Combining Static and Dynamic Data in Code Visualization

David Eng
Sable Research Group
McGill University, Montreal
flynn@sable.mcgill.ca

ABSTRACT
The task of developing and debugging compiler optimizations is a difficult one which is often facilitated by software visualization. There are many characteristics of the code which a developer must consider when studying the kinds of optimizations which can be performed. Both static data collected at compile-time and dynamic runtime data can determine the potential and result of a code transformation. In addition, the format and semantics of the programming language may depend on the tool being used or the operation being performed by the compiler. Visualization of such complex systems must include as much information as possible, and be able to adapt to the different sources from which this information is acquired.

This paper presents a software visualization framework with design goals which address these issues. Using an extensible intermediate language called JIL, both static and dynamic code data can be collected from supported compiler and software tools and then visualized.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors – *compilers, optimization.*

General Terms
Performance, Design, Experimentation, Languages.

Keywords
Combining static and dynamic data, visualization, intermediate languages, profiling, software understanding.

1. INTRODUCTION
Software visualization is a well-explored research area which has been applied to several aspects of computing [6]. Most applications encourage program understanding during development or maintenance. Object-oriented programs can easily obscure the original solution for which they were designed to implement. Visualization has been shown to improve programmer productivity, especially in complex systems which can span the work of several programmers over extended periods of time [1].

Object-oriented programming languages have also promoted the combination of static and dynamic data in program analysis and visualization. With such features as polymorphism and dynamic typing, it is important to consider both compile-time and runtime characteristics of modern object-oriented languages [9].

The framework presented in this paper is targeted at the visualization of generated code from an optimizing compiler. In this environment, the scope of the visualization covers both the software and the underlying characteristics of the execution platform. This data can be extracted from different intermediate languages and other representations, as well as sources of runtime data. In the following sections I will discuss the overall visualization framework and how we achieved interoperability between languages, tools, and machines.

2. VISUALIZATION FRAMEWORK
Price et al. describes the scope of a visualization in terms of the class and scale of the programs it can visualize [7]. In Figure 1 we present a visualization framework which is designed to be customizable and scalable. The foundation of this system is based on the Java Intermediate Language (JIL) which can encapsulate existing intermediate languages and support both static and dynamic program characteristics. Code and software tools will be presented which support JIL and provide suitable sources of both static and dynamic characteristics of the code. We then present an example visualization interface which can combine these characteristics with minimal implementation by using JIL as a data source.

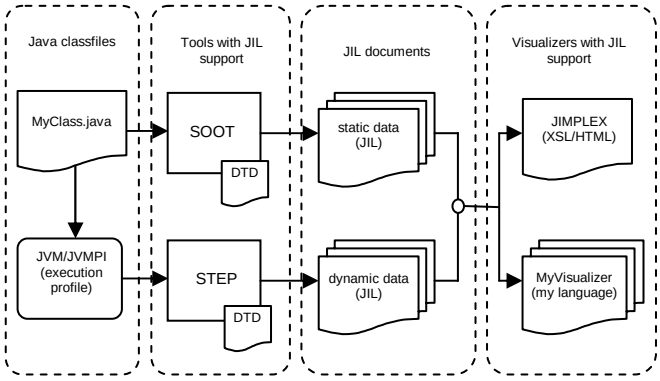

Figure 1. Overview of the visualization framework.

2.1 JIL
The Java Intermediate Language (JIL) is an Extensible Markup Language (XML) based format for encapsulating intermediate representations of Java source code [4]. We designed JIL as an extensible, but strictly defined language which supports extensions which might not have been envisioned yet. JIL benefits from many of the features which are intrinsic in XML,

such as portability between platforms and across networks, backwards compatibility with previous language versions, and the ability to be easily generated, validated, or parsed [2].

JIL is used as the data source in the visualization framework, providing persistent storage which is easily shared and managed. As a subset of XML, the production and consumption of JIL documents is facilitated by existing tools and APIs. Tools which support JIL can easily validate any incoming documents by using a Document Type Definition (DTD). DTDs are a form of XML schema which restricts the structure layout of a document. They define which elements can appear where, and which attributes can be used to describe them. DTDs are simple to extend, and provide a strict method of enforcing compatibility between JIL tools.

