Java Instrumentation Engine

REVISED DESIGN DOCUMENT

Eran Tromer <eran@tromer.org>

for Shmuel Ur, IBM Haifa Research Lab
as part of Course 236705
Technion, Israel Institute of Technology

1999/11/18

Table of Content

The Java Instrumentation Engine: Overview 5
Instrumentation Configuration 6
2.1 Overview 6
2.2 Opportunities 7
2.3 Actions 9
2.4 Generic Context Parameters 10
2.5 Generic Static filters 11
2.6 Generic Dynamic Filters 11
2.7 Global Parameters 11
2.8 JIE instrumentation IDs 11
2.9 XML Representation 12
The Java Parser Component 13
3.1 Overview 13
3.2 Parsing Tool Evaluation Criteria 13
3.3 Candidate Parsing Tools 14
3.4 Candidate Parsing Tools 14
3.4.1 JavaCC / Metamata Parse 14
3.4.2 JJTree 14
3.4.3 JTB 15
3.4.4 SableCC 15
3.4.5 ANTLR 16
3.5 Comparison Table 16
3.6 Choice of Parsing Tool 17
GUI Driver 19
Appendix A: Instrumentation Opportunities 24
Appendix B: Instrumentation Configuration DTD 32
Appendix C: Parsing Tool License Agreements 33
7.1 JavaCC (excerpts) 35
7.2 JTB 35
7.3 SableCC (excerpts) 36
Appendix D: Implementation Matrix 37
Bibliograpgy 39

Abstract

The Java Instrumentation Engine (JIE) is a general-purpose facility for sourcecode instrumentation of Java programs. This document provides a high-level specification of JIE, including capabilities, interfaces and key implementation issues.

Chapter 1 provides an overview of JIE, including description of uses and a schematic division into components.

Chapter 2 describes the capabilities of JIE and how these capabilities are used. It introduces the concept of instrumentation configuration – a list of rules specifying what sourcecode transformations to apply. In combination with the relevant appendices, this chapter defines the core functionality of JIE. Appendices A, B and C add details pertaining to this chapter.

Chapter 3 discusses one major component of JIE, the Java sourcecode parsing tool. This component is to be based on an externally available tool. Requirements and evaluation criteria are listed and candidates are evaluated. Appendix D provides highlights from their license agreements.

Chapter 4 depicts a design for a GUI-based interface to JIE, using screenshots of a UI prototype.

Appendix E specifies a proposal for a functionality subset that is appropriate as a current implementation goal.

What's New Since "Design Stage: Revised Design Document" (1999/08/19)

Minor changes in the transformation of the *Class Instantiation* instrumentation opportunity to create cleaner code if only an Exit action is defined (no Entry action).

Changes in the transformation of the *Nominal Basic Block* opportunity.

Noted that conveniently, Java reserves identifier names beginning with '`$`' for use of source processing tools such as JIE.

Generic context parameters: defined class name and identifiers for inner classes (including anonymous). Defined method identifiers for constructors, static initializers and instance initializers.

Reflected availability of JavaCC 1.1 (first release since JavaCC 0.8pre2, which was previously evaluated).

Added the ability to process a single file, in addition to a directory tree.

Minor changes to the XML configuration example.

In the Instrumentation Opportunities table, noted that the documented transformations are only a proof-of-concept. The actual transformations carried out by JIE are quite different, especially for *Nominal Basic Block*.

What's New Since "Design Stage: Final Report" (1999/03/19)

The names of several generic context parameters were changed to improve clarity and compliance with the Java Language Specification terminology.

Documented handling of nested and anonymous classes in generic context parameters.

The format of JIID changed to use 13 Base32 characters instead of 12 Base64 characters, and underlines instead of periods.

The transformation *Nominal Method* instrumentation opportunity was changed for compatibility with the annotation-based transformation implementation.

The proposed transformations *Field Read* and *Field Write* opportunities are incorrect. It is possible to perform them without global type information, but it's more complex than previously depicted. They are currently left greyed-out.

Minor change in the parsing of the *Logging Statement* global parameter.

1 The Java Instrumentation Engine: Overview

The goal of this project is construction of a general-purpose facility for sourcecode instrumentation of Java programs. In its basic mode of operation, the Java Instrumentation Engine (JIE) receives a Java source file and instrumentation instructions, and emits appropriately transformed Java sourcecode.

Sourcecode instrumentation is automated global modification of the sourcecode to include additional operations, usually for testing and analysis purposes. Uses include:

- Profiling (e.g., bottleneck identification)
- Execution logs (e.g., for off-site support)
- Debugging of time-sensitive systems or conditions
- White-box testing
- Coverage analysis
- Addition of error checking and error handling code

Sourcecode instrumentation normally does not affect program semantics – the typical instrumentation code merely reports some state information to an external agent. However, this is not necessarily the case (e.g., addition of error checking code).

The Java Instrumentation Engine is intended to provide a flexible and extendible tool for performing all but the most specialized instrumentation chores. At its current incarnation it is specific to the Java language, and is written in Pure Java code. Also, at this stage generality, clarity and rapid development take precedence over performance issues.

Schematically, JIE is composed of the following components:

1. Library of instrumentation opportunities
2. Instrumentation configuration, specifying what to do at each instrumentation opportunity
3. Input handling: Java sourcecode parser
4. Output handling: Java sourcecode emitter
5. Main driver: Invokes the parser to build an internal representation of the input, performs the transformation according to the instrumentation configuration, and invokes the emitter to produce output.
6. Command-line application
7. GUI-based driver application

Components 1, 2, 3, 6 and 7 are discussed in detail in this document. Components 4 and 5 consist of implementation details that do not form a part of this high-level specification.

2 Instrumentation Configuration

2.1 Overview

JIE performs instrumentation in a batch process based on an *instrumentation configuration.* An instrumentation configuration consists of:

- Global parameters (e.g., input and output paths)
- Global filters
- Instrumentation rules

An *instrumentation rule* determines the *instrumentation points* – portions of the sourcecode that will undergo *instrumentation transformation*. For each such instrumentation point, it determines what transformation will be applied. Code inserted by JIE during such transformation is called *instrumentation code.* An instrumentation rule is composed of four parts:

A. Name
B. Opportunity
C. Action
D. Static filters
E. Dynamic filters

A. For readability purposes, each rule can be assigned a name. Rule names are not necessarily unique.

B. The instrumentation *opportunity* determines which code locations are candidates for instrumentation, and what is the general nature of the transformation they will undergo to insert the instrumentation code. Examples include entry into method, entry into basic block and throwing of an exception.

C. The instrumentation *action* determines what the instrumentation code does. Common examples include writing trace data to a file and calling some methods. The general case is using an *action template*, which is an arbitrary string (usually a Java statement) that is textually inserted at the appropriate location, as defined by the instrumentation opportunity.

Action templates can include macros that expand to *context parameters*. For example, the following action template:

```
logFile.write(${line}, "${method}", "Impossible Event Occured");
```

might be expanded to:

```
logFile.write(523, "il.ac.technion.cs.sandwichMachine.fill",
"Impossible Event Occured");
```

Some context parameters, such as the method name and line number used above, are always available. These are called *generic context parameters*. Other context parameters are specific to certain instrumentation opportunities – for instance, the class name in the C*lass Instantiation* opportunity.

