JIL: an Extensible Intermediate Language

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

ABSTRACT

The Java Intermediate Language (JIL) is a subset of XML and SGML described in this document. Its goal is to provide an intermediate representation of Java source code suitable for machine use. JIL benefits from the features of XML, such as extensibility and portability, while providing a common ground for software tools. The following document discusses the design issues and overall framework for such a language, including a description of all fundamental elements.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *classes and objects, constraints, data types and structures, frameworks.*

General Terms

Design, Experimentation, Standardization, Languages.

Keywords

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

1. INTRODUCTION

Java Intermediate Language describes a restricted form of XML, the Extensible Markup Language [1]. It describes a class of documents which represent an intermediate representation of Java source code [3], suitable for use somewhere between the programmer and the executing operating system.

JIL documents are constructed from markup tags which contain textual data. Markup tags encode the documents storage layout and logical structure, applying constraints on a standard XML document. Every JIL document is a compliant XML document, and the W3C recommendation for XML 1.0 [1] can be used as a formal reference for the underlying syntax and document requirements.

1.1 Origin and Goals

JIL was developed as an alternate representation of an intermediate language (IL) used in an optimizing Java compiler. The extensible nature of the format allowed the source code to be annotated with analysis results and even runtime data. This representation provided a common format for interoperability, bridging the gaps between existing tools.

The design goals for JIL are:

1. JIL shall be strictly defined, but easily extensible.
2. JIL shall be supported across platforms and networks.
3. JIL shall be compatible with XML, SGML and related tools.
4. JIL shall be easy to parse and generate.

These goals do not assume a particular use for JIL, but suggest an open format that could be used in many different environments. The base language and extensions presented in this document are suited towards code understanding, optimization, and profiling.

2. OVERVIEW

The following sections will describe the design of the Java Intermediate Language from several different, but intersecting points of view.

2.1 JIL as a Java IL

Intermediate languages are widely used to provide an appropriate representation of Java for a specific process or analysis. The design of these kinds of languages is most commonly based on the application, either by convenience or in order to optimize the format for that particular task. This results in an abundance of languages for each individual task with subtly different semantics. JIL was designed to encapsulate much of the information provided by these intermediate languages, making it suitable for various applications.

In order to understand the types of data associated with an intermediate language we will examine the lifetime of Java source code.

```
[ MyApp.java ]  => [ javac ] => [ MyApp.class ]
```

Given a source file, the Java compiler creates Java bytecode in the form of a class file. Once compiled into bytecode, the source has taken a platform independent form which can be executed by any Java Virtual Machine (JVM). Java bytecode is one of the first Java ILs, as it is the final representation of the source code before it is passed to the JVM interpreter and executed.

```
[ MyApp.class ] => [ JVM ] => [ MyApp ]
```

Optimizing compilers can operate directly on these class files, making them the initial and target representation for such applications. JIL is not intended to replace Java bytecode, but to aide in its analysis, optimization, transformation, and visualization.

```
[ MyApp.class ] <=> [ Optimizing Compiler ]
```

Within the compiler, different intermediate representations can be used, each defining its own semantics but describing the same source object. JIL is designed to encapsulate each representation, along with any associated data extracted by the compiler.

2.1.1 Base Intermediate Language Constructs

As a common IL, JIL contains many code elements which are shared across ILs targeting Java bytecode. These elements form the framework of a class, as they provide a structure upon which extensions are applied. Every JIL document contains this framework of base elements, which is specified using a Document Type Definition (DTD).

The DTD specification enforces which elements and attributes are included in a JIL document, what they are allowed to contain, as well as their logical structure. Given a DTD, a validating parser can identify any inconsistencies in a JIL document where it does not follow the specification. These errors can result in the document being rejected by the parser, or in some cases they can be repaired or ignored. The DTD follows the premise that anything not specified is forbidden, enforcing constraints on what XML documents can be considered a valid JIL document. By using DTDs for validation, JIL-aware applications can ensure that they are generating or parsing documents that will be recognized and understood by other applications.

2.1.2 Language Extensions

One of the key features of JIL is its extensibility. JIL allows 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 code elements which would normally be hidden.

Language extensions are specified using additional DTDs which are included by the base definition. An application acting as a JIL generator must produce a documents which comply to the base DTD and any extension DTDs that it supports. The most common method of enforcing this requirement is by passing any documents to a simple DTD validator once they are generated. DTD provide are a fast and robust grammar which make extensions easy to specify.

Generators self describe their extensions by adding an identity element to the document’s header. Identity elements allow a generator to recognize its own work by logging the command or action which resulted in the added extensions. The document header provides a history of all contributing generators in order to allow a document consumer to identify what extensions to expect.

2.1.3 Static IL Extensions

A typical extension found in a JIL document might be the live variables associated with a statement of IL. This is static data which can be collected at compile-time and associated directly to the code. Using JIL this extension would be expressed as annotations applied to each statement, providing the list of variables which are live coming in and out of that statement.