JIL documents are composed of a hierarchy of nested tags which describe the layout of a Java class. This includes enumerations of the fields and methods of the class, including elements which are only found in bytecode such as labels. This basic skeleton of code elements provides a framework upon language extensions can be added. These extensions can include both static and dynamic characteristics of the data, and allow such information to be associated directly to each code element.

In order to demonstrate the kinds of extensions supported throughout this paper, we will consider polymorphism as a characteristic of the code with both static and dynamic properties. Figure 2a shows a fragment of Java code where polymorphism can be explored at compile-time by analyzing the potential targets of call sites, while the runtime behavior of these sites can reveal which targets are actually being invoked. The following sections will continue this example and describe two tools capable of exporting these types of analyses as JIL extensions.

2.2 SOOT

SOOT is an optimization framework for Java which uses intermediate languages to perform static analysis and transformations on Java bytecode [10]. SOOT can export a Java class as a JIL document, encapsulating the intermediate languages used within the optimization framework along with any associated analysis results.

SOOT supports several kinds of static analysis, and provides its own API for developing optimizations or transformations on Java bytecode. Developers working with SOOT must inspect different types of intermediate languages with subtly different semantics. Much like most intermediate languages, these are well suited for a specific operation, but conceal many aspects of the code. Flow analysis can reveal such information which can give a developer insight when designing future optimizations. JIL documents provide the SOOT framework with a format which can associate these native intermediate languages with their respective analysis results.

SOOT defines some basic language extensions for JIL which include such static information. For example, each statement is associated with a list of variables, indicating which are live before and after that line of code. This information would simply be nested within each statement tag, preserving the existing structure of the document.

Continuing our previous example of polymorphism as a code characteristic, SOOT supports several types of static analyses which can associate potential targets with call sites. Call sites within the JIL documents produced by SOOT can be easily identified as monomorphic or potentially polymorphic by extending each site with these analysis results. Figure 2b shows a simplified example of what a call site extended with class hierarchy (CHA) and variable type (VTA) might look like as JIL.

2.3 STEP

STEP is a customizable profiling framework for evaluating the performance and behavior of object-oriented applications; an early revision of the framework was published under the name STOOP [3]. Existing profiling systems can be difficult to apply to complex applications, or can only collect a limited set of runtime characteristics. Visualization of such data is typically limited to the same fixed domain of events, and is normally separated from any static analysis. STEP allows developers to select the kinds of data they want to collect by building custom profiling agents. These agents instrument existing code according to which aspects were requested to be profiled. This allows the rapid development of a variety of profilers which apply to both standard and unconventional profiling tasks.

STEP provides its own event-based language and accompanying compilers. Profiling agents define events using this language and pass them into an event pipe where the data is compressed and prepared for consumption. We provide a backend to this event pipe which can generate JIL documents. The code elements within these documents are annotated with the runtime data collected by the profiling agents. The JIL generator is an easily implemented example of an event pipe consumer, and the extensible nature of the profiling framework is well suited to persist its output in an extensible document format.

We can now revisit our polymorphism example using STEP to generate JIL dynamic data. The JIL fragment at the bottom of Figure 2c demonstrates how STEP associates actual runtime invocation targets to a particular call site. Although our static class hierarchy analysis indicated that this call site had three potential targets, according to the data collected from this execution run only two targets was invoked. Runtime data requires some extra information describing the execution environment and any other details that are required to uniquely identify the data associated with it. The separation and management of dynamic data will be described later in this document.

2.4 JIMPLEX

Internally, SOOT uses an intermediate language called Jimple before generating optimized bytecode. Jimple is typed three-address code suitable for applying transformations which might move or remove code elements. Optimizations such as algebraic manipulation, common sub-expression elimination, and constant propagation can be applied directly to Jimple, making it a useful language for code visualization. The JIL documents generated by SOOT associate Jimple with extended information, such as the behavior of any variables contained within each statement.

