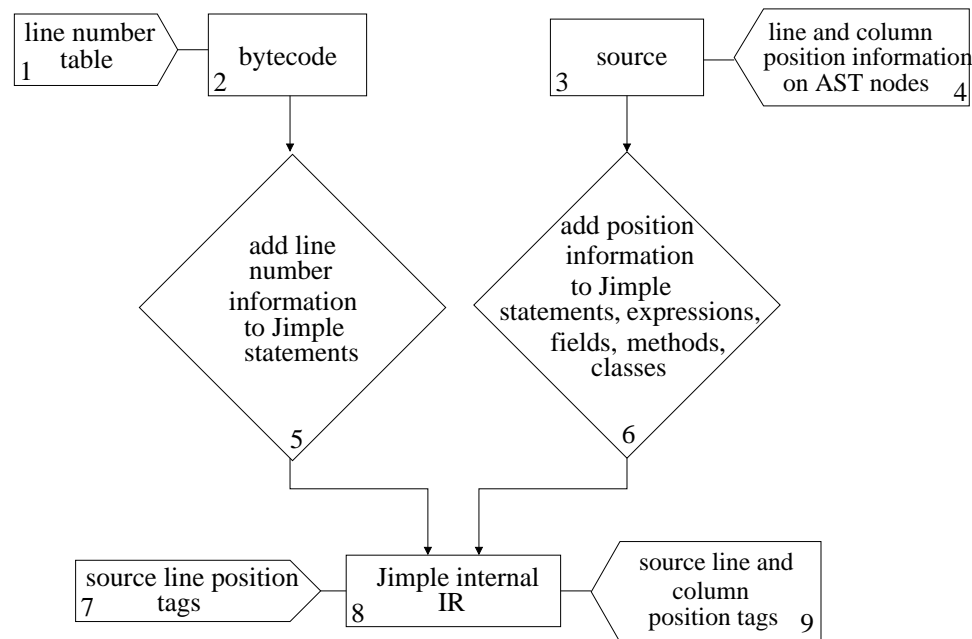


Figure 5.1: Overview of Generated Position Information

these line numbers are stored in **Tags** and attached to the corresponding Jimple statements, as part of the standard Soot attribute framework, shown in box 5 of Figure 5.1. When using bytecode as input there is no position information available for classes, methods or fields and there is no column information available. Consequently,

when using bytecode as input the visual information that can be displayed in the original source code is limited to only visual information relating to entire statements. When using source code as the input to Soot, line number and column position information is available in each node in the Polyglot **AST** as shown in boxes 3 and 4 of Figure 5.1. When Jimple is created the position information is stored in **Tags** on the appropriate **Hosts** (statements, expressions, fields, methods and classes) as described below in Section 5.2.2 and shown in Figure 5.1, box 6. This position information includes column positions. Figure 5.1 shows an overview of the entire collection and generation of position information when starting from bytecode or source and ending with Jimple, where the Jimple constructs have position **Tags**, containing all available position information, depending on the original input, attached as shown in boxes 7, 8 and 9. These position **Tags** contain as much position information as is available from the input. This implies that the level of precision of the visualizations is dependent on the availability of the position information.

5.2.2 Position Information Assignment for Source Input

Each node in the Polyglot **AST** has position information comprised of start and end line numbers and start and end column positions. This information is propagated to the corresponding Soot component as it is being created. This section describes the mapping between the Polyglot **AST** nodes and the Soot components and describes how the position information is assigned to the Soot components, when using source code as the input.

Each original source file may have one or several class declarations. Each class declaration is mapped to a **SootClass** and the **SootClass** is assigned a **Tag** with all position information associated with the class declaration. If the source file contains an anonymous class, a class declaration for the anonymous class is not created or represented in the Polyglot **AST**. In this case, we assign the start position of the **new** expression containing the anonymous class' declaration and the end position of the anonymous class' body as the position information assigned to the generated **SootClass**.

5.2. Mapping Source and IR Position Information

Each class declaration has zero or more class members given in any order. Here we describe how position information is assigned for each kind of class member. For each method declaration a `SootMethod` is created. If the method does not have a body, for example if it is an abstract method declaration, then the `SootMethod` is assigned the method declaration position information. If the method has a body, then the `SootMethod` is assigned the start position information from the method declaration and the end position information from the block representing the method body.

Each field declaration is translated to a `SootField` which is assigned position information from the field declaration. The Jimple statement that creates the field initializer is assigned the field declaration initializer position information.

Each initializer block is generated inside the class initializer method and the Jimple statements created are assigned position information corresponding to the individual statements in the block.

The position information for each type of statement is derived as described below. Whenever there is an expression `E` inside a statement in the original source code, it may be generated into several Jimple statements, each of which expresses part of the expression `E` and stores the partial result in a temporary variable. The temporary variable of the last generated statement is then used in the statement corresponding to the original statement. Each of these generated statements is assigned position information of the original expression `E`, however in the following discussion of position information assignment of statements we show the expression `E` in an unexpanded format.

Assert Statement:

An assert statement of the form `assert E1 [: E2]`, where `E2`, the error message expression is optional, is translated into several statements in Jimple. First a field access statement is generated which determines whether assertions are enabled and if they are not enabled, code to skip all assertion handling. Then an `if` statement to determine if the assertion is true is added. This `if` statement is assigned the position information of the original `assert` statement. The `if` statement condition

box is also assigned the position information from the `assert` statement condition. Finally, code to handle an assertion error is generated, including an invocation of the `AssertionError` class which takes an error message as an argument. This argument box is assigned the position of the optional error expression of the original `assert` statement. This assignment of position information is shown in Figure 5.2.

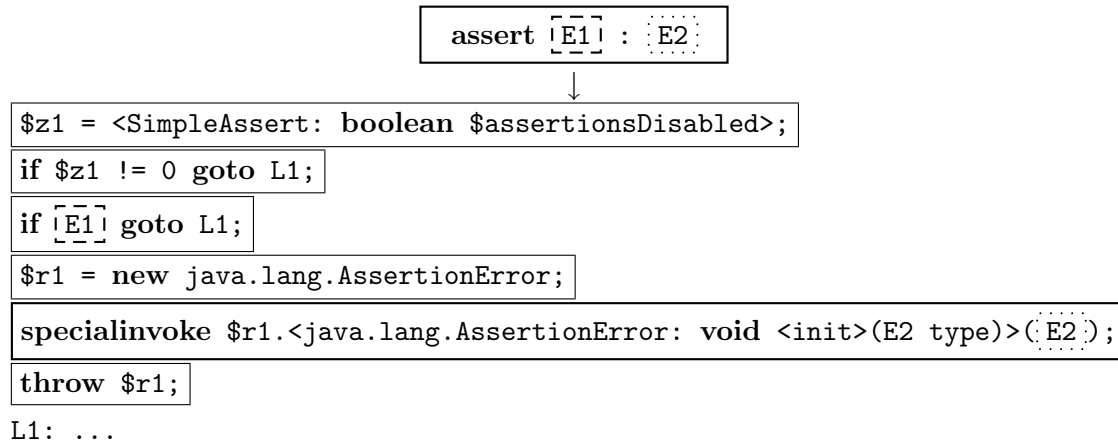


Figure 5.2: `Assert` Statement Position Information Generation

In this figure the box before the arrow represents the original source code `assert` statement and the boxes after the arrow represent the generated Jimple. The box line styles represent the position information and how it is propagated from the source to the Jimple IR. For example, in this example the original source statement could have `line number = 5`, `start column = 10`, `end column = 23`. This position is represented by the solid line box. Each of the generated Jimple statements which are explicitly encoding this source statement are assigned this position information. This is represented by the solid line boxes around each generated statement. The `assert` expression `E1` is a second construct which has position information associated with it. For example it may have the position information `line number = 5`, `start column = 17`, `end column = 18` which is represented using the dashed line. In the generated Jimple as indicated by the dashed line this position information is assigned to the `if` condition expression. Finally, the `assert` statement error expression `E2` could

5.2. Mapping Source and IR Position Information

have position information of `line number = 5, start column = 22, end column = 23` as indicated by the dotted line. In the generated Jimple the `specialinvoke` expression argument box is assigned this position information as shown with the dotted line.

Block Statement:

Each statement in a compound block statement is assigned the position information of the corresponding statement from the original source.

Branch Statement:

A branch statement of the form `break [label]` or `continue [label]`, where the `label` is optional is generated in Jimple as a `goto` statement. This `goto` statement is assigned the position information of the original branch statement.

Constructor Call Statement:

A constructor call statement is of the form `super([arguments])` or `this([arguments])` and is translated into several assignment statements to handle the arguments, followed by a `specialinvoke` statement. Each assignment statement created for handling an argument is assigned position information of the corresponding original argument. Each argument box in the invoke statement is also assigned the position information from the corresponding argument and the invoke statement is assigned the position information of the constructor call statement as shown by the box line styles in Figure 5.3.

Do Statement:

For a `do` statement of the form `do body while E`, each statement in the body is generated followed by an `if` statement to handle the loop. The `if` statement and the condition box of the `if` statement are assigned the position information of the `while` condition in the original source as shown in Figure 5.4.

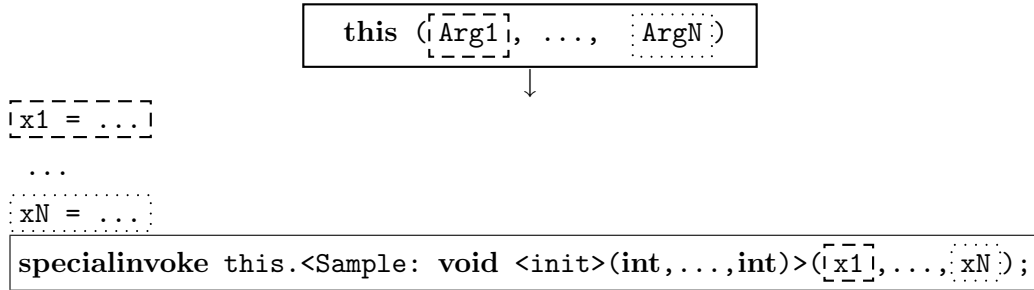


Figure 5.3: Constructor Call Statement Position Information Generation

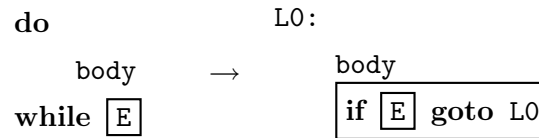


Figure 5.4: Do Statement Position Information Generation

For Statement:

A **for** loop of the form **for** (initializers, E, iterators) body is generated into several statements in Jimple. The initializers are generated as assignment statements which are assigned the position information of the original initializers. An **if** statement with a reversed condition expression is generated representing the **for** condition expression E. This **if** statement and the **if** statement condition box are both assigned the position information of the original **for** condition expression E. The body is generated and then the iterators are generated as binary assignment statements and are assigned position information from the original iterators. This assignment of position information is shown in Figure 5.5.

If Statement:

An **if** statement is generated as an **if** statement in Jimple. The generated **if** statement and its condition box are both assigned the position information of the original **if** statement condition expression as shown in Figure 5.6. The original **if** statement expression may actually be generated into several Jimple statements before being

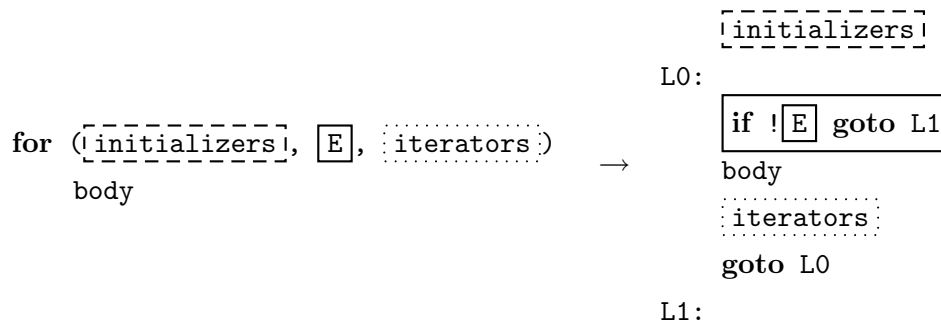


Figure 5.5: For Statement Position Information Generation

used in the generated if statement. Each of these statements are assigned position information of the original expression that they represent.

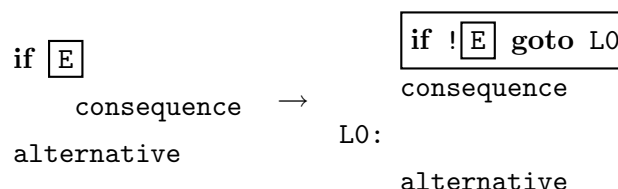


Figure 5.6: If Statement Position Information Generation

Local Declaration Statement

A local declaration statement of the form `T id [= E]`, where `T` represents the type and the initializing expression `= E` is optional is generated into an assignment statement in Jimple. The assignment statement and the identifier box are both assigned position information from the local declaration statement. The right operand box of the assignment statement is assigned the position information of the optional expression as shown in Figure 5.7.



Figure 5.7: Local Declaration Statement Position Information Generation

Return Statement:

A **return** statement is of the form **return** [E] and is generated to a **return** void or a **return** statement which is assigned the position information of the **return** statement from the source. If returning an expression E, then the generated expression is assigned position information of the original expression.

Switch Statement:

A **switch** statement is generated into either a **lookupswitch** statement or a **tableswitch** statement in Jimple. The key box in the generated expression is assigned position information of the original **switch** expression. The statements generated for the individual **switch** block statements are assigned corresponding position information based on the statement type. The position assignment is shown in Figure 5.8.

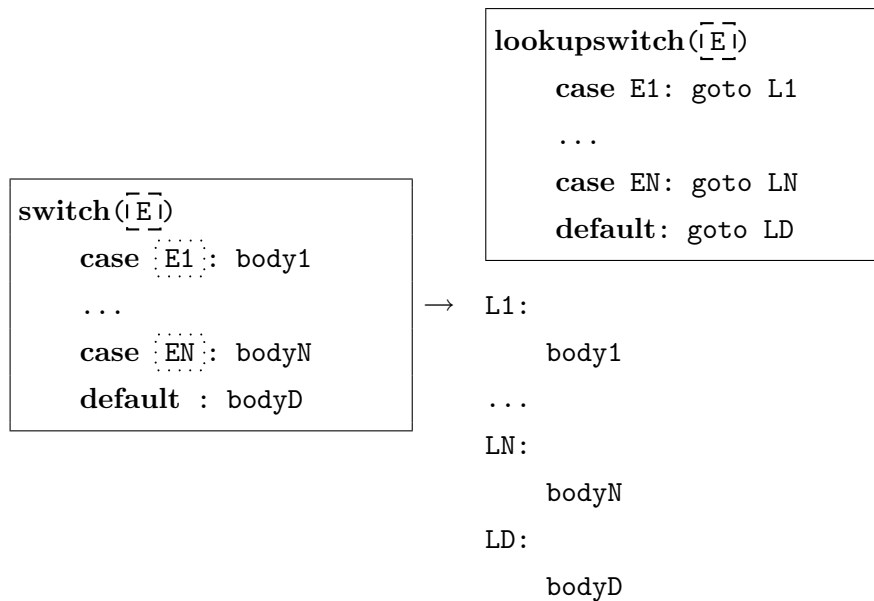


Figure 5.8: Switch Statement Position Information Generation

Synchronized Statement:

A **synchronized** statement is generated into a series of **entermonitor** and **exitmonitor** statements. The **entermonitor** statement, the **entermonitor** statement operand box, the **exitmonitor** statement and the **exitmonitor** statement operand box are all assigned the position information of the **synchronized** statement expression as shown in Figure 5.9. All statements created for the statements in the **synchronized** statement's body are assigned position information of the corresponding individual statements.

