COMBINING STATIC AND DYNAMIC DATA IN CODE VISUALIZATION

by

David Eng

School of Computer Science
McGill University, Montréal

August 2002

a thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science

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. These characteristics can include large amounts of data which would be difficult to inspect without the appropriate tools. Both static data collected at compile-time and dynamic runtime data can reveal opportunities for optimization and affect code transformations. In order to expose the behavior of such complex systems, visualizations should include as much information as possible and accommodate the different sources from which this information is acquired.

This thesis 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. We present new contributions which extend existing compiler and profiling frameworks, allowing them to export the intermediate languages, analysis results, and code metadata they collect as *JIL* documents. Visualization interfaces can then combine the *JIL* data from separate tools, exposing both static and dynamic characteristics of the underlying code. We present such an interface in the form of a new web-based visualizer, allowing *JIL* documents to be visualized online in a portable, customizable interface.

Résumé

La tâche de dvelopper, dentretenir et doptimiser les compilateurs est difficile mais peut être facilit par la visualisation des logiciels. Il existe plusieurs caractéristiques concernant les codes qui peuvent être considéré dans létude des différents genres doptimisation. Ces caractéristiques peuvent inclure de grandes quantités de données qui savèrent difficile inspecter sans les outils appropriés. Les données statistiques recueillies au moment de la compilation et les données dexécution peuvent nous indiquer des occasions doptimisation et ainsi influer sur la transformation du code apporter. Afin dexposer le comportement de systèmes si complexe, la visualisation devrait inclure autant dinformation que possible et, du même coup, accommoder les différentes sources doù cette information est acquise.

Cette thèse a pour but de présenter un cadre de visualisation conu pour aborder ces problèmes. Il est basé sur un nouveau langage qui savère être aussi un langage extensible. Ce langage est appelé *JIL* et fournit un fournit un format commun afin dencapsuler les représentations intermédiaires et les associer aux données au moment de la compilation et de lexécution. Nous présenterons les nouvelles contributions qui font en sorte de prolonger le compilateur existant et les cadres de profilage leurs permettant ainsi dexporter des langages intermédiaires, danalyser des résultats et de coder des méga-données sous forme de document *JIL*. Les interfaces de visualisation peuvent alors combiner les données *JIL* des autres outils exposant ainsi des caractéristiques statiques et dynamiques du code fondamental. Nous présenterons donc une telle interface sous forme dun nouveau visualisateur basé sur le web, permettant aux documents *JIL* dêtre visualiser en ligne laide dune interface portable et sur mesure.

Acknowledgments

The author would like to offer a special thanks to his supervisor, Laurie Hendren, for her guidance and insight throughout his years at McGill University, and especially during this research. Without her encouragement and support, this thesis would never have materialized.

The author would also like to thank the members of the Sable Research Group and the Adaptive Computation Laboratory at McGill University, especially Karel Driesen, Clark Verbrugge, Rhodes Brown, John Jorgensen, Qin Wang, Bruno Dufour, and Feng Qian. In addition to providing a friendly and collaborative research environment, their help and suggestions were invaluable in improving the quality of the papers which preluded the work presented in this thesis.

Contents

Appendices

List of Figures

List of Tables

Chapter 1
Introduction

Software visualization is a well-explored research area which has been applied to several aspects of computing. In most cases, visualization is used to encourage program understanding during the development or maintenance of software. Visualization has been shown to improve the productivity and effectiveness of programmers, especially when working with complex systems which can span the work of several people over extended periods of time [3, 22, 26].

Object-oriented programming has become a popular approach to creating such systems due to the powerful features of languages such as Java and C++. Unfortunately, object-oriented programs can easily obscure the original solution for which they were designed to implement; this has promoted the combination of static and dynamic data in the analysis and visualization of these languages. With features such as polymorphism and dynamic typing, it is important to consider both compile-time and runtime characteristics of modern object-oriented languages.

Static data is information which can be extracted from the program once it can be compiled. This includes the source code itself, any related intermediate languages, and the results of static analyses such as locations of variable uses and definitions or the potential invoke targets of call sites. Dynamic data is collected by profiling tools and includes characteristics of the runtime behavior of the code, such as the number of field accesses or the actual methods invoked at a virtual method call site.

Unfortunately, the tools which are able to perform analyses and extract static

information about program code typically associate this data with specific intermediate representations and data structures. These representations are optimized for a specific task and exist internally within the tools, making the data unavailable to developers who are working with the compilation of this code, or the original code itself. Many of these tools do not use the same intermediate languages and fail to provide a mechanism for exporting the valuable information collected in their analyses. Their inability to share data separates the functionality of the tools, and prevents developers from benefiting from their combined ability to expose a comprehensive model of the program behavior.

Tools which collect runtime data are subject to the same drawbacks. They are typically designed for a high-level evaluation of the software, which is inadequate for combining with low-level static information. In addition, the experimental nature of these tools and the algorithms used in code analyses make it difficult to achieve any kind of standardization. The data they collect is stored in a specific or proprietary format suited for a single visualization system.

By associating runtime data with low-level code elements and analyses from additional tools, it is possible to create a complete model of how the code behaves in relation to how it is compiled and executed.

1.1 Motivation

The work presented here addresses several different problems which have been encountered when studying the compilation and runtime behavior of Java as an object-oriented language:

- **Comprehensiveness**: How can we combine both static and dynamic information when visualizing software?
- **Generality**: Given the abundance of code and compiler tools, how can we combine their functionality and features when visualizing their results and related intermediate languages?

- **Extensibility**: How can we continue to visualize new and undiscovered code characteristics?

In order to solve each of these goals it was clear that we required a system which is highly extensible. Rather than propose an ideal visualization implementation, we present a framework which allows the creation of custom visualizers which use arbitrary tools as their data sources. Existing and future tools can be extended to export any static or dynamic data they can extract from the source code or related intermediate languages.

The framework presented in this thesis is targeted at the visualization of source code and intermediate languages used by an optimization framework for Java [21]. The scope of the visualization covers the code and structure of the software as well as its behavior when executed by the Java Virtual Machine (JVM) [30]. This data can be extracted from different intermediate languages and other representations, as well as sources of runtime data. In the following section we discuss the overall visualization framework, including all key contributions.

1.2 Framework overview

In Figure 1.1 we present an overview of the 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. *JIL* can be annotated with both static and dynamic program characteristics of the software which are abstracted by the source code. Figure 1.1 (a) shows Java class files where they exist as compiled bytecode or as programs being executed by a JVM.

In Figure 1.1 (b) we see two software tools which are capable of extracting these types of program characteristics. Our framework is designed to allow existing and future tools to be used for the analysis and inspection of Java bytecode at compile-time and runtime.

Any information extracted in (b) is then exported as *JIL* documents, shown in

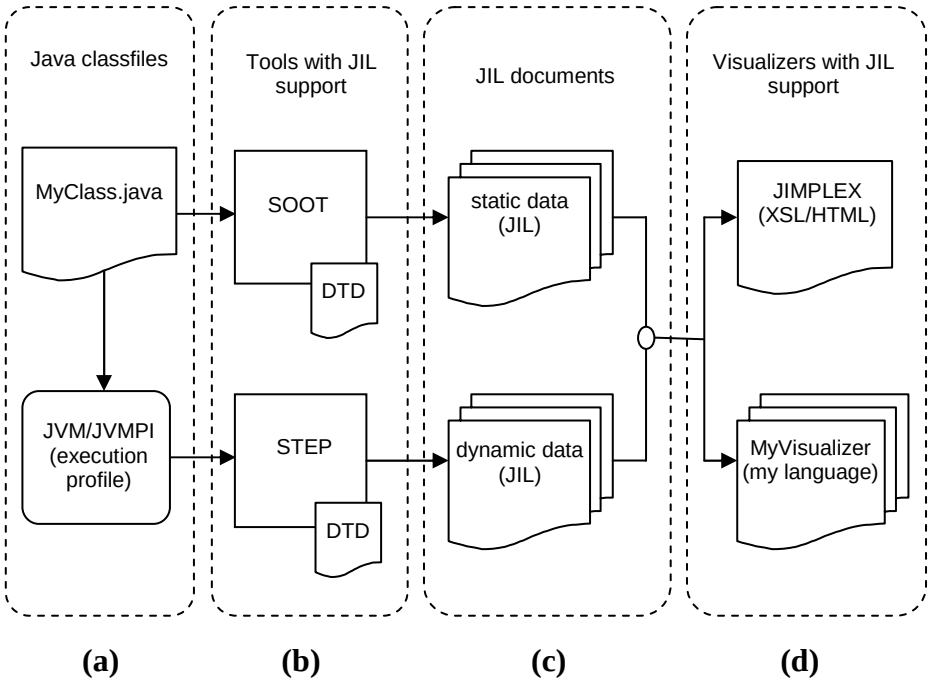

Figure 1.1: Overview of the visualization framework. (**a**) Java class files at compile-time and runtime. (**b**) Tools which are able to extract data from (a); the data provided by each tool is specified in an accompanying DTD. (**c**) *JIL* documents containing data extracted in (b). (**d**) Visualizers which use the *JIL* documents from (c) as sources of data.

Figure 1.1 (c). *JIL* documents can contain both static and dynamic data, and multiple documents can be used to describe a single class object. Documents can be merged in order to provide information collected by several tools in a common format.

By using these documents as a data source, the visualization interfaces shown in Figure 1.1 (d) can expose both static and dynamic characteristics to the user. We provide one interface which demonstrates the versatility of *JIL* as a data source, however our framework is designed to allow arbitrary tools to visualize the data contained within *JIL* documents.

The main contributions we present in this thesis are:

- A common intermediate language called *JIL*, capable of encapsulating Java intermediate representations and associating both static and dynamic data with individual code elements. *JIL* provides a portable and extensible format for exchanging data between arbitrary tools and visualizers.
- 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.

This thesis is structured as follows: Chapter 2 presents research related to our framework. Chapter 3 discusses the design and key features of *JIL*. Chapter 4 covers the creation of *JIL*, including descriptions of the tools used in our framework as *JIL* data sources. Chapter 5 describes the consumption and visualization of *JIL* documents, and presents the *JIMPLEX* interface as an example of the customization and ease of implementation possible when using *JIL* as a data source. Chapter 6 discusses possible future work that could benefit the framework, and the availability of the current tools and technologies used in the framework. Finally, we evaluate the framework and present conclusions in Chapter 7.

Chapter 2
Background and Related Work

The work presented here combines several aspects of research. Compiler development and program understanding are fundamental topics which have been studied for many years. Object-oriented programming has brought new challenges to these areas of research, and with them new solutions involving visualization and the combination of static and dynamic data. The following sections discuss some of the research in these areas which influenced our work.

2.1 Program understanding

As a requirement for software development and maintenance, program understanding and reverse engineering has become a popular research area. Many different approaches have been described for a variety of applications, such as the renovation of legacy systems or helping the cognitive development process [10,32,57]. When developers and documentation are either out-dated, unavailable, or unreliable, the source code often becomes the only reliable source of information about a program [42]. Our framework relies exclusively on the code as a source of information about the behavior of Java software, based on both compile-time and runtime data.

This requires the static analysis of Java source code and related intermediate languages [9,12,33]. Several compilers and optimization frameworks exist which use static analyses to optimize either Java source source or bytecodes, such as *FLEX* [41],

BLOAT [34], *Ajax* [35, 36], and *Briki* [11]. As a source of static data and analysis results, we chose to use the *SOOT* framework which includes an API for developing custom analyses and working with bytecode [51, 52, 54].

Runtime data has become more important in program understanding with the rise in popularity of systems which use virtual machines to execute platform independent code. Many profiling and benchmarking frameworks exist which can instrument Java bytecode in order to observe the runtime behavior of Java software such as *BIT* [29] and *ProfBuilder* [14]. In order to match the extensible nature of our framework, we used a profiling framework, called *STEP*, as a source of runtime data [5, 6]. *STEP* allows the development of custom profiling agents and allows a simple backend implementation to generate *JIL* documents using this data.

2.2 Language design

Our framework uses *JIL* in order represent both static and dynamic data suitable for program understanding. We chose to use a generic language to describe intermediate representations of Java and any associated metadata. This technique has been applied to similar applications, such as software re-engineering and language-independent compilation [16, 56]. In order to represent a system at a higher level of abstraction, a true language-independent system is required. Such systems use a generic language, such as *JIL*, to describe and annotate other unknown languages [55]. Generic language technology allows *JIL* to support both future languages and any associated data extracted from future tools.

2.3 Code and software visualization

Price et al. describes the scope of a visualization in terms of the class and scale of the programs it can visualize [39]. The goal of our framework is to allow developers to visualize the intermediate languages and associated metadata of arbitrary Java programs. In this thesis, we present a scalable framework which puts no limitations

on the size or complexity of the programs being visualized. Generality is also a key feature of the framework, which allows visualizations to include any source language.

Code visualization is a well-explored research area which has been shown to promote understanding and reduce the structural complexity of the software it represents [3]. Much of this effort has been spent creating an ideal representation of software for the visual and cognitive consumption by humans. These interfaces can range from basic code leads to complex graphical and multi-dimensional interfaces [26, 27]. Although such tools can efficiently display large amounts of data to the user, their design often depends on the data being visualized. Our framework uses *JIL* to separate the data from the interface and store it in an easily accessible format. This facilitates the creation of custom interfaces and encourages their implementation to be independent of the data and content.

2.4 Combining static and dynamic data

Object-oriented languages, such as Java, have introduced many new challenges to developers trying to understand the behavior of software. The dynamic nature of object-oriented programs has encouraged the combination of both compile-time and runtime data in code visualization [37, 46, 47, 49, 58]. Data extracted from object code at compile-time can describe static relations and structure obscured within the source code. However, the true behavior of the software involves events which occur at runtime such as object management, dynamic binding, and dynamic typing. These events are recorded at runtime and can reveal more information about the code.

Many existing software analysis tools include both static and dynamic data in their visualizations, such as *SCED* [28, 48] and *SoftArch* [22]. These tools incorporate dynamic data in order to model specific aspects of software architecture, such as state machine design and abstraction modelling, however they do not provide a sufficient solution for representing arbitrary characteristics of the source code. They allow varying degrees of abstraction in their visualizations, however they do not provide support for including new types of static and dynamic data, such as undiscovered

analysis results.

In the following chapter we describe a generic format, named Java Intermediate Language (*JIL*). It is used for storing static data, such as an intermediate representation of Java, as well as dynamic data, such as the runtime behavior of the code. The extensible representation of the data contained within *JIL* documents allows both static and dynamic code characteristics to be shared between tools and visualization implementations.

Chapter 3
Source and Representation

Now that we have presented an overview of the framework, we can examine the design of *JIL* and how it provides an extensible and language-independent platform for visualization.

3.1 Java Intermediate Language

The framework we present is based on a generic, language-independent representation called the Java Intermediate Language (*JIL*) [18,19]. This format is designed to allow arbitrary code tools to share data with custom visualizers. The key features of the format are its language independence, extensibility, and portability. The following sections discuss the features and design goals of the language in more detail.

3.1.1 Intermediate language design

Intermediate languages are used by optimizing compilers to give the separate modules a representation to work with which is independent of 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 [23]. However, when developing optimizations, transformations and analyses typically operate on a lower-level language, 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. The grammar and syntax of these intermediate representations are all optimized for their original purpose, which is typically a single process or operation. The differences between each representation complicate the development of tools and visualizers that support multiple languages. In order to provide a format which can support future Java intermediate languages without any knowledge of their grammar or syntax, we created *JIL* as a *metalanguage*.

3.1.2 JIL as a metalanguage

JIL documents are designed to store intermediate languages as *data*, and any hidden code characteristics and analysis results as *metadata*. *JIL* documents use arbitrary intermediate languages to describe the specification and structure of a Java class, and then describe any hidden relations and behavior of the code using annotations. Its design and use differ from traditional intermediate languages in that its content is independent of the tools that use it.

This is to say that tools which consume *JIL* as input may verify that a specific intermediate language is contained within a document, however there is no requirement placed on which languages a *JIL* document must contain. This encourages *JIL* consumers, which are typically visualizers, to support arbitrary intermediate languages in their interfaces, improving their usefulness and longevity. Similarly, tools which create *JIL* documents as output are able to store any intermediate languages they might deal with now or in the future. *JIL* treats each intermediate language it contains independently, allowing the relationships between code elements to be defined by the author of the *JIL* document. This generic strategy also applies to the metadata which describes the hidden relations between code elements in the form of analysis results and language extensions. Tools can programmatically support whatever extensions they wish to take as input or export as output, including both static and dynamic

types of data.