D. *Static filters* limit the set of actual instrumentation points to a subset of all the code constructs that match the opportunity. Each such filter provides a limiting criterion, such as: package name, class name, class modifiers and method visibility. They are called "static" because they determine where instrumentation code will be inserted at instrumentation time – where a static filter criterion is not matched, no code is inserted.

E. *Dynamic filters* are similar to static filters in that they determine whether the instrumentation action is executed in specific cases. These criteria are, however, evaluated at run-time. This means that the

instrumentation code is always inserted (assuming the static filters are matched), and additional code is inserted to control whether the action takes place or not every time the instrumentation point is reached.

Both static and dynamic filters may be generic to all opportunities or specific to certain opportunities. All the dynamic filters currently defined are generic.

Generic static filters can be used both inside instrumentation rules and as top-level *global filters*. Global filters are applied to all the rules in the instrumentation configurations, in addition to the rule-specific filters.

Both static and dynamic filter types are combined using a logical "AND" operation. Facilities for combining filters using "OR" and "NOT" and for building nested conditions may be added in the future.

Each instrumentation point is assigned an *JIE instrumentation ID (JIID)*. A JIID can be used to uniquely identify an instrumentation point in the context of a given set of sourcecode files.

Persistent storage and of instrumentation configurations is done via a custom XML DTD, using an externally available XML processor.

2.2 Opportunities

JIE provides a library of instrumentation opportunities that can be used to define instrumentation rules. This instrumentation opportunity library essentially defines the core capabilities of JIE.

The selection criteria for opportunities are usefulness and feasibility. First, opportunities were chosen based on expected uses for JIE and inspection of related tools. Some opportunities, such as *Method*, emerged as absolutely essential. Others, like *Exception Throw*, are not in obvious and immediate demand but it's easy to see their potential usefulness.

The feasibility of adding an instrumentation opportunity to JIE depends on the transformation complexity and on the required data. The sourcecode transformation required to achieve certain transformations are non-trivial, and require non-traditional use of some Java language constructs (e.g., see the transformation for *Class Instantiation*). Other potential opportunities require complete rewriting of sourcecode portions.

The main feasibility limitation, however, is the data required by some opportunities – both for identifying potential instrumentation points and for performing the transformation. JIE processing is performed purely on per-file basis. This means that any opportunity that requires information that is not necessarily available in the currently processed `.java` file cannot be achieved under the current model. Specifically, this means that opportunities may not depend on availability of *global type information* – method signatures and modifiers, field types and inheritance relationships for classes that are not defined in the current `.java` file. One must remember that JIE does not have at hand all the information available to a Java compiler handling the very same source file. See Appendix C: Global Type Information for a possible way to address this.

The following is a brief list of all currently envisioned instrumentation opportunities, subject to the above criteria.

- *Method (Entry/Exit)*
- *Nominal Method (Entry/Exit)*
- *Class (Entry/Exit)*
- *Class Instantiation*
- *Wait Set Operation*
- *Field Read*
- *Field Write*
- *Exception Throw*

- *Exception Catch*
- *Nominal Basic Block*

For a full specification, see Appendix A: Instrumentation Opportunities.

In the above list, "Nominal" means that the instrumentation code is affected by exceptions differently than by other control flow mechanisms, and is therefore suitably for use mainly in the absence of exceptions.

Noticeably missing are the following opportunities:

- *Basic Block*
  An exception-safe version of *Nominal Basic Block*
- *Method Invocation*
  Instrumentation points just before and just after a call to certain methods, on the *caller*'s side. This is in contrast to *Method* and *Nominal Method*, which put the instrumentation in the called method and are therefore not applicable for library methods.
- *Synchronized Operation*
  Instrumentation points just before and just after every operation that may incur use of Java's built-in thread synchronization mechanism.

While these opportunities are in demand, they are not currently feasible in the above sense. Specifically, all of these opportunities require availability of global type information.

To see the problem with *Basic Block*, consider the following Java expression:

```
someObject.someMethod(argObject.argMethod())
```

We'd like to add an instrumentation point just after `argMethod` returns, but before `someMethod` is called. Alas, apparently there is no way to achieve this that does not involve knowing either `argMethod`'s return value or `someMethod`'s argument type. Both of these fall under the category of global type information: they are likely to be declared outside the current `.java` file, and are therefore unavailable for JIE.

The problem with *Method Invocation* is that given method invocation expression, we don't necessarily know the type of the object whose method is invoked – this requires global type information. Lacking that, the only possibility is to filter solely based on the method name (regardless of the object), which is clearly of very limited usefulness.

The issue with *Synchronized Operation* is similar but even more extreme. In Java, operations related to threads and synchronization include:

A. `synchronized` statements
B. Invocation of methods declared `synchronized`
C. The following `java.lang.Object` methods:
   `wait` (in all overloaded forms), `notify`, `notifyAll`
D. The following `java.lang.Thread` methods:
   `start`, `stop`, `suspend`, `resume`, `join`, `interrupt`, `yield`, `sleep`, `destroy`
E. The following `java.lang.ThreadGroup` methods:
   `stop`, `suspend`, `resume`, `destroy`

Of the above, only categories A and C are feasible in the general case. Category A requires simple syntax analysis. Category C can be handled by matching method names regardless of the object, since the relevant methods are declared `final` in `java.lang.object`, and therefore belong to all Java classes and cannot be overridden.

Category B requires knowing method modifiers, which are a particularly problematic case of global type information: they are not necessarily available in sourcecode form at all. For instance, in order to

learn whether a method of a library class is synchronized, JIE would have to parse compiled `.class` files placed inside compressed `.jar` archives. This is clearly outside the current scope of this project.

Categories D and E are special cases of the *Method Invocation* problem described above – they requires knowing, for every object whose methods are called, whether it's an instance of `java.lang.Thread` or `java.lang.ThreadGroup` (or their descendants). Filtering merely by method name is insufficient, since names such as `start`, `notify` and `join` are likely to be used by other classes as well – for example, the method `start` is also defined (with totally different semantics) for `java.applet.Applet`.

Note that Category C, the only consistent subset of the above categories that is feasible, is handled by the *Wait Set Operations* instrumentation opportunity.

2.3 Actions

Each instrumentation rule specifies an action: either an action template or one of a few specific action types.

An action template is a string that is inserted at the appropriate location in the code. Note that the instrumentation transformation may include additional code that is not a part of the action template, but is required for correct insertion of the action. Action templates can include macros that are expanded as the instrumentation code is inserted. All macros have the form "`${macro-text}`".

The following macros are defined:

Macro	Description
`${${}`	Expands into the string "`${`", which cannot be specified otherwise.
`${insert "file"}`	Inserts the specified file in the current location. Macro expansion will *not* occur in the included file.
`${include "file"}`	Includes the specified file in the current location. Recursive macro expansion is performed.
`${param}`	Expands into the appropriate context parameter. This can be either a generic context parameter (see Generic Context Parameters) or an opportunity-specific context parameter (see Appendix A: Instrumentation Opportunities).