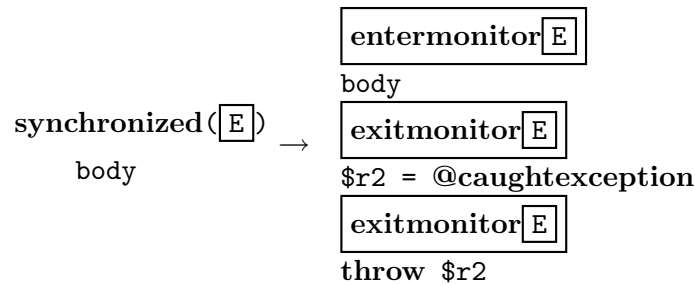


Figure 5.9: Synchronized Statement Position Information Generation

Throw Statement:

A **throw** statement is generated into a Jimple **throw** statement. The Jimple **throw** statement is assigned position information of the original **throw** statement. The generated **throw** statement operand box is assigned the position information from the **throw** statement expression.

Try/Catch Statement:

Statements created from the **try** and **catch** bodies are assigned position information on a per statement basis. The `@caughtexception` statement and the operator on the left hand side are assigned position information from the formal in the **catch** clause as show in Figure 5.10

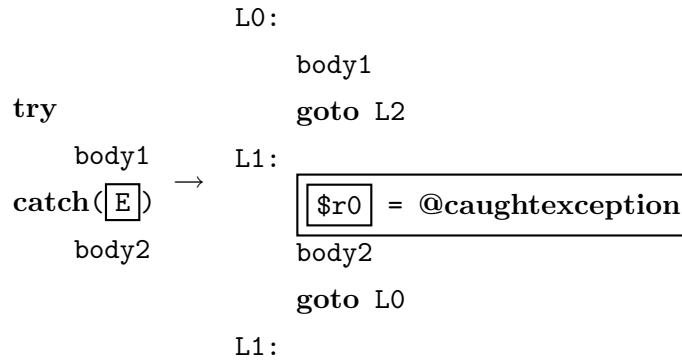


Figure 5.10: Try/Catch Statement Position Information Generation

While Statement:

A `while` statement of the form `while E body` is generated into an `if` statement which breaks out of the loop if unsatisfied. This `if` statement and the `if` statement condition box are assigned the position information of the original `while` condition as shown in Figure 5.11. Each statement in the generated body is assigned position information of the corresponding original statement.

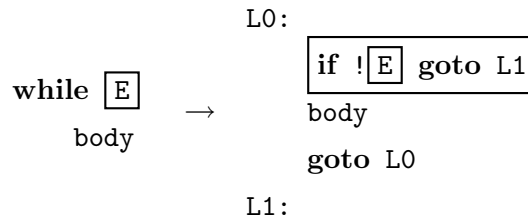


Figure 5.11: While Statement Position Information Generation

The expressions are assigned position information as described below.

Array Access Expression:

An array access expression results in an array reference statement in Jimple. The array reference statement is assigned the position information of the array access expression. The array reference statement base box is assigned the position information from the array access expression array and the array reference statement index box

is assigned the position information of the array access index, which may be a complicated expression. This assignment of position information is shown in Figure 5.12.

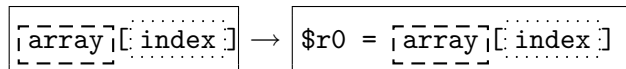


Figure 5.12: Array Access Expression Position Information Generation

New Array Expression:

A **new** array expression results in a **newarray** expression or a **newmultiarray** expression, embedded in an assignment statement in Jimple. The generated assignment statement and the **newarray** or **newmultiarray** expression are assigned the position information of the original **new** array expression. The **newarray** or **newmultiarray** expression box is also assigned the position information of the original **new** array expression. The **newarray** size box and the **newmultiarray** size boxes are assigned position information of the original **new** array expression dimensions, which may be complicated expressions. This assignment of position information is shown in Figures 5.13 and 5.14.

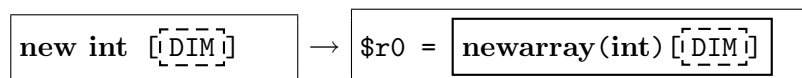


Figure 5.13: New Array Expression Position Information Generation

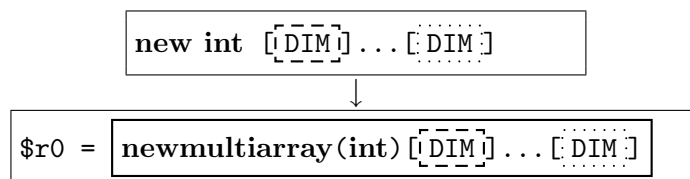


Figure 5.14: New Multi-Array Expression Position Information Generation

Array Initializer Expression:

An array initializer expression is generated into a series of assignment statements, each of which are assigned position information of the corresponding original expression as shown in Figure 5.15.

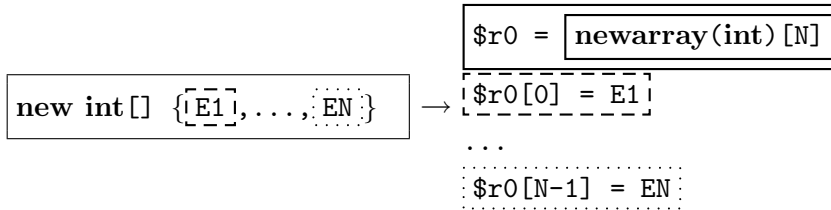


Figure 5.15: Array Initializer Expression Position Information Generation

Assignment Expression:

An assignment statement of the form `id [op] = E`, where the operator `op` is optional and `id` represents an identifier, is generated into a Jimple assignment statement of the form `id = [id op] E`. The generated assignment statement is assigned position information of the original assignment expression. The left box is assigned the identifier position information and the right box is assigned the expression position information. If there is an operator then a binary expression is created and assigned the position information of the original expression. The binary expression left box is assigned the position information of the identifier and the right box is assigned the position information of the expression. This assignment of position information is shown in Figure 5.16 and 5.17.



Figure 5.16: Assignment Expression Position Information Generation

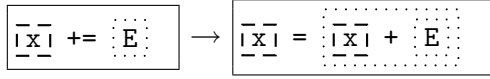


Figure 5.17: Assignment with Operator Expression Position Information Generation

Binary Expression:

A simple binary expression of the form $E1 \text{ op } E2$ is generated into a Jimple binary operator expression, where the left operand box is assigned position information of the left expression $E1$ and the right operand box is assigned position information of the right expression $E2$. The generated binary operator expression is assigned position information of the original binary expression.

A conditional and binary expression of the form $E1 \ \&\& \ E2$ is generated into a series of `if` and assignment statements. The first `if` statement which evaluates $E1$ is assigned position information of $E1$, as is the `if` statement condition box. The second `if` statement which evaluates $E2$, if $E1$ was found to be true, is assigned position information of $E2$, as is the `if` statement condition box. The assignment statements which set the result of the conditional and binary expression are assigned position information of the original binary expression as shown in Figure 5.18.

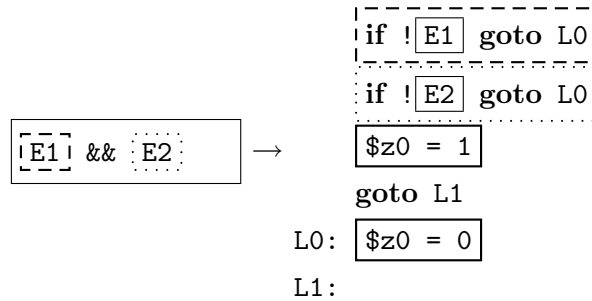


Figure 5.18: Conditional And Binary Expression Position Information Generation

A conditional or binary expression of the form $E1 \ || \ E2$ is generated into a series of `if` and assignment statements. The first `if` statement which evaluates $E1$ is assigned position information of $E1$, as is the `if` statement condition box. The second `if` statement which evaluates $E2$ is assigned position information of $E2$, as is the `if`

statement condition box. The assignment statements which set the result of the conditional or binary expression are assigned position information of the original binary expression as shown in Figure 5.19.

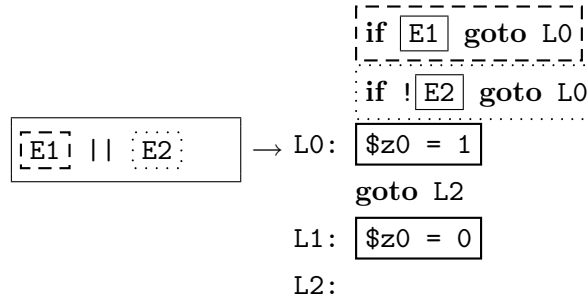


Figure 5.19: Conditional Or Binary Expression Position Information Generation

Call Expression:

A call expression is generated as an invoke statement expression, which is assigned the call expression position information. The invoke statement expression argument boxes are assigned position information of the call expression arguments and the the invoke statement expression base box is assigned position information of the call expression target as shown in Figure 5.20.

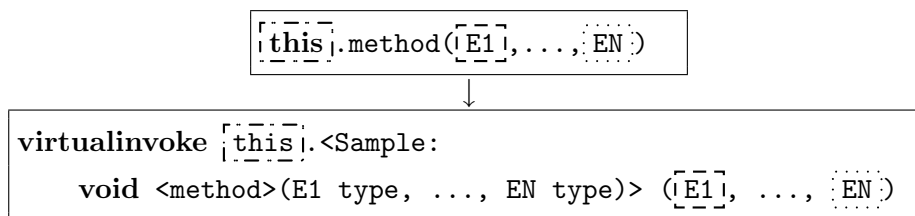


Figure 5.20: Call Expression Position Information Generation

Cast Expression:

A cast expression is generated as a cast expression statement. The cast expression statement is assigned position information of the original cast expression. The cast

expression statement operand box is assigned position information of the cast expression as shown in Figure 5.21.

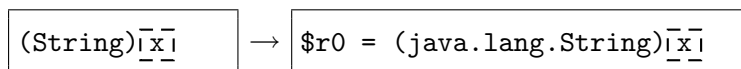


Figure 5.21: Cast Expression Position Information Generation

Conditional Expression:

A conditional expression of the form $E1 \ ? \ E2 \ : \ E3$ is generated into several Jimple statements. The condition expression $E1$ is generated as an `if` statement, which is assigned position information of the conditional expression. The `if` statement condition box is assigned position information of the condition expression $E1$. The consequence expression $E2$ is generated as an assignment statement which is assigned position information of the conditional expression and whose right operand box is assigned position information of the consequence expression $E2$. The alternative expression $E3$ is generated as an assignment statement which is assigned position information of the conditional expression and whose right operand box is assigned position information of the alternative expression $E3$. This assignment of position information is shown in Figure 5.22.

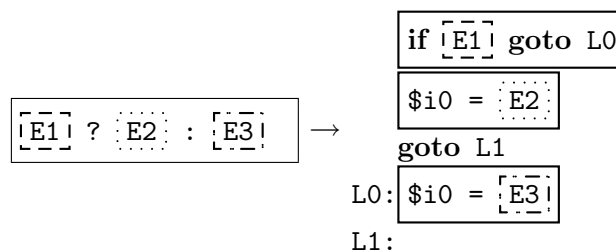


Figure 5.22: Conditional Expression Position Information Generation

Field Expression:

A field access expression is generated as a field reference as part of an assignment statement in Jimple. This assignment statement is assigned position information of the field expression, as is the right operand box of the assignment statement. The base box of the field reference is assigned the position information of the field target as shown in Figure 5.23.

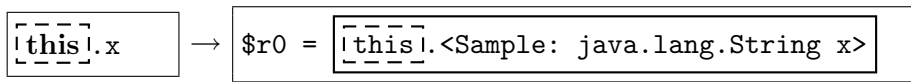


Figure 5.23: Field Expression Position Information Generation

Instanceof Expression:

An `instanceof` expression of the form `E instanceof T`, where `T` is a type, is generated as an `instanceof` expression in Jimple and embedded in the right hand side of an assignment statement. This assignment statement and the `instanceof` expression are assigned the position information of the original `instanceof` expression. The `instanceof` expression operand box is assigned position information of the expression `E` as shown in Figure 5.24.

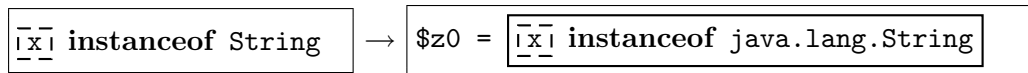


Figure 5.24: Instanceof Expression Position Information Generation

Literal Expression:

Position information is not explicitly assigned to generated literal expressions as they are always part of some other expression. They do not exist on their own.

Local Expression:

Position information is not explicitly assigned to generated locals as they are always part of some other expression. They do not exist on their own.

New Expression:

A **new** expression is generated into a Jimple **new** expression, which is embedded in an assignment statement, and a **specialinvoke** expression statement. The **new** expression and **new** expression assignment statement are assigned position information of the original expression, as is the **specialinvoke** expression statement. The argument boxes in the **specialinvoke** expression statement are assigned position information of the corresponding arguments in the original **new** expression as shown in Figure 5.25

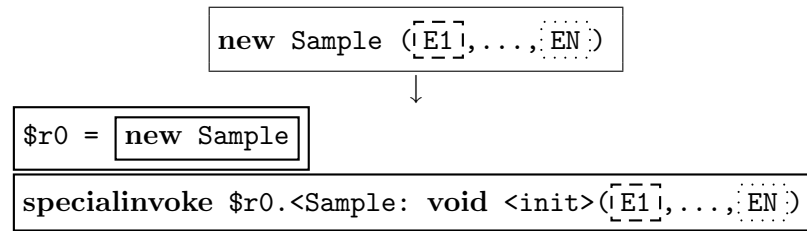


Figure 5.25: New Expression Position Information Generation

Special Expression:

Position information is not explicitly assigned to code generated for special expressions of the form **this.E** or **super.E** as special expressions are always part of the target of field or call expressions.