It is important to note that *JIL* is not designed to be manipulated or decompiled into bytecode, or replace the intermediate languages it contains. It merely provides a bridge between tools and visualization interfaces. These tools use traditional intermediate languages, and the visualization interfaces are designed to present these languages and related data to developers.

3.1.3 JIL as an XML application

In order to provide a metalanguage that can address our goal of interoperability between tools, we based *JIL* upon the Extensible Markup Language (XML) [4], a subset of the Standard Generalized Markup Language (SGML) [25]. SGML is an international standard for defining a language, and XML allows generic SGML to be used on the Web much like common Hypertext Markup Language (HTML) [40]. *JIL* benefits from many of the features of these well established formats. This includes a growing number of tools and APIs as well as support 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 that use this standard schema for validating their input can identify which extensions are supported as well as their version information.

Some key features of *JIL* are a direct result of using XML as the defining metalanguage:

- **Accessible**: *JIL* is human readable and editable using a common text editor, which aides in debugging.
- **Extensible**: *JIL* is strictly defined using DTDs, which are themselves easily extensible; documents and their specifications never require compilation.
- **Robust**: *JIL* is easy to generate and parse, encouraging the development of tools with good reliability and performance.

- **Modular**: *JIL* is modular and manageable through schemas and basic processing; documents can be separated and combined as required without breaking structure or compatibility.
- **Portable**: *JIL* is portable across languages, platforms, and networks.
- **Compatible**: *JIL* is compatible with XML, SGML, and related tools and APIs.

Every *JIL* document is a valid SGML and XML document, which means *JIL* is compatible with existing and future SGML and XML tools. This includes software for almost any task, including processing and authoring, and since these formats are license-free, they are widely supported:

- **Client-side**: *JIL* can be browsed using Internet Explorer, Mozilla, Opera, and other popular browsers.
- **Server-side**: *JIL* can be imported, exported, and natively served in popular databases such as Microsoft SQL Server 2000 and Oracle 8i.
- **Support**: Programming APIs are available in many different programming languages such as C, C++, Java, Perl, Python, etc.

XML also has some disadvantages which it passes on to *JIL*. For example, *JIL* is extremely verbose, and a *JIL* document is typically much larger than the corresponding source code it represents. However, the decreasing cost of disk space and the progress of compression algorithms for both storage and network transfer are working towards minimizing this disadvantage. XML is not always the best choice for an application, but in the case of *JIL* its features address many of the design goals.

3.1.4 JIL specification

The formal specification of *JIL* is available as a DTD. DTDs are a standard schema for defining constraints on SGML documents. They define the order and properties of required and optional elements using a grammar similar to Extended Backus-Naur Form (EBNF). DTDs are a standard format used by XML validation tools and

APIs in order to verify that the content of a document complies to the grammar they specify. The validation process can also identify the inconsistencies where a *JIL* document does not comply to the DTD specification. These inconsistencies can often be repaired or ignored, or as a worst case cause the document to be rejected completely. There are several features of DTDs which make them a natural choice in our framework:

- DTDs are text-based and do not require compilation, making them easily extensible when language constructs need to be added, updated, or removed.
- DTDs can reference other DTDs, making them modular.
- DTDs can be referenced using a URL, allowing them to exist on any device on a network and to be shared.

By using DTDs, applications can ensure that they are generating, parsing, or validating documents that will be recognized and understood by other applications. The DTDs used in our framework are available in appendix A.1 and online [20]; however, the following section uses *JIL* examples to describe them.

3.2 Modelling Java

We have introduced an XML-based format for modelling Java classes using intermediate languages. The following sections present the origins of these intermediate languages and associated metadata, and show how they can be encapsulated within our document format.

3.2.1 The life of a Java class

In order to understand the kinds of data associated with intermediate languages and code elements we examine the life of a Java class.

Given a Java source file, the Java compiler creates a Java class file consisting of bytecode instructions; see Figure 3.1 (a). Once compiled into bytecode, the source

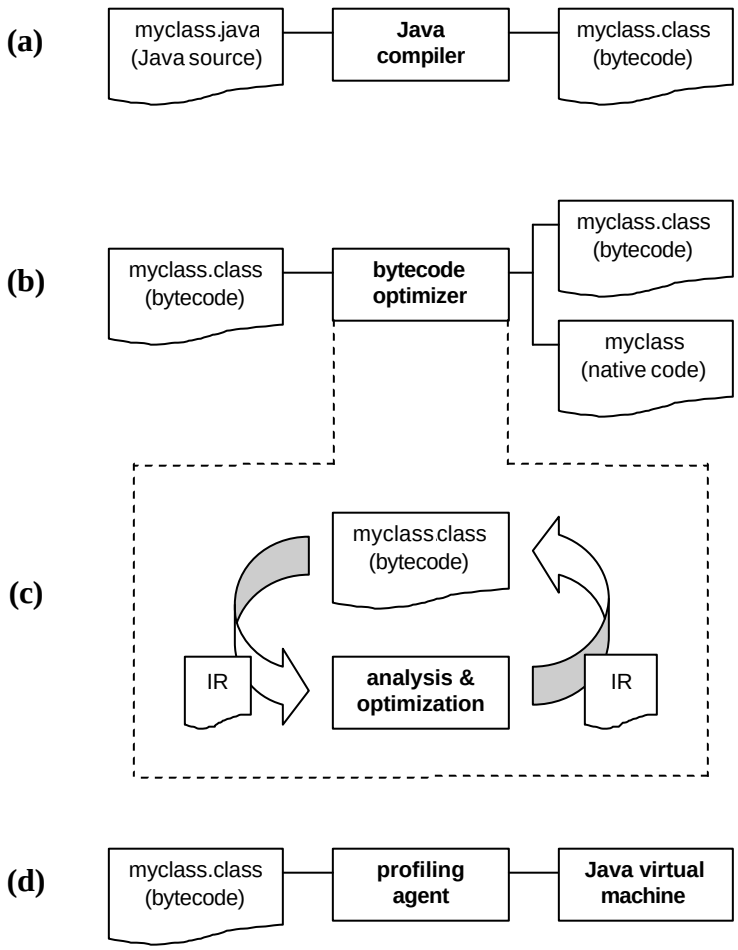

Figure 3.1: The life of a Java class. (**a**) Compilation of Java source code into bytecode; (**b**) Static optimization and analysis of bytecode; (**c**) Intermediate representations (IR) used in (b); (**d**) Runtime profiling of a Java class.

has taken a platform independent form which can be executed by a Java Virtual Machine (JVM) [30]. Java bytecode is itself an intermediate language, as it is the final representation of the source code before it is passed to a JVM and executed.

Optimizing compilers often operate directly on Java bytecode, using class files as both input and output; see Figure 3.1 (b). By targeting expensive bytecode instructions, such as virtual method calls and object allocations, these compilers can create optimized class files without affecting the functionality and original solution provided by the Java class. This kind of implementation provides versatile optimizers which create optimized bytecode executable on any JVM implementation.

Intermediate languages are used within optimizing compilers in order to provide a representation suitable for a specific process or operation; see Figure 3.1 (c). Java bytecode is designed to be executed by a JVM, but is not necessarily the ideal representation for optimization. Bytecode is stack based, where operations can be spread across several lines of code, and it also relies on additional data structures, such as the constant pool, making it difficult to manipulate and transform. Multiple intermediate representations are often used within a compiler at different stages throughout the optimization process. Each of these representations is designed with a specific motivation in order to facilitate a particular optimization process. Our visualization framework uses *JIL* to associate these intermediate representations with static characteristics and analysis results extracted by optimization frameworks.

The life of a Java class does not end after static optimization and analysis; see Figure 3.1 (d). By using a profiling agent such as *JVMPI* [43], an instrumented virtual machine based on an open source VM such as *Kaffe* [50], or by instrumenting the bytecode directly, dynamic information can be extracted which describes the runtime behavior of the code. We discuss the dynamic characteristics of code supported by *JIL* in Section 4.3.

3.2.2 Document structure

The content of XML documents is stored in a logical structure which describes a hierarchy of elements. This structure supports single inheritance by simply nesting

markup tags, allowing an element to have a parent, any number of siblings, and any number of children. This makes XML (and therefore *JIL*) a suitable foundation for modelling Java objects and constructs. By nesting elements according to an abstract tree representation of the class, they can be annotated with extensions while preserving the underlying structure. This allows backwards compatibility with *JIL* consumers which are unaware of the extensions or how to interpret them. Any unknown extensions can be ignored or handled separately.

Each *JIL* document describes a single Java class. In order to describe inner classes or an entire package of Java classes, multiple *JIL* documents must be used. There is no current specification for the packaging of multiple *JIL* documents. Their only association is the Java class or package they represent. Each document contains a required set of base elements which describes the underlying structure or skeleton of a Java class. These elements include enumerations of the fields and methods of the class, and provide a foundation upon which language extensions can be applied. These extensions allow any number of tools to annotate base elements with both static and dynamic information. This information can come in any form, such as analysis results or metadata, exposing characteristics of the code elements which would normally be hidden.

The logical structure of a *JIL* document is shown in Figure 3.2. The logical structure of the document is represented as a tree, where the directed arrows point towards child elements. Note that there also exists a virtual structure which includes associations between unrelated elements. For example, the use of a parameter is stored as a line number which acts as an index to the enumeration of statements in the same method. These kinds of abstract relations are defined by the authors of the corresponding elements, and they help to reduce the existence of redundant data in the document.

Markup

JIL is designed to provide a scalable framework where an arbitrary number of documents can be merged and processed with good performance. Attributes are used

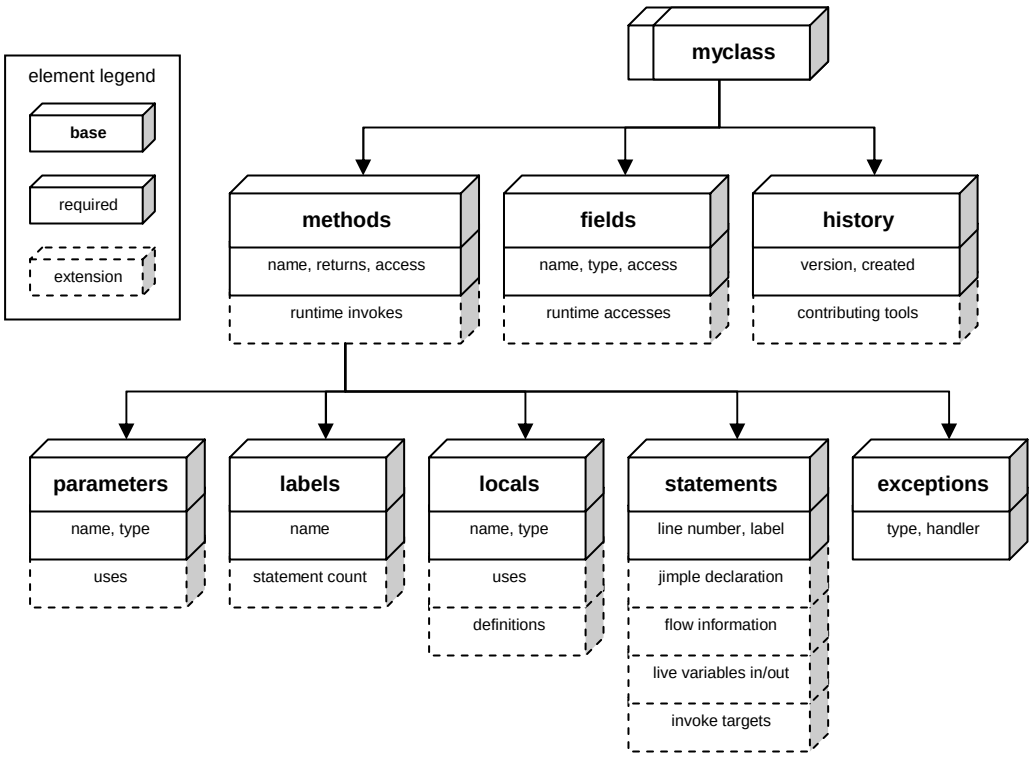

Figure 3.2: *JIL* document structure. The single inheritance hierarchy of XML is ideal for modelling Java constructs. Base elements provide a structure upon which intermediate languages and metadata can be added. Required elements contain basic information about base elements which is necessary for identification. Language extensions can be associated with any element and provided by arbitrary tools.

where possible to annotate and describe objects, since they typically perform better than enclosing the data between tags when processed by XML parsers. The properties of a programming element, such as the name of a field or the type of a local, are stored within the attributes of a tag. Attributes can also be weakly typed using a DTD, limiting them to a set of keywords or a name token. The following is an example of an element representing a `local` variable which uses attributes to store the name and type information:

```
<local name="MyDouble" type="double" />
```

The following is an example of what the above code element would look like if attributes were not used in the markup:

```
<local>
  <name>MyDouble</name>
  <type>double</type>
</local>
```

By not using attributes, the markup elements become extremely verbose and do not provide any benefit to an XML parser or another form of interpreter. Therefore, attributes are used where possible in order to simplify and minimize the number of characters required for the XML representation of code elements. However, there are cases where code elements are represented by this kind of markup. Data is enclosed between tags when it contains special characters or requires enumeration. Also, if there might be more than one property of the same name then this style of markup is used. The following is an example of markup representing a statement of *Jimple*, an intermediate language introduced in section 4.2.2. The statement contains some special characters which are not allowed to be stored as attributes in XML:

```
<jimple>
  <![CDATA[ $r0 = $r1 + $r2; ]]>
</jimple>
```

Enumerations

Enumerations are used widely in *JIL* to group and give order to lists of programming elements. An optional attribute `count` can be used to mark the number of nodes to expect in the enumeration. A *JIL* consumer can use this number to decide if, when, and how to process the nested nodes. Elements within an enumeration require unique identifiers, indicated by the attribute `id`.

```
<modifiers count="2">
  <modifier id="0" name="public" />
  <modifier id="1" name="abstract" />
</modifiers>
```

Note that these attributes are omitted from some of the other examples in this thesis in order to save space and highlight the other markup being demonstrated.

Language extensions

JIL language extensions include any data which is not explicitly expressed by an intermediate language. This type of information is typically extracted directly from the Java source, a related intermediate language, or Java bytecode. Extensions can also include runtime data which is collected during the execution of the program. There are no constraints on what kind of data can be represented as language extensions, however they must be defined using a DTD, just like the base *JIL* elements. Extension DTDs are typically referenced by the base *JIL* DTD and are associated to a specific tool. Their modularity allows them to be easily swapped in and out depending on the information required or the tools being used.

The following fragment follows the example from Section 3.2.2, adding a static extension which indicates which variables are live coming in and out of the statement:

```
<statement>
  <soot_statement>
    <jimple><![CDATA[ $r0 = $r1 + $r2; ]]></jimple>
```

```
    <livevariables>
      <in>$r1</in>
      <in>$r2</in>
      <out>$r0</out>
    </livevariables>
  </soot_statement>
</statement>
```

Extension elements are typically named by taking the extended element's name preceded by the extending generator and an underscore. In the above example, the element being extended is a `statement` and the extending generator is `soot`, therefore the resulting extension element is labelled as `soot_statement`. Generators can extend any element defined in the base DTD, including attributes of existing elements. The extension DTDs for the tools used in our framework are available in appendices A.2 and A.3.

3.3 JIL document elements

Each *JIL* document begins with some header tags for XML compliance and self-description, and can only contain those elements defined in the DTDs being used for validation. An example *JIL* document containing the minimum number of elements that are required according to the *JIL* 1.0 DTD can be found in appendix B.2.

Please note that some of the examples in this section do not reflect the complete syntax but merely highlight the particular elements or attributes being demonstrated; ellipsis have been included where syntax has been omitted which is not required for understanding.

3.3.1 Naming

There is no requirement placed on the naming of *JIL* documents, however they are typically associated with a single Java class. The relation between a *JIL* document and the source object is represented internally by the class name, allowing multiple *JIL* documents to refer to the same class file. If multiple *JIL* documents refer to

different versions of the same class file they can be distinguished by their document history element, described in section 3.3.3.

