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, tuning, and debugging compiler optimizations is a difficult one which can be facilitated by software visualization. There are many characteristics of the code which must be considered when studying the kinds of optimizations which can be performed. Both static data collected at compile-time and dynamic runtime data can reveal opportunities for optimization and affect code transformations. Visualization of such complex systems must include as much information as possible, and accommodate the different sources from which this information is acquired.

This paper presents a visualization framework designed to address these issues. The framework is based on a new, extensible language called JIL which provides a common format for encapsulating intermediate representations and associating them with compile-time and runtime data. Custom visualization interfaces can then combine JIL data from separate tools, exposing both static and dynamic characteristics of the underlying code.

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 [7]. Visualization has been shown to improve efficiency and productivity, especially in complex systems which can span the work of several programmers over extended periods of time [1]. The object-oriented languages found in such systems can easily obscure the original solution they were used to implement. With such features as polymorphism and dynamic typing, it is important to consider both the compile-time and runtime characteristics of these languages during visualization [10].

The framework presented in this paper permits the visualization of intermediate representations of Java used by an optimizing compiler. In this environment, the scope of the visualization covers both static information about the software (Java code) and dynamic information about the execution of the program (Java Virtual Machine). This data can be extracted from different intermediate languages and other representations, as well as sources of runtime data.

Specifically, we present the following:

- A common intermediate language called JIL, capable of encapsulating Java intermediate representations and associating both static and dynamic data with individual code elements.
- Extensions to existing software tools which allow static characteristics and analysis results, as well as dynamic runtime data, to be exported as JIL documents.
- A new visualization implementation using JIL as a data source, allowing data from the multiple tools to be combined in a customizable interface.

2. VISUALIZATION FRAMEWORK

In Figure 1 we present a visualization framework which is designed to be customizable and scalable. The foundation of this system is the new Java Intermediate Language (JIL) which can encapsulate existing intermediate languages, including both static and dynamic program characteristics obscured within the compiled code and the execution platform (a). In the following sections we discuss the design of this language and present two tools (b) which provide suitable sources of both static and dynamic characteristics of the code (c). We then present an example visualization interface which combines these characteristics with minimal implementation by using JIL as a data source (d). Finally, we discuss some of the data management details and present our conclusions.

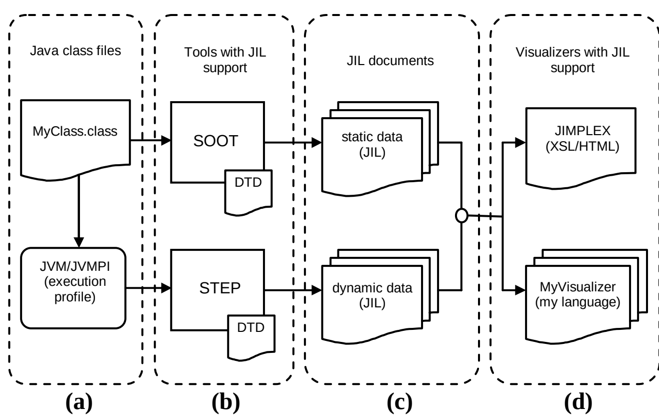

Figure 1. Overview of the visualization framework.

2.1 Intermediate Language Design

Intermediate languages are used by optimizing compilers to give the separate modules a representation to work with which is independent of the code being generated. This encourages modularity throughout the compiler, and allows the code generator 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 [6]. 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. These lower-level intermediate languages are typically designed for a single process or operation, creating variations which complicate the development of supporting tools and visualizers.

2.1.1 Java Intermediate Language

In order to combine the information which can be associated with these low-level languages, a common format was required which could act as metadata for code. We designed the Java Intermediate Language (JIL) for this purpose of encapsulating intermediate representations of Java source code. Although strictly defined, JIL remains extensible and able to support extensions which might not have been envisioned yet. 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 visualization interfaces and tools which use traditional intermediate languages. We only describe the relevant highlights of JIL in this document, but a complete specification is available separately as a technical report [5].

JIL is based on the Extensible Markup Language (XML) and 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 [2]. 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 (DTDs). DTDs are simple to extend, and provide a strict method of enforcing compatibility between JIL tools; applications which use this standard XML schema for validating their input can identify which extensions to expect from a JIL document.