Unary Expression:

The pre- and post- increment and decrement unary expressions are generated as assignment statements with a simple binary add or subtract expression on the right hand side. The assignment statement is assigned position information of the unary

expression and the binary expression left operand is assigned position information of the unary operand expression as shown in Figure 5.26.

$$\boxed{\overline{\overline{E}}++} \rightarrow \boxed{\overline{\overline{E}} = \overline{\overline{E}} + 1}$$

Figure 5.26: Simple Unary Expression Position Information Generation

A unary plus expression of the form $+E$ is generated as the right hand side of an assignment statement. The assignment statement is assigned position information of the unary plus expression and the assignment statement right operand is assigned position information of the unary plus expression operand E , as shown in Figure 5.27.

$$\boxed{+\overline{\overline{E}}} \rightarrow \boxed{\$i0 = \overline{\overline{E}}}$$

Figure 5.27: Unary Plus Expression Position Information Generation

A unary minus expression of the form $-E$ is generated as a negative expression on the right hand side of an assignment statement. The assignment statement and its right operand box are assigned position information of the unary minus expression. The negative expression operand box is assigned position information of the unary minus expression operand E , as shown in Figure 5.28.

$$\boxed{-\overline{\overline{E}}} \rightarrow \boxed{\$i0 = \boxed{\text{neg}\overline{\overline{E}}}}$$

Figure 5.28: Unary Minus Expression Position Information Generation

A unary bitwise complement expression of the form $\sim E$ is generated as an exclusive or expression on the right hand side of an assignment statement. The assignment statement and its right operand box are assigned position information of the unary bitwise complement expression. The exclusive or expression left operand box is assigned position information of the unary bitwise complement expression operand E , as shown in Figure 5.29.

$$\boxed{\sim[E]} \rightarrow \$i0 = \boxed{[E] \wedge -1}$$

Figure 5.29: Unary Bitwise Complement Expression Position Information Generation

A unary logical complement expression of the form `!E` is generated as an `if` statement and some assignment statements. The `if` statement and the `if` statement condition box are assigned position information of the unary logical complement expression operand `E`. The assignment statements are assigned position information from the unary logical complement expression as shown in Figure 5.30

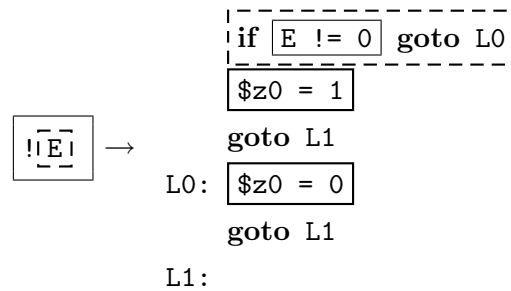



Figure 5.30: Unary Logical Complement Expression Position Information Generation

5.3 Visual Results

Adding **Tags** with the position information for each Jimple construct, as described above, makes line number and column position information available for **SootClasses**, **SootMethods**, **SootFields**, statements and **ValueBoxes**, which hold expressions. When analysis results are generated the results can be easily stored in these **Hosts** also, using the three new visualization **Tags**. In this section we describe the three visualization **Tags**, how they are used, and how they are presented in a visual way.

5.3.1 StringTags and Tool-tips

StringTags encode textual information and results, and are recommended, even if other **Tag** types are used, for ensuring the clarity information being visualized. A **StringTag** is displayed in the plugin as a pop-up tool-tip when the mouse hovers over the text in the line corresponding to the textual information. Each statement in the code that has a tool-tip available has a small Soot icon  in the margin to indicate that there is information available at that line. **StringTags**, like all **Tags** available for visualizations, are easy to create. For example, `stmt.addTag(new StringTag(analysisResult))` puts the analysis result string in a **StringTag** and attaches the **Tag** to the statement. When the results are displayed in the editor with the source file or IR file and when the mouse is hovering above the text in the line containing the statement, the tool-tip corresponding to the analysis results is displayed as shown in Figure 5.31.

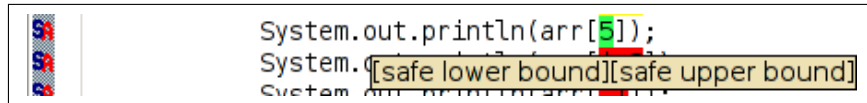


Figure 5.31: Tool-tip with Analysis Information

5.3.2 ColorTags and Color Highlighting

ColorTags can be used to encode a set of distinct results with different colours, or to encode a set of related statements in one colour, or to highlight a specific result. **ColorTags** are displayed in the plugin as colour highlighting. By default, the background of the code is highlighted with the appropriate colour but it is possible to highlight the foreground text instead. In the event of overlapping colour regions the colours are set from longest region to shortest region so that shorter regions of highlighting are not hidden. For example, to add a **ColorTag** to a **ValueBox** we could use `vb.addTag(new ColorTag(ColorTag.BLUE))`, which will add the **Tag** to the **ValueBox** which will cause the background of the expression contained in the

`ValueBox` to be blue when the source code or IR code is displayed in the editor as shown in Figure 5.32. Figure 5.32 shows all odd variables in blue and all even variables in yellow and is a part of a result for a parity analysis. `ColorTags` require

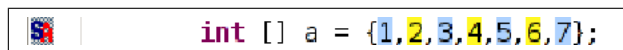



Figure 5.32: `ColorTags` Representing Analysis Information

a high degree of precision in order to highlight regions of the code more specifically than on a per statement basis. This precise information about column positions is available for the source only when using source code as the input when invoking Soot. When using bytecode as the input to Soot, the required information is not available. `ColorTags` are however always available in the Jimple IR, as the needed level of position information can always be generated. This is not really a problem though, because if one actually wants to see `ColorTag` visualizations in the source then presumably one has the source and can use it as input to Soot.

5.3.3 LinkTags and Links

`LinkTags` encode information associating one part of the code with another part. Each statement, in the editor, that has an associated `LinkTag` has a small Soot icon  in the margin beside it on the line in the editor. Clicking on this icon pops up a list of links available. Selecting one of the links causes the cursor to move to the line containing the statement of the link target. This target may be in a different file, in which case the file will be opened or activated in an editor as needed. Creating a `LinkTag` can be done, for example, with the following code `stmt.addTag(new LinkTag(analysisString, linkToHost, className))`, which will add a link from the statement with the text of the analysis string to the target `Host linkToHost` in the file specified by `className`. For example, in Figure 5.33 we see two links corresponding to the statement `int z = x + 2`. These links represent possible definitions of the variable `x` which is used in the statement.

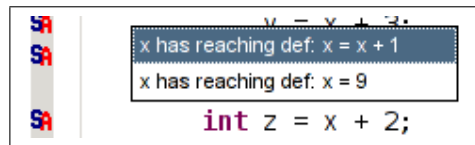


Figure 5.33: LinkTag with Analysis Information

5.3.4 KeyTags and Legends

It is also possible to generate a legend which explains the encoding of the analysis result information. In order to provide legends, a **KeyTag** is available. It may be added to the **SootClass** and contains the colour and the corresponding analysis result text for each distinct result. This is especially useful for **ColorTags** which represent a set of discrete results. For example, the following code

```
sootClass.addTag(new KeyTag(ColorTag.Green,
    "Safe Lower and Safe Upper", "ArrayBounds"))
```

will add legend information to the **sootClass** indicating a colour **ColorTag.GREEN** and associated text **Safe Lower and Safe Upper** for the array bounds check analysis. If a legend is specified it is automatically displayed in small view beside the editor as shown in Figure 5.34, which shows the four different colours used to visualize the four different possible outcomes of the array bounds check analysis.

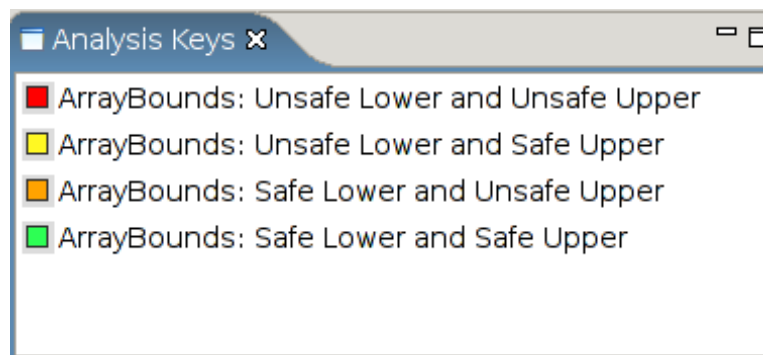


Figure 5.34: Analysis Visualization Results Legend View

5.4 Collecting Tags for Output

Position information and results of the analyses are stored in **Tags** on **Hosts** in the Jimple IR as shown in boxes 1, 2 and 3 of Figure 5.35. Box 1 shows the source position **Tags** when using bytecode as input and box 3 shows the source position **Tags** when using source code as input. If and when the textual Jimple IR is printed as output, line number and column position information is generated about this textual representation and is also attached as **Tags** to the corresponding Jimple **Hosts** as shown in boxes 4 and 5 of Figure 5.35. For the purpose of displaying the resulting information in a visual way, all **Tags** are collected from each Jimple **Host** and collated and output in an XML format as shown in box 6 of Figure 5.35.

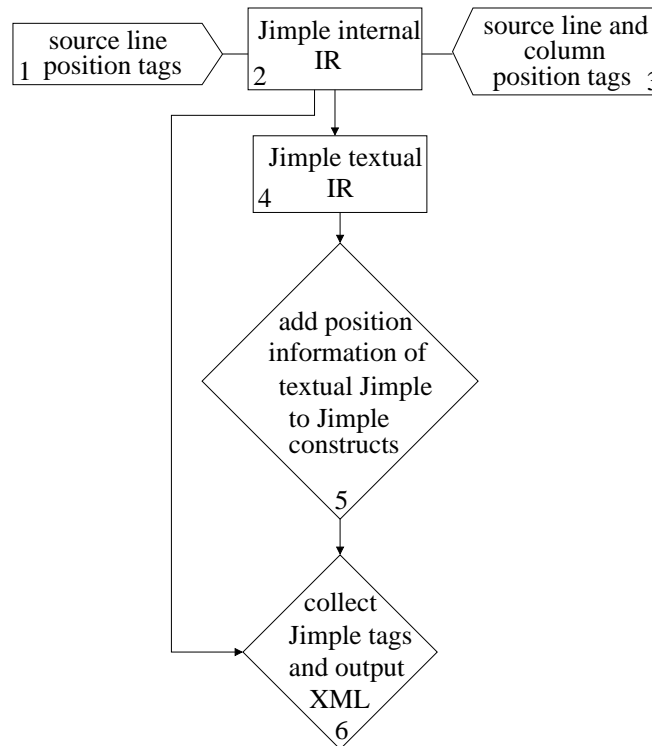


Figure 5.35: Tag Collection Overview

5.5 Managing the Display in Eclipse

When a Jimple file or a source file is opened in Eclipse, if there is an associated XML file containing analysis results and position information, then attribute information is displayed in the plugin editor in a visual way. The type of visualization depends on how the information is encoded in the **Tag**. The plugin handles the management of displaying the correct information for the file in focus and for updating the visualizations when new information is generated. The visual display is updated when Soot is run and if a new attribute file becomes available.

Having this attribute file handling mechanism to update the visual display allows the results to be generated outside of the Eclipse plugin and then have the results viewed within the Eclipse plugin. This is especially useful for long-running memory intensive analyses, or for visualizing the results of many analyses that are generated using complicated scripts.

Several analyses and their results may be generated at the same time and in some cases the visualizations may conflict. To handle this situation, another view, shown in Figure 5.36, is automatically displayed, which lists all of the generated analyses

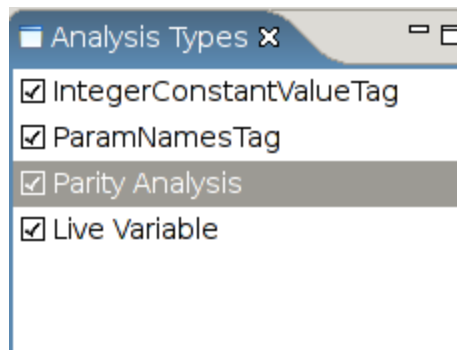


Figure 5.36: Analysis Visualization Results Types View

with visual information for the file in focus and permits selection of which ones to display, by manipulating the check boxes on the left. It is possible to switch between analysis results and the plugin manages the display and updating.

5.6 Summary

This chapter described the collection, generation and use of position information required to display analysis results in the original source code and in the Jimple IR. First we gave the motivation of why such tools for viewing analysis results are needed. We discussed where the position information, needed to relate results produced on the IR back to the original source code, comes from and how it is assigned to Jimple constructs. We also gave an overview of the visualization **Tags** available within Soot and how they are displayed in Eclipse. In the next chapter we will give several concrete examples which use this framework.

Chapter 6

Applications of Tools

6.1 Motivation

To demonstrate the usefulness of our visualization framework we examined two applications: visualizations for compiler research and visualizations for using compiler analyses for program understanding. Applications in this chapter also provide concrete examples of the framework.

6.2 Applications for Compiler Research

In this section we describe several examples which are available within the Soot framework, for which we added visual information. These analyses are useful for teaching, compiler research, advanced users and programmers in general. The general process for using the framework for viewing analysis results is shown in Figure 6.2. To use the framework, one may write a new Soot analysis as shown in box 1a, or use an existing Soot analysis as shown in box 1b. These analyses may be intra- or inter- procedural analyses. Soot contains standard extendable classes for easily writing intra-procedural analyses.¹ Once the analysis is available within Soot, one must write a tagger class as

¹soot/toolkits/scalar/ForwardFlowAnalysis.java, soot/toolkits/scalar/BackwardFlowAnalysis.java, or soot/toolkits/scalar/ForwardBranchedFlowAnalysis.java

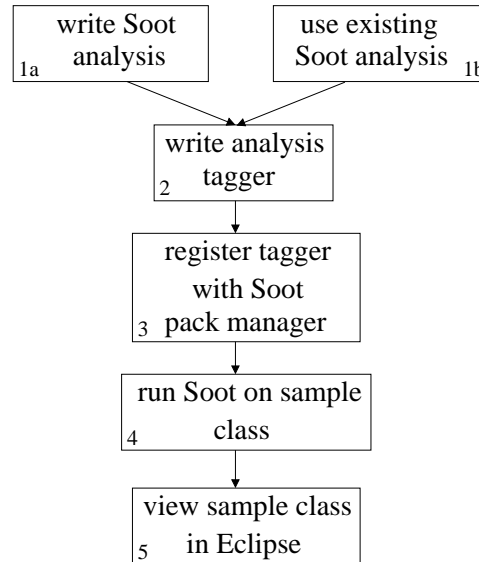


Figure 6.1: Process for Using Framework for Viewing Analysis Results

shown in box 2. This tagger class must extend the `BodyTransformer` class² for intra-procedural analyses or the `SceneTransformer` class³ for inter-procedural analyses. Extending the `BodyTransformer` will cause Soot to apply the tagger code to every body processed, typically for adding visualization `Tags` to all statements with some property, and extending the `SceneTransformer` will give access to the whole program being processed at one time. As shown in box 3, one must then register the tagger class with the Soot `PackManager`.⁴ The pack manager decides which transformers will be applied and in what order. After all is setup, one can run Soot on a sample class to test the implementation as shown in box 4. This will produce the all of the visualization information needed for viewing the results within the plugin. Finally, as shown in box 5, the results will automatically be displayed when the sample class is displayed within the plugin.