JIL documents should typically use an `.xml` file extension in order to be identified by XML and SGML tools, however the `.jil` file extension is also used when this kind of compliance is not required. For example, when serving *JIL* documents online, those documents with an `.xml` extension will be recognized by client and server-side applications as XML documents.

3.3.2 Headers

JIL documents are textual, but contain header information in order to self-describe the content within. Header tags come at the beginning of the document and exist at the root level. They uniquely identify a *JIL* document, while associating it with any related documents. Separate *JIL* documents might refer to the same Java source code, while containing different types or versions of annotated data. These annotations must be recognized in order to be accurately parsed and understood.

The following sections describe those elements included in a *JIL* document which do not directly represent a characteristic of source code or an intermediate language.

XML declaration

Since every *JIL* document is a valid XML document, it must begin with appropriate XML declaration tag [4].

```
<?xml version"1.0" ?>
```

JIL declaration

JIL documents include a header tag at the root level in order to indicate the version of the *JIL* contained within. The version information indicates to a consumer which version of *JIL* it must be prepared to parse. This version corresponds to the version of the validating DTD.

```
<jil version"1.0" />
```

3.3.3 Document history

JIL documents are associated with a single class, but they may be created from multiple sources throughout their lifetime. One *JIL* generator might create a *JIL* document while another might extend the document with additional code characteristics of which the original generator had no understanding.

The history element indicates which applications were used to create the *JIL* document. It's an enumeration of identity elements which self-describe a generator and the action it took when contributing to the *JIL* document. Typical information found in an identity node would include a time-stamp of when the operation was performed and the command line which triggered it.

```
<history>
  <soot version="1.2.3" cmd="-X MyClass" />
  <step version="1.0" mode="field-accesses" />
</history>
```

3.4 JIL class elements

JIL documents contain a single class tag at the root level. All source code characteristics are represented with *JIL* tags contained within the class tag. Nested classes are not supported, and should be handled using separate *JIL* documents.

The class name is stored in the `name` attribute. If this class has a parent in the class hierarchy, it can be indicated in the `extends` attribute. Currently *JIL* mimics Java and supports only single inheritance.

```
<class name="MyClass" extends="MyParent" />
```

Class modifiers

Class modifiers indicate the accessibility or hierarchical attributes of the class. *JIL* supports any number of modifiers, but only keywords which are used as modifiers in Java. Note that some other *JIL* elements also use the `modifiers` tag.

```
<modifiers>
  <modifier name="public" />
  <modifier name="final" />
</modifiers>
```

Typical class modifiers include `public`, `final`, and `abstract`. For a complete list of accepted modifiers see the base *JIL* DTD in appendix A.1.

Interfaces

If the class implements one or more interfaces this is indicated using the interfaces enumeration.

```
<interfaces>
  <interface name="my.package.interface" />
</interfaces>
```

Extensions

Class extensions are defined using the standard notation.

```
<class>
  <generator_class>
    ...
  </generator_class>
</class>
```

Java attributes are planned to be included as a class extension.

3.4.1 Fields

Member variables which are global to the entire class are contained within the `fields` tag. Each field is enumerated and assigned a unique identifier, a name, and a type.

```
<fields>
  <field name="MyDouble" type="double" />
  <field name="MyLong" type="long" />
</field>
```

Field modifiers

Each field can indicate its accessibility and behavior by including a modifiers tag within the field tag.

```
<field ...>
  ...
  <modifiers>
    <modifier name="private" />
    <modifier name="static" />
  </modifiers>
</field>
```

Typical field modifiers include `public`, `private`, `protected`, `static`, `final`, `transient`, and `volatile`. A complete list of accepted modifiers is specified in the base *JIL* DTD found in appendix A.1.

Extensions

Field extensions are applied to each field and allow tools to associate metadata to each field.

```
<field ...>
  ...
```

```
  <generator_field>
    ...
  <generator_field>
</field>
```

3.4.2 Methods

Methods are enumerated within the `methods` tag. Each `method` tag indicates the method's name and return type.

```
<methods>
  <method name="main" returntype="void" />
</methods>
```

Method modifiers

Method accessibility and behavior is described using an enumeration of modifiers. Usage is similar to the class and field modifiers.

```
<method ...>
  ...
  <modifiers>
    <modifier name="native" />
    <modifier name="synchronized" />
  </modifiers>
</method>
```

Parameters

Parameters are enumerated within the `parameters` tag for each method.

```
<parameters>
  <parameter name="MyString" type="String" />
  <parameter name="MyDouble" type="double" />
</parameters>
```

Extensions

Method extensions use the standard notation, but they can also exist for child nodes as well.

```
<method>
  <generator_method>
    ...
  </generator_method>
</method>
```

One tool used in our framework supports parameter extensions which indicate the statements where the associated parameter was used or defined. This static data is associated to another element through the statement line numbers, creating an abstract relation known only to the generators and consumers supporting this extension.

```
<parameter...>
  ...
  <soot_parameter uses="1" defines="1">
    <definition line="1" />
    <use line="2" />
  </soot_parameter>
</parameter>
```

3.4.3 Locals

Variables which are local to each method are represented by a locals enumeration tag, which is a child of each method tag.

```
<locals>
  <local name="MyLocal" />
</locals>
```

Locals by type

Local variables are also stored by type. This is a grouping which could be computed by a *JIL* consumer, but by storing this basic grouping within the *JIL* it can simplify the implementation of a consumer.

```
<types>
  <type name="MyType">
    <local name="MyLocal" />
  </type>
</types>
```

Extensions

Extensions to locals are stored using the standard notation.

```
<local ...>
  ...
  <generator_local>
    ...
  </generator_local>
</local>
```

The tool used as a source of static data in our framework generates *JIL* documents which contain local extensions indicating the statement where each local was used or defined, much like it does for fields.

```
<local ...>
  ...
  <soot_local>
    <definition line="1" />
    <use line="2" />
  </soot_local>
</local>
```

3.4.4 Labels

Labels are used in Java bytecode to indicate basic blocks of code which can be used as targets for branch operations. *JIL* contains an enumeration of a method's labels, and each statement is associated with a label.

```
<labels>
  <label name="MyLabel" />
</labels>
```

3.4.5 Statements

Statements represent the actual lines of code stored in an intermediate language.

```
<statements>
  <statement label="Mylabel" />
</statements>
```

Bytecode statements would include an operation and any associated parameters. For other intermediate languages, statements can range in complexity and might contain special characters. The natural representation of a statement is kept in its own tag as content.

```
<statement label="MyLabel">
  <jimple>
    <![CDATA[ $r0 = $r1 + $r2; ]]>
  </jimple>
</statement>
```

Extensions

Statement extensions associate data to each individual statement of an intermediate language.

```
<statement ...>
  ...
  <generator_statement>
    ...
  </generator_statement>
</statement>
```

In our framework we extend each statement with annotations, some of which relate to other elements such as fields or locals. Analysis results which apply to each statement are also stored as statement extensions, such as which variables are live coming in and out of a given statement.

```
<statement ...>
  ...
  <soot_statement>
    <livevariables incount="1" outcount="1">
      <in local="MyLocal" />
      <out local="MyLocal" />
    </livevariables>
  </soot_statement>
</statement>
```

3.4.6 Exceptions

Exceptions are also represented in *JIL* as an enumeration contained within each method. Exceptions reference three labels which indicate where the specified exception catching begins, ends and which handler represents the location of the exception handler.

```
<exceptions>
  <exception type="MyException">
    <begin label="MyBeginLabel" />
    <end label="MyEndLabel" />
    <handler label="MyHandlerLabel" />
  </exception>
</exceptions>
```

3.5 Document structure and management

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.

3.5.1 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 take 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* [13] and *PHP* [38]. 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.

3.5.2 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.

Versioning is also used within *JIL* content to describe the source and operation

which resulted in the *JIL* being interpreted. Multiple versions of extensions can describe the same characteristics using different data. Code transformations and optimizations can be versioned in order to compare the resulting code and any associated data visually. Dynamic code extensions are typically associated with an execution run and a runtime environment. The separation of these data streams is handled by versioning the appropriate 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.

Chapter 4
Creation and construction of JIL

We have introduced *JIL* as an extensible metalanguage for encapsulating intermediate languages and associated data. This chapter discusses the creation of *JIL* documents and the construction of the data contained within. We present two existing code tools which can extract various kinds of information which describe the behavior of Java programs. These tools have been extended to support the creation of *JIL* and serve as sources of static and dynamic data.

4.1 Creation of JIL

JIL requires no special encoding and can be created by hand using a common text editor. This facilitates debugging *JIL* documents and prototyping new elements and extensions. This allows the creation of *JIL* documents to be implemented using standard libraries without any proprietary code. Applications which generate *JIL* documents can also do so by using one of the many APIs available for most major programming languages. These APIs provide a quick and efficient way to generate compliant *JIL* without having to worry about the implementation details of an additional output format.

4.1.1 XML processing models

The implementation of a stand alone *JIL* generator or an extension to an existing system depends largely on the application. Two major processing models exist for XML-based documents: *DOM* [2] and *SAX* [31].

The *Document Object Model* (*DOM*) builds an internal image of the document in memory, and allows the non-linear construction of documents. *DOM*-based parsing can require large amounts of memory, however by modelling the entire document tree, programmers are given more freedom when adding, removing, or modifying elements.

The *Simple API for XML* (*SAX*) is an event based model which is designed to use as little memory as possible by not building an internal tree representation of the document. Instead, *SAX* traverses the tree and triggers events based on the elements and content it encounters.

These processing models can be combined to provide the ideal model based on the application. For example, when building a document during a computationally expensive analysis, a developer would chose to build a *DOM*-based document tree and add elements and attributes as the computation proceeds. However, if a developer is working with a profiling tool where they expect the document to comprise of an unpredictable amount of data, they would chose to build the tree on the fly using a *SAX* model for processing.

Many packages exist which allow the developer to use both these kinds of processing models. The recent emergence of web-based applications and platforms has increased both public and commercial interest in XML related technologies and APIs. Some popular programming frameworks for XML are *JDOM* [24], Apache's *XERCES* [1], and Sun's *JAXP* [44] and related toolsets.

Generated documents should be validated using a DTD or another schema for defining a document grammar. This ensures that the documents contain all required elements, and helps tools to identify any unsupported elements or attributes. DTD validation helps debug *JIL* generation, and is also supported programmatically in most XML APIs.

4.2 Static data

There are many different kinds of static data that can be associated with code elements. Some are very simple, such as the line number where a code element is stored in a source file. Other, less obvious characteristics are computed by program analyses, like those found in optimizing compilers. For example, such an analysis can determine the locations where a certain variable is defined and used. The framework we present allows all of these kinds of static data to be associated to code elements by using *JIL*.

4.2.1 Static data in JIL

Every *JIL* document must contain some basic static data. The framework of base elements which describes a class is static, since it is based on the unchanging bytecode representation of a Java class. For example, the methods and fields of a class are aspects of the program which are known once it is compiled. Once a Java program is compiled into bytecode, the only way to affect its structure or properties is to either directly modify the class file, or change the source code and recompile. Since *JIL* is meant to provide different representations of Java bytecode, its behavior is similar to class files; as long as a class file is unchanged, its *JIL* representation is valid.

Beyond the basic layout and structure of a Java class file, there are other static characteristics of the code which can be discovered once it's compiled. This type of static information is not contained within the bytecode directly, but can be computed through analysis.

Figure 3.1 (b) shows the static optimization of a Java class file. This is where the static information about a class can be extracted and analyzed by a variety of tools. Thus far we have only assumed that such tools exist, however our framework required such a tool in order to provide a source of static data. The following section describes an existing tool, called *SOOT*, which is capable of performing the types of analyses on Java classes, which expose opportunities for optimization [51, 52, 54]. We extended this tool to be able to convert Java classes into *JIL* documents, which

include the results of these analyses, making it the primary source of static data in our framework.

4.2.2 SOOT bytecode optimization framework

SOOT is an optimization framework for Java which uses intermediate languages to perform static analysis and transformations on Java bytecode [51, 52, 54]. *SOOT* was a natural choice as an extensible source of static data, providing its own API for developing code optimizations and transformations. Some of the key features of *SOOT* which influenced the decision to use it in our framework are as follows:

- **Extensibility**: *SOOT* is used as an experimental framework for working with Java bytecode, and provides a perfect testbed for new optimization and analysis techniques and algorithms; the extensible nature of the *JIL* document format provides a suitable way for exporting and visualizing current and future data computed by *SOOT*.

- **Programmability**: *SOOT* provides its own API for developing code optimizations and transformations, allowing programmatic access to its internal representations of Java bytecode and the analyses it is capable of performing.

- **Availability**: *SOOT* is an ongoing research project of the Sable Research Group at McGill University, where this framework was developed as well; this meant that we had access to some of the original developers and other resources which facilitated the addition of support for *JIL* as an output format.

Internally, *SOOT* applies code transformations and analyses using three different intermediate representations of Java bytecode. *Baf* is an abstract version of bytecode, *Jimple* is a three-address code where the stack is replaced by local variables, and *Grimp* is an aggregated three-address code suitable for generating stack code. These intermediate languages are used as alternative representations for code analysis and manipulation operations, which are difficult to apply to stack-based bytecode.

Jimple is a typed three-address code suitable for applying transformations which might move or remove code elements [53]. Code transformations such as algebraic manipulation, common sub-expression elimination, and constant propagation can be performed directly on *Jimple* code, making it an ideal representation for visualization. In order to translate stack-based bytecode to *Jimple*, an algorithm is used which converts stack references to temporary variables which can be explicitly used in three-address code. This can result in verbose code, however this conversion concentrates on correctness since *Jimple* can be further simplified by applying basic optimizations such as constant and copy propagation. *Jimple* is the target of visualization in our framework, since it exposes many optimizations to a developer which would not be apparent or possible with another representation.

4.2.3 SOOT as a source of static program data

The *SOOT* optimization framework is capable of decompiling Java classes into *Jimple* and other intermediate languages, as illustrated in Figure 4.1. As a contribution of this thesis, support for *JIL* was added to *SOOT* version 1.2.3 as an optional XML output format. *SOOT* takes a Java class as input and creates a *JIL* document compliant with the base and *SOOT* DTDs, found in appendices A.1 and A.2. This makes the current version of *SOOT* the first bytecode-to-*JIL* converter.

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

In order to demonstrate how *SOOT* exports analyses as *JIL*, we consider virtual method polymorphism as a characteristic of the code. Figure 4.2 (a) shows a fragment of Java code where polymorphism can be explored at compile-time by analyzing the potential targets of call sites. *SOOT* supports several types of static analyses which can associate potential targets with call sites. Call sites within the

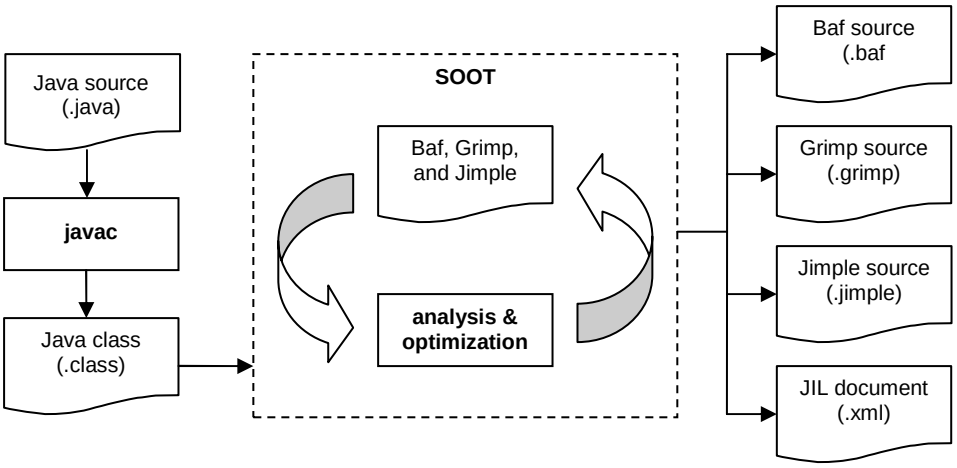

Figure 4.1: The internal use and generation of intermediate languages in *SOOT*. *JIL* was added to version 1.2.3 of *SOOT* as an additional output format.