JIMPLEX is a visualization implementation for JIL with the goal of providing a customizable visualization framework for browsing Jimple. Developers working with SOOT and Jimple required a tool which would allow them to develop and debug optimizations. By using JIL as a data source, JIMPLEX can annotate Jimple code elements with any static or dynamic data collected by SOOT and STEP.

a)

```
// myclass.java
...
myInterface myObject;
if( branch1 ) myObject = new A();
else if( branch2 ) myObject = new B();
else myObject = new C();
myObject.myMethod();
...
```

b)

```
<!-- JIL: SOOT output of myclass -->
...
<statement>
 <jimple>interfaceInvoke 48.myInterface:
         void MyMethod()</jimple>
 <soot_invoketargets method="myMethod">
  <targets analysis="CHA" count="3">
   <target class="A"/>
   <target class="B"/>
   <target class="C"/>
  </targets>
  <targets analysis="VTA" count="2">
   <target class="A"/>
   <target class="B"/>
  </targets>
 </soot_invoketargets>
</statement>
...
```

c)

```
<!-- JIL: STEP output for myclass -->
...
<statement>
 <jimple>interfaceInvoke 48.myInterface:
         void MyMethod()</jimple>
 <step_callsite method="myMethod">
  <targets count="2">
   <target class="A" invokecount="55"/>
   <target class="B" invokecount="45"/>
  </targets>
 </step_callsite>
</statement>
...
```

d)

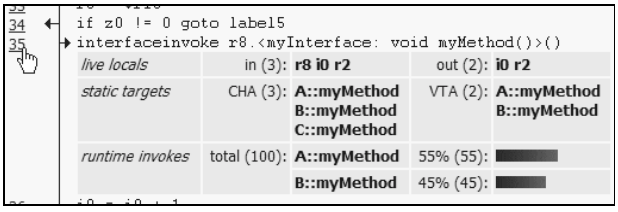

Figure 2. a) a simple polymorphic Java fragment with an interface invoke; b) a JIL fragment produced by SOOT indicating the possible targets of the call site; c) a JIL fragment produced by a STEP profiling agent indicating the actual number of runtime invokes; d) a slice from the JIMPLEX interface focused on the call site, where the user can browse the extensions from b) and c).

This allows the user to browse Jimple annotated with data from multiple tools, such as we saw in the JIL fragments in Figure 2b and 2c. Figure 2d is a screenshot of the JIMPLEX interface, where the user is focused on the Jimple statement containing the polymorphic call site. By comparing the results of static analyses to the actual runtime behavior we can verify that the variable type analysis (VTA) performed by SOOT was accurate in eliminating the potential target from class C, based on the last profiling run.

In order to facilitate the visualization of future JIL extensions, the implementation of JIMPLEX is easily customizable. JIMPLEX is an XML application which uses XSLT to stylize and transform JIL documents, delivering visualization interfaces as HTML to common web browsers. XML transformations can be performed on the fly in modern browsers, allowing the interface to be customized without having to recompile code or learn a complex API. These technologies also encourage collaboration, allowing visualizations to be shared between platforms and devices across the Internet.

3. INTERMEDIATE LANGUAGES

Intermediate languages are used by optimizing compilers to give the separate modules a representation to work with which is independent of a machine or platform. This encourages modularity throughout the compiler, and allows generated code to be retargeted towards another platform without having to re-implement any analyses or optimizations. Java itself has been described as an ideal intermediate language, providing strong types and other language features which help debug a new language compiler [5]. However, when developing optimizations, transformations and analyses typically operate on a lower-level language which is closer to the target representation. Three-address code and other kinds of intermediate languages can represent control flow graphs and reveal optimizations which would be obscured or much more complicated in a higher-level or stack-based language.

Jimple is used as an intermediate language for applying optimizations in the SOOT bytecode optimization framework [11]. In order to translate stack-based bytecode to Jimple, an algorithm is used which converts stack references to temporary variables which can be used in three-address code explicitly. This can result in verbose code, however this conversion concentrates on correctness since Jimple can simplified further by applying constant propagation and other optimizations.

In the following sections we will present some of the problems solved by JIL as a common intermediate language, and discuss the role it plays in the visualization framework.

3.1 Intermediate Language Design

Intermediate languages are typically designed with specific goals which address a single process or operation. In the case of Jimple, the designers wanted a language which was easy to analyze, optimize, and convert to and from bytecode. This leads to a range of intermediate languages which are all optimized for their individual purpose, but with different semantics and language characteristics. These subtle differences prevent the development of tools and visualizers which can work with multiple languages, since their implementation is usually not worth the trouble.

In section 2.1 we presented an intermediate language for Java designed as a visualization data source. JIL could be described as metadata, as it describes intermediate languages in a common format. Its use differs from traditional intermediate languages in that its content is independent of the tools which use it. JIL is not designed to be manipulated or decompiled into bytecode, but provides a bridge between tools which use traditional intermediate languages and visualization interfaces.