²soot/BodyTransformer.java

³soot/SceneTransformer.java

⁴soot/PackManager.java

6.2.1 Analysis Results for Teaching

Parity Analysis

The parity analysis is a simple analysis which determines which variables are always odd or always even. It is useful for teaching basic data flow analysis and is an example which may be used in an introductory optimizing compilers course. In this section we give a concrete example of how to use the visualization framework from start to finish.

First, we need a data flow analysis which computes the parity information for each local variable in the program. The complete intra-procedural parity analysis, which computes the parity information for each local variable and stores the results in a map, is available in Soot⁵, so we will use it in order to demonstrate the application of **Tags**. For the purpose of visualizing the analysis results we want to make all odd variables be coloured blue, all even variables be coloured yellow and all others be coloured red.

The tagger class required is shown in Figure 6.2. This figure shows the first part of the class needed. It shows that the **ParityTagger** class extends the **BodyTransformer** class as the tagger will add **Tags** to locals in one method at a time. The critical method which must be overridden is the **internalTransform** method which will be called by the Soot **PackManager** when it is processing a method body. The method body being processed will be passed in as a parameter. The first step in the tagger code is to invoke the **ParityAnalysis**, which will cause Soot to compute the required results. Then we iterate over all the statements in the body. For each statement we access the parity information for all the local definitions and add **Tags** to the definition boxes, and access the parity information for all the local uses and add **Tags** to the use boxes. The **addTags** method is shown in Figure 6.3. In this code we iterate over all the boxes requiring **Tags**. For each local we access the textual representation on the result and store it in a **StringTag** which we attach to the box. We then determine the parity of the variable and make a new **ColorTag** with the corresponding colour: yellow for even, blue for odd, red for neither (or top) and green for unspecified (or bottom) and

⁵soot/jimple/toolkits/annotation/parity/ParityAnalysis.java

```

public class ParityTagger extends BodyTransformer {

    protected void internalTransform( Body body) {

        // compute analysis
        ParityAnalysis pa = new ParityAnalysis(new BriefUnitGraph( body ));

        // iterate over all stmts
        for( Iterator it = body.getUnits(); it.hasNext(); ) {
            Stmt stmt = (Stmt) it.next();
            // tag all definitions with flow information after statement
            addTags( stmt.getDefBoxes(), (Map) pa.getFlowAfter( stmt ) );
            // tag all uses with flow information before statement
            addTags( stmt.getUseBoxes(), (Map) pa.getFlowBefore( stmt ) );
        }
    }
    ...
}

```

Figure 6.2: Code to Visualize Parity Analysis Results

add the `ColorTag` to the box. The complete code for the parity tagger is available in Soot.⁶

The next step is to register this tagger with the Soot `PackManager` so that it may be run by Soot. One way to do this is to create a new `ParityTagger` and simply add it to the Jimple transformation pack (`jtp`) as shown in the code snippet in Figure 6.4.

Finally, when we run Soot on a sample program, the parity results will be computed, the local variables will be tagged with `StringTags` and `ColorTags` and this visualization information will be available with the plugin. When the sample program is viewed within Eclipse, each local in the editor has its background coloured in blue, yellow or red as shown in Figure 6.5.

⁶soot/jimple/toolkits/annotation/parity/ParityTagger.java

```
private void addTags( Collection boxes, Map parityMap ) {

    // iterate over all local boxes
    for( Iterator it = boxes.iterator(); it.hasNext(); ) {
        ValueBox box = (ValueBox) it.next();
        Value local = box.getValue();

        // get the flow information
        String parity = (String) parityMap.get( local );
        if( parity == null ) return; // no parity information for this value
                                   // (only computed for variables of int type)

        // add a String Tag
        box.addTag( new StringTag( local.toString()+" is "+parity+"." ) );

        // add a Color Tag
        if( parity.equals( ParityAnalysis.EVEN ) )
            box.addTag( new ColorTag( ColorTag.YELLOW ) );
        else if( parity.equals( ParityAnalysis.ODD ) )
            box.addTag( new ColorTag( ColorTag.BLUE ) );
        else if( parity.equals( ParityAnalysis.TOP ) )
            box.addTag( new ColorTag( ColorTag.RED ) );
        else if( parity.equals( ParityAnalysis.BOTTOM ) )
            box.addTag( new ColorTag( ColorTag.GREEN ) );
        else throw new RuntimeException( "Unknown parity value "+parity+"." );
    }
}
```

Figure 6.3: Code to Add Tags to Visualize Parity Analysis Results

```
PackManager.v().getPack( "jtp" ).add( new Transform( "jtp.parity",
    new ParityTagger() ) );
```

Figure 6.4: Code to Register Tagger with PackManager

```

public static void main(String[] args) {
    int x = 0;
    int y = 1;
    int z = 2;
    int [] w = {x, y, z};
    int[] s = {z, y, x, x, y, z};

    for (int i = 0; i < 10; i++){
        x = x + 2;
    }

    do {
        y = y + 1;
        s[y] = 9;
    }while (y < 10);
}

```

Figure 6.5: Parity Analysis with Visualization Results

Using our visual tools here allows us to easily see where the analysis computes an incorrect result as parity information is very easy to compute in one’s head. This analysis provides an example for teaching basic flow analysis and describes a concrete way of using the visualization framework for displaying analysis results.

6.2.2 Analysis Results Specifically for Compiler Research

In compiler research there are several well defined analysis which are often used as the basis for other analyses. These kinds of analyses are often readily available in optimizing compiler frameworks. In this section, we describe how visual information can be added to these existing analyses within Soot, so they may be displayed to allow us to verify the correctness of the analyses and to understand them more easily, and to then use them in more complex analyses.

Call Graph

Soot generates precise call graph information [Lho02] which is used in many different analyses, for example in the unreachable fields and methods and the tightest qualifiers analyses which are described later in this chapter in sections 6.3.1 and 6.3.2. To be

able to easily verify the correctness of the call graph and to be able to use this information while designing other analyses we make the results of the call graph analysis available in **LinkTags** so that it may be viewed in the source code and the IR code. These **LinkTags** allow the user to easily navigate to any methods called by a statement in the source and for any method the user can also easily navigate to any of the callers of the method. The code required to add **LinkTags** representing the call graph is quite simple for our framework and is shown in Figure 6.6. Again, like in the parity tagger, we extend the **BodyTransformer** class and override the **internalTransform** method. The first step is to get the call graph which is already available in Soot, when running Soot in whole-program mode, which enables inter-procedural analyses. We do this by accessing it from the **Scene**. The **Scene** is a part of Soot which stores generated information on a global basis about all the classes being analyzed or used by Soot. We then iterate through the statements and for each iterate through all the edges going out of the statement. For each of these edges we find the target method and attach a **LinkTag** to the statement, connecting the statement to the target method. We also want to tag all of the incoming calling edges for the method of the body being processed. We use the call graph to find all of the edges into the method, then find the source of the caller and add a **LinkTag** to the method for each edge going into it, connecting the method and the source which calls it. The complete code for adding visualization **Tags** for the call graph is available within Soot.⁷

In Figure 6.7 we give an example showing the call graph tagger being applied to a very small method, with **LinkTags** being displayed showing the two virtual methods which could be called at the statement `o.foo()`. Clicking on one of the links causes the cursor to navigate to the corresponding code. In this case, selecting the first link brings **Class1** into focus and positions the cursor at the `foo` method as shown in Figure 6.8.

⁷soot/jimple/toolkits/annotation/callgraph/CallGraphTagger.java

```

protected void internalTransform( Body body) {
    // get call graph
    CallGraph cg = Scene.v().getCallGraph();
    // iterate over all stmts
    for( Iterator it = body.getUnits(); it.hasNext(); ) {
        Stmt stmt = (Stmt) it.next();
        // iterate over all call graph edges out of stmt
        for( Iterator edges = cg.edgesOutOf(stmt); edges.hasNext(); ) {
            Edge edge = (Edge) edges.next();
            // get call graph edge target method
            SootMethod target = edge.tgt();
            // add LinkTag - linking stmt to the target method
            stmt.addTag(new LinkTag("CallGraph: TargetMethod: "+target.getName(),
                                   target,
                                   target.getDeclaringClass()));
        }
    }
    // get this method
    SootMethod thisMethod = body.getMethod();
    // iterate over all edges calling the method
    for( Iterator sources = cg.edgesInto(thisMethod); sources.hasNext(); ) {
        Edge callEdge = (Edge) sources.next();
        // get the source method of caller
        SootMethod caller = callEdge.src();
        // determine actual source
        Host src = caller;
        if (callEdge.srcUnit() != null) {
            src = callEdge.srcUnit();
        }
        // add LinkTag - linking calling source to method
        thisMethod.addTag(new LinkTag("CallGraph: Source: "+src.toString(),
                                      src,
                                      caller.getDeclaringClass().getName()));
    }
}

```

Figure 6.6: Code to Add LinkTags to Visualize the Call Graph



Figure 6.7: Call Graph Analysis with Visualization Results

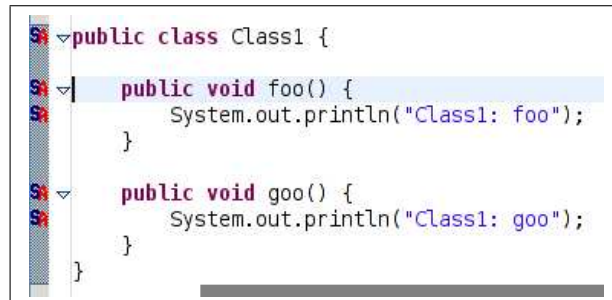


Figure 6.8: Call Graph Analysis LinkTag with Visualization Results

Live Variables Analysis

Another common compiler analysis is a live variable analysis. Soot generates a live variable analysis which determines which variables are live (may be used again) at each statement in the program.⁸ A live variable analysis, like the call graph, may be used in many more complicated analyses such as a constructor folder analysis, a constant propagation analysis, or a copy propagation analysis. Again we can easily add a tagger class which adds **ColorTags** to each live variable and **StringTags** to each statement with live variables to be able to indicate visually which variables are live. We show this tagger code in Figure 6.9 In this code first we compute the liveness analysis. Then for every local in every statement, we add a **StringTag** to the statement if the local is live and add a **ColorTag** to every live local to assign it the colour green. Once we run this tagger within Soot on a sample program and display it within the Eclipse plugin editor, we can see the background colour for all the live locals is green, as shown in Figure 6.10. The complete code for the tagger is also

⁸soot/toolkits/scalar/SimpleLiveLocals.java

```

public class LiveVariablesTagger extends BodyTransformer {

    protected void internalTransform( Body body) {

        // compute analysis
        LiveLocals sll = new SimpleLiveLocals(new ExceptionalUnitGraph( body ));

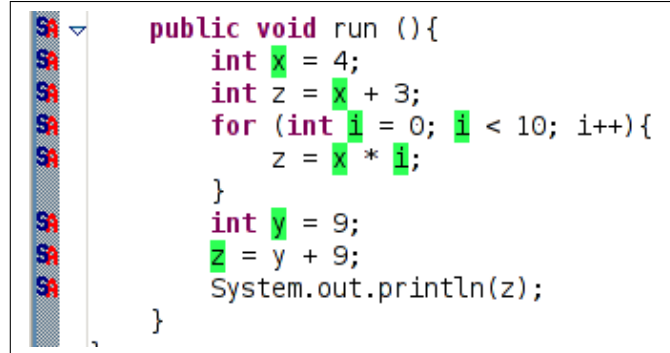
        // iterate over all stmts
        for( Iterator it = body.getUnits(); it.hasNext(); ) {
            Stmt stmt = (Stmt) it.next();
            // iterate over all live locals in stmt
            for( Iterator locals = sll.getLiveLocalsAfter(stmt).iterator();
                locals.hasNext(); ) {
                Value value = (Value) localsIt.next();
                // add StringTag to stmt for each live local
                stmt.addTag(new StringTag("Live Variable: "+value));

                // iterate over the use and definition boxes in stmt
                for(Iterator boxes = stmt.getUseAndDefBoxes().iterator();
                    boxes.hasNext(); ) {
                    ValueBox box = (ValueBox) boxes.next();
                    // add ColorTag to box if its value is live
                    if (box.getValue().equals(value)) {
                        box.addTag(new ColorTag(ColorTag.Green));
                    }
                }
            }
        }
    }
}

```

Figure 6.9: Code to Add StringTags and ColorTags to Visualize the Live Variables

available in Soot.⁹



```
public void run () {
    int x = 4;
    int z = x + 3;
    for (int i = 0; i < 10; i++) {
        z = x * i;
    }
    int y = 9;
    z = y + 9;
    System.out.println(z);
}
```

Figure 6.10: Liveness Analysis with Visualization Results

Reaching Definitions Analysis

Soot also generates reaching definition information which computes for all the uses of variables in a statement, which statement defines them.¹⁰ A reaching definition analysis is another example of a common analysis which is used in many other analyses such as a static method binding analysis, or a local splitting analysis. To show this information in a visual way, we again represent it using **LinkTags**, in order to connect with each local in a statement, the statement or statements where the local is defined. As shown in Figure 6.11, for every statement we determine all of the locals used in the statement. For each of these locals we iterate over all of the definitions and add a **LinkTags** to the statement, connecting the local to each of its potential definitions. This is again an intra-procedural tagger which extends the **BodyTransformer**. These **LinkTags** provide a list of all the statements which may be the definition for each variable used in the statement. In the visual editor, when a sample program is analyzed by Soot and displayed, these links can be used to navigate to any of the potential definitions for the locals in each statement. Again the full tagging code is available in Soot.¹¹

⁹soot/jimple/toolkits/annotations/liveness/LiveVarsTagger.java

¹⁰soot/toolkits/scalar/SmartLocalDefs.java

¹¹soot/jimple/toolkits/annotation/defs/ReachingDefsTagger.java

```

protected void internalTransform( Body body ) {
    // get control flow graph for method body
    UnitGraph graph = new ExceptionalUnitGraph( body );
    // compute analysis
    LocalDefs sld = new SmartLocalDefs( graph, new SimpleLiveLocals( graph ) );
    // iterate over all statements
    for( Iterator it = body.getUnits(); it.hasNext(); ) {
        Stmt stmt = (Stmt) it.next();
        // iterate over all use boxes
        for( Iterator uses = stmt.getUsesBoxes().iterator(); uses.hasNext(); ) {
            ValueBox box = (ValueBox)uses.next();
            Value value = box.getValue();
            // only consider locals
            if ( value instanceof Local ) {
                Local local = (Local) value;
                // iterate over definitions of each use
                for( Iterator defs = sld.getDefsOfAt( local, stmt ).iterator();
                    defs.hasNext(); ) {
                    Stmt def = (Stmt)defs.next();
                    // add Link Tag
                    def.addTag( new LinkTag(
                        local + " has reaching def: " + def.toString(),
                        def,
                        body.getMethod().getDeclaringClass().getName() ));
                }
            }
        }
    }
}

```

Figure 6.11: Code to Visualize Reaching Definition Analysis Results

6.2.3 Analysis Results for Research and Advanced Users

There are several analyses available in Soot which compute advanced results for code optimization. In this section we describe how we use the visualization framework to view these results.