JIL documents produced by *SOOT* can be easily identified as monomorphic or potentially polymorphic by extending each site with these analysis results. Figure 4.2 (b) shows a simplified example of *JIL* describing a call site extended with the results of class hierarchy analysis (CHA) [17] and variable type analysis (VTA) [45].

We chose virtual method polymorphism as an example since it demonstrates both static and dynamic properties of the code. In addition to the static information extracted by *SOOT*, the runtime behavior of these call sites can reveal which targets are actually being invoked. We continue this example in Section 4.3.3 when we describe another tool which is capable of exporting dynamic data as *JIL*.

4.3 Dynamic data

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

(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)

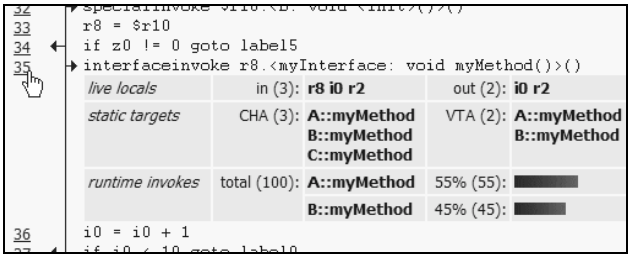

Figure 4.2: Polymorphism example. (**a**) A simple polymorphic Java fragment with an interface invoke; (**b**) A *JIL* fragment produced by *SOOT* indicating the possible targets of the call site; (**c**) A *JIL* fragment produced by a *STEP* profiling agent indicating the actual number of runtime invokes; (**d**) A slice from the *JIMPLEX* interface focused on the call site, where the user can browse the extensions from (b) and (c).

optimizations can not always be implemented based on static information alone.

One such optimization is virtual method inlining, which can compensate for the unnecessary use and overhead of virtual methods by directly inlining methods at their respective call sites [45]. Although effective, this optimization is complicated by the dynamic dispatching of virtual methods. Some static analyses can prove that a call site is monomorphic, allowing the called method to be inlined without breaking the code. Those call sites which can not be proven to be monomorphic require a more complicated optimization strategy. The runtime behavior of the code can reveal call sites which execute a single method most often, but are still provably polymorphic. Optimization of these virtual calls is still possible by inserting a runtime check to verify that inlined code is only executed when the dynamic type of the call site matches that of the inlined method.

4.3.1 Dynamic data in JIL

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. The separation and managment of *JIL* documents was discussed in Section 3.5.

4.3.2 STEP profiling framework

STEP is a customizable profiling framework for evaluating the performance and behavior of object-oriented applications [7]. Existing profiling systems can be difficult to apply to such 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 was originally introduced as a common trace format in the *STOOP* framework [5, 8]. *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, such as *JIL*.

Some of the key features of the *STEP* system mimic those of our own visualization framework, since they were both designed to be flexible and extensible:

- **Generality**: *STEP* provides a flexible format for storing traces which is not bound to a particular tool, data type, or encoding.
- **Self-descriptive**: *STEP* describes its own traces by specifying the structure, proper interpretation, and encoding of data.

As a result of these features, *STEP* is able to remain independent of any event producer or consumer. Event producers can include the Java Virtual Machine Profiler Interface (*JVMPI*) [43], a customized JVM, or another source of runtime events. Consumers are typically visualization or analysis tools, and can include our own *JIMPLEX* visualizer described in chapter 5, or any other tool capable of interpreting generic profiling events. *JIL* matches this kind of interoperability by allowing arbitrary tools to produce or consume *JIL* documents.

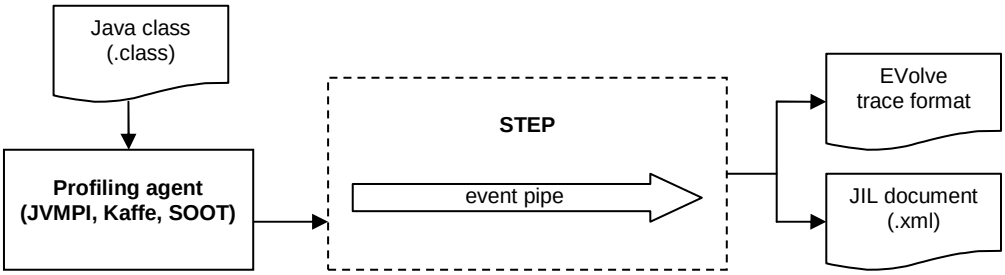

Figure 4.3: The profiling of Java programs in *STEP*. Profiling agents, such as *JVMPI* [43], *Kaffe* [50], or *SOOT*, are used to pass events to to the event pipe. The backend of *STEP* can then output the trace format in several formats, including *JIL*.

4.3.3 STEP as a source of dynamic program data

STEP allows runtime information to be efficiently exported by backend implementations. These implementations pull data from the event pipe and convert it to a specific format; this process is illustrated in Figure 4.3. Since the data coming from the event pipe is designed to be visualized, the implementation of the backend is typically dependent on the visualization system which consumes this data. For example, *EVolve* is a visualizer which is capable of interpreting large amounts of data [59]. It creates graphical representations which the user can filter and customize. The backend implementation which allows *STEP* to act as a data source for *EVolve* converts the trace data to a specific format with strict requirements. For the framework we present here, we designed a simple backend for generating *JIL* data, following the guidelines described in Section 4.1.

We now revisit our polymorphism example using *STEP* to generate *JIL* dynamic data. The *JIL* fragment at the bottom of Figure 4.2 (c) 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.

Chapter 5
Visualization and Consumption

We have now described how code tools which can extract both static and dynamic characteristics of Java code can be used to create *JIL* documents. This chapter discusses how these documents are visualized.

5.1 JIL as a data source

XML has been used as a format for storing data in many different scenarios. As a truly portable data source, it glues together many different complex systems by allowing data to be quickly and reliably queried, much like a common relational database. However, there are some fundamental differences between a relational database and XML. Data in XML is arranged in a hierarchy and lends itself well to parent-child and sibling relationships, as elements can only have a single parent. This structure applies well when modelling language constructs and associating metadata to them. Most code elements in a Java class can be represented using a hierarchical structure, as was demonstrated in Section 3.2. For the application programmer who is considering using *JIL* as input data, they must understand these differences and avoid treating the data as a typical database. By taking advantage of the previous work and research that has been applied to XML data mining, they can save both time and code.

Several programming models exist for consuming XML data such as *JIL*, some of which are optimized to save memory while others are suited towards repeated

processing of random elements. When developing applications which consume *JIL*, developers have a rich library of APIs and tools to choose from, which continues to grow. In Section 4.1.1 we discussed the two most popular XML processing models, *DOM* [2] and *SAX* [31], which are both supported by most XML APIs. The choice of which model to use when designing a *JIL* generator is based on different considerations, but is equally important when designing an application which consumes *JIL* as input.

Applications which consume *JIL* can vary both in their implementation and their functionality. *JIL* documents typically contain more data about a Java class than a user could, and would want to, absorb. Even the Java source code for a complex class can be overwhelming for a developer to build a cognitive model, especially if they are unfamiliar with the code. For this reason, most *JIL* consumers build an internal representation of the document and present the user with a high level interface, allowing them to focus on the elements of interest and request more detail when required. The *DOM* processing model is ideal for such an implementation, however it requires large amounts of memory depending on the size of the *JIL* document. As a result, when building a scalable visualization application the choice of processing model is very important.

As an XML-based format, *JIL* consumers can base their design and implementation on existing XML frameworks and applications. This is an excellent way to decrease the cost of development, both in time and lines of code. Since its inception in 1996, XML has proven to be an effective format for transmitting data across the Internet. It is often used as a data source when generating HTML web pages as a means of separating the interface from the data. This allows the interface to be implemented as a stylesheet, a generic layout and design which can be rendered dynamically as HTML. XML is used as a data source from which the stylesheet can extract those elements of the web site which vary from page to page.

For example, let us consider providing this thesis online in a browsable interface. Instead of rendering a large number of different HTML web pages representing the different chapters and sections of the document, we could store the text as XML and design a single interface to the documentation as a stylesheet. The stylesheet would

include generic layout information such as margin measurements and how to typeset headings, titles, and paragraphs. Using this strategy, when the text of the document changes the interface is updated dynamically the next time the XML is transformed by the stylesheet into HTML. Otherwise, every affected HTML page would need to be identified and reconstructed. Maintaining an updated version of the document under these circumstances would require much more work.

Since the data we want to present when visualizing intermediate languages is primarily textual, and because of how XML lends itself well to dynamic HTML interfaces, we chose to develop a web-based visualizer for *JIL*. The following section describes the design and implementation of *JIMPLEX*.

5.2 JIMPLEX visualization framework

JIMPLEX is a visualization implementation for *JIL* documents. It was designed 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 visually annotate *Jimple* code elements with any static or dynamic data collected by *SOOT* and *STEP*.

This allows the user to browse *Jimple* annotated with data from multiple tools, such as we saw in the *JIL* fragments in Figure 4.2 (b) and (c). Figure 4.2 (d) 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 performed by *SOOT* was accurate in eliminating the potential target from class `C`, based on the last profiling run.

In order to provide an interface which is portable across networks and platforms, *JIMPLEX* runs in a common web browser using HTML and basic scripting languages, such as Javascript, to implement its interface. Figure 5.1 shows a screenshot of the *JIMPLEX* interface running in a web browser. Rather than describing the details

of the arbitrary interface, the following sections will focus on how the interface is generated dynamically using *XSLT* and *JIL* as a data source.

5.2.1 JIMPLEX as a dynamic web page

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.

In Figure 5.2 we present a general overview of the visualizer. `Server A` represents a computer connected to the internet and equipped with a web server. *JIL* documents stored on this server can be accessed like any other web page, however as an XML file they are not useful to a human or a machine without translation. Instead, they are referenced using a Uniform Resource Locator (URL) by `Server B`. `Server B` hosts the *JIMPLEX* stylesheet, which is stored locally on this server as an XSL file with an `.xsl` extension. This stylesheet is the implementation of *JIMPLEX* and any changes to the interface or behavior of the visualizer are applied to this stylesheet on `Server B`. Multiple clients, represented in the figure as `Client 1` through `Client n`, can access the stylesheet on `Server B` as a common web page. The web server on `Server B` does not transmit the actual XSL file to the clients, but instead transmits HTML data which is interpreted as a web page by a browser. This HTML data is the result of the transformation of the XML document on `Server A` by the *JIMPLEX* stylesheet.

5.2.2 XSLT transformation in JIMPLEX

The transformation process shown in Figure 5.3 is often applied by the web server itself, requiring no additional software running on the server and simply a web browser

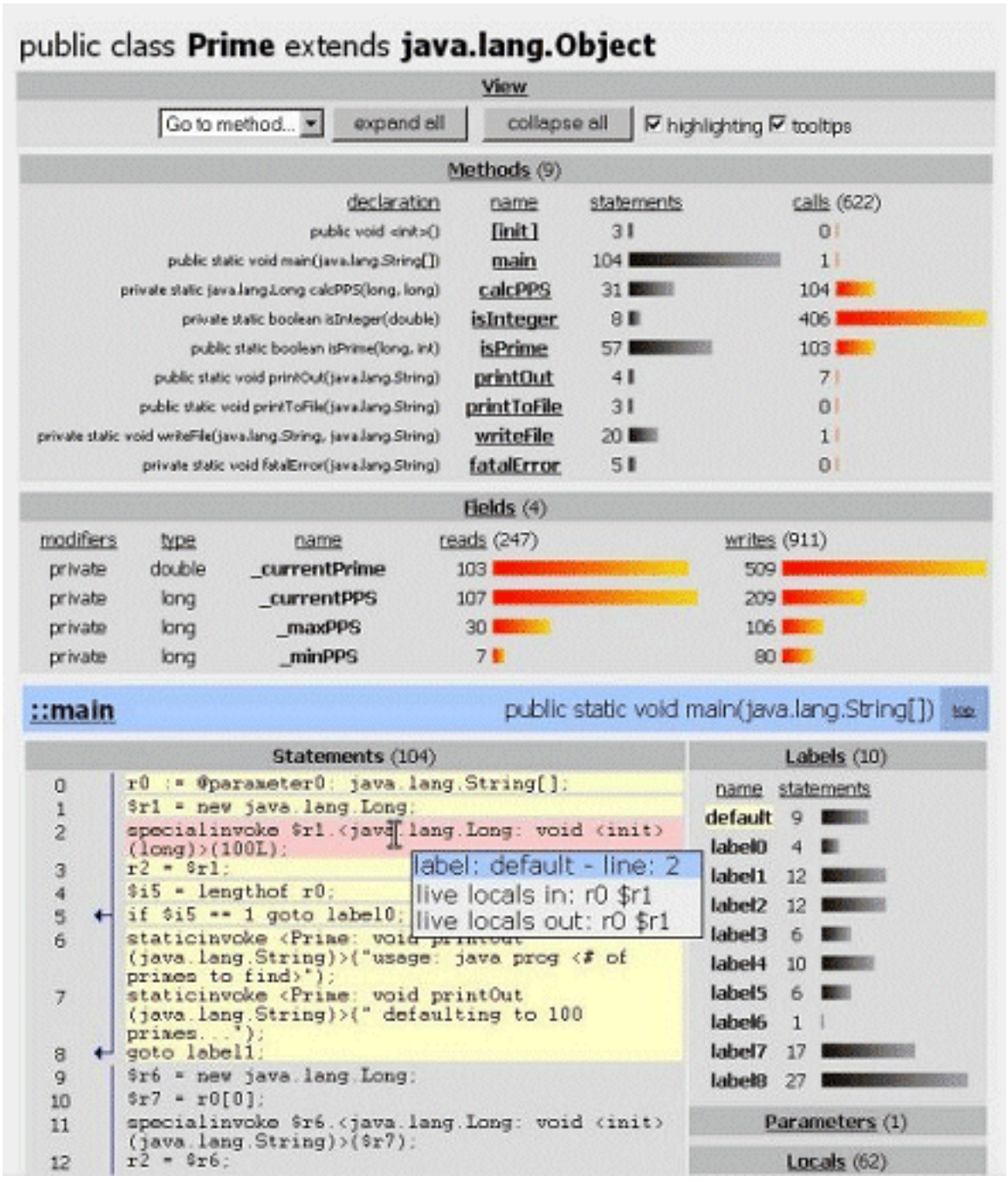

Figure 5.1: *JIMPLEX* running in a common web browser.

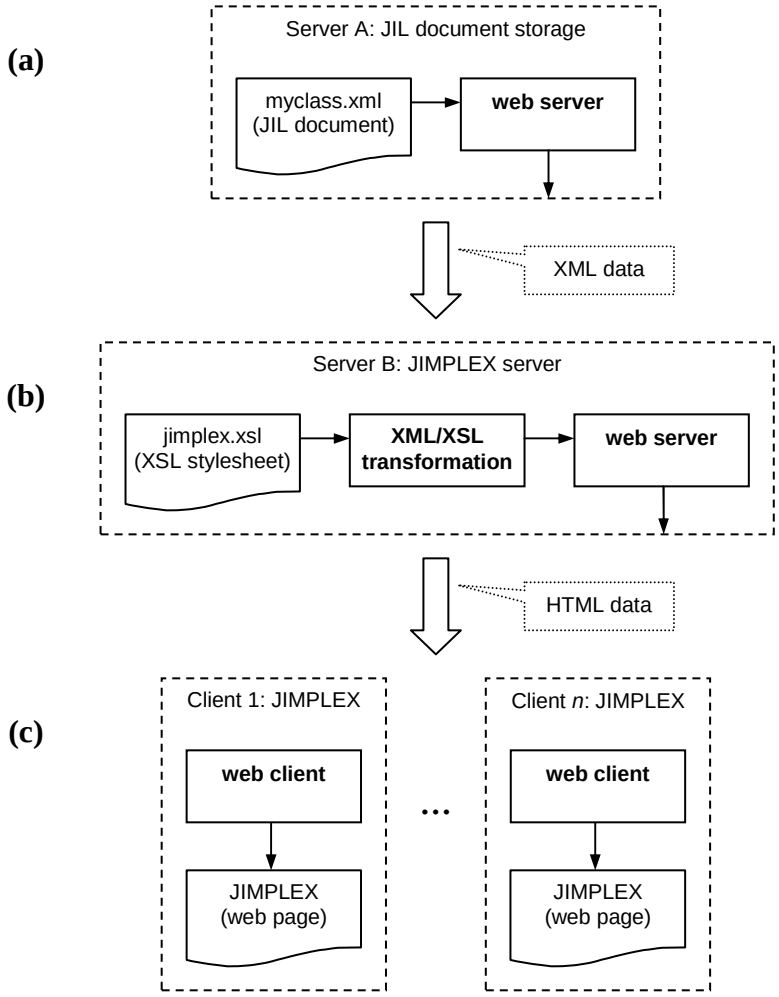

Figure 5.2: Overview of the *JIMPLEX* visualizer. (**a**) *JIL* documents are provided online by a web server. (**b**) The *JIMPLEX* interface is hosted online and queries data from (a) across the Internet; the XML data is transformed by the XSL stylesheet on the web server. (**c**) Any number of clients can then access the interface using a web browser as HTML from (b).