Currently only one specific (i.e., non-template) action is defined: *Write to Log*. It relies on the *Logging Statement* global parameter (see Global Parameters) to simplify creation of rules that write strings to a logging stream, such as a trace file. This action type receives one string argument, *message*, which undergoes macro expansion and is then inserted into the logging statement.

For instance, if *Logging Statement* is set to

```
java.lang.System.out.println("${msg}");
```

then a *Write to Log* action with the message

```
I'm at ${methodName} (line {$line})
```

will expand to something similar to

```
java.lang.System.out.println("I'm at getData (line 42)");
```

Note that some instrumentation opportunities define two separate actions (typically *Entry* and *Exit*). While the opportunity filters are shared, each action is specified and applied separately.

It's possible (in fact, likely) for several rules to insert instrumentation code at the same location. In this case, all actions are guaranteed to be executed (subject to dynamic filters), at an unspecified order.

2.4 Generic Context Parameters

The following context parameters are implicitly available in the actions of all opportunities.

Macro and type[1]	Example	Description
`${filename}` path	`/home/tromer/org/tromer/jie/Demo.java`	Fully qualified path to `.java` source file
`${packageName}` name	`org.tromer.jie`	Fully qualified package name
`${packageIdent}` identifier	`jie`	Unqualified package name
`${className}` name	`org.tromer.jie.Demo`	Fully qualified class name[2]
`${classIdent}` identifier	`Demo`	Unqualified class name[3]
`${methodName}` name	`org.tromer.jie.Demo.show`	Fully qualified method name
`${methodNameSig}` method header + qualification	`void org.tromer.jie.Demo.show(String)`	Fully qualified method name with type signature
`${methodIdent}` identifier	`show`	Unqualified method name[4]
`${methodIdentSig}` method header	`void show(String)`	Unqualified method name with type signature
`${line}` integer	`753`	Line number of the instrumentation point (in the original `.java` file)
`${col}` integer	`12`	Column number of the instrumentation point (in the original `.java` file)
`${position}` integer**#**integer	`753#12`	Line and column number of the instrumentation point (in the original `.java` file)
`${jiid}` JIID	`jiid:IIGLJMEMJ697A_75_12`	JIE Instrumentation ID of the instrumentation point (see JIE instrumentation IDs)

For nested classes, the `className` is created by concatenating the nested class's identifier to the name of the encoding class, separated by '`$`'. This is in compliance with the Inner Classes Specification. Similar action is taken for anonymous classes, except that the unique ID is not a decimal

---

[1] Type descriptions based on [JAVA-LANG] where possible.

[2] For inner classes, after the package name '`$`' is used as a separator instead of '`.`' (as specified in [JAVA-INNER]).

[3] For named local classes (block-scoped), an illegal identifier of the form "*`line_col_ClassName`*" where the class declaration starts on line *`line`* column *`col`*. This is a slight variation on [JAVA-INNER].
For anonymous classes, same as the above using "`_anonymous_`" as the class name.

[4] For constructors, same as the class identifier.
For static initializers, the illegal identifier "`*static-initializer@`*`line`*`#`*`column`*`*`".
For instance initializers, the illegal identifier "`*instance-initializer@`*`line`*`#`*`column`*`*`".
For field initializers, the illegal identifier "`*field-initializer@`*`line`*`#`*`column`*`*`".

number (as specified in the standard) but a slightly more readable representation containing the line and column number of the class declaration.

In both cases, `classIdent` contains only the last part of `className`, after all '`.`' and '`$`' separators. These rules apply to the class portions of `methodName` and `methodNameSig` as well.

2.5 Generic Static filters

Static filters cause rules to transform only a subset of the potential instrumentation points defined by the rule's opportunity. The following filters can be applied to for all instrumentation rules regardless of opportunity types. In addition, they can be used as global filters.

Name	Parameter	Description
Package	Fully qualified package name	Limit to specific package and sub-packages
Class	Fully qualified class name	Limit to specific class
Method	Fully qualified method name[5]	Limit to specific method
Class visibility	public/package/nested/local	Limit to classes with given visibility
Method visibility	public/protected/package/private	Limit to methods with given visibility
Excluded JIIDs	URL of file containing a list of JIIDs, one per line	Skip all instrumentation points whose JIID is mentioned in the file

2.6 Generic Dynamic Filters

Currently envisioned dynamic filters include:

- **Enable/Disable**
  Enable/disable instrumentation at run-time.
- **Once Only**
  Execute action only the first time the instrumentation point is reached.
- **Every N Times**
  Execute action only once every N times the instrumentation point is reached.
- **Minimum Time Interval**
  Do not execute action if less than the specified time has elapsed since the last time it was executed.

2.7 Global Parameters

Beside instrumentation rules and global filters, an instrumentation configuration includes a few general parameters:

- **Input**
  The root directory of the source files to be instrumented. This directory will be scanned recursively. Alternatively, a single input file.
- **Output**
  The root directory for output files. The instrumented files will be placed here in a structure reflecting that of the input directory. Alternatively, a single output file (iff a single input file was specified).
- **Logging Statement**
  The text inserted when using the *Write to Log* instrumentation action. "`${msg}`" will be replaced by the logged message. See Actions for an example.

2.8 JIE instrumentation IDs

A Java Instrumentation ID (JIID) is used to uniquely identify an instrumentation point in the context of a given set of sourcecode files. This means that after running JIE, each instrumentation point has a JIID different than all other instrumentation points in the output, and after running JIE again, JIIDs may change in the following ways:

- Input files changed: undefined (no guarantees whatsoever).

[5] This does not provide a way to distinguish overloaded methods.

- Instrumentation rules changed: instrumentation points caused by rules whose opportunity and action fields were left intact retain their JIIDs. Others have JIIDs different from all other JIIDs, new and old alike.
- None changed: No change in produced JIIDs.

A JIID has the following structure:

`jiid:`*`hash.line.column`*

For example:

`jiid:IIGLJMEMJ697A_75_12`

`line` and *`column`* specify the location of the instrumentation point in the original `.java` file. *`hash`* is a 64-bit long integer represented as 13-character Base32 string (all characters are in the range 'A'..'V', '0'-'9'). It's produced by hashing together the following values:

- Fully qualified class name
- Opportunity and action fields of the rule that created the instrumentation point.

It's virtually impossible for JIE to create two identical JIIDs except when the above values are identical.

The purpose of the "`jiid:`" prefix is to ease automated extraction of JIIDs from instrumented sourcecode files, as well as facilitate further manipulation, using text-processing tools such as `sed`.

JIIDs' primary use is in coverage testing, using the *Excluded JIIDs* static filter to limit the instrumentation scope to code that has not undergone coverage analysis yet.

2.9 XML Representation

Instrumentation configurations can be represented using XML structured storage (see [XML]). This allows for convenient external manipulation of instrumentation configurations, and provides ample room for future extensions. Using existing XML parsers, this representation may actually be easier to provide than a non-standard textual storage format.

There are several XML parsers written in Java that are publicly available:

Name	Source	URL
XML Parser for Java	IBM	http://www.alphaWorks.ibm.com/tech/xml4j
XP	James Clark	http://www.jclark.com/xml/xp/
Lark	Tim Bray	http://www.textuality.com/Lark/

All of the above are suitable for our purpose. The above list represents the order of preference – if no particular issues arise, the IBM parser will be used.