Cast Check Analysis

The cast check analysis in Soot verifies for statements containing casts whether or not the cast check is required. The code for tagging the results of this analysis is slightly more complicated than the previous ones and is shown in Figure 6.12. In this tagging code we again iterate through each statement, but only consider assignment statements which have a cast expression of a reference type on the right hand side. In this example there is a pre-computed set of locals and types used. These pre-computed sets are common for more complicated analysis, to store information which can be calculated in one pass of the code. We then add a **StringTag** to each statement containing a cast expression indicating if the cast check may be eliminated. This information is displayed in a visual way as shown in Figure 6.13 and again this complete example is available within Soot.¹²

Arraybounds Check Analysis

The array bounds check analysis in Soot determines statically if array indexes may be out of bounds or if they are definitely safe within bounds. This is a comprehensive analysis [QHV02] consisting of three related analyses: a *variable constraint analysis*, an *array field analysis*, and a *rectangular array analysis*. Although the analysis is quite complex, the code to add tags so that the results of the analysis may be verified is quite straight-forward as shown in Figure 6.14. In this tagging code we again iterate through all the statements, considering only those containing array reference expressions. We get the results of the array bounds safeness and add **ColorTags** as needed. There are four distinct results that can arise and we add **ColorTags** to

¹²soot/jimple/toolkits/pointer/CastCheckEliminator.java

```

public class CastCheckTagger extends BodyTransformer {

    protected void internalTransform( Body body) {

        // compute cast check eliminator analysis
        CastCheckEliminator cce = new CastCheckEliminator(
            new BriefUnitGraph( body));

        // iterate over all stmts
        for( Iterator it = body.getUnits(); it.hasNext(); ) {
            Stmt stmt = (Stmt) it.next();
            // only consider assign stmts with casts of reference types in rhs
            if( stmt instanceof AssignStmt) {
                AssignStmt as = (AssignStmt)stmt;
                Value rhs = as.getRightOp();
                if( as instanceof CastExpr) {
                    CastExpr cast = (CastExpr) rhs;
                    Type type = cast.getCastType();
                    if( type instanceof RefType) {
                        if (cast.getOp() instanceof Local) {
                            Local local = (Local)cast.getOp();

                            // get local type set for stmt
                            LocalTypeSet set =
                                (LocalTypeSet) unitToBeforeFlow.get(stmt);
                            // add StringTag to stmt indicating if cast check
                            // is needed
                            stmt.addTag(new StringTag(
                                cce.getResult(local, type, set)));
                        }
                    }
                }
            }
        }
    }
}

```

Figure 6.12: Code to Visualize Cast Check Elimination Analysis Results

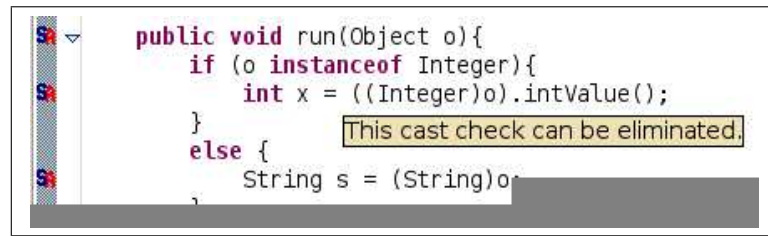


Figure 6.13: Cast Check Analysis with Visualization Results

indicate the safeness of the array index with respect to the bounds. The colour green is used to indicate the index is definitely in bounds, yellow is used to indicate the index is potentially out of bounds at the lower end, orange is used to indicate that the index is potentially out of bounds at the upper end, and red is used to indicate that the index is potentially out of bounds at both the lower and upper ends as shown in Figure 6.15.

Null Pointer Analysis

The null pointer analysis determines if expressions are definitely null, definitely not null or unknown. This is another fairly complex analysis, as it must consider code along different paths of branches in the code. Even so, the code to add visualization tags is again quite uncomplicated. First we add visualizations **Tags** to the locals used in the statement and then add visualization tags for the locals defined in the statement. The two steps are needed to be able consider the branching in the code. The code to handle the boxes is shown in Figure 6.16. The code that actually adds the **Tags** is shown in Figure 6.17, where we obtain the results and add **ColorTags** and **StringTags** to enable visualizing the results of this analysis by making null expressions red, non null expressions green and unknown expressions blue. The results of this analysis are shown for a sample program in Figure 6.18. In the figure we see that formal **args** is blue indicating the nullness is unknown which makes sense considering it comes from outside the method, the local **y** is red as it is assigned to the value **null** and the local **ne** is green, as it is definitely not null, being created in

```

public class ArrayCheckTagger extends BodyTransformer {
    protected void internalTransform( Body body) {
        // compute array bounds check eliminator analysis
        ArrayBoundsCheckerAnalysis analysis =
            new ArrayBoundsCheckerAnalysis( body);
        // iterate over all stmts
        for( Iterator it = b.getUnits(); it.hasNext(); ) {
            Stmt stmt = (Stmt) it.next();
            // only consider stmts containing array references
            if( stmt.containsArrayRef() ) {
                ArrayRef aref = stmt.getArrayRef();

                // get analysis results for stmt
                int result = interpret(analysis.getFlowBefore(stmt), aref,
                    stmt, new IntContainer(0));
                // add ColorTags indicating array bounds to array references
                switch(result){
                    case ArrayBoundsCheck.UNSAFE_LOWER_UPPER:{
                        aref.addTag(new ColorTag(ColorTag.RED));
                        break;
                    case ArrayBoundsCheck.UNSAFE_LOWER:{
                        aref.addTag(new ColorTag(ColorTag.YELLOW));
                        break;
                    case ArrayBoundsCheck.UNSAFE_UPPER:{
                        aref.addTag(new ColorTag(ColorTag.ORANGE));
                        break;
                    case ArrayBoundsCheck.SAFE:{
                        aref.addTag(new ColorTag(ColorTag.GREEN));
                        break;
                    }
                }
            }
        }
    }
}

```

Figure 6.14: Code to Visualize Array Bounds Check Analysis Results

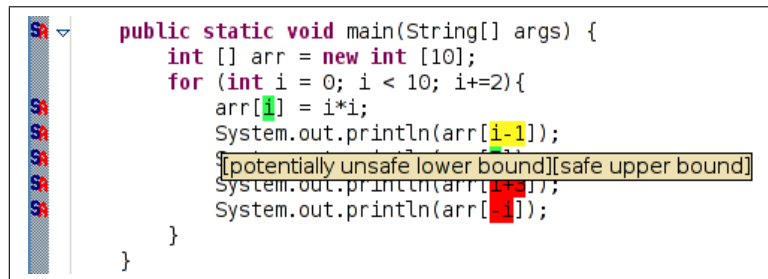


Figure 6.15: Array Bounds Checks Analysis with Visualization Results

the same statement. However, if we saw that the local `y` was blue, we would know our analysis was imprecise and missing being able to determine that `y` is null by assignment, just by looking at the results. In fact, when we first started using the visualization framework we quickly found areas where the analysis could be improved, simply by glancing at the results in the editor, within the plugin.

6.3 Applications for Program Understanding

Now that we have shown several uses of the visualization framework for analysis results in compiler research and educational areas, we must determine if this visualization framework may be useful for generating visual information, for program understanding, that may be useful to regular programmers.

We created three new analyses: to find unreachable fields and methods, to determine the tightest qualifiers possible for fields and methods, and to find loop invariant expressions. These three analyses are ones that could be used by programmers in general. For each one, we ran it on a set of Java benchmarks, collected in a graduate level compiler course, to determine if the analyses are useful at finding suggested areas of code improvement in real-live code.

```

public class NullCheckTagger extends BodyTransformer {

    protected void internalTransform( Body body) {
        // compute null check analysis
        BranchedRefVarsAnalysis analysis = new BranchedRefVarsAnalysis(
            new ExceptionalUnitGraph(body));

        // iterate over all stmts
        for( Iterator it = body.getUnits(); it.hasNext(); ) {
            Stmt stmt = (Stmt) it.next();

            // get analysis results for uses
            FlowSet beforeSet = (FlowSet)analysis.getFlowBefore(stmt);
            // iterate over all use boxes in stmt
            for(Iterator uses = stmt.getUseBoxes().iterator(); uses.next(); ) {
                ValueBox useBox = (ValueBox) uses.next();
                // add visualization tags
                addVisualizationTags(useBox, beforeSet, stmt, analysis);
            }

            // get analysis results for definitions
            FlowSet afterSet = (FlowSet)analysis.getFallFlowAfter(stmt);
            // iterate over all definition boxes in stmt
            for(Iterator defs = stmt.getDefBoxes().iterator(); defs.next(); ) {
                ValueBox defBox = (ValueBox) defs.next();
                // add visualization tags
                addVisualizationTags(defBox, afterSet, stmt, analysis);
            }
        }
    }
    ...
}

```

Figure 6.16: Code to Visualize Null Check Analysis Results


```
private void addVisualizationTags(ValueBox box, FlowSet set, Stmt stmt,
    BranchedRefVarsAnalysis analysis) {

    // only consider reference types
    if (box.getValue() instanceof RefLikeType) {

        // get result for box
        int result = analysis.anyRefInfo(box.getValue(), set);

        // add StringTags to stmt and ColorTags to box
        switch (result) {
            case analysis.NULL :{
                stmt.addTag(new StringTag(box.getValue()+" : Null"));
                box.addTag(new ColorTag(ColorTag.RED));
                break;
            }
            case analysis.NON_NULL :{
                stmt.addTag(new StringTag(box.getValue()+" : Not Null"));
                box.addTag(new ColorTag(ColorTag.GREEN));
                break;
            }
            case analysis.UNKNOWN :{
                stmt.addTag(new StringTag(box.getValue()+" : Unknown"));
                box.addTag(new ColorTag(ColorTag.BLUE));
                break;
            }
        }
    }
}
```

Figure 6.17: Code to Add StringTags and ColorTags for Null Check Analysis Results



Figure 6.18: Null Checks Analysis with Visualization Results

Benchmark Programs

ConFour is an implementation of the game connect four. Dmd is a package to compute the strongly connected components of a directed graph. Hull computes the convex hull of a set of points in a plane. Imagematch is a program that performs Template Matching using sum of square differences. JWBench is a discrete event simulation program which simulates a cell in a personal communications system (PCS). Knight-sTour finds a knights tour on an $n \times n$ grid. Telecom is a telecom system simulation program. Vignere is a program to break the Vignere cipher.

6.3.1 Unreachable Fields and Methods Analyses

The unreachable fields and unreachable methods analyses use the precise call graph analysis generated by Soot to determine which fields and methods are unreachable. These unreachable fields and methods are coloured red in the source code to indicate the results to the programmer. Unreachable fields and methods may be removed from the code for future maintainability and these analyses provide a simple way of finding such opportunities. In Tables 6.1 and 6.2 we show the results of running this analysis on several small applications. For each application we show the total

6.3. Applications for Program Understanding

benchmark	number methods	unreachable methods
ConFour	22	2 (9%)
Dmd	22	0
Hull	23	1 (4%)
ImageMatch	6	0
JWBench	36	0
KnightsTour	21	0
Telecom	43	9 (21%)
Vignere	24	2 (8%)

Table 6.1: Unreachable Methods Analysis Results

benchmark	number of fields	unreachable fields	non static final unreachable fields
ConFour	34	17 (50%)	0
Dmd	13	0	0
Hull	8	3 (4%)	0
ImageMatch	2	0	0
JWBench	35	0	0
KnightsTour	13	0	0
Telecom	24	11 (42%)	4 (17%)
Vignere	11	1 (9%)	1 (9%)

Table 6.2: Unreachable Fields Analysis Results

number of methods and fields and the number of unreachable methods and fields. Up to 20% of methods are unreachable in the one application - Telecom. We find that there are many unreachable fields found, however, upon closer observation of the visual results we can see that the majority of the fields are `final static` fields whose values have been propagated throughout the code at compile time. These unreachable fields are still useful for code readability. An example of how the visual results for an unreachable method are displayed is shown in Figure 6.19.

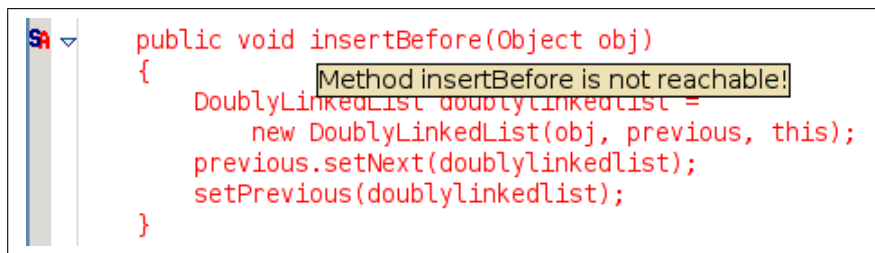


Figure 6.19: Unreachable Methods Analysis with Visualization Results

This is an example which shows some of the flexibility of the framework, showing that **Tags** can be added to **Hosts** other than statements, in this case to methods, and that the foreground of the text, rather than the background may be coloured. This is done by setting the second parameter as a `true` flag when creating the `ColorTag`. The code used for tagging unreachable methods is shown in Figure 6.20. Here we extend the `SceneTransformer` because we are interested in the whole application. We look at all the application classes, in this case we do not consider library classes, and find all of the declared methods. We then determine if the method is unreachable and add a `StringTag` and a `ColorTag`. We describe this example to demonstrate how simple it is to use the visualization framework even on whole program analyses and how easy it is to access the precise information generated by Soot.