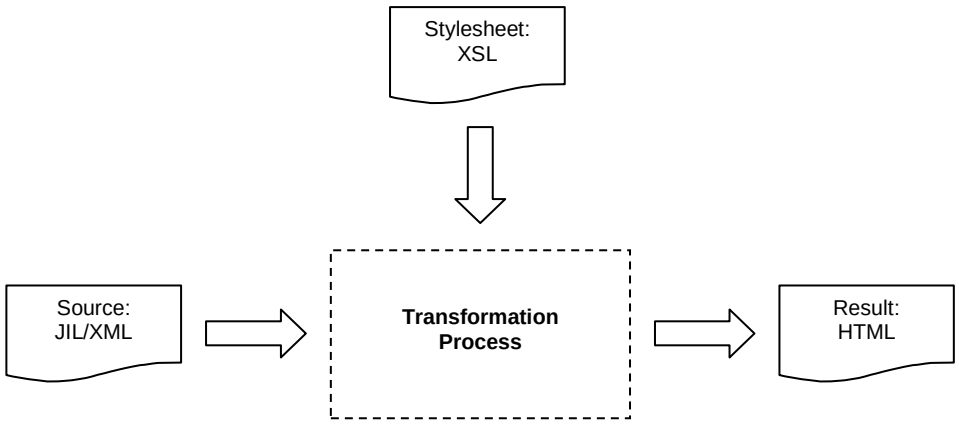

Figure 5.3: XSLT transformation in *JIMPLEX*.

capable of understanding HTML on the client. It is often the case that `Server A` and `Server B` in Figure 5.2 are actually the same server, and the XML data being transmitted between them is not required to travel across the Internet. However, *JIMPLEX* allows the user to specify an arbitrary *JIL* data source using an URL, which means that the *JIL* document is referenced as an online resource regardless of where it is actually stored. Modern web browsers, such as Microsoft's Internet Explorer 5 can even apply *XSLT* transformations on the client, allowing both the XML source, the *XSLT* stylesheet, and the interface to exist on the same machine. This allows the entire visualization process to take place locally without any requirement for network transmission. The choice to implement *JIMPLEX* as an *XSLT* stylesheet was based on the versatility and functionality the technologies make possible. We present this *XSLT* stylesheet in appendix B.3, and describe the kind of functionality it makes possible in the following section.

5.2.3 Dynamic interface generation

Now that we have outlined the process of transforming *JIL* data into HTML, we examine how we use HTML to create a visualization interface. The basic idea behind

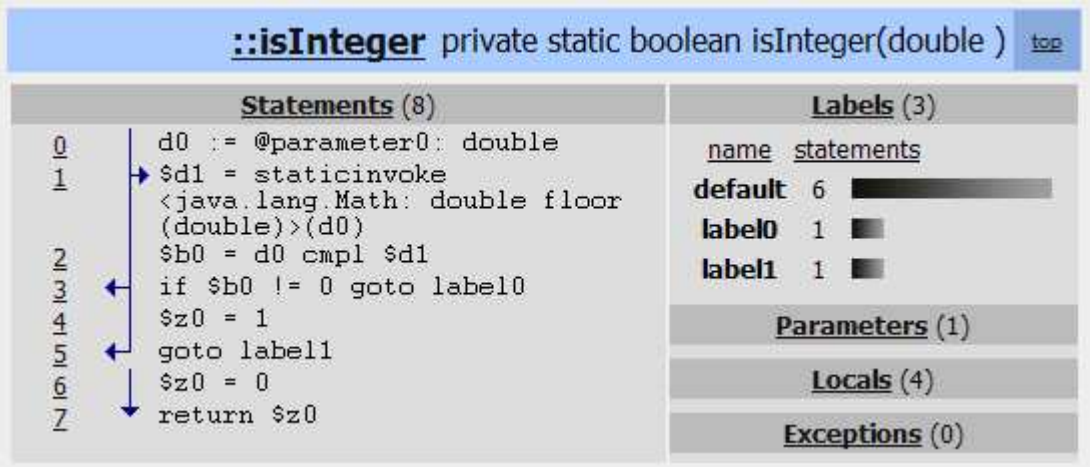

Figure 5.4: *JIMPLEX* interface: the *Jimple* statements from a simple method.

the *JIMPLEX* visualizer is to present the user with a textual representation of the intermediate languages stored in a *JIL* document. We use the basic formatting features of HTML in order to *pretty-print* the program code, which enhances the readability and promotes human understanding [15]. However, readability is not the focus of the visualizer since intermediate languages tend to lack the types of relationships and semantics found in high level languages. For example, the indentation of code within a `for` loop written in C allows the programmer to quickly identify which code is called repeatedly. If this code were examined in assembly, the same indentation would not be possible since the assembly instructions which comprise the `for` loop do not follow the semantics of C. Instead, we focus on presenting the user with the metadata associated with each code element as intuitively as possible. It is this metadata which the user is interested in, otherwise they could simply browse the intermediate code itself in their favorite text editor. *JIL* makes it possible to associate this metadata with the intermediate code, and *JIMPLEX* allows the user to visualize this metadata alongside the code elements.

In order to illustrate how basic metadata is visualized in *JIMPLEX* we present a slice from the interface in Figure 5.4. It depicts the *JIMPLEX* representation of a simple method `IsInteger`. The eight statements of *Jimple* source code which comprise the method are displayed in the section on the left. These statements are numbered,

and include the flow information using graphical lines and arrows. An arrow pointing to the right indicates that the statement contains an invoke. An arrow pointing to the left indicates that the program can branch on that statement. The line indicates the possible control flow; when this line has a section missing, such as next to the statement labelled 5, it means that the program can not continue past that statement without branching. An arrow pointing down indicates that the method returns to its callee after the accompanying statement. Although this is simple example of metadata associated to intermediate language statements, it demonstrates how this information can be displayed to the user intuitively using only basic HTML.

Although there are limitations in the functionality of a web-based interface, the visualization of programming languages and related metadata is largely text based. In addition to HTML, scripting languages, such as Javascript and CSS, which are supported by modern browers are used to implement the interface. These languages can provide the kind of advanced functionality and interactivity associated with application-based interfaces.

This type of functionality can be seen in Figure 5.5. On the right side of this section of the *JIMPLEX* interface there are several enumerations available to the user: `Labels`, `Parameters`, and `Locals`. An enumeration of the `Exceptions` is also available, but is not visible in this screenshot. By clicking on the titles of these enumerations, the user can expand and collapse the data associated with them in order to select which enumerations they want to view. In Figure 5.5 the user has expanded the list of `Locals` associated with this method, and can see the number of times each variable is defined and used according to the `set/get` counts. As the user passes their mouse over a particular local variable, the variable is highlighted as well as all the statements in which it is defined and used. In Figure 5.5, the user has highlighted the local variable `l0` which has triggered the highlighting of its definition in statement 0 and the highlighting of its first use in statement 3. Although colors are not visible in a monochrome printing of this thesis, it is enough to note that the use of color is a feature that even a text-based web interface can take advantage of. This kind of highlighting functionality is useful for quickly identifying statements, or sections of statements, and is available in each of the enumerations associated with

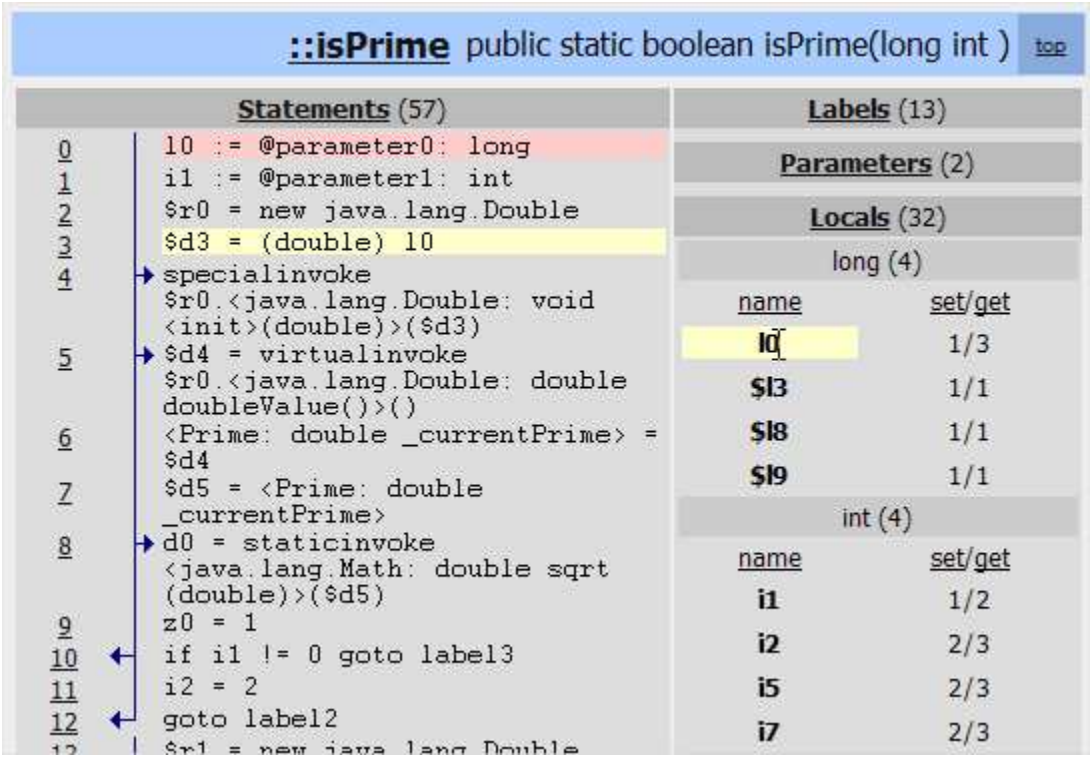

Figure 5.5: *JIMPLEX* interface: highlighting of variable definitions and uses.

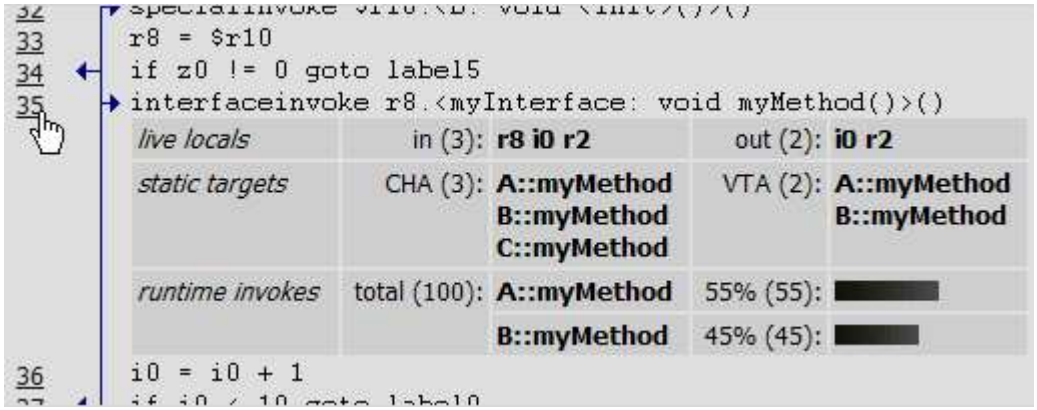

Figure 5.6: *JIMPLEX* interface: analysis results associated with statements.

each method.

More advanced metadata, such as the results of static or dynamic analyses are also exposed by the *JIMPLEX* interface. Figure 5.6 shows a slice of the statement view in a *JIMPLEX* interface. By clicking on the statement numbers to the left of each statement, the user can expand the metadata associated with that particular statement. In this case, the user has clicked on the statement labelled 35, which contains an invoke. The metadata presented to the user includes the results of three static analyses and one runtime analysis. The local variables which are live passing in and out of the statement are listed first. This allows the user to quickly identify which variables are not used before and after this particular statement. Next, the results of a class hierarchy analysis (CHA) [17] and a variable type analysis (VTA) [45] are listed as invoke targets. CHA and VTA are static analyses which can identify which methods are potential targets of the invoke contained with this statement, allowing the user to explore the polymorphism of the code. The last section of metadata shows the results of a dynamic analysis of which methods were actually invoked at this call site. This is data which was collected at runtime during an execution run which caused this invoke to be executed 100 times. Out of these 100 invokes, the data shows that 55 of them selected method `A::myMethod` as the target and 45 of them selected method `B::myMethod`. The different analyses shown here complement each other, and often expose redundant data. For example, the user will notice that the

runtime invokes only involve those methods identified by the variable type analysis. If this were not the case, the user might speculate that the VTA was inaccurate and investigate why the runtime invokes told a different story. *JIMPLEX* allows the user to develop a clear and complete cognitive model of both the intermediate languages and characteristics of the code which are contained within *JIL* documents.

Chapter 6
Availability and Future Work

We have presented all the tools and technologies used in our visualization framework, however there is no substitute for a hands-on demonstration. The following chapter describes the availability of the tools and resources used in our framework, all of which are open source. Finally, we will discuss some of the ideas for future improvements that we were unable to explore or implement due to a lack of time and resources.

6.1 Availability

The work presented in this thesis is based on the *JIL* 1.0 specification which is available online as a Document Type Definition (DTD). These DTDs are meant to demonstrate the online specification of this intermediate format, allowing documents and tools to reference them directly as online resources. They can also be downloaded and modified as required, allowing other researchers to improve or build upon the current specification. Table 6.1 contains the URLs where the base *JIL* DTD, as well as the extension DTDs, can be found online.

This initial version of *JIL* includes some basic elements of a Java class, and provides a structure upon which extensions can be added. This base specification references another DTD for the *SOOT* optimization framework which adds extensions

DTD	Current Version	URL
Base *JIL*	1.0	`http://www.sable.mcgill.ca/jil/jil10.dtd`
SOOT	1.0	`http://www.sable.mcgill.ca/jil/jil-soot10.dtd`
STEP	1.0	`http://www.sable.mcgill.ca/jil/jil-step10.dtd`

Table 6.1: Document Type Definitions which are available online.

Software Tool	Current Version	URL
JIL	1.0	`http://www.sable.mcgill.ca/jil`
JIMPLEX	1.0	`http://www.sable.mcgill.ca/jil`
SOOT	1.2.3	`http://www.sable.mcgill.ca/soot`
STEP	beta	`http://www.sable.mcgill.ca/step`

Table 6.2: Software framework homepages.

for some static analyses. An extension DTD is also provided for the *STEP* profiling framework which defines some basic dynamic profiling extensions.

Figure 6.2 contains the links to the homepages of the tools used in our framework. These tools are constantly evolving and frequently benefit from public contributions and input. The *JIL* homepage has links to related papers and online tools, such as the *JIMPLEX* visualization framework which is provided as a package of client and server-side scripts. Current implementations allow the online validation and visualization of *JIL* documents produced by *SOOT* and *STEP*.

The *SOOT* optimization framework has an extensive website which includes documentation and tutorials, as well as links to public discussion lists and related papers. The current version of the *SOOT* binaries and source code are available for download there. The *STEP* framework is not currently available for release, however a site exists where the source and documentation will soon be available. Currently, only related papers are available for download.

6.2 Future work

As an open framework, there are many different areas for future work. The *JIL* specification itself is in its infancy, and only basic extensions are supported. This 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 Document Type Definitions 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. Current implementations target the optimization of Java classes, but the framework can be applied to many areas of visualization.