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 lower-level intermediate languages, such as labels. This basic skeleton of code elements provides a framework upon which language extensions can be added. These extensions can include both static and dynamic characteristics of the code, and allow such information to be associated directly with each code element.

In order to demonstrate the kinds of extensions supported by JIL, we consider virtual method polymorphism throughout this paper 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 continue this example and describe two tools capable of expressing the results of such analyses as JIL extensions.

2.2 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, to which annotations are added. Most common code annotations are analysis results which can be calculated by inspecting bytecode or intermediate representations. These are static characteristics of the code which are known once the Java source is compiled.

2.2.1 Generating static data

The top portion of Figure 1b shows a code tool capable of performing static analyses on Java bytecode being passed a compiled class file. These results are then exported as extensions in the JIL documents it produces. These extensions are defined in an accompanying DTD, allowing tools to validate and identify the JIL documents being produced. By associating extensions with the tools that generate them, several tools can annotate the same code elements independently. Redundant or related extensions may also be compared or combined by tools which support them.

As part of the framework presented in this paper, we added support for the output of JIL to SOOT, an existing optimization framework which uses intermediate languages to perform static analysis and transformations on Java bytecode [11] [12] [13]. Most of these analyses are performed on Jimple, an intermediate language where some basic optimizations can be applied before generating bytecode [14]. For example, 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. With the addition of JIL as an output format and the associated DTD specifying the supported extensions, SOOT provides an ideal source of static data.

Continuing our running example of polymorphism as a code characteristic, SOOT supports several types of static analyses which can associate call sites with potential targets. 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 JIL describing a call site extended with the results of class hierarchy (CHA) [4] and variable type (VTA) analysis [8].

2.3 Dynamic Code Elements

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

2.3.1 Collecting dynamic data

The bottom portion of Figure 1b shows runtime data being collected by a profiling tool which supports JIL output. Annotating JIL with dynamic data works much like with static data. In order to manage both static and dynamic extensions, each JIL document contains a history of all the tools and operations which have authored it. This allows dynamic data from separate executions of the same tool to coexist in a single document. Tools which support JIL uniquely identify each execution with an entry in the document's history, typically including the profiling or benchmarking agent which was used, a timestamp, and information about the execution environment. This allows JIL tools to compare and process multiple runtime data sets.

In order to collect dynamic data within our framework we extended a tool called STEP, a customizable profiling framework for evaluating the performance and behavior of object-oriented applications [3]. It helps developers build custom profiling agents to collect data on the runtime behavior of programs; 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 provided a backend to this event pipe which generates 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 preserve 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 (CHA) indicated that this call site had three potential targets, according to the data collected from this execution run only two targets were invoked. Runtime data must include 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 is described in section 3.

2.4 Visualization of Static and Dynamic Data

We have demonstrated how JIL documents can combine static and dynamic data, and how existing tools can be extended to produce JIL documents as shown in Figure 1c. These documents provide versatile data sources which are easily merged and accessed, facilitating the creation of a variety of visualization interfaces.

2.4.1 JIMPLEX

Developers working with SOOT and Jimple required a tool which would allow them to develop and debug optimizations. Once we had extended SOOT and STEP to produce JIL documents, we developed a visualization implementation called JIMPLEX with the goal of providing a customizable visualization framework for browsing Jimple. 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

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)

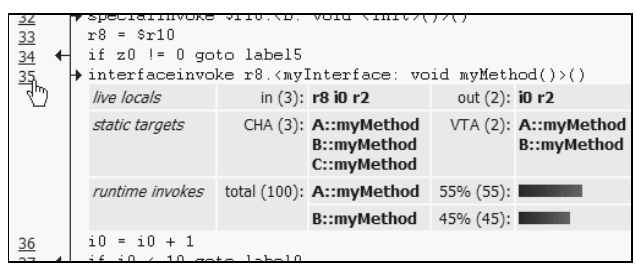

Figure 2. a) A simple polymorphic Java fragment with an interface invoke; b) A JIL fragment produced by SOOT indicating the possible targets of the call site; c) A JIL fragment produced by a STEP profiling agent indicating the actual number of runtime invokes; d) A slice from the JIMPLEX interface focused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

browsers, allowing the interface to be customized without having to recompile code or learn a complex API. These technologies also encourage collaboration, allowing both the data sources and the visualization interface to be shared between platforms and devices across the Internet.