6.3.2 Tightest Qualifiers Analysis

The tightest qualifiers analysis computes for fields and methods whether or not they could have, for example, a `private` qualifier where they currently have a `public`

```
public class UnreachableMethodTagger extends SceneTransformer {
    protected void internalTransform() {

        // iterate through all application classes
        for (Iterator classes = Scene.v().getApplicationClass().iterator();
             classes.hasNext();) {
            SootClass appClass = (SootClass) classes.next();

            // iterate through all methods
            for (Iterator methods = appClass.getMethods().iterator();
                 methods.hasNext(); ){
                SootMethod method = (SootMethod)methods.next();

                // determine if method is unreachable
                if (!Scene.v().getReachableMethods().contains(method)){
                    // add StringTags and ColorTags
                    method.addTag(new StringTag("Method is unreachable"));
                    method.addTag(new ColorTag(ColorTag.RED, true));
                }
            }
        }
    }
}
```

Figure 6.20: Code for Tagging Unreachable Methods with Visualization Results

qualifier. This analysis displays the results by colouring in red all methods and fields with qualifiers which could be tightened and indicates with a **StringTag** to what level as shown in Figure 6.21. This analysis uses the call graph analysis to compute

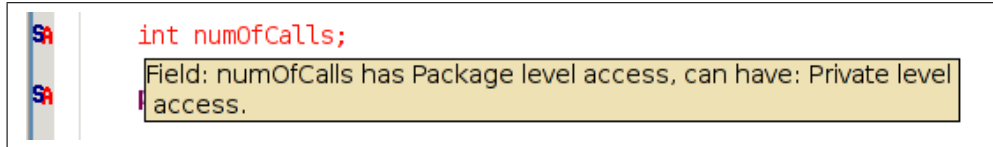


Figure 6.21: Tightest Qualifiers Analysis with Visualization Results

the needed results. Tightening the qualifiers, particularly to private may increase the runtime speed in that a special invoke can be used for method invocation which is faster than a virtual invoke necessary for a public method. Additionally, tighter qualifiers may improve the security of the code. Tables 6.3 and 6.4 show the results

benchmark	methods	pb ¹³ -> pkg ¹⁴	pb -> prv ¹⁵	pkg -> prv	pvt ¹⁶ -> pkg	pvt -> prv
ConFour	22	2	2	4	0	0
Dmd	22	6	0	0	0	0
Hull	23	7	8	0	0	0
ImageMatch	6	0	5	0	0	0
JWBench	36	0	0	0	0	0
KnightsTour	21	12	7	0	0	0
Telecom	43	11	0	0	3	3
Vignere	24	0	0	0	0	0

Table 6.3: Tightest Qualifiers on Methods Analysis Results

of the analyses searching for tightest qualifiers for methods and fields. Each table lists the total number of methods or fields and the number of methods or fields which could be declared with a tighter qualifier.

¹³public

¹⁴package

¹⁵private

¹⁶protected

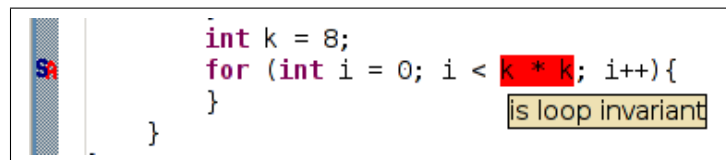
6.3. Applications for Program Understanding

benchmark	fields	pb -> pkg	pb -> prv	pkg -> prv	pvt -> pkg	pvt -> prv
ConFour	34	0	0	8	1	1
Dmd	13	0	0	2	0	0
Hull	8	0	0	0	0	0
ImageMatch	2	0	0	0	0	0
JWBench	35	0	0	0	0	0
KnightsTour	13	0	1	8	0	0
Telecom	24	2	1	2	0	0
Vignere	11	0	0	0	0	0

Table 6.4: Tightest Qualifiers on Fields Analysis Results

6.3.3 Loop Invariants Analysis

The loop invariant analysis uses a dominator analysis to find all loops and then determines for all the statements in the loop if the statement is loop invariant, in other words if the statement could safely be executed outside the loop body without changing the semantics of the code. The most common loop invariant found was the recalculation of some expression in the condition part of a **for** loop. This type of loop invariant was found in half of the benchmark applications. Other loop invariants found were repeatedly accessing fields whose values did not change, repeatedly setting fields with unchanging values and other unnecessary binary re-computations. Each loop invariant expression is coloured red and is indicated with a `StringTag` as shown in Figure 6.22.



```
int k = 8;
for (int i = 0; i < k * k; i++){
}
}
```

Figure 6.22: Loop Invariant Analysis with Visualization Results

6.4 Summary

In this chapter we looked at various applications of the visual analysis framework. In particular, we considered examples where this framework is useful for compiler teaching and research and for general programmers use. We described the steps required to use the visualization framework and gave a concrete example of the steps from starting with a data flow analysis to finishing with the results displayed within the plugin. We then discussed several analyses and how to add visualization **Tags** in order to use the results in our visualization framework. We then looked at several analyses that may be used for program understanding, giving experimental results for their ability to find code improvement areas in real life code and showed how these results can be displayed within the visualization framework.

Chapter 7

Interactive Tools

7.1 Motivation

In chapter 5 we discussed viewing the final analysis results. Up until this point the Soot framework has functioned very much like a black box. Soot could be invoked with a set of options and any extra result information from the analysis had to be encoded somehow in relation to the input or output. While it is very useful to be able to see the final results, it is necessary to be able to reach that stage, so it is also useful to have a set of interactive tools which consider partially generated results as they are being computed and results, consisting of possibly large data sets, in formats unrelated to the representations available for input or output. In this chapter we discuss two such tools: an interactive control flow graph tool which is like a debugging tool for intra-procedural analyses and a tool for displaying partial views of large graphs.

7.2 Interactive Control Flow Graph Tool

In order to view the analysis results as they are being generated we provide a tool that displays control flow graphs and annotates them with data flow analysis information as the information is being generated.

This tool is useful for researchers when they are debugging complex analyses because it allows one to easily view partial data flow results. It is possible that the final results may conceal errors that can be easily caught when viewing the results as they are being generated. It can also indicate at which kinds of statements the analysis is being incorrectly computed.

For students learning about flow analysis, this tool gives insight into how fixed-point iterations in data flow analysis work. It can be used, by students, for debugging their first analyses and also for viewing simple, standard analyses, such as a liveness analysis, to explore how they function.

This tool is also useful for teaching data flow analysis and explaining control-flow graphs. It allows professors to generate the examples instead of using the blackboard, and as it provides back-up functionality, no blackboard erasing is necessary and re-drawing the previous step is trivial, when needed, when students have questions.

Soot analyzes each method for each analysis sequentially. When running Soot from within the Eclipse plugin in interactive mode, each time it begins a new analysis on a new method, it displays the control flow graph using an extension to the GEF framework [GEF]. Initially the control flow graph is displayed with only the statements as the nodes in the graph. For each new analysis, of each method, a new editor window is opened with the control flow graph. These graphs can also be saved to dot files [GKN02] for saving and printing purposes.

7.2.1 Running

Once the control flow graph is generated, the system becomes under the control of the user and pauses for user direction. The commands available to manipulate the graph are `step forward`, `step backward`, `finish method`, `next method` and `stop interaction`. The `step forward` command causes the data flow sets to be displayed one at a time as they are being generated. After going forward, it becomes possible to `step backwards` to remove the display of the previously generated flow sets. It is also possible to use the `finish method` command to run through the entire method in an animation automatically. The animation stops at the end of the analysis for

the method to display the final results for a particular analysis. After the method has been processed the **next method** command can be used and Soot will continue to the next method. At this point the display of previous method is fixed and can no longer be manipulated. The **stop interaction** command causes Soot to continue without further user interaction. Figures 7.2 - 7.5 show the progression of generating the data flow sets on several nodes in a control flow graph while computing the sets of live variables for part of the trivial method listed in Figure 7.1. In Figure 7.2 the

```
public void run () {  
    int x = 0;  
    for( int i = 0; i < 10; i++ ) {  
        x += 2;  
    }  
}
```

Figure 7.1: Live Variable - Interactive Control Flow Graph Example Code

first live variable **i** is generated as the data flow set on the **i = i + 1** node. This set is propagated to the **x = x + 2** node in Figure 7.3. Then in Figure 7.4 the variable **x** is added to the set of live variables. This is set is then propagated to the **if i >= 10 goto return** node in Figure 7.5.

7.2.2 Debugging

Sometimes it is necessary or desirable to see the data flow sets on each iteration for a particular node or nodes. A mechanism is provided for adding a small stop sign icon to each node to stop at on each iteration. Then the **finish method** command can be used to animate the process of generating data flow sets for each node in the method, and the animation will stop at each node with a stop sign icon, highlighting the node to draw attention to it as shown in Figure 7.6. Then the user can proceed with any of the action commands.

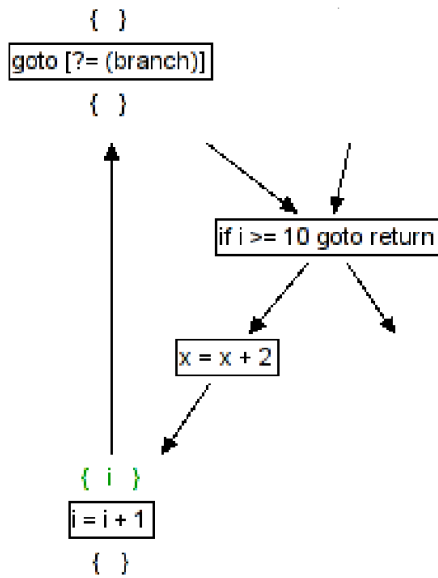


Figure 7.2: Add i to Data Flow Set

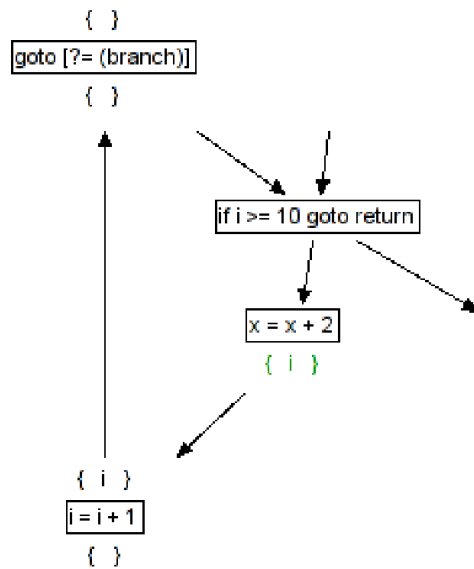


Figure 7.3: Propagate Set

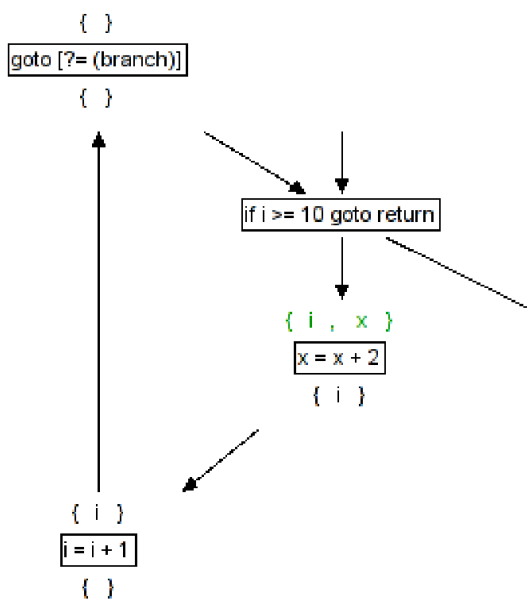


Figure 7.4: Add x to Set

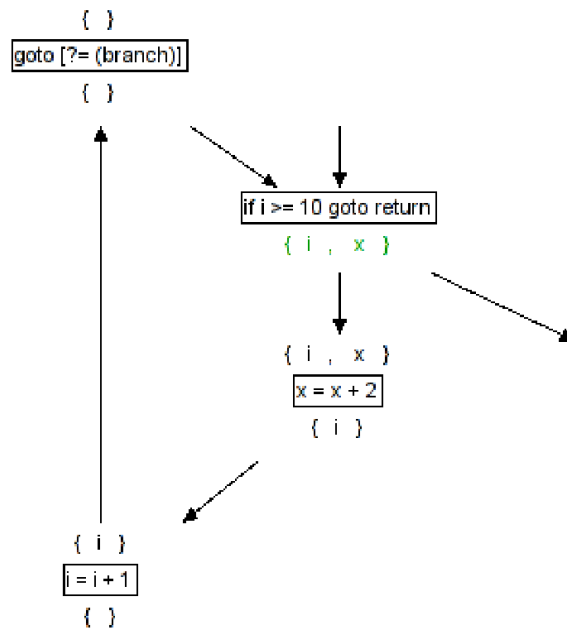


Figure 7.5: Propagate Set

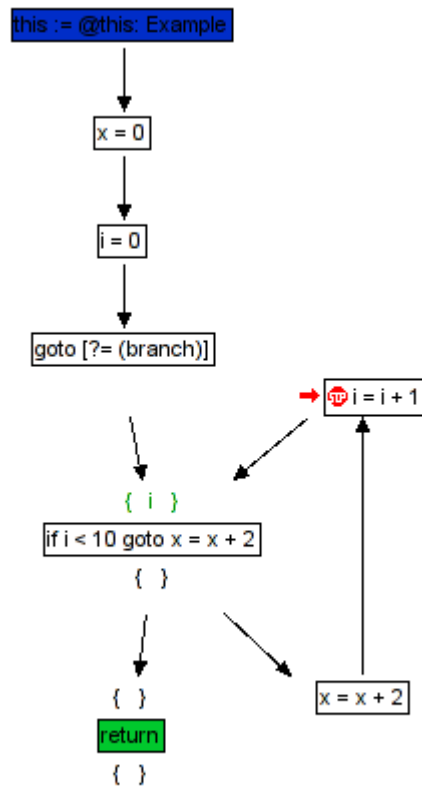


Figure 7.6: Partially generated flow sets on cfg with Liveness Analysis

7.2.3 Filtering Data Flow Sets for More Relevant Displays

Sometimes the data flow sets generated for an analysis are quite large and only a subset of the data is relevant. We provide a filtering mechanism where the data flow analysis will be performed on the full sets generated, but the displayed data will be only the filtered subsets. For example, for the parity analysis for each node a set mapping each local in the method to its parity is generated, but the filtered set of only locals that are live is displayed at each node as shown in Figure 7.7.