6.2.1 JIL language extensions

The framework presented here is designed to allow visualization interfaces to include any data from any source, with minimal development. It allows visualizers to be developed based on the information the user wants to analyze rather than what information is available.

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 a benchmarking suite using a popular XML API matching 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.

6.2.2 Visualization implementations

The *JIMPLEX* visualizer presented in Section 5.2 only explores a few of the possibilities available when presenting the user with an interface to the data contained within *JIL* documents. There are two major features of XML and *JIL* which facilitate the design and implementation of a variety of visualizers:

- **Separation of data from presentation**: In the case of *JIL* the data consists of a detailed description of Java classes, included intermediate languages and associated metadata. The presentation of this data is the visualization interface used to expose the data to a user. This allows visualization developers to worry more about the interfaces and functionality they can provide, rather than how the data is stored and accessed.

- **Interoperability**: *JIL* is easily shared between applications, both because it is easy to test for compatibility and to transmit as data. The widespread acceptance of XML as a universal format for interchanging data between applications and across networks gives developers a strong foundation upon which new visualizers can build.

6.2.3 Metadata-enhanced software development

Although visualization is the current focus of this framework, it is only one example of an application for *JIL*. The future use of *JIL* could target any application where metadata is associated to code. A *JIL*-aware Integrated Development Environment (IDE) could 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.

Chapter 7
Discussion and Conclusion

This chapter reviews the contributions of our framework and the goals it was designed to achieve. We evaluate the success of the overall system, and of each module which comprises it.

7.1 Contributions

In order to examine the success of the framework we revisit each of the contributions made towards this thesis. The problems that were stated in Section 1.1 were solved collectively by these contributions, and in this respect they rely on one another. However, these contributions were designed and implemented independently, and should therefore be evaluated as separate solutions.

7.1.1 JIL as a common document format

In order to provide a common document format for sharing data between arbitrary code tools, we designed the Java Intermediate Language (*JIL*). Many of the benefits of this language come from its roots as an XML-based format. We now list the key benefits and issues we discovered when designing and using this format.

Benefits

- As an XML-based format, the generation, parsing, and processing of *JIL* is facilitated by the numerous applications and APIs available both commercially and in the public domain. The popularity of and support for this format allow developers to concentrate on the functionality of their tools rather than the performance or reliability of their *JIL* input and output routines.

- Another benefit of XML compliance is the ability to serve and share *JIL* documents online, allowing both people and software to collaborate across existing networks and protocols. The *JIMPLEX* visualizer is an example of this functionality, as an online tool which can provide a visualization interface to any *JIL* document on the web.

- Extensibility is one of main benefits of *JIL*. The format is designed to store both existing and future intermediate representations of Java, as well as related metadata. The DTD language used to specify the *JIL* document format is easily extended in order to formally define new elements or attributes. Elements can be added or removed from *JIL* documents without breaking any existing structure, allowing documents to evolve freely while maintaining compatibility.

- *JIL* elements are self-describing, allowing visualizers to provide support without prior knowledge of the languages or metadata contained within a document. They also allow applications which consume *JIL* as input to implement intelligent data mining, evaluating which elements to parse based on the user's queries or the computational cost.

Issues

- Since much of the data contained within *JIL* documents is used to describe the structure and relations between document elements, they tend to be much larger and more verbose than their corresponding source code and bytecode. In the case of large, complex Java classes this can seriously affect the performance

of tools which do not handle *JIL* efficiently. However, the performance of tools and APIs is improving as the XML community researches new compression techniques.

7.1.2 SOOT as a static data source

The choice of *SOOT* a static data source was based largely on its nature as an experimental platform for developing new static analyses for Java bytecode. As a contribution to this thesis, support for *JIL* was added as an additional output format. We avoid examining the merits of *SOOT* as an optimization framework and focus on the benefits it provides as a source of static data for *JIL*.

Benefits

- *SOOT* provides an ideal source of content for *JIL* documents. Before support for *JIL* was added, *SOOT* already used several intermediate languages internally. These languages were used for performing analyses and transformations upon. With the addition of *JIL* support, *SOOT* acquires a new mechanism for exporting the metadata associated with these languages and *JIL* acquires a source of static data.

- *SOOT* provides a source of undiscovered data, taking advantage of the extensibility of *JIL*. As an open source API, *SOOT* can be used to develop new optimizations, analyses, and transformations on Java bytecode, using *JIL* to export and visualize them.

Issues

- For the user who simply wants to create a *JIL* document representation of a Java class, the *SOOT* framework provides much more functionality than they require. This comes at the cost of execution speed and the high complexity of the API and framework.

7.1.3 STEP as a dynamic data source

The *STEP* backend implemented for this thesis allows dynamic characteristics of Java classes to be stored within *JIL* documents.

Benefits

- The customizable nature of the *STEP* framework allows developers to collect arbitrary runtime data, taking full advantage of the extensibility of *JIL* as a format for storing this data.

Issues

- One of the major issues with the *STEP* framework is its unfinished state of release at the time of the writing of this thesis. This has prevented the development of a backend compatible with the current specification of the *STEP* trace format, as well as a complete evaluation of *STEP* as a source of dynamic data.

7.1.4 JIMPLEX as a visualization backend

JIMPLEX was developed to demonstrate two fundamental properties of using *JIL* as a visualization data source. Firstly, the ability to create a customizable interface which is as easily updated with new code characteristics as the *JIL* specification itself. And secondly, the ability to provide this functionality online.

Benefits

- The *JIMPLEX* visualizer is both simple and customizable. The interface consists only of a dynamic web page which queries its data from *JIL* documents and builds its interface using common web languages such as XSL, HTML, and Javascript.

- As a web-based interface, *JIMPLEX* encourages collaboration and compatibility. The tool can run on a web server allowing the interface to exist on any

machine with a web browser connected to the network. This allows users to share the same version of the visualizer and never have to worry about updating the tool to receive the latest features or bug fixes.

Issues

- As a web page, the functionality provided by *JIMPLEX* is limited. The languages and technologies supported by web browsers were not designed to implement an interactive visualization interface. Although web pages are effective at displaying textual information, they are not as efficient when providing graphical and interactive functionality.

7.2 Summary

In this thesis we have presented an open framework for developing visualization interfaces in the interest of studying the compilation and runtime behavior of Java programs. We now revisit our solutions for the underlying goals of our framework described in Section 1.1.

- **Comprehensiveness**: In order to include both static and dynamic data in our visualization framework, we designed *JIL* as a generic format capable of associating any type of data to each code element. This includes both compile-time and runtime characteristics of Java programs, allowing the user to construct a comprehensive model of the software's behavior.

- **Generality**: We allow any code or compiler tool to contribute content to *JIL* documents, creating a many-to-many relationship between applications which can generate and consume *JIL*; this relationship is illustrated in Figure 7.1. Typically, code tools share a one-to-one or a one-to-many relationship with the applications which can utilize the intermediate languages they produce. For example, *SOOT* can produce *Jimple* representations of Java classes, however no other application can contribute information to the *Jimple* code making this the final representation of the class.

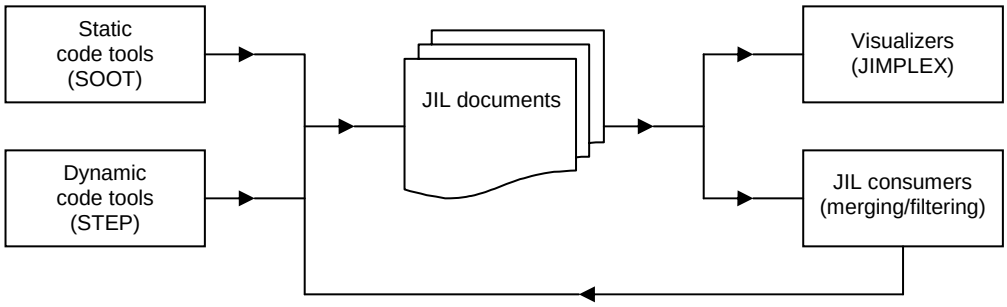

Figure 7.1: The many-to-many relationship of *JIL* generators and consumers. Any number of static or dynamic code tools can contribute to the *JIL* representation of a Java class. That *JIL* document can then be used as a data source by visualizers or other applications designed to consume *JIL*; these applications can also be designed to merge, filter, or otherwise process elements and output a new *JIL* document.

- **Extensibility**: In order to continue visualizing new and undiscovered characteristics of code elements, we designed *JIL* and our visualizer *JIMPLEX* to be highly extensible. The formal specification of *JIL* is stored as a Document Type Definition (DTD), which can be modified without any special tools or compilation in order to accept new document elements, attributes, or structures. Elements which are added or removed from *JIL* documents do not break any existing structure, allowing even existing documents to evolve as their generating applications are extended to support future metadata or intermediate languages. *JIMPLEX* is implemented as an XSLT stylesheet, allowing the interface to be extended to support these new *JIL* elements by simply updating the stylesheet, requiring no special tools or compilation.

7.3 Discussion

The key features of the framework allow existing and future tools to contribute both static and dynamic code elements to these visualizations. By using an extensive and

portable document format for storing and separating data, our framework 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, as many APIs exist which can already validate, parse, and generate compliant *JIL*.

The approach to visualization presented in this thesis extends object-oriented design to software management and visualization. The framework mimics much of the flexibility and management possible when using modern object-oriented languages and technologies to build extensible and reusable systems. It is a natural progression for our compiler development and visualization systems to inherit a modular and language-independent design. By encouraging developers to design tools which can share the data they extract, they can increase their user audience and prolong the lifetime of their software. Although it is not a new concept to attach attributes and metadata to source code, this technique has rarely been applied to low level languages and code tools. Even though this framework has a narrow focus of Java visualization, it has demonstrated that the gaps between static and dynamic data, code tools, and intermediate representations can be bridged by applying new techniques with existing technologies.

Bibliography

[1] Apache XML Project. *Xerces XML Parser*, 2000. Available at `http://xml.apache.org`.

[2] V. Apparao, S. Byrne, M. Champion S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. *Document Object Model (DOM) Level 1 Specification (Second Edition)*. W3C DOM Working Group, September 2000. Available at `http://www.w3.org/DOM`.

[3] T. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C XML Working Group, October 2000. Available at `http://www.w3.org/TR/REC-xml`.

[5] R. Brown, K. Driesen, D. Eng, L. Hendren, J. Jorgensen, C. Verbrugge, and Q. Wang. STOOP: The Sable toolkit for object-oriented profiling. Technical Report SABLE-2001-2, McGill University, Sable Research Group, November 2001.

[6] R. Brown, K. Driesen, D. Eng, L. Hendren, J. Jorgensen, C. Verbrugge, and Q. Wang. STEP: A framework for the efficient encoding of general trace data. Technical Report SABLE-2002-7, McGill University, Sable Research Group, June 2002.

[7] R. Brown, K. Driesen, D. Eng, L. Hendren, J. Jorgensen, C. Verbrugge, and Q. Wang. STEP: A framework for the efficient encoding of general trace data. In *Proceedings of PASTE '02*, pages 27–34, November 2002.

[8] R. Brown, J. Jorgensen, Q. Wang, L. Hendren, K. Driesen, and C. Verbrugge. Poster 32: STOOP: The Sable toolkit for object-oriented profiling. In *Poster Abstracts of OOPSLA '01*, October 2001.

[9] Z. Budimlic and K. Kennedy. Optimizing Java: Theory and practice. *Concurrency: Practice and Experience*, 9(6):445–463, 1997.

[10] E. B. Buss, R. de Mori, W. M. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, E. Merlo, H. A. Muller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S. R. Tilley, J. Troster, and K. Wong. Investigating reverse engineering technologies for the CAS program understanding project. *IBM Systems Journal*, 33(3):477–500, 1994.

[11] M. Cierniak and W. Li. Briki: A flexible Java compiler. Technical Report TR621, University of Rochester, May 1996.

[12] M. Cierniak and W. Li. Optimizing Java bytecodes. *Concurrency: Practice and Experience*, 9(6):427–444, 1997.

[13] J. Clark. *XSL Transformations (XSLT) Version 1.0*. W3C XML Working Group, November 1999. Available at `http://www.w3.org/TR/xslt`.

[14] B. Cooper, H. Lee, and B. Zorn. ProfBuilder: A package for rapidly building Java execution profilers. Technical Report CU-CS-853-98, University of Colorado, April 1998.

[15] D. D. Cowan, D. M. Germán, C. J. P. Lucena, and A. von Staa. Enhancing code for readability and comprehension using SGML. In *Proceedings of ICSM '94*, pages 181–190, September 1994.

[16] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. VORTEX: An optimizing compiler for object-oriented languages. In *Proceedings of OOPSLA '96*, pages 83–100, 1996.

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

[18] D. Eng. Combining static and dynamic data in code visualization. In *Proceedings of PASTE '02*, pages 43–50, November 2002.

[19] D. Eng. JIL: an extensible intermediate language. Technical Report SABLE-2002-3, McGill University, Sable Research Group, June 2002.

[20] D. Eng. JIL: an extensible intermediate language, January 2002. Available at `http://www.sable.mcgill.ca/jil`.

[21] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, May 1996.

[22] J. Grundy and J. Hosking. High-level static and dynamic visualization of software architectures. In *Proceedings of the IEEE International Symposium on Visual Languages*, pages 5–12, September 2000.

[23] J. C. Hardwick and J. Sipelstein. Java as an intermediate language. Technical Report CMU-CS-96-161, Carnegie Mellon University, 1996.

[24] J. Hunter and B. McLaughlin. JDOM. Available at `http://www.jdom.org`.

[25] International Standard Organization. *Standard Generalized Markup Language (SGML)*, October 1986. International Standard ISO 8879.

[26] C. Knight and M. Munro. Visualising software - a key research area. In *Proceedings of the IEEE International Conference on Software Maintenance*, page 436, September 1999. Short paper.

[27] Claire Knight and Malcolm C. Munro. Virtual but visible software. In *Proceedings of International Conference on Information Visualisation*, pages 198–205, July 2000.

[28] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi. SCED: A tool for dynamic modelling of object systems. Technical Report A-1996-4, University of Tampere, 1996.

[29] H. B. Lee and B. G. Zorn. BIT: A tool for instrumenting Java bytecodes. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 73–83, December 1997.

[30] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, September 1996.

[31] D. Megginson. SAX: The simple API for XML, May 2000. Available at `http://www.saxproject.org/`.

[32] H. A. Muller, J. H. Jahnke, D. B. Smith, M. D. Storey, S. R. Tilley, and K. Wong. Reverse engineering: a roadmap. In *ICSE - Future of SE Track*, pages 47–60, 2000.

[33] N. Nystrom. Bytecode-level analysis and optimization of Java class files. Master's thesis, Purdue University, May 1998.

[34] N. Nystrom and T. Hosking. BLOAT: Bytecode level optimization and analysis tool, 1999. Available at `http://www.ex-compiler.lcs.mit.edu`.

[35] R. O'Callahan. The design of program analysis services. Technical Report CMU-CS-99-135, Carnegie Mellon University, June 1999.

[36] R. O'Callahan. Optimizing a solver of oplymorphism constraints: SEMI. Technical Report CMU-CS-99-136, Carnegie Mellon University, June 1999.

[37] D. P. Olshefski and A. Cole. A prototype system for static and dynamic program understanding. In *Proceedings of the Working Conference in Reverse Engineering*, pages 93–106, 1993.

[38] PHP Group. *PHP: Hypertext Preprocessor*. Available at `http://www.php.net`.

[39] B. A. Price, I. S. Small, and R. M. Baecker. A principled taxonomy of software visualisation. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[40] D. Raggett, A. Le Hors, and I. Jacobs. Html 4.01 specification, December 1999. Available at `http://www.w3.org/TR/html401`.

[41] M. Rinard, C. S. Ananian, B. Demsky, M. C. Marinescu, D. Marinov, R. Rugina, and A. Salcianu. The FLEX compiler infrastructure, 1999. Available at `http://www.ex-compiler.lcs.mit.edu`.

[42] D. J. Robson, K. H. Bennett, B. J. Cornelius, and M. Munro. Approaches to program comprehension. *Journal of Systems and Software 14*, pages 79–84, February 1991.

[43] Sun Microsystems. *Java Virtual Machine Profiling Interface (JVMPI)*, 2000. Available at `http://java.sun.com/j2se/1.3/docs/guide/jvmpi`.