By using the JIL documents produced by SOOT and STEP as data sources (seen in Figures 2b and 2c), JIMPLEX can browse Jimple annotated with data from both tools. 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.

Figure 3 is a screenshot of JIMPLEX visualizing a Java class in a web browser. Two JIL documents are used as data sources to dynamically generate the interface. Static information, such as the relative number of statements per method or label is derived from

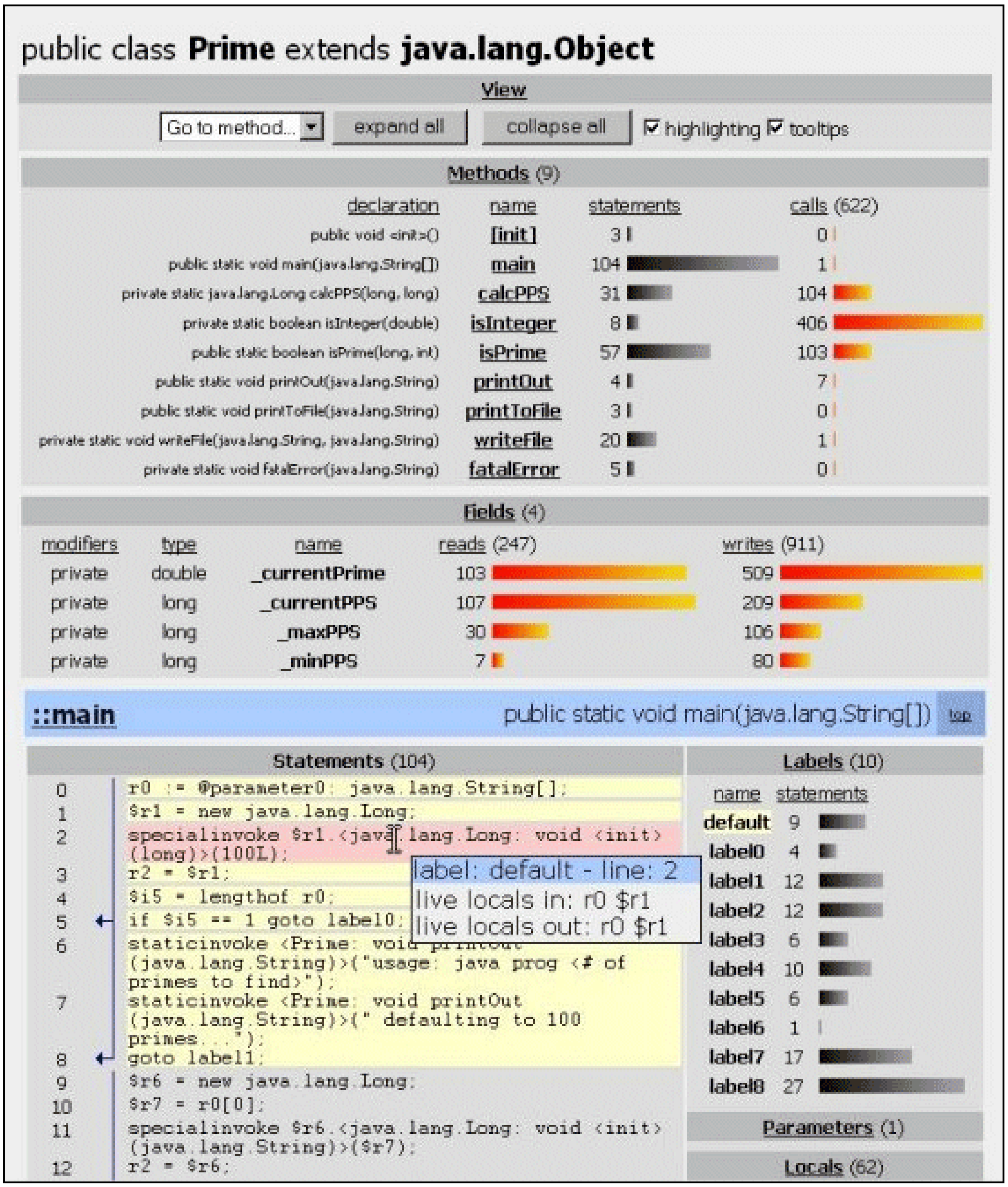

Figure 3. Screenshot of JIMPLEX browsing an intermediate representation of a Java class; the interface is generated dynamically from separate JIL data sources. Enumerations of the methods and fields of the class include runtime data, such as method invokes and field accesses. The statement listings for each method expose static analysis results to the user, such as variable liveness and flow information.