JIL is an intermediate language which is both common and extensible. JIL can encapsulate other intermediate languages, associating Java classes with multiple code representations. JIL treats each language independently, allowing the relationships between code elements to be defined by the JIL author. SOOT, for example, defines a static relationship between variables and the statements where they were defined or used. STEP defines a similar relationship, by associating variables with runtime accesses. These kinds of data associations can be static or dynamic, and JIL provides an extensible format for defining them.

As an XML application, JIL benefits from many of the features of this well established format, including a growing number of tools and APIs as well as support coming from a large community of developers. Like XML, JIL documents are portable across platforms and networks, with native support in most modern web servers and clients. Compatibility is achieved by defining language semantics and restrictions using document type definitions (DTD); applications which use this standard XML schema for validating their input can identify which extensions are supported as well as their version information.

3.2 Static Code Elements

All JIL documents are required to contain some basic elements of a Java class file. These elements make up the fundamental structure of the document, upon which annotations can be added. Most common code annotations are analysis results which can be calculated by inspecting bytecode. These are static facts and characteristics of the code which are known once the Java source is compiled.

The SOOT optimization framework can perform several kinds of static analysis on Java bytecode. Most of these analyses are performed on Jimple code, where some basic optimizations can be applied before generating bytecode. Some simple analysis of which variables are live at each statement can be used to perform copy propagation and simple aggregation, where unneeded variable definitions can be collapsed.

SOOT exports these types of analysis results as extensions in the JIL documents it produces. These extensions are defined in an accompanying DTD, allowing tools to validate and identify JIL documents produced by SOOT. By associating extensions with the tools that generate them, several tools can annotate the same code elements independently. This also allows redundant or related extensions to be compared or combined by tools which support them.

3.3 Dynamic Code Elements

Data and code characteristics discovered at runtime can play a major role in the optimization of object-oriented programs. Languages such as Java feature many expensive features, such as polymorphism and garbage collection, which can drastically affect performance. The most effective optimizations target the expensive bytecodes which are generated by Java compilers to provide these features. However, these optimizations can not always be implemented based on static information alone.

Annotation of dynamic data in JIL works much like with static data. In order to manage both static and dynamic extensions, JIL documents each contain a history of all the tools and operations which have authored it. This allows dynamic data from separate execution runs of the same tool to coexist in a single document. STEP uniquely identifies each execution run by an entry in the history of the document indicating the profiling agent which was used, a timestamp, and information about the execution environment. This allows JIL tools to compare and process multiple data sets of runtime information.

4. MANAGING JIL DOCUMENTS

The framework presented here encourages interoperability between code tools, regardless of their implementation or supported languages. Any number of tools can contribute to a JIL document, allowing collaboration and modularity between tools which would normally be difficult to achieve. Adding support to existing tools requires very little implementation, and many APIs exist which can already validate, parse, and generate compliant JIL. The following sections will discuss the non-trivial task of providing this kind of interoperability in more detail.

4.1 Document Structure

JIL documents use the nesting structure of XML to represent the hierarchy of code elements in a Java class. The current JIL definition requires that documents contain a base structure consisting of several required elements. These describe the basic code elements which all classes will contain, such as enumerations of fields and methods. By nesting elements, JIL can contain optional extensions to these base elements which will not break the underlying structure of the document when removed. This makes document extensions easier to manage, since they can be swapped in or out, and new extensions can be introduced while maintaining backwards compatibility with previous versions of the same document.

JIL documents achieve this kind of manageability by strictly defining the grammar of their contents and self-describing any included extensions. A DTD is used to define the restrictions on which elements and attributes can and must be included. This DTD can also be extended with references to extension definitions, allowing the base JIL grammar to evolve independently of any extensions. Definitions are also independently versioned, so that validation can indicate which generation of JIL to expect or which extensions are available. Following the trend of other XML technologies, DTDs can be referenced locally or across the internet during validation.

4.2 Merging

There are several different kinds of merging which are possible when combining data from multiple JIL documents. In some cases, an entire package of JIL documents might describe the target of visualization. A single class could also be persisted as separate JIL documents, and then later combined during processing in order to improve performance or manageability. This section will describe the different types of merging which are possible.