[44] Sun Microsystems. *Java API for XML Processing (JAXP)*, 2001. Available at `http://java.sun.com/xml`.

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

[46] T. Systa. Dynamic reverse engineering of Java software. In *Proceedings of the ECOOP Workshop on Experiences in Object-Oriented ReEngineering, FZI Report 2-6-6/99*, June 1999.

[47] T. Systa. On the relationships between static and dynamic models in reverse engineering Java software. In *Proceedings of the Working Conference on Reverse Engineering*, pages 304–313, 1999.

[48] T. Systa. *Static and Dynamic Reverse Engineering Techniques and for Java Software Systems*. PhD thesis, University of Tampere, May 2000.

[49] T. Systa. Understanding the behaviour of java programs. In *Proceedings of the Working Conference on Reverse Engineering*, pages 35–44, 2000.

[50] Transvirtual Technologies Inc. *Kaffe Open VM*, 1998. Available at `http://www.kaffe.org`.

[51] R. Vallee-Rai. Soot: A Java bytecode optimization framework. Master's thesis, McGill University, July 2000.

[52] R. Vallee-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: is it feasible? In *Proceedings of the International Conference on Compiler Construction*, pages 18–34, March 2000.

[53] R. Vallee-Rai and L. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical Report SABLE-1998-4, McGill University, Sable Research Group, July 1998.

[54] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON '99*, pages 125–135, 1999.

[55] M.G.J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In *Proceedings of SOFSEM '96*, pages 235–254, 1996.

[56] M.G.J. van den Brand, P. Klint, and C. Verhoef. Reengineering needs generic programming language technology. Technical Report P9618, University of Amsterdam, Programming Research Group, 1996.

[57] A. von Mayrhauser and A. Vans. Program understanding – a survey. Technical Report CS-94-120, Colorado State University, August 1994.

[58] R. J. Walker, G. C. Murphy, B. N. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings of OOPSLA '98*, pages 271–283, October 1998.

[59] Q. Wang, R. Brown, K. Driesen, L. Hendren, and C. Verbrugge. EVolve: An extensible software visualization framework. Technical Report SABLE-2002-6, McGill University, Sable Research Group, July 2002.

Appendix A
Grammars

A.1 Base JIL DTD

The specification of the Java Intermediate Language (*JIL*) is provided here as a package of Document Type Definitions (DTDs). These DTDs are also provided digitally on the web [20]. A formal syntax of DTDs can be found in the W3C XML specification [4], however we will describe some basic syntax here:

- `ENTITY`: Entities in DTDs are variables which represent other values. We use entities to define special keywords or values, as well as to reference external DTDs.

- `ELEMENT`: DTD elements correspond to XML elements, meaning XML markup which consists of a start and end tag, such as:

  ```
  <variable>a</variable>
  ```

  In this example, `variable` is an XML element and would be defined by a DTD element with the same name. DTD elements also specify which children, if any, this element should contain as well as how many of each children the element can contain.

- `ATTLIST`: Attributes of XML elements are declared using the DTD `ATTLIST` keyword. Attributes are associated to elements and describe additional properties about them, such as:

  ```
  <variable type="int">a</variable>
  ```

  This continues our previous example, giving the variable `a` the type `int`. Attributes in a DTD specify which element they are associated to, as well as their name, basic type, and default value, if any. Attributes can also be marked as required or optional, which will tell a validating application if a missing attribute should result in the document being rejected or not.

The base *JIL* DTD specifies a set of fundamental code elements common in every Java class. These includes the class itself, fields, methods, exceptions, parameters, locals, labels, and associated attributes for each. Future extensions should be associated to these elements, although new elements may be added. Each element definition is preceded by a corresponding *JIL* fragment, commented out using `<!--` and `-->`.

```
<!-- internal entities -->
<!ENTITY UNKNOWN "unknown">
<!ENTITY UNKNOWN_COUNT "-1">

<!-- external generator includes -->
<!ENTITY % JIL-GENERATOR-SOOT SYSTEM "jil-soot10.dtd">
%JIL-GENERATOR-SOOT;
<!ENTITY % JIL-GENERATOR-STEP SYSTEM "jil-step10.dtd">
%JIL-GENERATOR-STEP;

<!-- root node: <jil version="1.0" /> -->
<!ELEMENT jil (history, class)>
<!ATTLIST jil version CDATA #FIXED "1.0">

<!-- <history> (supported generators) </history> -->
<!ELEMENT history (soot*, step*)>
<!ATTLIST history created CDATA "&UNKNOWN_COUNT;">

<!-- <class name="myclass" extends="unknown"> -->
<!ELEMENT class (modifiers?, interfaces?, fields?, methods?)>
```

```
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class extends CDATA "&UNKNOWN;">

<!-- <modifiers count="1" /> -->
<!ELEMENT modifiers (modifier*)>
<!ATTLIST modifiers count CDATA "&UNKNOWN_COUNT;">

<!-- <modifier type="public" /> -->
<!ELEMENT modifier EMPTY>
<!ATTLIST modifier name (public|protected|private|package|abstract|final|
                         static|transient|volatile) #REQUIRED>

<!-- <interfaces count="1" /> -->
<!ELEMENT interfaces (interface*)>
<!ATTLIST interfaces count CDATA "&UNKNOWN_COUNT;">

<!-- <interface name="myinterface" /> -->
<!ELEMENT interface EMPTY>
<!ATTLIST interface name CDATA #REQUIRED>

<!-- <fields count="1" /> -->
<!ELEMENT fields (field*)>
<!ATTLIST fields count CDATA "&UNKNOWN_COUNT;">

<!-- <field id="0" name="myfield" type="mytype" /> -->
<!ELEMENT field (modifiers, stoop_field?)>
<!ATTLIST field id NMTOKEN #REQUIRED>
<!ATTLIST field name NMTOKEN #REQUIRED>
<!ATTLIST field type CDATA "&UNKNOWN;">

<!-- <methods count="1" /> -->
<!ELEMENT methods (method*)>
<!ATTLIST methods count CDATA "&UNKNOWN_COUNT;">

<!-- <method name="mymethod" returntype="mytype" /> -->
<!ELEMENT method (declaration, parameters, locals, labels, statements,
                  exceptions, stoop_method?)>
<!ATTLIST method name NMTOKEN #REQUIRED>
<!ATTLIST method returntype CDATA "&UNKNOWN;">

<!-- <declaration>my declaration</declaraction> -->
<!ELEMENT declaration (#PCDATA)>
<!ATTLIST declaration length NMTOKEN "&UNKNOWN_COUNT;">

<!-- <parameters count="2">...</parameters> -->
<!ELEMENT parameters (parameter*)>
<!ATTLIST parameters count CDATA "&UNKNOWN_COUNT;">
<!ATTLIST parameters method CDATA "&UNKNOWN;">
```

```
<!-- <parameter id="0" name="myparam" type="mytype" /> -->
<!ELEMENT parameter (soot_parameter?)>
<!ATTLIST parameter id NMTOKEN #REQUIRED>
<!ATTLIST parameter name NMTOKEN #REQUIRED>
<!ATTLIST parameter type CDATA "&UNKNOWN;">

<!-- <locals count="2">...</locals> -->
<!ELEMENT locals (local*, types?)>
<!ATTLIST locals count CDATA "&UNKNOWN_COUNT;">

<!-- <local id="0" name="mylocal" type="mytype"> --> <!ELEMENT
local (soot_local?)> <!ATTLIST local id NMTOKEN #REQUIRED>
<!ATTLIST local name CDATA #REQUIRED> <!ATTLIST local type CDATA
"&UNKNOWN;">

<!-- <types count="2" /> -->
<!ELEMENT types (type*)>
<!ATTLIST types count CDATA "&UNKNOWN_COUNT;">

<!-- <type id="0" count="2" name="mytype" /> -->
<!ELEMENT type (local*)>
<!ATTLIST type id NMTOKEN #REQUIRED>
<!ATTLIST type count NMTOKEN #REQUIRED>
<!ATTLIST type type CDATA #REQUIRED>

<!-- <labels count="2">...</labels> -->
<!ELEMENT labels (label*)>
<!ATTLIST labels count CDATA "&UNKNOWN_COUNT;">

<!-- <label id="0" name="mylabel" /> -->
<!ELEMENT label EMPTY>
<!ATTLIST label id NMTOKEN #REQUIRED>
<!ATTLIST label name CDATA #REQUIRED>

<!-- <statements count="1">...</statements> -->
<!ELEMENT statements (statement*)>
<!ATTLIST statements count CDATA "&UNKNOWN_COUNT;">

<!-- <statement id="0" label="mylabel" branches="false"
                                       fallsthrough="true" /> -->
<!ELEMENT statement (soot_statement*)>
<!ATTLIST statement id NMTOKEN #REQUIRED>
<!ATTLIST statement label CDATA #REQUIRED>
<!ATTLIST statement labelid NMTOKEN #IMPLIED>

<!-- <exceptions count="1" /> -->
<!ELEMENT exceptions (exception*)>
```

```
<!ATTLIST exceptions count CDATA "&UNKNOWN_COUNT;">

<!-- <exception id="0" type="java.io.IOException"> -->
<!ELEMENT exception (begin, end, handler)>
<!ATTLIST exception id NMTOKEN #REQUIRED>
<!ATTLIST exception type CDATA #REQUIRED>

<!-- <begin label="mylabel" /> -->
<!ELEMENT begin EMPTY>
<!ATTLIST begin label CDATA #REQUIRED>

<!-- <end label="mylabel" /> -->
<!ELEMENT end EMPTY>
<!ATTLIST end label CDATA #REQUIRED>

<!-- <handler label="mylabel" /> -->
<!ELEMENT handler EMPTY>
<!ATTLIST handler label CDATA #REQUIRED>
```

A.2 SOOT extensions

A separate DTD is used to specify the extensions supported by the *SOOT* optimization framework [51]. The extensions found here are specific to this framework, although similar extensions might also be provided by another tool capable of static analysis.

```
<!-- <soot version="1.0" /> -->
<!ELEMENT soot EMPTY>
<!ATTLIST soot version CDATA #REQUIRED>
<!ATTLIST soot command CDATA #REQUIRED>
<!ATTLIST soot timestamp CDATA "&UNKNOWN_COUNT;">

<!-- <... method="myclass" /> -->
<!ATTLIST method class CDATA #IMPLIED>
<!ATTLIST local method CDATA #IMPLIED>
<!ATTLIST label method CDATA #IMPLIED>
<!ATTLIST statement method CDATA #IMPLIED>
<!ATTLIST parameter method CDATA #IMPLIED>
<!ATTLIST exception method CDATA #IMPLIED>

<!-- <parameter ...><soot_parameter id="0"
                     type="mytype" name="myparam" /></parameter> -->
```

```
<!ELEMENT soot_parameter (use*)>
<!ATTLIST soot_parameter uses CDATA "&UNKNOWN_COUNT;">

<!-- <local ...><soot_local uses="1" defines="1" /></local> -->
<!ELEMENT soot_local (use*, definition*)>
<!ATTLIST soot_local uses CDATA "&UNKNOWN_COUNT;">
<!ATTLIST soot_local defines CDATA "&UNKNOWN_COUNT;">

<!-- <soot_local...><use id="0" line="0" /></local> -->
<!ELEMENT use EMPTY>
<!ATTLIST use id NMTOKEN #REQUIRED>
<!ATTLIST use line NMTOKEN #REQUIRED>
<!ATTLIST use method CDATA "&UNKNOWN;">

<!-- <local...><definition id="0" line="0" /></local> -->
<!ELEMENT definition EMPTY>
<!ATTLIST definition id NMTOKEN #REQUIRED>
<!ATTLIST definition line NMTOKEN #REQUIRED>
<!ATTLIST definition method CDATA "&UNKNOWN;">

<!-- <label id="0" name="mylabel" stmtcount="1" stmtpercentage="100" /> -->
<!ATTLIST label stmtcount CDATA "&UNKNOWN_COUNT;">
<!ATTLIST label stmtpercentage CDATA "&UNKNOWN_COUNT;">

<!-- <soot_statement ... branches="true/false" fallsthrough="true/false" /> -->
<!ELEMENT soot_statement (uses*, defines*, invoketargets*, livevariables?,
                                                          jimple?)>
<!ATTLIST soot_statement branches ( true | false ) #IMPLIED>
<!ATTLIST soot_statement fallsthrough ( true | false ) #IMPLIED>

<!-- <uses id="0" local="$r1" /> -->
<!ELEMENT uses EMPTY>
<!ATTLIST uses id NMTOKEN #REQUIRED>
<!ATTLIST uses local CDATA #REQUIRED>
<!ATTLIST uses method CDATA #IMPLIED>

<!-- <defines id="0" local="$r1" /> -->
<!ELEMENT defines EMPTY>
<!ATTLIST defines id NMTOKEN #REQUIRED>
<!ATTLIST defines local CDATA #REQUIRED>
<!ATTLIST defines method CDATA #IMPLIED>

<!-- <livevariables incount="1" outcount="1"><in /><out /></livevariables> -->
<!ELEMENT livevariables (in*, out*)>
<!ATTLIST livevariables incount NMTOKEN #REQUIRED>
<!ATTLIST livevariables outcount NMTOKEN #REQUIRED>

<!-- <in id="0" local="$r1" /> -->
```

```
<!ELEMENT in EMPTY>
<!ATTLIST in id NMTOKEN #REQUIRED>
<!ATTLIST in local CDATA #REQUIRED>
<!ATTLIST in method CDATA #IMPLIED>

<!-- <out id="0" local="$r2" /> -->
<!ELEMENT out EMPTY>
<!ATTLIST out id NMTOKEN #REQUIRED>
<!ATTLIST out local CDATA #REQUIRED>
<!ATTLIST out method CDATA #IMPLIED>

<!-- <jimple length="10"><![CDATA[ $r1 = $r2 ]]></jimple> -->
<!ELEMENT jimple (#PCDATA)>
<!ATTLIST jimple length NMTOKEN "&UNKNOWN_COUNT;">

<!-- <invoketargets analysis="CHA" count="1" /> -->
<!ELEMENT invoketargets (target*)>
<!ATTLIST invoketargets analysis NMTOKEN #REQUIRED>
<!ATTLIST invoketargets count NMTOKEN #REQUIRED>

<!-- <target id="0" class="myclass" method="mymethod" /> -->
<!ELEMENT target EMPTY>
<!ATTLIST target id NMTOKEN #REQUIRED>
<!ATTLIST target class CDATA #REQUIRED>
<!ATTLIST target method CDATA #REQUIRED>
```

A.3 STEP extensions

The following DTD is used to specify the extensions supported by the *STEP* profiling agent used in this framework.

```
<!-- <step version="1.0" /> -->
<!ELEMENT step EMPTY>
<!ATTLIST step version CDATA #FIXED "1.0">
<!ATTLIST step id NMTOKEN #REQUIRED>
<!ATTLIST step trace CDATA "&UNKNOWN;">

<!-- <method ...><step_method calls="1" /></method> -->
<!ELEMENT step_method (stepdeclaration, trace)>
<!ATTLIST step_method calls NMTOKEN #REQUIRED>

<!-- <stepdeclaration length="10"><![CDATA[ void main(java.lang.String[]) ]]>
                                  </stepdeclaration> -->
<!ELEMENT stepdeclaration (#PCDATA)>
```

```
<!ATTLIST stepdeclaration length NMTOKEN "&UNKNOWN_COUNT;">

<!-- <trace nesting="1"><entries count="1" /><exits count="1" /></trace> -->
<!ELEMENT trace (entries, exits)>
<!ATTLIST trace nesting NMTOKEN #REQUIRED>
<!ELEMENT entries EMPTY>
<!ATTLIST entries count NMTOKEN #REQUIRED>
<!ELEMENT exits EMPTY>
<!ATTLIST exits count NMTOKEN #REQUIRED>

<!-- <field ...><stoop_field /></field> -->
<!ELEMENT stoop_field (accesses)>

<!-- <accesses count="1"><reads /><writes /></accesses> -->
<!ELEMENT accesses (reads, writes)>
<!ATTLIST accesses count NMTOKEN "&UNKNOWN_COUNT;">

<!-- <reads count="1"><access /></reads> -->
<!ELEMENT reads (access*)>
<!ATTLIST reads count NMTOKEN "&UNKNOWN_COUNT;">

<!-- <writes count="1"><access /></writes> -->
<!ELEMENT writes (access*)>
<!ATTLIST writes count NMTOKEN "&UNKNOWN_COUNT;">

<!-- <access declaringtype="mytype" /> -->
<!ELEMENT access EMPTY>
<!ATTLIST access declaringtype CDATA #REQUIRED>
```