Representation of instrumentation configuration uses a simple XML Data Type Definition (DTD), which reflects the content of this chapter in a fairly straightforward way. For an example, see Appendix B: Instrumentation Configuration DTD.

3 The Java Parser Component

3.1 Overview

The purpose of the Java Parser component is to convert the input sourcecode into an internal representation suitable for further processing. While the Java language is relatively simple compared to alternatives, the task of constructing a full Java parser is not trivial. Therefore, the project will make use of existing tools to accomplish this task. Typically, such tools accept a grammar specification of some kind, and generate the sourcecode for an appropriate parser.

Strictly speaking, the parser component is composed of four parts: a lexical analyzer (lexer), the parser per-se, a tree builder and a visitor pattern. The lexer and parser should have full support for the Java language, including recent extensions and Unicode escape codes. Numerous Java tools fulfill these requirements. Our main concern is therefore with the more advanced functionality.

It is possible to perform the instrumentation transformation in either of two ways:

1. Using ad-hoc parser callbacks – performing all work during the parser operation.
2. Building a complete in-memory Abstract Syntax Tree (AST), and then operating on this tree.

The second alternative is easier to work with, and allows greater flexibility and generality in the design of the instrumentation framework. Since the whole syntax tree is available simultaneously, such operations as inspection of context and multiple passes become much simpler and more efficient. On the down side, this entails non-negligible memory consumption (on the order of 0.5-3KB per line of source), and the related memory allocation overhead.

For our needs, flexibility and clarity are more important than performance optimization. Therefore we choose the second alternative, and require tree-building functionality from the parsing tool.

Operations on the AST (namely looking for instrumentation opportunities) are best carried out using the Visitor design pattern [GAMMA]. The Visitor pattern provides a flexible enumeration scheme for heterogeneously typed data structures, such as the AST. It provides the means to achieve encapsulation and proper modularity, while maintaining strong typing.

3.2 Parsing Tool Evaluation Criteria

The following evaluation criteria are used to evaluate potential parsing tools for this project.

1. **Java language support**
   Availability of up-to-date grammars including JDK 1.1 extensions[6]. Support for Unicode escape codes.
2. **Functionality**
   Level of functionality offered by the tree and Visitor pattern implementation. This includes provided and autogenerated utility classes.
3. **AST structure**
   Clarity, convenience and safety of the AST object structure. This includes strong typing, readable identifier names and encapsulation of unsafe operations.
4. **Documentation and support**
   Quality of documentation and availability of interactive support via mailing lists, newsgroups and personal e-mail.
5. **Development status**
   Preferable are active projects with frequent releases, short response time to bug reports and high rate of feature extension.

[6] And, non-critically, the JDK 1.2 `strictfp` keyword

6. **License and source availability**
   The tool must be publicly available. Its licensing terms must not limit the ability to distribute JIE. Limitations on commercial use (e.g., distribution for a charge) may be acceptable. Sourcecode availability is preferable but not critical, since our main reliance is on the code autogenerated by the tool code.
7. **Pure Java**
   Any code generated or provided for included in JIE must be Pure Java. Tools used only during development should also preferably (but not critically) have Pure Java implementations.
8. **Performance**
   Time and space efficiency is impossible to reliably evaluate without further work, and are therefore not considered here. This may change if it becomes evident that performance may be unacceptable for some alternatives.
9. **Robustness**
   Frequency of problems and how well they're handled (identified, announced and solved). All inspected products are satisfactory in this regard. As precise comparison is difficult and unnecessary, it is not carried out here.
10. **Parser type**
    Different types of parsers and grammars (e.g., LL(1), LL(k), LALR(1)) affect the structure of the AST. It is assumed that the differences are inconsequential, as long as full Java grammar is available.

3.3 Candidate Parsing Tools

This section lists the potential parsing tools available, under the limitations listed above. Basic details and an overview are given for each. Further details and comparative evaluation are provided in the next section. Searching was based on comprehensive Internet indices of compiler tools [CFCI] [CCCT].

3.4 Candidate Parsing Tools

3.4.1 JavaCC / Metamata Parse

Name: Java Compiler Compiler (JavaCC)
Version: 1.1 *(1999/08/10)*
Source: SunTest, Sun Microsystems
Home page: http://www.suntest.com/JavaCC/

Name: Metamata Parse
Source: Metamata Inc.
Home page: http://www.metamata.com/JavaCC/

Note: The authors of JavaCC have left SunTest and founded Metamata Inc. Future development of JavaCC will be under the Metamata Parse title, under similar licensing terms.

JavaCC is the most widely used Java parser generator. Originally written in SunTest and now under the auspices of Metamata Inc., JavaCC is guaranteed to be compatible and efficient in handling of Java programs. JavaCC includes a lexer generator and a parser generator, and numerous sample grammars (including Java 1.1). It also includes a separate AST builder, JJTree (see below).

JavaCC has a feature not supported by other alternatives, called "special tokens". This features enables inclusion of whitespace and comments in the AST, which provides an easy way to fully preserve the sourcecode content after transformation.

3.4.2 JJTree

Name: JJTree (part of JavaCC)
Version: 1.1 *(1999/08/10)*
Source: SunTest, Sun Microsystems

Home page: http://www.suntest.com/JavaCC/

Bundled with the JavaCC is a JJTree, an AST tree builder built upon the JavaCC lexer and parser. JJTree works by augmenting the grammar file with AST node construction code.

JJTree provides powerful extension hooks during the tree construction stage. These can be used, for instance, to reduce memory consumption in an application-specific way.

On the down side, JJTree's AST structure is lacking from an OO design perspective. It can either use a single class for all tree nodes, or produce weakly typed references (all access via typecast). Both greatly hamper ease of development and code reliability. In addition, its support for the Visitor pattern is very rudimentary.

3.4.3 JTB

Name: Java Tree Builder (JTB)
Version: 1.1 *(1999/01/10)*
Source: Kevin Tao, Purdue University
Home page: http://www.cs.purdue.edu/homes/taokr/jtb/

JTB, like JJTree, is an AST tree builder that works by augmenting JavaCC's input grammar with AST node construction code. Unlike JJTree, it is developed separately from JavaCC, but compatibility is consistently maintained. JTB does not deny access to JavaCC's low-level functions, such as token location information and special token.

The AST object structure constructed by JTB is mostly strongly typed, with notable exceptions such as grammar productions involving alternatives, lists and optional parts. Field names are numerical (e.g., `f0`, `f1`), which hampers readability.

JTB has a very good implementation of the Visitor pattern (with the exception of nodes containing multiple alternatives, which are still handled by a "switch" statement). In this respect, it is clearly superior to JavaCC's own JJTree[7].

JTB can generate several predefined visitor classes which implement commonly required functionality related to output and formatting. A simple Java pretty-printer example is also available. These classes may prove useful in the construction of the Java sourcecode emitter component of JIE.

3.4.4 SableCC

Name: SableCC
Version: 2.6 *(1998/11/09)*
Source: Étienne Gagnon, Sable Research Group
Home page: http://www.sable.mcgill.ca/sablecc/

SableCC originated as Étienne Gagnon's M.Sc. thesis [GAGNON]. Unlike JavaCC and its AST-building extensions, SableCC is designed from the grounds up to operate as an AST builder, and does not have independent lexer and parser functionality. Several grammars are included with SableCC, including Java 1.1. A Unicode preprocessor class is also provided.

