

# Visualizing Program Analysis with the Soot-Eclipse Plugin<sup>1</sup>

Jennifer Lhoták and Ondřej Lhoták

*Sable Research Group, McGill University, Montreal, Canada*  
{jlhotak,olhotak}@sable.mcgill.ca

---

## Abstract

Our integration of the Soot bytecode manipulation framework into the Eclipse IDE forms a powerful tool for graphically visualizing both the progress and output of program analyses. We demonstrate several examples of the visualizations that we have developed, and explain how they are useful for both compiler research and teaching.

*Key words:* program analysis, visualization

---

## 1 Introduction and Motivation

Soot<sup>2</sup> [3] was developed as a bytecode analysis and transformation toolkit that performs both intra- and inter-procedural static analyses and transformations. It has been used extensively by researchers both within the Sable Research Group and elsewhere to experiment with their new analyses, and provides a common framework for comparison of empirical results. In addition, Soot is used regularly for teaching optimizing compiler courses at McGill and other universities, forming a base for student assignments and projects.

In integrating Soot into the Eclipse IDE, our goals were to make Soot more accessible to students and researchers, and to make possible greater interaction between Soot and its users. We have focused on providing a set of graphical tools to control the behaviour of Soot and for Soot to report information back to users. The key parts of the Soot-Eclipse plugin are a Soot launcher, which allows the user to configure Soot's many options and launch Soot, an intermediate representation (IR) editor for viewing and editing Soot's IRs, and a visualization component which displays program analysis results in the

---

<sup>1</sup> This work was supported in part by NSERC and an IBM Eclipse Innovation Grant.

<sup>2</sup> The Soot package including the Soot-Eclipse plugin is freely available from the Soot homepage: <http://www.sable.mcgill.ca/soot>

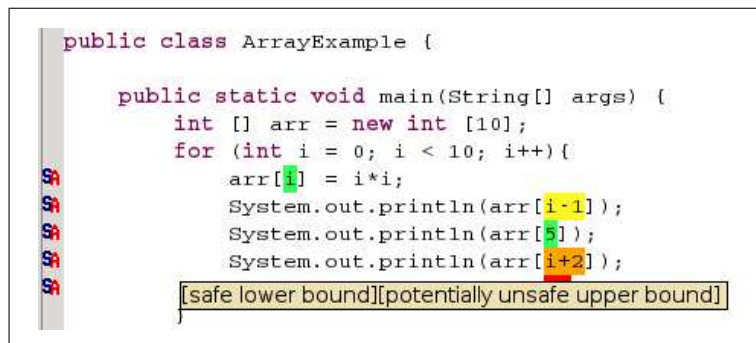
IR and source editors. A common theme in the development of Soot, and of the Eclipse plugin in particular, has been the effort to make it as generic as possible, making it a suitable framework for the development of new analyses and transformations, rather than just a collection of existing ones.

Both main uses of the Soot framework, compiler research and teaching, benefit from the program analysis visualizations. For researchers, being able to see the progress and results of an analysis makes it easier to develop new, complicated analyses. In teaching, the visualizations can be used for instructor-led demonstration of analysis algorithms, as well as by students to better understand and debug program analyses that they implement.

## 2 Visualizing Program Analysis Results

Soot includes a generic annotation framework [1] for encoding the results of analyses in *tags* attached to parts of the IR (expressions, statements, methods, fields and classes). The annotation framework propagates the tags between the various IRs in Soot, and can also encode them in class file attributes for use by other tools. The Soot-Eclipse plugin reads the tags as the source of information to be visualized. This means that a user wanting to visualize the results of a new analysis need only encode them using the standard annotation framework tags, and they will immediately become visible in Eclipse.

For visualizing different types of results, we have introduced three new types of tags. The information stored in these tags is read into Eclipse and displayed with any of the IRs or original source code. *String tags* encode textual information, and are displayed as tooltips when the mouse is moved over the associated piece of code. *Colour tags* are used to highlight sections of code (such as variables or expressions) in colour. *Link tags* encode connections between different statements, and show up as lists of links with which one can navigate to another piece of code.



```

public class ArrayExample {
    public static void main(String[] args) {
        int [] arr = new int [10];
        for (int i = 0; i < 10; i++){
            arr[i] = i*i;
            System.out.println(arr[i-1]);
            System.out.println(arr[5]);
            System.out.println(arr[i+2]);
        }
    }
}

```

Fig. 1. Analysis results for array bounds check analysis

Figure 1 shows the visualization of the result of one example analysis, array bounds check analysis [2]. For each array access, there are four possible analysis results: each of the upper and lower bounds is either safe or potentially unsafe. We encode each possible outcome using one of four colours, and add

colour tags to the array accesses, which causes them to be highlighted with the appropriate colour in the IR and source code. In addition, the analysis results are encoded in textual form using string tags. The text appears in a tooltip when the mouse is moved over each array expression.