Appendix B
Samples

B.1 JIL document

Here is a sample Java class represented as a *JIL* document. This example is used to show the minimal elements required to represent a class.

```
<!-- declare this a JIL document using the DTD --> <!DOCTYPE jil
SYSTEM "jil10.dtd"> <jil>

  <history>
  </history>

  <!-- public MyClass implements MyInterface extends MySupperClass -->
  <class name="MyClass" extends="MySupperClass">
    <modifiers>
      <modifier name="public" />
    </modifiers>
    <interfaces>
      <interface name="myinterface" />
    </interfaces>

    <!-- public MyField -->
    <fields>
      <field id="0" name="MyField">
        <modifiers>
          <modifier name="public" />
        </modifiers>
```

```
    <!-- dynamic field accesses instrumented by STOOP -->
    <stoop_field>
      <accesses>
        <reads>
          <access declaringtype="MyType" />
        </reads>
        <writes>
          <access declaringtype="MyType" />
        </writes>
      </accesses>
    </stoop_field>

  </field>
</fields>

<!-- public void MyMethod( MyType MyParam ) -->
<methods>
  <method name="MyMethod" class="MyClass">

    <declaration>
      <![CDATA[ public void MyMethod( MyType MyParam ) ]]>
    </declaration>

    <parameters>
      <parameter id="0" name="MyParam">
        <soot_parameter uses="1">
          <use id="0" line="123" />
        </soot_parameter>
      </parameter>
    </parameters>

    <!-- MyType MyLocal -->
    <locals>
      <local id="0" name="MyLocal">

        <!-- static local uses instrumented by SOOT -->
        <soot_local>
          <use id="0" line="1" />
          <definition id="0" line="1" />
        </soot_local>

      </local>

      <!-- locals by type -->
      <types count="1">
        <type id="0" count="1" type="MyType">
          <local id="0" name="MyLocal">
            <soot_local>
```

```
            <use id="0" line="123" />
            <definition id="0" line="123" />
          </soot_local>
        </local>
      </type>
    </types>

</locals>

<!-- labels -->
<labels>
  <label id="0" name="MyLabel" />
</labels>

<!-- statements -->
<statements>
  <statement id="0" label="MyLabel">

    <!-- static per-statement analysis results instrumentated by SOOT -->
    <soot_statement>
      <uses id="0" local="MyLocal" />
      <defines id="0" local="MyLocal" />
      <livevariables incount="1" outcount="1">
        <in id="0" local="MyLocal" />
        <out id="0" local="MyLocal" />
      </livevariables>
      <jimple>
        <![CDATA[ $l5 = virtualinvoke $r13.
                  <java.lang.Long: long longValue()>(); ]]>
      </jimple>
    </soot_statement>

    </statement>
  </statements>

  <!-- exceptions -->
  <exceptions>
    <exception id="0" type="MyException">
      <begin label="MyLabel" />
      <end label="MyLabel" />
      <handler label="MyLabel" />
    </exception>
  </exceptions>

  <!-- dynamic method calls instrumented by STOOP -->
  <stoop_method calls="1">
    <stepdeclaration>
      <![CDATA[ void MyMethod( MyType ) ]]>
```

```
          </stepdeclaration>
          <trace nesting="1">
            <entries count="1" />
            <exits count="1" />
          </trace>
        </stoop_method>

      </method>
    </methods>
  </class>

</jil>
```

B.2 Hello world example

The following section demonstrates the differences between the Java source, the Jimple intermediate representation, and the *JIL* representation of a Java class. In order to conserve space within this document, a simple 'Hello world' program is used as an example. Although the Java and Jimple source code for the program are relatively small, the *JIL* representation becomes quite verbose. Here are the relative number of lines in each source listing:

- Java source: 7 lines.
- *Jimple* source: 23 lines, a 229% increase from the Java source.
- *JIL* document: 151 lines, a 557% increase from the *Jimple* source, and a 2,057% increase from the Java source.

B.2.1 Java source

The Java source for the 'Hello world' program consists of a class declaration and a `main` method containing a single call to `println`.

```
public class myHelloWorld
{
    public static void main( String[] args )
```

```
    {
        System.out.println( "Hello world!" );
    }
}
```

B.2.2 Jimple source

Once the Java source is compiled into a class file using a Java compiler, we used *SOOT* in order to generate the *Jimple* representation of the class. This representation includes the class initialization routine `<init>` which is included automatically by the Java compiler, even if it is not explicitly defined in the Java source. The *Jimple* source for a class will typically contain additional lines of code since it includes *Jimple* statements for code elements, such as temporary variables, inserted by the Java compiler.

```
public class myHelloWorld extends java.lang.Object
{

    public void <init>()
    {
        myHelloWorld r0;

        r0 := @this: myHelloWorld;
        specialinvoke r0.<java.lang.Object: void <init>()>();
        return;
    }

    public static void main(java.lang.String[])
    {
        java.lang.String[] r0;
        java.io.PrintStream $r1;

        r0 := @parameter0: java.lang.String[];
        $r1 = <java.lang.System: java.io.PrintStream out>;
        virtualinvoke $r1.<java.io.PrintStream: void println(java.lang.String)>
          ("Hello world!");
        return;
    }
}
```

B.2.3 JIL document

Following the steps in Appendix C.1, the following *JIL* document was generated using *SOOT*. With the extra text required to form markup elements and attributes, the document becomes much more verbose than the Java or *Jimple* sources.

```
<?xml version="1.0" ?>
<!DOCTYPE jil SYSTEM "http://www.sable.mcgill.ca/jil/jil10.dtd">
<jil>
  <history created="Thu Aug 22 13:15:36 EDT 2002">
    <soot version="1.2.3 (build 1.2.3.dev.4)" command="--xml myHelloWorld"
          timestamp="Thu Aug 22 13:15:36 EDT 2002" />
  </history>
  <class name="myHelloWorld" extends="java.lang.Object">
    <modifiers count="1">
      <modifier name="public" />
    </modifiers>
    <interfaces count="0" />
    <fields count="0" />
    <methods count="2">
      <method name="_init_" returntype="void" class="myHelloWorld">
        <declaration length="20"><![CDATA[public void <init>()]]></declaration>
        <parameters method="_init_" count="0" />
        <locals count="1">
          <local id="0" method="_init_" name="r0" type="myHelloWorld">
            <soot_local uses="1" defines="1">
              <use id="0" line="1" method="_init_" />
              <definition id="0" line="0" method="_init_" />
            </soot_local>
          </local>
          <types count="1">
            <type id="0" type="myHelloWorld" count="1">
              <local id="0" method="_init_" name="r0" type="myHelloWorld">
                <soot_local uses="1" defines="1">
                  <use id="0" line="1" method="_init_" />
                  <definition id="0" line="0" method="_init_" />
                </soot_local>
              </local>
            </type>
          </types>
        </locals>
        <labels count="1">
          <label id="0" name="default" method="_init_" stmtcount="3"
                 stmtpercentage="100" />
        </labels>
```

```
  <statements count="3">
    <statement id="0" label="default" method="_init_" labelid="0">
      <soot_statement branches="false" fallsthrough="true">
        <defines id="0" local="r0" method="_init_" />
        <livevariables incount="0" outcount="1">
          <out id="0" local="r0" method="_init_" />
        </livevariables>
        <jimple length="26"><![CDATA[r0 := @this: myHelloWorld]]></jimple>
      </soot_statement>
    </statement>
    <statement id="1" label="default" method="_init_" labelid="1">
      <soot_statement branches="false" fallsthrough="true">
        <uses id="0" local="r0" method="_init_" />
        <invoketargets analysis="CHA" count="1">
          <target id="0" class="java.lang.Object" method="_init_" />
        </invoketargets>
        <livevariables incount="1" outcount="0">
          <in id="0" local="r0" method="_init_" />
        </livevariables>
        <jimple length="53"><![CDATA[specialinvoke r0.<java.lang.Object:
                          void <init>()>()]]></jimple>
      </soot_statement>
    </statement>
    <statement id="2" label="default" method="_init_" labelid="2">
      <soot_statement branches="false" fallsthrough="false">
        <livevariables incount="0" outcount="0" />
        <jimple length="7"><![CDATA[return]]></jimple>
      </soot_statement>
    </statement>
  </statements>
  <exceptions count="0" />
</method>
<method name="main" returntype="void" class="myHelloWorld">
  <declaration length="44"><![CDATA[public static void main
                                  (java.lang.String[] )]]></declaration>
  <parameters method="main" count="1">
    <parameter id="0" type="java.lang.String[]" method="main"
               name="_parameter0">
      <soot_parameter uses="1">
        <use id="0" line="0" method="main" />
      </soot_parameter>
    </parameter>
  </parameters>
  <locals count="2">
    <local id="0" method="main" name="r0" type="java.lang.String[]">
      <soot_local uses="0" defines="1">
        <definition id="0" line="0" method="main" />
      </soot_local>
```

```
  </local>
  <local id="1" method="main" name="$r1" type="java.io.PrintStream">
    <soot_local uses="1" defines="1">
      <use id="0" line="2" method="main" />
      <definition id="0" line="1" method="main" />
    </soot_local>
  </local>
  <types count="2">
    <type id="0" type="java.lang.String[]" count="1">
      <local id="0" method="main" name="r0" type="java.lang.String[]">
        <soot_local uses="0" defines="1">
          <definition id="0" line="0" method="main" />
        </soot_local>
      </local>
    </type>
    <type id="1" type="java.io.PrintStream" count="1">
      <local id="1" method="main" name="$r1" type="java.io.PrintStream">
        <soot_local uses="1" defines="1">
          <use id="0" line="2" method="main" />
          <definition id="0" line="1" method="main" />
        </soot_local>
      </local>
    </type>
  </types>
</locals>
<labels count="1">
  <label id="0" name="default" method="main" stmtcount="4"
         stmtpercentage="100" />
</labels>
<statements count="4">
  <statement id="0" label="default" method="main" labelid="0">
    <soot_statement branches="false" fallsthrough="true">
      <defines id="0" local="r0" method="main" />
      <livevariables incount="0" outcount="0" />
      <jimple length="38"><![CDATA[r0 := @parameter0:
                                  java.lang.String[]]]></jimple>
    </soot_statement>
  </statement>
  <statement id="1" label="default" method="main" labelid="1">
    <soot_statement branches="false" fallsthrough="true">
      <defines id="0" local="$r1" method="main" />
      <livevariables incount="0" outcount="1">
        <out id="0" local="$r1" method="main" />
      </livevariables>
      <jimple length="50"><![CDATA[$r1 = <java.lang.System:
                                  java.io.PrintStream out>]]></jimple>
    </soot_statement>
  </statement>
```

```
        <statement id="2" label="default" method="main" labelid="2">
          <soot_statement branches="false" fallsthrough="true">
            <uses id="0" local="$r1" method="main" />
            <invoketargets analysis="CHA" count="1">
              <target id="0" class="java.io.PrintStream" method="println" />
            </invoketargets>
            <livevariables incount="1" outcount="0">
              <in id="0" local="$r1" method="main" />
            </livevariables>
            <jimple length="88"><![CDATA[virtualinvoke $r1.<java.io.PrintStream:
                                 void println(java.lang.String)>("Hello world!")]]>
                                 </jimple>
          </soot_statement>
        </statement>
        <statement id="3" label="default" method="main" labelid="3">
          <soot_statement branches="false" fallsthrough="false">
            <livevariables incount="0" outcount="0" />
            <jimple length="7"><![CDATA[return]]></jimple>
          </soot_statement>
        </statement>
      </statements>
      <exceptions count="0" />
    </method>
  </methods>
 </class>
</jil>
```

B.3 JIMPLEX

In order to demonstrate how the *JIMPLEX* visualization interface described in Section 5.2 separates the data from the visualizer, the *XSLT* source is provided online on the *JIL* web page found in Table 6.2. The *XSLT* source was too extensive to provide a listing directly in this appendix, however we will outline an example of how *XSLT* is used to transform *JIL* data.

Those readers who are unfamiliar with *XSLT* but have experimented with web page design will notice that most of the content of *XSLT* is HTML. *XSLT* describes a web page in a programmatic way, allowing content to be structured and stylized using general templates. *XSLT* is useless by itself, and requires an XML data source in order to describe a web page. This is why *XSLT* is said to transform XML into a human-readable format.

Let us consider an example of how parameters are stored within a *JIL* document. The following *JIL* fragment includes two parameters with different types.

```
<parameters method="myMethod" count="2">
  <parameter id="0" type="java.lang.String[]" name="_parameter0" />
  <parameter id="1" type="java.lang.Long" name="_parameter1" />
</parameters>
```

Now, consider the following *XSLT* segment which creates an HTML table with two columns labelled `class` and `name`. It enumerates the parameters and creates a row for each, printing the type and name of each parameter in their respective columns. Note that although only enough HTML code is included to create one row, the *for-each* command will enumerate through each parameter, copying the HTML code for each parameter in the *JIL* data source. The *value-of* command is replaced with the value of the named attribute. If an additional attribute were to be added to the parameters in the *JIL* document, the *XSLT* code could be updated to include the new data with minimal changes.

```
<table>
<!-- parameter column headings -->
<tr>
  <td>class</td>
  <td>name</td>
</tr>
<!-- create a row for each parameter -->
<xsl:for-each select="parameters/parameter">
<tr>
  <!-- echo the type attribute of the parameter -->
  <td><xsl:value-of select="@type" /></td>
  <!-- echo the name attribute of the parameter -->
  <td><xsl:value-of select="@name" /></td>
</tr>
</xsl:for-each>
</table>
```

After the transformation process, the following HTML would be the result; this is the code which would be generated dynamically each time a web browser accessed the interface. Note that comments have been added in order to explain the code, and would not normally be included by the *XSLT* processor.

```
<table>
<!-- parameter column headings -->
<tr>
  <td>class</td>
  <td>name</td>
</tr>
<!-- row for the first parameter -->
<tr>
  <!-- echo the type attribute of the 1st parameter -->
  <td>java.lang.String[]</td>
  <!-- echo the name attribute of the 1st parameter -->
  <td>_parameter0</td>
</tr>
<!-- row for the second parameter -->
<tr>
  <!-- echo the type attribute of the 2nd parameter -->
  <td>java.lang.Long</td>
  <!-- echo the name attribute of the 2nd parameter -->
  <td>_parameter1</td>
</tr>
</table>
```

Appendix C
How To's

C.1 Generating JIL with SOOT

The following information is based on the *JIL* support in version 1.2.3 of *SOOT*. For the most current syntax of commands, please refer to the command line help of the version of *SOOT* you are working with. Complete documentation and tutorials on the use of *SOOT* can be found online at the URL in Table 6.2.

In order to generate *JIL* with *SOOT*, it is as simple as selecting an alternative XML output format. This is done in the same way *Jimple* is selected as an output format, but by using the `--xml` command line argument instead of `--jimp`. For the following examples we will assume your Java source `MyClass.java` has been compiled into a Java class `MyClass.class` by `javac`.

The following command will generate the *JIL* representation of `MyClass.class` as `MyClass.xml`:

```
> java soot.Main --xml MyClass
```

C.2 Generating JIL with STEP

At the time of the writing of this thesis, the *STEP* framework was not at a release state where we could describe the commands to generate *JIL* documents. In order to learn

more about the current state of *STEP* and for all related papers and documentation, please visit the *STEP* web site found in Table 6.2.

C.3 Visualizing JIL

The visualization interfaces implemented as part of this thesis include several online tools, include the *JIMPLEX* visualizer described in Section 5.2. These tools were designed to be accessible and compatible with any computer connected to the Internet and equipped with a modern web browser. These tools are available from the *JIL* web site [20].

In order to visualize a *JIL* document using an online visualizer, the document must be accessible over the Internet. This can be achieved by placing the *JIL* file in a web accessible folder so that the file can be referenced using an URL. This URL is then used in the online tools in order to access the document remotely.