A JIL document augmented with such static data now provides a consumer with all the content of the original IL along with the results of a static intra-procedural analysis. This kind of information can provide insight to a developer or subsequent JIL consumer working with either the original source code or the encapsulated IL.

An ideal generator for static JIL extensions is the SOOT framework for optimizing Java bytecode [8]. Support for JIL was recently added to SOOT as an output format. SOOT is able to perform various analyses directly to Java bytecode, and the extended data it persists in its JIL output is a rendering of data it uses internally to perform optimizations and transformations. This data aides the debugging and development of new optimizations and analyses.

2.1.4 Language Extensions

Dynamic data describes the kinds of information which can vary for the same code in different execution or processing environments. Dynamic IL extensions are typically collected at runtime in order to benchmark or profile code for a complete description of its behavior. This kind of data is rarely associated with low level code elements allowing programmers to easily ignore the runtime behavior of their code. Investigation into such details is usually triggered only when searching for a bug or optimization.

The Sable Toolkit for Object-Oriented Profiling (STOOP) is a typical source of dynamic data [2]. STOOP provides a framework for building custom profilers which can collect runtime data on almost any aspect of programs written in Java. This data is collected by profiling agents and then passed through an event pipe and on to a visualizer. We provide a backend which can consume data events from the pipe and produce compliant JIL with profiling extensions.

Benchmarking data is another runtime characteristic of code which can be stored in JIL documents. Data can be associated with any element at any level in the hierarchy, making JIL a comprehensive benchmarking format. The kinds of benchmarking information can vary from general timings to hotspot counters. JIL provides a format where this information can be associated directly to the code elements.

2.2 JIL as XML

JIL exploits many of the natural features of XML. The use of XML tags in JIL is very straightforward since JIL is simply a well formed XML document. As an XML document, JIL takes advantage of the extensibility and hierarchical structure of XML. The following sections will describe JIL as an XML application

2.2.1 Features

XML is a universal language for describing a structured format which is widely used in many applications. JIL exploits many of the features of XML:

- JIL is human readable and editable using text editor, which aides debugging.
- JIL is easy to generate and parse, encouraging the development of tools and good reliability and performance.
- JIL is modular and manageable through schemas and basic processing.
- JIL is portable across languages, platforms, and networks.

XML is also license-free, making it a widely used format with support in many popular packages:

- JIL can be browsed on a client using Internet Explorer, Netscape, and Opera.
- JIL can be served as a native database using Microsoft SQL Server 2000 or Oracle 8i.

- Programming APIs are available in C, C++, Java, Perl, Python, COM etc.

XML also has some disadvantages which it passes on to JIL. For example, JIL is extremely verbose, and a corresponding JIL document will typically be much larger than the source code it resulted from. However, the cost of disk space and the current state of compression algorithms for both storage and network transfer trivialize this disadvantage. XML is not always the best choice for an application, but in the case of JIL it's features cover most of the design goals.

2.2.2 Structure

JIL exploits the natural hierarchy and nesting of XML for describing the structure of code elements and extensions. By nesting elements according to a specified framework 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.

2.2.3 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 where possible to annotate and describe objects, since they 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.

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

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.

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

2.2.4 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 examples in this document in order to save space and highlight the other markup being demonstrated.

2.2.5 Extensions

JIL is naturally extensible, allowing any element to be annotated with additional data. These annotations are associated and defined by a generator, so that a compliant JIL consumer which supports these annotations knows what to expect when parsing the document. A JIL generator will typically be accompanied by a corresponding DTD. Supported extensions are then defined in this additional DTD which is referenced by the base DTD when a document is validated.

```
<statement>
  <stoop_statement>
     …
  </stoop_statement>
</statement>
```

Extension elements are typically named by taking the extended element's name preceded by the extending generator and an underscore. Generators can extend any element defined in the base DTD, including attributes of existing elements.

2.3 JIL as Storage

JIL provides physical and logical storage for both static and dynamic data. The following sections will discuss the creation, management, and processing of JIL as a source of data.

2.3.1 Creating JIL

JIL requires no special encoding and can be created by hand using a common text editor. This makes debugging JIL documents and prototyping new elements or extensions quick and easy. This also makes JIL generation easy to implement using standard libraries.

Applications which generate JIL documents can also do so using some of the many programming APIs available for every major language. These APIs provide a quick and easy way to generate compliant JIL without having to worry about implementation details.

Generated documents should be validated using the JIL Document Type Definition. This ensures that the documents contain all required elements, as well as identifying any unsupported elements or attributes. DTD validation helps debug JIL generation, and is also supported programmatically in most XML APIs. Support has been recently added to SOOT to support JIL generation, making it the first bytecode to JIL converter which complies with the JIL DTD.

2.3.2 Managing Multiple Documents

Java applications typically consist of several classes organized into a hierarchy. This object-oriented design is mimicked by the organization of JIL documents. However multiple JIL documents can exist for a single class file by including different extensions in each. The ability to include dynamic data also means that even though the same extensions are used, they can contain different data resulting from several runtime environments or cases.