7.3 Motivation - Displaying Large Graphs

Call Graphs in Java can grow very large, very quickly, because of the large standard library which is referenced by even the most basic code. Even for a simple Hello World program the call graph may contain thousands of nodes or reachable methods, depending on the precision of the call graph. In our small study, the simple program shown in Figure 7.8 had 3000 - 6000 nodes in the call graph depending on whether the call graph was generated using Spark or a less precise class hierarchy based analysis as shown in Table 7.1

	Spark	CHA
classes	1424	1424
methods	3602	6784

Table 7.1: Reachable Method Counts for HelloWorld Call Graph

Displaying and viewing a large graph is very difficult and we sought a way to make a manageable display. To accomplish this goal, we developed an interactive, graph display tool, which is built on top of the GEF framework and interacts with information generated by Soot. This tool provides the basic functionality for laying out the graph, updating the graph with new nodes and allowing interaction between the graphical display and the code controlling the display. The problem of displaying a call graph has been built on top of this tool and we discuss it here. However, other

7.3. Motivation - Displaying Large Graphs

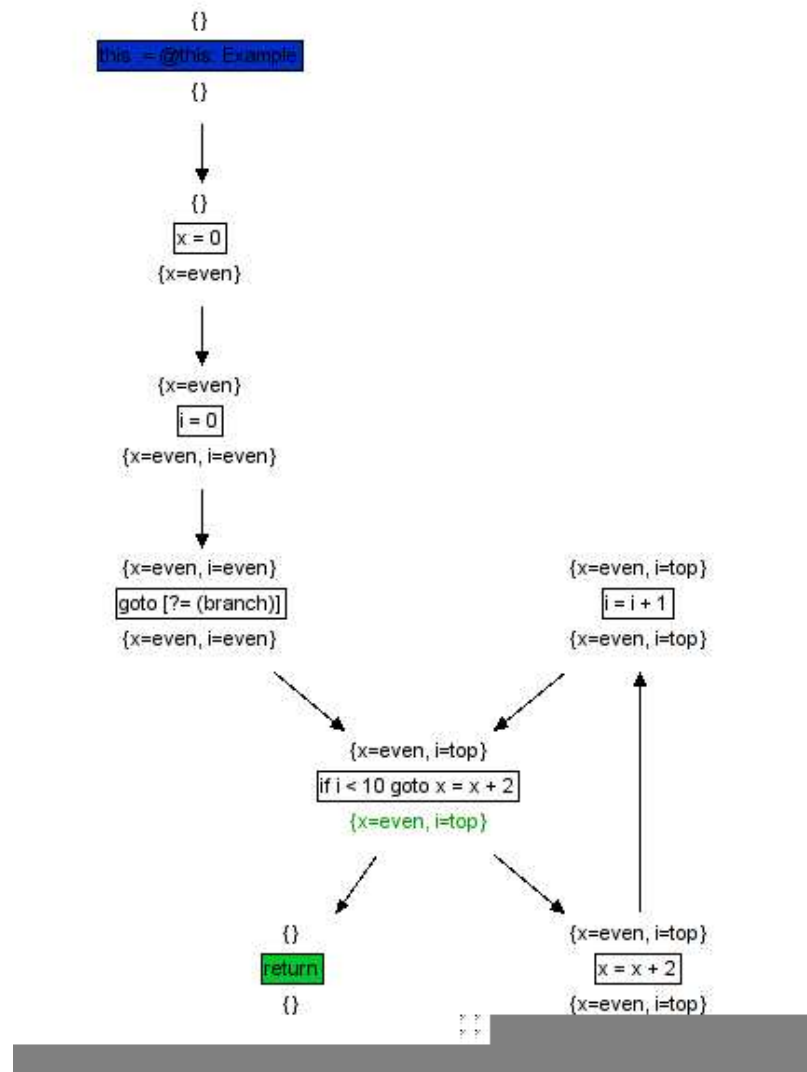


Figure 7.7: Annotated cfg with filtered Parity Analysis

```

public class HelloWorld {
    public static void main (String [] args){
        System.out.println("HelloWorld");
    }
}

```

Figure 7.8: Hello World Java Program

types of large graphs could also benefit from this approach, such as points-to graphs or some other compiler generated graphs.

7.4 Interactive Call Graph Tool

This tool displays an interactive call graph. It interacts with Soot while Soot is running. Soot processes the call graph and then using a message system this tool requests required information from Soot to display the call graph as required. It starts off by displaying the first main method that it finds as the start node. From there the user can expand the graph, collapse the graph and jump to the corresponding source code. Each node represents a method and each arc represents a call. Right clicking on a node displays a list of the options as shown in Figure 7.9 . Selecting the

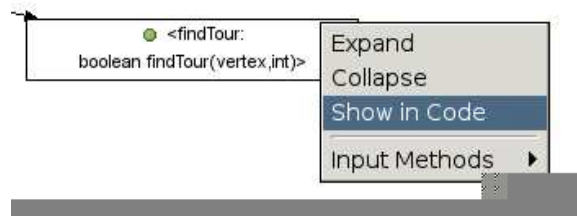


Figure 7.9: Interactive Call Graph Tool Options

expand option adds to the graph all methods called by the selected method. Each of these can then be further expanded. Selecting the **collapse** option will cause

7.4. Interactive Call Graph Tool

any nodes that haven't themselves been selected, to disappear. This functionality provides a pruned call graph where the user can focus on the interesting or important calls while ignoring insignificant ones. Selecting the **show in code** option opens a Java editor window with the corresponding method highlighted. There are also two toolbar options: **stop** which ends the interaction with Soot, and **collapse all** which resets the display of the graph to its original state. Each node shown in the graph indicates whether it is **public**, **private** or **protected** and we also show the call type on the arcs as well. This tool also allows the suppression of the display of library methods, in order that the user may focus on the application. A sample call graph of the telecom application is shown in Figure 7.10. In this example, the library methods are not displayed.

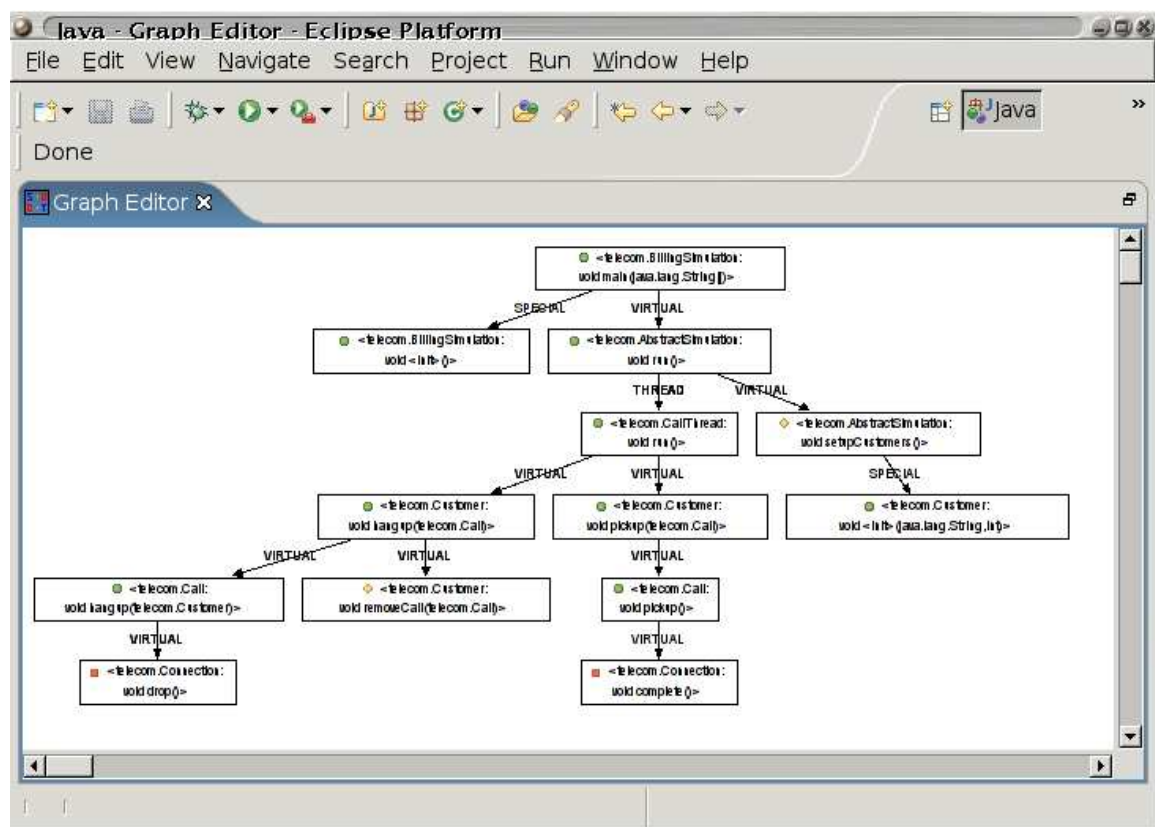


Figure 7.10: Interactive Call Graph Tool

Chapter 8

Related Work

There are three broad areas of previous work related to the visualization of compilers and compiler generated information. First, there are several projects which focus on displaying information generated by compilers, visually, in various different formats. Second, there is some past work which explores the issue of mapping source code to various intermediate representations (**IRs**) and the challenges associated with generating information based on the **IR** and displaying it in relation to the original source. Finally, there has been recent related work in the area of using compilers to find programming problems and to present the discovered information in visual environments. In this section, each area of related work is discussed in turn.

8.1 Views Displaying Compiler Information

In this section we look at research projects which use different kinds of views to display the information generated by the compiler. We compare these views to the views and displays in our project. We also explore the issue of propagating information between the underlying compiler and the user interface system.

VISTA [ZCW⁺02] is a system for interactive code improvement which arose out of a need to improve tools for compilers for embedded systems. The main features include displays of register transfer lists, the **IR** used for analyses, which include views of the code before the transformations which highlight instructions to be modified or

deleted and views after the transformations which highlight instructions which were modified or inserted, views showing the basic block structure, functionality for users to query for a list of loops, a list of live or dead registers at a specific program point, as well as dominators, successors and predecessors of a node. This system is one of the systems which is the most similar to our framework. The differences are that it provides tools for C, Pascal, and Ada code, as opposed to our tools which are primarily for Java code, and as well the user interface (UI) components are built as a stand alone system and not integrated into a popular IDE (although realistically, there probably did not exist a popular extensible IDE at the time this system was developed). One interesting feature about this set of tools is a view where users can determine the order in which to run the compiler transformations. This is an area that would be quite interesting to explore in our framework. Currently Soot, upon which our visualization framework is based, runs analyses in a very specific order.

A graphical user interface for compiler optimizations with Simple-SUIF [Har96] is a research project which provides a visual framework where the user may step through compiler components of the back-end of compiler optimizations as the optimizations are being performed. It is most related to our interactive control flow graphs, although our graphs are for analyses which do not affect the structure and this research deals with optimizations that change the structures of graphs. Similarly to the VISTA system, this framework allows for transformations to occur in any order and allows optimizations to be undone. It provides a variety of graph representations and it would be an interesting future work project to explore these representations and to determine which ones could be integrated into our Soot - Eclipse plugin.

The UW Illustrated Compiler [AHY88] is one of the earliest attempts at visualizing compilers. Its main focus was for helping students to learn about compilers and focused mostly on the front-end, whereas our work focuses on visualization of the back-end of the compiler. It graphically displays the control and data structures used during compilation.

Xvpodb [BW95, Boy93] is a stand-alone graphical tool designed to aid in porting compilers to different platforms. More specifically, its purpose is to provide a graphical debugging like environment specific to the back-end of a compiler, to help developers

catch bugs when re-targeting compiler optimizations for different target platforms. Unlike the tools in this project which are primarily for analyzing Java code, it is developed on top of the **VPO** [VPO] an optimizing code generator for C, Pascal and Ada. It provides a tool similar to our interactive control flow graph tool, which displays before and after states of the control flow graphs of a human readable form of register transfer lists, upon which the analyses are performed. This tool functions by using a message passing system from the compiler to the graphical view and allows for breakpoints and single stepping through the control flow graph as the optimization is being performed much like the interactive control flow graph tool described in this project. The main difference is that **xvpodb** shows optimizations which actually change the structure of the methods being analyzed and attempts to highlight the instructions that will change or have changed, whereas our interactive control flow graph display data flow sets which are being generated by the analyses as extra annotations for each node. This project is quite similar to the **VISTA** project.

The **Feedback Compiler** [BDJW98] provides visualization tools, which show optimizations performed in the back-end of the **LCC** [LCC] compiler, a compiler for C code. Instead of a general visualization framework, like what has been developed in this project, the feedback compiler attempts to provide a variety of different displays each suited to a specific compiler analysis. Two visualization displays are available in this compiler. It provides a visualization tool which animates the process of common sub-expression elimination and a separate tool for graphically visualizing the results of an optimized loop for an iteration space reshaping analysis. The main objectives of these compiler visualizations are to aid in debugging optimized code, to help in porting the compiler to new architectures and for showing programmers how compilers affect the code, whereas the goals of our work are to provide generic tools for compiler researchers and students, with only a small emphasis on visualizations for general programmers.

JAnalyzer [Bod03] is a stand alone graphical user interface (GUI) tool which provides an interactive visual representation of a call graph generated by Soot. It shows a node, representing a method, and all the caller and callee immediate neighbours. One can navigate through the graph by clicking on a node. This re-centers the graph

with the selected node as the center. This tool was built on top of Soot and thus can visualize any of the call graphs built by the different algorithms available in Soot (class hierarchy analysis (CHA), rapid type analysis (RTA), variable type analysis (VTA)) like our call graph tool can. It limits the view however to one main node and its neighbours only, whereas we allow the user to expand the view to include an unlimited number of nodes. Also, one of our goals was to reuse existing tools as much as possible and thus our tools are built into Eclipse taking advantage of the graphical editor and graph layout tools available, much of the JAnalyzer project focused on developing the required graphical tools. Finally, this tool allows a user to start from a source method and create a call graph, whereas our tools first provide the call graph for the class being analyzed and then, from the graph, the user can navigate to the corresponding source code method.

A static analysis tool for visualizing exception propagation for Java [CJH02] has been developed as a stand-alone tool, which displays the exception propagation path using an exception propagation graph. It is built on top of the IDE Jipe [JIP] and the analysis is done at the AST level and built upon the Barat [BAR] analysis system. This tool is specific to this exception propagation analysis and does not generalize to other analyses or visualizations, which was an important goal for the development of our tools.