SableCC's great strength is its clean AST design, which is strictly strongly typed. Beyond code safety and clarity, this has the important practical benefit of the ability to easily inspect the AST content during debugging. The AST node classes support safe tree transformation with guaranteed consistency invariants. Field names are user-determined, and the provided Java grammar provides human-readable names.

The Visitor pattern implementation matches the AST in elegance. Depth-first and reversed-depth-first visitor classes are autogenerated.

---

[7] This was acknowledged by one of JavaCC's authors, in e-mail communication.

3.4.5 ANTLR

Name: ANTLR
Version: 2.5.0
Source: MageLang Institute
Home page: http://www.antlr.org
Note: ANTLR replaces PCCTS (last version: 1.33), which was implemented in C++

ANTLR is a powerful lexer/parser generator that is actively developed and widely used. It is flexible and feature-laden with respect to grammar flexibility and parser generation. Tree construction instructions can be specified in the grammar. A Java 1.1 grammar is available, but is overly condensed and may need to be modified to be usable. Unicode is not yet supported.

The AST generated by ANTLR can be tailored for specific needs by simple grammar modifications. However, it uses a single tree node class for all nodes, which is inconvenient and unsafe.

ANTLR does not have a Visitor pattern, but provides an interesting alternative called "tree parsers" or "tree walkers" [ANTLR-REF]. A tree walker is essentially a grammar that operates on the Abstract Syntax Tree, after its construction. It is greatly simplified relative to the normal grammar, and can harness the full power ANTLR's grammar facilities, including EBNF, syntactic predicates and semantic predicates.

By attaching code to tree walker productions, a tree walker can behave similarly to a Visitor. The advantage of this approach is that grammar rules can be used to declaratively and concisely specify the places in the grammar where the instrumentation code should be activated. Arbitrary procedural processing can then be performed in each of these places.

3.5 Comparison Table

	JavaCC + JJTree	JavaCC + JTB	SableCC	ANTLR
Java language support	Full	Full	Full	No Unicode. Java grammar requires modification.
Functionality	• Visitor pattern very rudimentary. • Powerful AST creation hooks. • Support for comment preservation.	• Visitor pattern mostly well-implemented • Several visitor classes autogenerated. • Support for comment preservation. • Java pretty-printer included	• Visitor pattern well-implemented • Basic visitor classes are autogenerated.	• Offers a powerful semi-declarative alternative to the Visitor pattern
AST structure	• Either uniform node type or weak typing (access via typecast) • Numeric field names	• AST partly strongly typed (some access via typecast) • Numeric field names	• AST strictly strongly typed • Meaningful field names	• Uniform node type • Meaningful field names
Documentation and support	Basic documentation available. Active mailing list and newsgroup.	Basic documentation available.	Basic documentation available but static. Author responds helpfully to inquires.	Comprehensive documentation available. Active mailing list and newsgroup.

			Low-traffic mailing list.	
References: Web, Usenet, Mailing lists[8]	(918) 179 [9] (639) 35 (747) 123	(918) 95 (639) 35 (747) 0	30 28 261	826 (incl. PCCTS) 481 (incl. PCCTS) 130
Development status	Development status unclear – announcements contradict reality.	Actively developed	Developed in author's free time. Recent releases did not add features. Extensions planned.	Actively developed
License and source availability	Source not available. Distribution of generated code not limited. See 8.1.	Open source, free for noncommercial use, commercial use status unclear. See 8.2.	Open source, LGPL (no significant limitations). See 8.3.	Public-domain (no limitations)
Pure Java	Yes	Yes	Yes, but does not run under JDK 1.2[10]	Yes

Sources of information:

- Information published on the tools' web pages, including meta-information such as update frequency.
- E-mail and Usenet discussions with the authors of JavaCC, SableCC and ANTLR.
- Practical experimentation with JavaCC, JJTree and JTB. Using each tool, I generated a full Java parser / tree builder and inspected the generated code. I then run it on sample Java code and inspected the actual AST built in memory.
- Reference statistics generated using AltaVista Advanced Search queries on product name with appropriate filters.

3.6 Choice of Parsing Tool

JJTree is ruled out, since it is inferior to JTB in almost any respect. This leaves JavaCC+JTB, SableCC and ANTLR.

ANTLR's main strength is in its semi-declarative visitor replacement. During more detailed design stages, it will become evident whether this feature is desirable. My current expectation is that this mechanism will prove somewhat too esoteric for our needs, and will not be used. Under such circumstances, ANTLR's AST structure deficiencies and lack of other functionality render it inappropriate.

The two remaining candidates are JavaCC+JTB and SableCC. Both are satisfactory on all accounts, but provide somewhat different tradeoffs.

The following table reiterates the significant differences between these two alternatives.

	JavaCC+JTB	**SableCC**
Functionality	Pretty printer included. Comment preservation available.	
AST structure	Hybrid	Strongly typed
Development status	Actively developed, with large user community.	Development in author's free time, with a smaller user

[8] The mailing list count estimates the number of messages on the product's support mailing list

[9] Numbers in parenthesis refer to JavaCC, which is relevant to both of the first columns.

[10] Due to newly introduced enforcement of method size limit (64K). This problem was discovered and reported by me during this evaluation. According to the author, it will be resolved by the time JIE reaches implementation stage.

		community.
License	JavaCC – closed source. JTB – open source. Commercial use limitations unclear.	LGPL: open source, no significant limitations.

Since both alternatives are satisfactory and neither is clearly superior, the final choice is deferred until future design details become evident. The architecture and usage of these tools are similar, so this will not have any adverse effect on the design process.

It should be noted that some of the deficiencies mentioned above can be resolved by modification of the code generated by the parsing tools. This may require non-negligible work, and has the disadvantage of precluding future upgrades. However, should the choice of alternatives narrow down for whatever reason, this can form a fallback strategy.

Future design decisions and tool evolution will determine the final choice.

4 GUI Driver

The basic functionality of JIE is provided in a Java class with no user interface. Two driver applications are provided for executing this class: a command-line utility and a GUI application.

The command-line utility is trivial – it receives an XML specification of the instrumentation configuration as its only parameter, and activates JIE on that configuration, with rudimentary progress indication.

The GUI driver offers additional functionality: it provides a GUI-based interface for editing instrumentation configurations. This chapter provides screenshots of a UI prototype that illustrate how this functionality is presented and used.

The GUI driver uses a fairly simple (albeit detail-laden) user interface. It's composed of two windows: a main window which provides access to top-level details of the instrumentation configuration, and a *Rule Settings* dialog that provides access to rule details.

4.1 Main Window

The main window of the GUI application offers access to the top-level details: global parameters, global filters and the list of rules. It also provides the means to execute the instrumentation configuration, with detailed progress indication. Each of the above is placed in its own tab in a tab control. The following screenshots demonstrate the UI design.