The most straightforward merging involves JIL documents which represent different classes; in this case there is no merging

required. Data in these documents will not intersect or conflict since it is referring to an entirely separate class. Tools that want to browse a complete package or compare the data collected on separate classes can load each document and process them independently. There are no notable caveats in this case, and the details of how to handle each document is left up to the implementation.

Once a tool is handling JIL documents which refer to the same class, they can start to take advantage of the object-oriented document model when merging. Such tools typically treat these documents as a single entity describing a Java class. Each document's extensions and data can then be associated with this common entity, and their interactions are left up to the implementation. A simple union of all the data can be performed which will present the user with a single class representation which includes all the extensions from each document. This is a common case when a tool is passed JIL documents from separate sources, each containing different extensions on the same class. These extensions can be combined when loaded by the tool in order to hide their logical separation from the user. This can be convenient when visualizing multiple documents from different remote sources, where their physical separation becomes more of a convenience. For example, if one tool if still being developed and debugged, its extensions can be kept separate from those generated by other more stable tools. The ability to separate extensions in this way is also convenient for research groups working on different extensions independently across the Internet.

In some cases tools will generate document extensions which intersect, meaning they describe the same characteristics of the code with different empirical data. Such data is typically collected at runtime by profiling or benchmark tools over several execution runs, while visualization is targeted at the same code objects. Visualizers can display averages and other calculations by identifying and processing this intersecting data. This process requires some basic heuristics used by the visualizer to decide how to combine the data and present the user with code characteristics of interest.

The visualization framework we present separates the interface from the data. An open data representation allows custom interfaces to define how the user visualizes intersecting data. The format and structure of JIL is designed to give interfaces more flexibility when deciding how to interpret the data. Simple interfaces can allow basic filtering of datasets where the user can browse the evolution of the code's performance, while complex interfaces might use statistical operations and graphics in order to provide a more comprehensive representation. JIL tools which can offer some insight on the interpretation of multiple JIL documents can export any data they produce as additional JIL extensions. For example, given a JIL document describing the local variables which are live at each statement, another tool could interpret this data and export an additional JIL document containing lists of variables which are unused and could be eliminated in each method. By chaining the processing and interpretation of JIL documents, visualizations can become more complex and cover a larger scope of code characteristics. This also allows a many-to-one relationship between code tools and visualizers.

The process of merging JIL document extensions is not trivial, but is facilitated by the wide array of libraries and APIs which can process and parse XML. Many basic combinatory operations are supported by basic interface languages such as XSLT and PHP. These kinds of interpreted languages can allow quick prototyping of new and experimental visualizations. By using a scalable data format, JIL tools can process as many documents or extensions as required by the visualization. Most APIs also support the loading of documents using the well established HTTP protocol for network transmission. This will encourage visualizers to support the visualization of remote JIL documents, allowing collaboration between tools which might exist on different machines or networks.

The labeling and versioning of JIL document extensions allows tools to identify and perform advanced operations when merging and combining data. The following section describes the details of how the current implementation of JIL handles the versioning of code elements and extensions.

4.3 Versioning

JIL documents are unambiguous descriptions of Java classes, allowing tools to directly construct a hierarchical structure of code elements. Document type definitions allow the format of these elements to be recognized and validated using existing XML parsers. DTDs can be referenced using a Uniform Resource Locator (URL), allowing JIL tools to provide a unique specification for the JIL they support online. Versioning of DTDs allows tools to identify documents based on different versions of JIL. Extensions are versioned independently of each other, allowing JIL documents to be formed from any combination of supported extensions.

Each JIL document contains a history of contributors. This history is a list of tools with attributes which uniquely identify a set of tags within the document. A JIL tool uses the document history to describe the operation or command it performed when generating the tags in the document. Version information is usually associated to a history element, which indirectly represents the output of a particular version of a tool. This allows code elements and extensions to be traced back to a specific tool, and then separated by a visualizer when parsing the document. By maintaining this versioned history within each JIL document, it allows tools to manage and separate both supported and unsupported extensions.

5. CONCLUSIONS

We have presented an open framework for developing visualization and software understanding interfaces. The key features of the framework allow existing and future tools to contribute both static and dynamic code elements to these visualizations, allowing interfaces to be developed based on the information the user wants to analyze rather than what information is immediately available. Collaboration and interoperability are facilitated by using an extensive and portable document format for persisting and separating data.