the JIL document produced by SOOT. The runtime data, such as the cumulative number of method invokes and field accesses, is extracted from another document produced by a STEP profiling agent. By exposing both kinds of data in the same interface the user can identify possible optimization targets, such as the relatively small method *isInteger*, which was executed the most frequently.

3. 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 discuss the non-trivial task of providing this kind of interoperability in more detail.

3.1 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 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.

3.2 Merging

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

The most straightforward merging involves JIL documents which represent different classes; in this case there is no merging required. Data in these documents does 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.

When a tool merges JIL documents which refer to the same class, it can advantage of the object-oriented document model. 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 presents 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 is 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 can 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. Visualizers can display averages and other calculations by identifying and processing this intersecting data. This process requires some basic algorithms used by the visualizer to decide how to combine the data and present the user with code characteristics of interest.

The visualization framework presented in this paper 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 into 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. Such 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 encourages visualizers to support the visualization of remote JIL documents, allowing collaboration between tools which might exist on different machines or networks.

3.3 Versioning

JIL documents are unambiguous descriptions of Java classes, allowing tools to 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 with 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.

4. 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 extensible and portable document format for persisting and separating data.

This framework has been applied to both a static analysis and profiling tool 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 preserve the code characteristics they can extract. The visualization interface presented in this paper has demonstrated the interoperability and customization that is possible when using JIL as a data source, as well as the usefulness of exposing both static and dynamic data to the user.

4.1 Current Progress

Current visualization tools are based on the JIL specification 1.0 available online as a Document Type Definition at: http://www.sable.mcgill.ca/jil. The JIMPLEX visualization framework is also available at this URL as a package of client and server-side scripts to provide visualization interfaces supported by common web browsers. The current release of SOOT, which supports JIL as an output format, is available at: http://www.sable.mcgill.ca/soot.

4.2 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 should first decide what code elements would be associated with their benchmarking results, such as extending individual methods with timing information. These extensions should be defined in a DTD, and support for JIL should be added to their benchmarking suite using a popular XML API in their favorite programming language. The extension DTD should 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 with 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.

5. ACKNOWLEDGMENTS

This work was supported by NSERC. Many thanks to Karel Driesen, Laurie Hendren, John Jorgensen, Bruno Dufour, and the Sable Research Group at McGill University for their help in improving the quality of this paper.

6. 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. Sable Technical Report 2001-2, McGill University, 2001.

[4] J. Dean, D. Grove, and C. Chambers. *Optimizations of object-oriented programs using static class hierarchy analysis.* Proceedings of ECOOP '95, pp. 77-101, August 1995.

[5] D. Eng. *JIL: an Extensible Intermediate Language.* . Sable Technical Report 2002-3, McGill University, March 2002.

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

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

[8] V. Sundaresan, L. Hendren, C. Razafimahera, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. *Practical virtual method call resolution for Java.* Proceedings of OOPSLA '00, pp. 264-280, October 2000.

[9] 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.

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

[11] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, V. Sundaresan. *Optimizing Java Bytecode using the Soot Framework: is it Feasible?* Proceedings of the International Conference on Compiler Construction, pp 18-34, March 2000.

[12] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. *Soot - a Java Optimization Framework.* Proceedings of CASCON '99, pp. 125-135, 1999.

[13] R. Vallée-Rai. *Soot: A Java Bytecode Optimization Framework.* Master's thesis, McGill University, July 2000.

[14] R. Vallée-Rai and L. Hendren. *Jimple: Simplifying Java Bytecode for Analyses and Transformations.* Sable Technical Report 1998-4, McGill University, July 1998.