### 3 Visualizing Progress of Dataflow Analysis

Soot includes a generic fixed-point dataflow analysis framework. To implement a dataflow analysis, a user of the framework implements the relevant flow equations, and the framework performs the fixed-point computation. The dataflow framework is intended for Soot users to implement their own analyses, but it also underlies many of Soot's internal analyses. In the optimizing compiler course at McGill, students are required to implement an analysis of their choice using the framework.

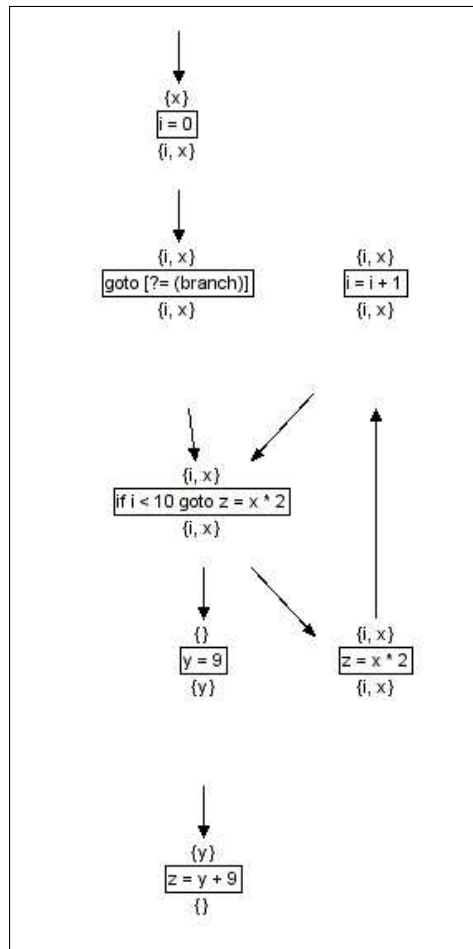


Fig. 2. Fixed point dataflow analysis for live variables

The Soot-Eclipse plugin makes it possible to visualize the progress of any analysis implemented using the fixed-point framework. As shown in Figure 2, the plugin displays a graphical representation of the control flow graph of

the method being analyzed. As the fixed-point computation proceeds, the dataflow facts being computed by the analysis are successively displayed at the nodes of the graph. In this case, we are visualizing a live variable analysis, so the sets of live variables are displayed. However, the progress of an arbitrary analysis can be traced as long as appropriate methods are provided to convert the dataflow facts to strings. A button in the user interface allows the user to single-step through the analysis, performing the computation one node at a time. A back button is also provided to undo each computation and move backward in the analysis progress.

Although Soot started as a framework mainly for research, its use for teaching has become increasingly important, and we expect the visualization tools to make it even more suitable for this purpose. In a typical optimizing compiler course, the instructor teaches dataflow analysis by tracing through an example analysis on a control flow graph on the blackboard. The same tracing can now be done using the plugin using the actual analysis, making it possible for students to trace through analyses of their choice at their own pace. One of the biggest difficulties that students face when they implement and debug their own dataflow analysis is understanding what is happening as their analysis proceeds. Being able to trace through the analysis that they have implemented one step at a time, and even going backwards, will hopefully help them to gain a better understanding of dataflow analysis, and make the task of implementing and debugging them easier and more enjoyable.

## 4 Summary

We have briefly demonstrated the program analysis visualization tools implemented in the Soot-Eclipse plugin. We hope researchers, instructors, and students alike will find them helpful for implementing and understanding their own program analyses.

## References

- [1] Pominville, P., F. Qian, R. Vallée-Rai, L. Hendren and C. Verbrugge, *A framework for optimizing Java using attributes*, in: *Compiler Construction, 10th International Conference (CC 2001)*, LNCS **2027**, 2001, pp. 334–554.
- [2] Qian, F., L. Hendren and C. Verbrugge, *A comprehensive approach to array bounds check elimination for Java*, in: *Compiler Construction, 11th International Conference*, LNCS **2304**, 2002, pp. 325–341.
- [3] Vallée-Rai, R., E. Gagnon, L. J. Hendren, P. Lam, P. Pominville and V. Sundaresan, *Optimizing Java bytecode using the Soot framework: is it feasible?*, in: *Compiler Construction, 9th International Conference (CC 2000)*, LNCS **1781**, 2000, pp. 18–34.