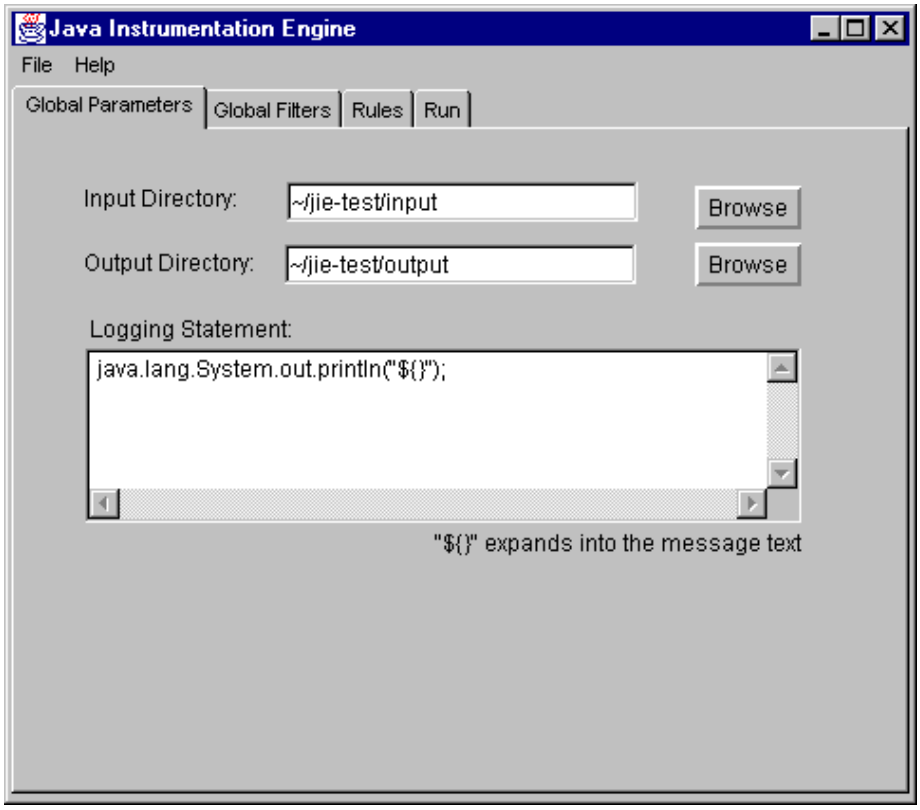

Figure 1 - Global parameters

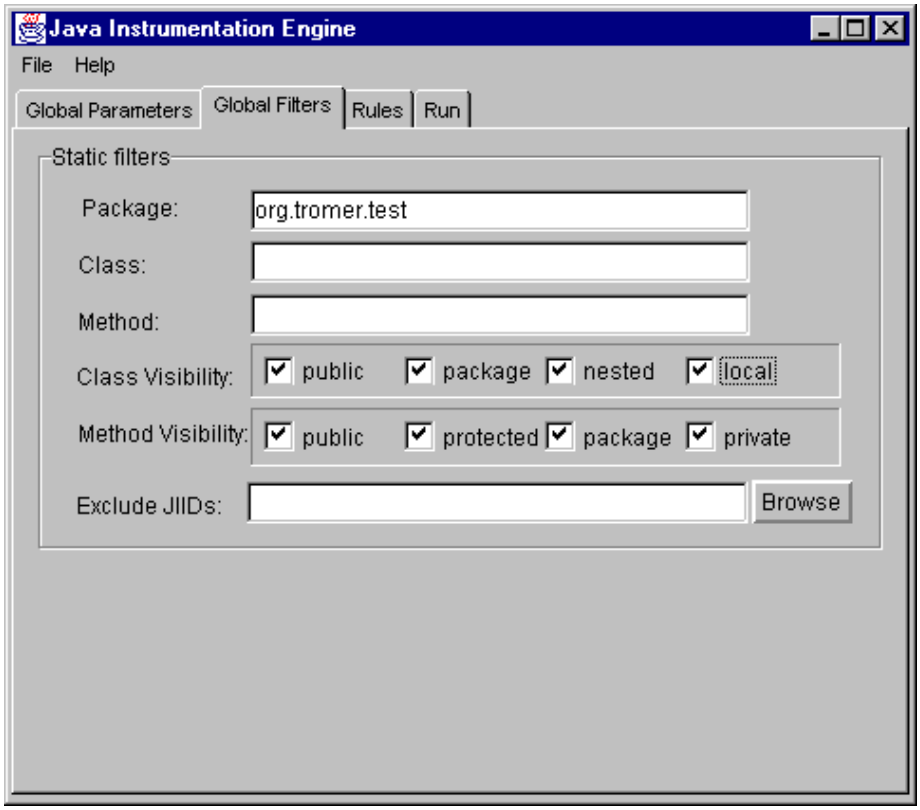

Figure 2 - Global filters

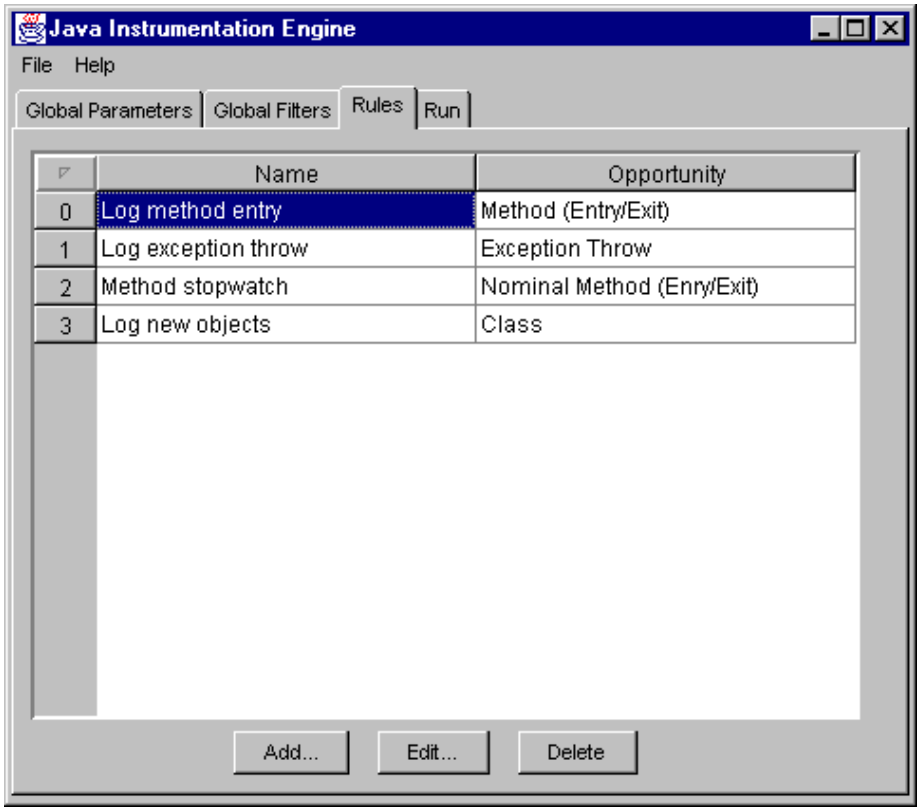

Figure 3 - Rule list

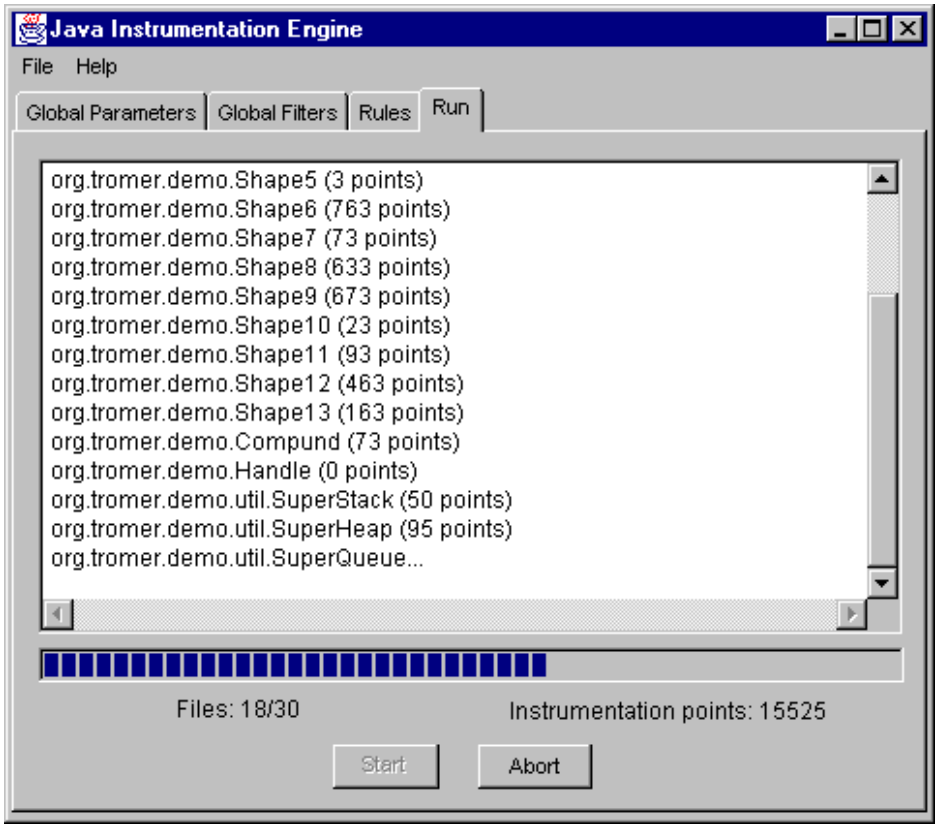

Figure 4 – Progress indication of instrumentation in progress

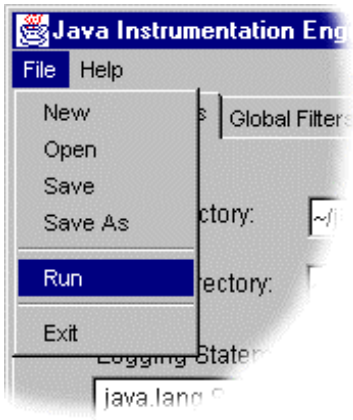

Figure 5 - Pull-down men

4.2 Rule Settings dialog

Clicking on the *New* or Edit button in the *Rules* tab of the main window opens the *Rule Settings* dialog. This dialog presents all the fields pertaining to the edited rule. As in the main window, the information is arranged in a tab control.

The first tab contains the rule name, opportunity and filters. In addition, there is one or more tabs dealing with action details. Most rules have a single *Action* tab, but rules that have both an entry action and an exit action have two independent action tabs, called *Entry Action* and *Exit Action* tabs, respectively.

The following screenshots illustrate the UI design. The rule depicted is identical to the second rule listed in the XML example provided in Appendix B: Instrumentation Configuration DTD.