JIL documents self-describe the extensions they contain using header markup which is specified in the base JIL definition. This markup comes in the form of a history list of all contributing generators. Each generator which has contributed markup to the JIL document signs the document with its own identity tag which indicates a time stamp as well as the action or command it performed.

2.3.3 JIL as a Data Source

XML has been used as a data source in many different scenarios in the past few years. 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.

Several programming models exist for consuming XML data such as JIL, some which are optimized to save memory while others are suited towards repeated processing of random elements. Developers will have a rich library of APIs and tools to choose from, which will continue to grow.

3. DOCUMENTS

A JIL document represents a single source code object, such as a Java class. Each document begins with some header tags for XML compliance and self-description, and can only contain those elements defined in the JIL Document Type Definition including any supported extensions.

3.1 JIL as a Java Class

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

3.1.1 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.

3.1.2 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.

3.1.3 XML Declaration

Since every XIL document is a valid XML document, it must begin with appropriate XML declaration tag. Refer to the XML specification for extended syntax information.

```
<?xml version”1.0” ?>
```

3.1.4 JIL Declaration

JIL documents will have 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” />
```

3.1.5 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 was 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=”1.2.2” cmd=”–X MyClass”>
  <stoop version=”1.0” mode=”field-accesses”>
</history>
```

3.2 Classes

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” />
```

3.2.1 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.

3.2.2 Interfaces

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

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

3.2.3 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.

3.3 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>
```

3.3.1 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 can be found in the base JIL DTD.

3.3.2 Extensions

Field extensions apply to each field, and are typically partnered with an associated extension to statements.

```
<field>
  <generator_field>
   …
  <generator_field>
</field>
```

JIL documents generated by the JIL backend for STOOP support a profiling mode which records field reads and writes. These counts, and any other profiling data provided by STOOP which applies to each field, are attached to each field through the use of a *stoop_field* tag.

3.4 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>
```

3.4.1 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>
```

3.4.2 Parameters

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

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

3.4.3 Extensions

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

```
<method>
  <generator_method>
   …
  </generator_method>
</method>
```

SOOT 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, however this association is defined internally within the generators and consumers supporting this extension.

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

3.5 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>
```

3.5.1 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>
```

3.5.2 Extensions

Extensions to locals are stored using the standard notation.

```
<local>
  <generator_local>
   …
  </generator_local>
</local>
```

The JIL generated by SOOT contains local extensions which indicate 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>
```

3.6 Labels

Labels are used in Java bytecode to indicate basic blocks of code which can be used as targets for branch operations. Every statement must be associated to a label, and in JIL this association is stored in each statement.

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

3.7 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>
```

3.7.1 Extensions

Statement extensions associate date to each individual statement.

```
<statement>
  <generator_statement>
   …
  </generator_statement>
</statement>
```

SOOT extends 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>
```

3.8 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>
```

4. DISCUSSION

The following sections discuss the language presented in this paper, with respect to the original design goals, as well as related and future work.

4.1 Related Work

Much work has been done towards the design of intermediate languages, however the design goals of these languages are usually driven by a particular application.

4.1.1 Language Extensions

Compilers typically use intermediate languages internally as a specialized format for efficient processing. Typed intermediate languages have received much interest for their ability to preserve type information throughout the compilation process [5], [6]. Compilers can use this information to guide optimizations and generate strongly typed code. The cost of processing type information at such a low-level can be offset by caching it within an extensible language.

4.1.2 Interoperability

Runtime systems which use a common IL have also been able to provide type-safe JIT compilation, as well as providing language and platform independence [7]. Some ILs are borrowing language features directly from high-level programming languages such as Java, so that they can be executed and debugged using commonly available tools [4]. JIL takes a different approach by not assuming anything about the tools which produce and consume it. As a result, JIL provides a format which is able to encapsulate many of features found in typical ILs.

4.2 Future Work

JIL represents an effort to consolidate the work that goes into the design, generation, and processing of intermediate languages for Java. This work is ongoing, and new analyses and transformations are constantly being developed. These analyses can produce new information about source code which will be beyond the scope of current ILs. JIL provides a language which can be extended in parallel with tool development, so that data can be quickly visualized and shared with existing tools. With the proper support, JIL can help developers fine tune their code and tools with an expandable, object-oriented framework.

4.3 Availability

The proposed specification of JIL and related extensions are available online as Document Type Definitions at this web site:

`http://www.sable.mcgill.ca/~flynn/jil/`

5. REFERENCES

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

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

[3] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1997.

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

[5] S. L. P. Jones and E. Meijer. *Henk: A typed intermediate language.* TIC 1997.

[6] G. Morrisett, K. Crary, N. Glew, and D. Walker. *Stack-based typed assembly language.* ACM Workshop on Types in Compilation, 1998.

[7] D. Syme. *ILX: Extending the .NET Common IL for Functional Language Interoperability.* BABEL 2001.

[8] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. *SOOT: a Java bytecode optimization framework.* CASCON 1999.