This framework has been applied to several tools in order to combine the data they provide into a single customizable visualization interface. With the addition of JIL support, these tools have benefited from the addition of a visualization backend and the ability to export and persist the code characteristics they can extract. This has improved both the usefulness and interoperability of these tools, with a low development and implementation cost.

5.1 Current Progress

Current visualization tools are based on the JIL specification 1.0 available online as a Document Type Definition at the following URL: http://www.sable.mcgill.ca/~flynn/jil/.

The JIMPLEX visualization framework is provided online as a package of client and server-side scripts to provide visualization interfaces supported by common web browsers. Current implementations allow the online validation and visualization of JIL documents produced by SOOT and STEP.

5.2 Future Work

As an open framework, there are many different areas for future work. The JIL specification itself is in its infancy, and although only basic extensions are included, the specification is designed to be extended based on the tools that support it. Current support is limited to SOOT and STEP, but any tool which can provide some insight into software understanding can extend the JIL definition and generate JIL data. Both static and dynamic extensions are easily specified by DTDs or another form of XML schema. When adding support to a tool for generating or modifying JIL documents with data extensions, supplying a DTD allows other tools to validate and recognize those extensions.

Let us consider a user who wants to inspect some generated code in relation to its benchmarking data. They would first decide what code elements would be associated to their benchmarking results, such as extending individual methods with timing information. These extensions would be defined in a DTD, and support for JIL would be added to their benchmarking suite using a popular XML API in their favorite programming language. The extension DTD could then be used by a visualization interface to validate JIL documents containing these dynamic extensions. The visualizer could present the user with statistical information based on the benchmarking data recorded in the JIL documents. Future improvements to the code generator could then be evaluated by comparing successive JIL documents containing the benchmarking extensions.

Visualization is the current focus of this framework, however it is only one example of an application for JIL. Future use of JIL could target any application where metadata is associated to code. A JIL-aware Integrated Development Environment (IDE) might remind the user about methods which are called frequently or suggest the most effective strategy to modularize the code. Software development rarely involves the inspection of static analyses beyond compiler errors, and dynamic information is typically not available until changes to the code may be too costly. Much effort is spent debugging software during development, and most developers target a single problem or area of the code when debugging. A debugger which supported JIL could preserve such information with the code allowing the developer to reference this data without having to execute another costly debugging run. By combining static and dynamic data into an extensible document format, tools can provide a developer with information and insight normally obscured by the code.

6. ACKNOWLEDGMENTS

Thanks.

7. REFERENCES

[1] T. Ball and S. G. Eick. *Software Visualization in the Large.* IEEE Computer, Vol. 29, No. 4, pp. 33-43, 1996.

[2] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. *Extensible Markup Language (XML) 1.0 (Second Edition).* http://www.w3.org/TR/REC-xml.

[3] R. Brown, K. Driesen, D. Eng, L. Hendren, J. Jorgensen, C. Verbrugge, and Q. Wang. *STOOP: The Sable Toolkit for Object-Oriented Profiling.* McGill University, Technical Report SABLE-2001-2, 2001.

[4] D. Eng. *JIL: an Extensible Intermediate Language.* McGill University, March 2002.

[5] J.C. Hardwick and J. Sipelstein. *Java as an Intermediate Language.* Carnegie Mellon University, Technical Report CMU-CS-96-161, 1996.

[6] C. Knight and M. Munro. *Visualising Software – A Key Research Area.* Proceedings of the IEEE International Conference on Software Maintenance, 1999.

[7] B. A. Price, I. S. Small, and R. M. Baeker. *A Taxonomy of Software Visualization.* Proceedings from the Hawaii International Conference on System Sciences, Vol. 2, pp. 597-606, 1992.

[8] T. Systa. *On the Relationships between Static and Dynamic Models in Reverse Engineering Java Software.* Proceedings of the Working Conference on Reverse Engineering, pp. 304-313, 1999.

[9] T. Systa. *Understanding the Behaviour of Java Programs.* Proceedings of the Working Conference on Reverse Engineering, pp. 35-44, 2000.

[10] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. *SOOT: a Java bytecode optimization framework.* Proceedings of CASCON '99, pp. 125-135, 1999.

[11] C. Verbrugge and L. Hendren. *Simplifying Java Bytecode for Compiler Analyses and Transformations.* McGill University, July 1997.