A partial evaluation visualization tool [WD98] has been developed to visualize the effects of partial evaluation by keeping the source position information and highlighting in the source code the line being optimized. Side by side with the source editor, this tool shows a pretty printed version of the IR (Residual Program) relating to the optimization. The pretty printed version is not an editor like our IR editor but serves a similar function in that it displays the low-level IR in a human-meaningful way. These visualizations are for analyses performed on Scheme and Lisp code. The main purpose of this project is for promoting comprehension of legacy code but the secondary benefits include helping implementors and users understand the process and results of partial evaluation, which ties in with our goals of tools for compiler researchers and students to better understand the processes in the back-end of a compiler.

Integrating software productivity tools into Eclipse [RD03] is a project which aims to aid programmers of tools in visualizing the information generated. They provide a general system which attempts to link external tools which generate large amounts of structured data into two generic Eclipse views, specifically an XML tree and graph views. Further, they present two inter-procedural, external, static analyses that use this framework: a memory leak detector and a security vulnerability detector. They provide a standard XML based mechanism for integrating the generated data into Eclipse, somewhat like our tools which provide an XML format for generated analysis data, which is used to view analysis results in Eclipse.

An Architecture for Interoperable Program Understanding Tools [WOL⁺98] is a project which provides a framework for combining all different tools related to program understanding. It standardizes the IRs to accept different languages from the front-end, to provide common tools for building and accessing the different structures, such as control flow graphs, built by most modern compilers and to standardize the visualization tools used to display the information. It was designed to be a plug and play architecture much like Eclipse. The researchers have extended this framework by integrating a data-slice computation and visualization tool with a concept recognizer. They integrate the visualization tools that show the common graphs created by the framework without depending on the specific representation of the decomposition slice graph originally produced by the data-slice tool. This is similar to our tools which seek to be extensible to any kind of new analyses, however our tools all use the IRs available in the Soot framework.

8.2 Correlating IR and Source Results

In this section we look at work which has explored the area of correlating the results of analyses generated on an IR with the original source code. For each we discuss what challenges are faced and how they are handled.

OPTVIEW is an approach for examining optimized code [TG98]. This research indicates that while it is worthwhile to show at the source code level changes made

by the optimizer, it is difficult to make a correct and precise mapping between low-level and source code. Their solution to this problem is to present a view of partially modified source code with visual indications of how it was affected by the optimizer. We also show, at the source level, information generated by the compiler, but do not find as many difficulties mapping the IR to the original source. This may be because Jimple, our main IR, is much more similar to the Java source code, in contrast to assembly compared to C, used in the OPTVIEW project. We discuss our mapping between the source and IR in section 5.2.

Some research has been done on using visualization tools to teach compiler design [Veg01]. This teaching project was essentially a visualizer for the front-end of a compiler (mainly the AST) and was created for helping students learn about compilers. It is similar to our goals in a sense however as it seeks to provide a mapping between the source code and the IR using an annotated display and is used for students to see the relation between the source and AST (which in this case is the IR). The primary objectives for this teaching project were to aid in the debugging of the students' compiler construction project. Our tools have been used to aid students in the development of back-end compiler analyses in an advanced compiler optimization course.

8.3 Static Analysis to Highlight Coding Problems

In this section we discuss work related to finding programming problems, using static analyses and displaying the results in a graphical format. For each area of related work we look at whether there are analyses that are already handled or could be handled or could benefit from our Soot - Eclipse framework.

Finding Bugs is Easy [HP04] is a research project which makes use of static analyses to automatically detect bugs in large software projects. It performs the analyses on Java bytecode, using the BCEL library [BCE] to manipulate the bytecode and is integrated in Eclipse. It finds and highlights bugs in much the same way the front-end of a traditional compiler in an IDE highlights syntax errors with a list with links to

and highlighting of the source code line. This project includes many bug checks, some of which are related to standard compiler analyses such as null dereference checks, synchronization checks and security access checks. In our project we provide a visualized view of the results on a null checks analysis as described in section 6.2.3. It would be an interesting future work project to determine which of the checks performed in the finding bugs is easy project could benefit from advanced compiler analyses and be incorporated into the Soot framework.

Models of Thumb - Assuring Best Practice Source Code in Large Java Software Systems [HS02] is research which is similar to the previous research about finding bugs, except instead of finding bugs it highlights questionable programming practices using a different type of static analysis (not standard data flow analysis). In particular it finds places where code written with a lack of attention to quality or maintainability, checking areas such as over-specifying variable types or not handling exceptions in catch blocks. The results of the analyses are displayed in a graphical tree structure in a view in Eclipse, from which the user may navigate to the problematic source code. As this work performs very simple analyses on the Java AST in Eclipse, it is not necessary to correlate position information of the IR and the original source code. As well, as it performs analyses which may be undecidable it contains a system in which the programmer can help the analysis by specifying information in `javadoc` annotations. Additionally, it would be interesting, as future work, to perform and display the results of the over-specified variable types in the Soot - Eclipse framework, as this analysis may benefit from the more precise points-to information available in Soot.

SABER: Smart Analysis Based Error Reduction [RSS⁺04] is recent work in which deep static analysis is used to find errors in enterprise Java (J2EE) code and to display the results in a graphical way. It identifies many potential coding problems and categorizes them into six categories. It then uses six different techniques to find occurrences of the problems in the code. Like Soot, it computes a call graph and points-to analysis and other standard compiler analyses to use as the basic structures for information to find the coding problems. The results are displayed in a stand-alone user interface where the problem category is highlighted and the specific instance of

the problem found in the code is given, then the user may navigate to the actual code. It would be a very interesting future work project to perform these analysis in Soot and display the results with our visualization framework. The main goal of this work is to find code problems which are very difficult to track down during production.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

This thesis introduced the Soot - Eclipse plugin, a framework for visualizing an optimizing compiler for the purposes of developing compiler optimizations, learning about compiler optimizations and using compiler optimizations to provide extra program information to general programmers. It presented the basic integration of Soot into Eclipse, the analysis visualization framework and the interactive control flow and call graph tools.

The basic plugin makes the Soot optimizing compiler framework accessible to new users and students learning about compilers, in a simple graphical environment. This basic plugin provides standard menu options for invoking commonly used Soot functions and a view for displaying the output generated by Soot. The generated options dialog, included in the basic plugin, ensures that the plugin always stays synchronized with the compiler framework and this dialog, along with the managing configurations dialog, give advanced users the ability to manipulate the Soot framework. The IR editor, along with its associated content outline, provides support for learning about the various IRs produced in the Soot framework.

The analysis result visualization framework provides mechanisms for attaching and displaying visual information regarding the results of analyses, relating to both the IR and the original source code, in an extensible and flexible way. It includes Java

To Jimple a source to Jimple code generator which associates position information of the original source code with the constructs in the Jimple IR. Three visualization **Tags**: **StringTags**, **ColorTags** and **LinkTags**, are provided within Soot to be used to encode analysis results. These **Tags** are easy to use to encode analysis results in a flexible way and are adaptable for future types of analysis results. These **Tags** are collected, displayed and updated within the plugin automatically. This framework provides tools for verifying the correctness of analyses to advanced compiler researchers and new compiler students alike. Additionally, this framework along with the sophisticated compiler analyses which can be computed within Soot provide additional program information to general programmers.

The interactive control flow graph tool provides an integrated mechanism for learning about compiler analyses and debugging analyses as they are being generated. This tool displays the control flow graphs as they are being analyzed by Soot and updates the data flow sets of generated analysis information, allowing the user to step-through the analysis to find problems and imprecision.

The extensible graph tool provides a way to display large amounts of data generated by the compiler, by limiting the amount of information shown at one time. Additionally, the example of the interactive call graph can be used to verify correctness of the call graph and as reference when building analyses on top of the call graph in Soot. This interactive call graph tool allows the user to expand and collapse the call graph and navigate to the corresponding source code.

As part of this work Java to Jimple, a source to IR code generator was developed and it has been used further for the **abc** project¹: an extensible, optimizing research compiler for the AspectJ language.

Additionally, the analysis results visualization framework has been used by students in an advanced optimizing compiler course, where the students successfully used the results to help figure out where their analyses were broken. It has also been used in other projects for visualizing analyses results for side effect analyses and security flow analyses.

¹<http://aspectbench.org>

9.2 Future Work

There are several areas of future research which could be developed.

As mentioned in the section on related work there are several analyses and visualization displays that could be integrated into this framework, such as some of the analysis from the Finding Bugs is Easy project [HP04] or the SABER project [RSS⁺04], which could benefit from the precision of information generated by Soot, or could be enhanced in our visualization environment. Additionally, there are several kinds of graphs, such as directed acyclic graphs or register interference graphs, and other information which we do not display but could display as interesting extensions to this work.

Additionally, views could be added to enable the specification to run the Soot analyses in any order. This would require support for such a feature in Soot, but would enable researchers and students to explore the possibilities of how different analyses interact with each other.

The visualization framework discussed here has already been used to aid advanced compiler students in designing analyses and learning about compiler optimizations. More resources could be developed to further enhance this learning opportunity for future students.

The framework is now being used for different kinds of visualizations and as a future work more compiler analysis projects that can benefit from the visualization framework could be implemented.

In this work we describe the mechanisms for communication between the underlying compiler system and the user interface environment, however, there could be more tightly integrated approaches possible to streamline the whole system.

Appendix A

User Guide

The Soot - Eclipse plugin is freely available under the Lesser General Public License as part of the Soot framework. It is available for download as part of the Soot package. The Soot framework can be found at:

- <http://www.sable.mcgill.ca/soot>

Documentation specific to the Soot - Eclipse plugin can be found at:

- <http://www.sable.mcgill.ca/soot/eclipse>

Bibliography

- [AHY88] K. Andrews, R. R. Henry, and W. K. Yamamoto. Design and implementation of the uw illustrated compiler. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 105–114, New York, NY, USA, 1988. ACM Press.
- [BAR] Barat.
URL: <<http://sourceforge.net/projects/barat>>.
- [BCE] Bcel.
URL: <<http://jakarta.apache.org/bcel/>>.
- [BDJW98] D. Binkley, B. Duncan, B. Jubb, and A. Wielgosz. The feedback compiler. In *IEEE Sixth International Workshop on Program Comprehension*, June 1998.
- [Bod03] Eric Bodden. A high-level view of java applications. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 384–385, New York, NY, USA, 2003. ACM Press.
- [Boy93] Mickey Boyd. Graphical visualization of compiler optimizations. Master's thesis, Florida State University, July 1993.

- [BW95] Mickey R. Boyd and David B. Whalley. Graphical visualization of compiler optimizations. *Journal of Programming Languages*, 3:69–94, 1995.
- [CJH02] Byeong-Mo Chang, Jang-Wu Jo, and Soon Hee Her. Visualization of exception propagation for Java using static analysis. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, pages 173–182. IEEE Computer Society, 2002.
- [ecl03] Eclipse platform technical overview. Technical report, Object Technology International, 2003.
URL: <<http://www.eclipse.org/>>.
- [GEF] The graphical editing framework.
URL: <<http://www.eclipse.org/gef/>>.
- [GKN02] Edmen Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing Graphs with dot*, 2002.
- [Har96] Brian Keith Harvey. Graphical user interface for compiler optimizations with Simple-SUIF. Master’s thesis, University of California, Riverside, December 1996.
- [HP04] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOP-SLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136. ACM Press, 2004.
- [HS02] T.J. Halloran and W.L. Scherlis. Models of thumb: Assuring best practice source code in large java software systems. Technical report, Carnegie Mellon University, 2002.
- [INN96] Sun microsystems: Inner classes specification, 1996.
URL: <<http://java.sun.com/products/archive/jdk/1.1/index.html>>.
- [JIP] Jipe a free java ide.
URL: <<http://jipe.sourceforge.net/>>.

- [LCC] lcc: A retargetable compiler for ansi c.
URL: <<http://www.cs.princeton.edu/software/lcc//>>.
- [LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.
- [Lho02] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master’s thesis, McGill University, December 2002.
- [LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., second edition, April 1999.
- [MH01] Jerome Miecznikowski and Laurie Hendren. Decompiling java using staged encapsulation. In *WCRE ’01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE’01)*, pages 368–374, Washington, DC, USA, 2001. IEEE Computer Society.
- [MH02] Jerome Miecznikowski and Laurie J. Hendren. Decompiling java byte-code: Problems, traps and pitfalls. In *CC ’02: Proceedings of the 11th International Conference on Compiler Construction*, pages 111–127, London, UK, 2002. Springer-Verlag.
- [Mie03] Jerome Miecznikowski. New algorithms for a java decompiler and their implementation in soot. Master’s thesis, McGill University, February 2003.
- [NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 138–152, April 2003.

- [PQVR⁺01] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing Java using attributes. In *Compiler Construction, 10th International Conference (CC 2001)*, volume 2027 of *LNCS*, pages 334–554, 2001.
- [QHV02] Feng Qian, Laurie Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for Java. In *Compiler Construction, 11th International Conference*, volume 2304 of *LNCS*, pages 325–341, April 2002.
- [RD03] Will Robinson and Ben D’Angelo. Integrating software productivity tools into eclipse. In *eclipse ’03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 40–44. ACM Press, 2003.
- [RSS⁺04] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, and Larry Koved. Saber: smart analysis based error reduction. In *ISSTA ’04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 243–251, New York, NY, USA, 2004. ACM Press.
- [SHR⁺00] Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA ’00)*, pages 264–280, 2000.
- [TG98] Caroline Tice and Susan L. Graham. OPTVIEW: a new approach for examining optimized code. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 19–26. ACM Press, 1998.
- [Veg01] Steven R. Vegdahl. Using visualization tools to teach compiler design. *The Journal of Computing in Small Colleges*, 16(2):72–83, 2001.

- [VPO] Zephyr: Very portable optimizer.
URL: <<http://www.cs.virginia.edu/zephyr/vpo/>>.
- [VR00] Raja Vallée-Rai. Soot: A java bytecode optimization framework. Master's thesis, McGill University, July 2000.
- [VRGH⁺00] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, volume 1781 of *LNCS*, pages 18–34, 2000.
- [WD98] Oscar Waddell and R. Kent Dybvig. Visualizing partial evaluation. *ACM Comput. Surv.*, 30(3es):24, 1998.
- [WOL⁺98] S. Woods, L. O'Brien, T. Lin, K. Gallagher, and A. Quilici. An architecture for interoperable program understanding tools. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, pages 54–63. IEEE Computer Society, 1998.
- [ZCW⁺02] Wankang Zhao, Baosheng Cai, David Whalley, Mark W. Bailey, Robert van Engelen, Xin Yuan, Jason D. Hiser, Jack W. Davidson, Kyle Galivan, and Douglas L. Jones. Vista: a system for interactive code improvement. In *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 155–164. ACM Press, 2002.