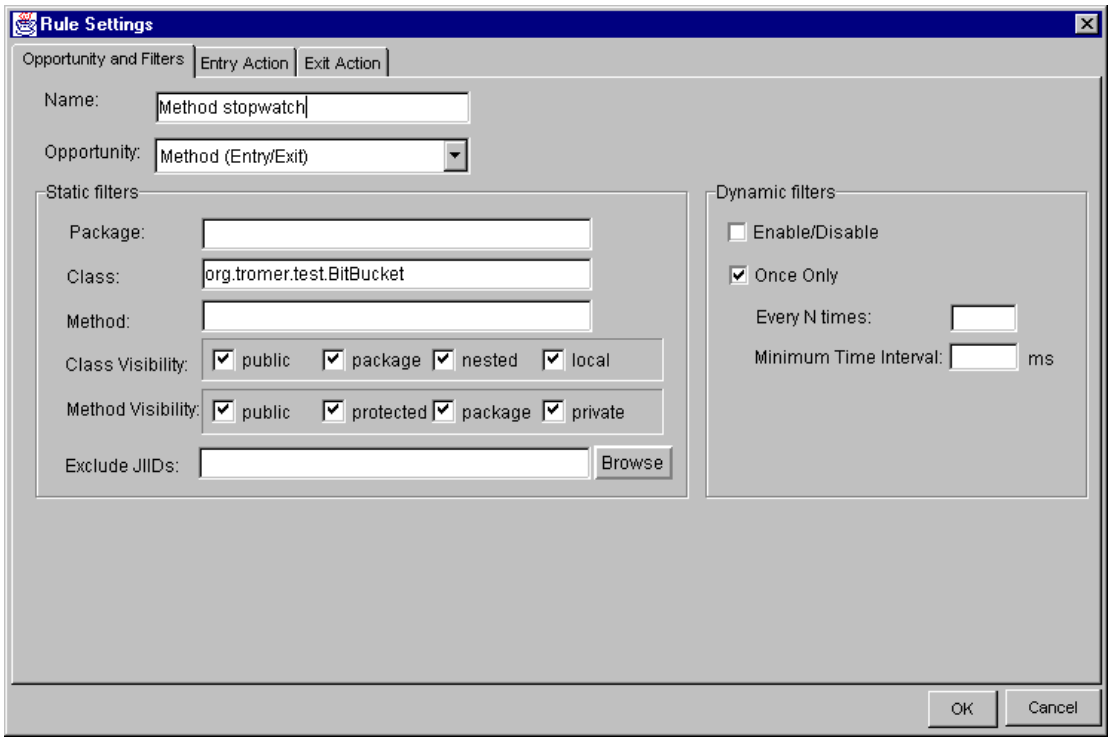

Figure 6 - Rule Opportunity and Filters

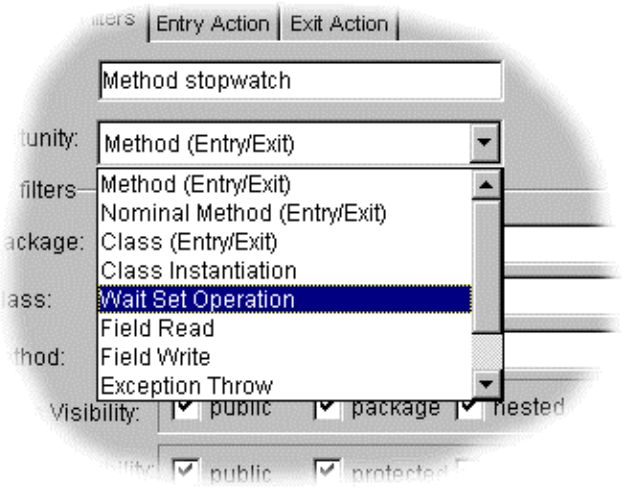

Figure 7 - Opportunity selection

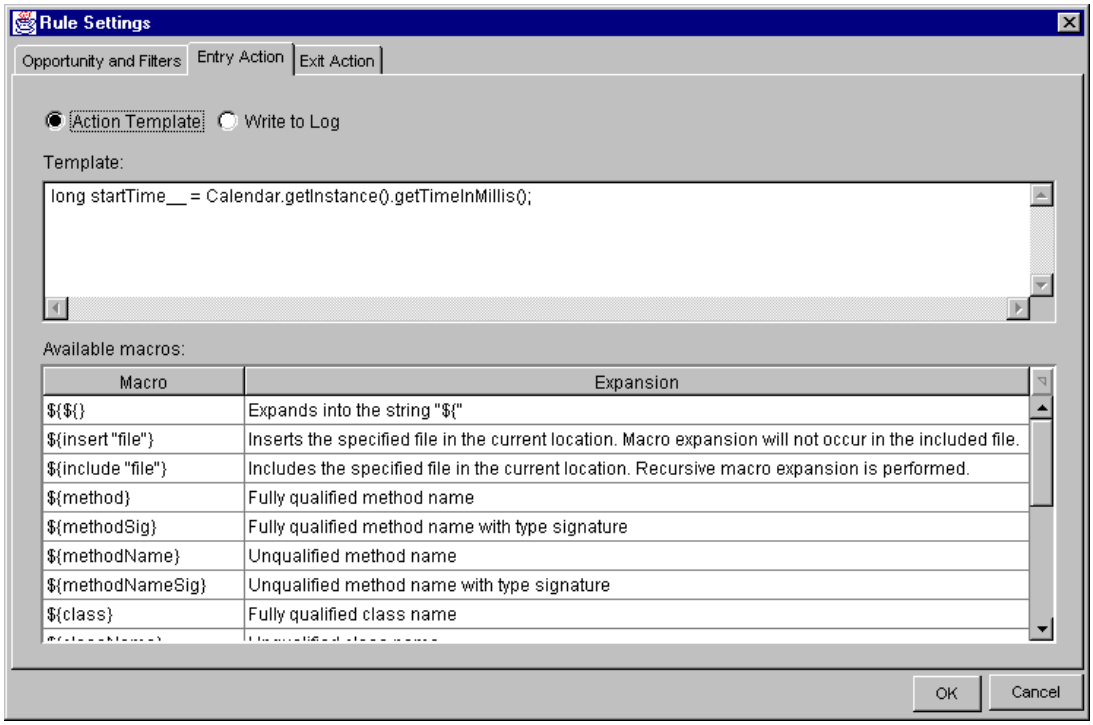

Figure 8 – Rule entry action

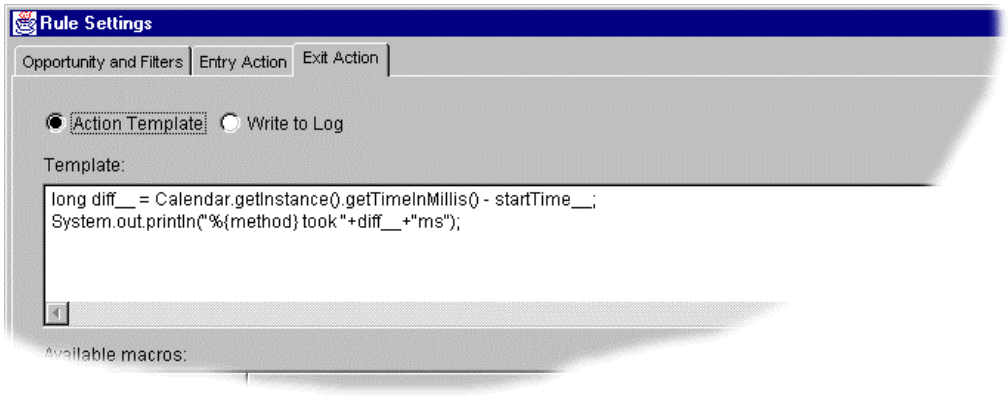

Figure 9 - Rule exit action

5 Appendix A: Instrumentation Opportunities

This appendix presents the full specification for all currently envisioned instrumentation opportunities, subject to the criteria listed in 2.2. Each opportunity is completely summarized in a table, using the following format:

<table>
<tr><td>Name</td><td colspan="2">Name of instrumentation opportunity</td></tr>
<tr><td>Description</td><td colspan="2">Description of the precise stage at which the instrumentation code is called.
If there are several sub-opportunities, each is described.</td></tr>
<tr><td>Filters</td><td colspan="2">Names, types and descriptions of filters specific to this opportunity. Apart from these, all generic filters apply.</td></tr>
<tr><td>Context</td><td colspan="2">Context parameters provided to the instrumentation actions, in addition to the generic context parameters.</td></tr>
<tr><td>Template</td><td>Generic form of the grammar element to which the opportunity is applied. Terminals are marked in bold. Words in non-bold specify non-terminals.</td><td>The transformed form of the grammar element. This transformation is for illustration purposes only, and is not a formal specification.

This template is a proof-of-concept that demonstrates the feasibility of the transformation. Actual transformations may differ.

Identifier names (temporary variables, etc.) will be chosen to be unique in their context. Conveniently, Java reserves identifier names beginning with ‘$’ for this type of use.

ACTION, ENTRY-ACTION and EXIT-ACTION specify the macro-expanded form of the rule action. Where applicable, they are followed by the opportunity-specific context parameters, in parenthesis.</td></tr>
</table>

Names for non-terminals in the templates, as well as general terminology, are based on [JAVA-LANG].

Name	**Method (Entry/Exit)**	
Description	Entry: Before first line of a method. Exit: After the last executed line of a method. This can be the last line, a return statement or any statement that caused method exit by exception throw. Constructors, *static initializers* and *instance initializers* are considered as methods. Note that code in *field initializers* is not considered to be inside any method (see [JAVA-LANG], [JAVA-INNER]).	
Filters	none	
Context	none	

Template[11]	`METHOD-HEADER {`
	`  STATEMENTS`
	`}`
		`METHOD-HEADER {`
		`  ENTRY-ACTION`
		`  try {`
		`  STATEMENTS`
		`  } finally`
		`  EXIT-ACTION`
		`  }`
		`}`

Name	**Nominal Method (Entry/Exit)**	
Description	Entry: Before first line of the method. Exit: After the last executed line of the method. In contrast with the Method opportunity, the exit action will occur only if the end of the method code is reached or a return statement is executed. If the method is quit via an exception throw the exit action will not be executed. Constructors, *static initializers* and *instance initializers* are considered as methods. Note that code in *field initializers* is not considered to be inside any method (see [JAVA-LANG], [JAVA-INNER]).	
Filters	none	
Context	none	

Template	`RESULT METHOD-DECLARATOR`
	`{`
	`  STATEMENTS`
	`}`
	
	inside `STATEMENTS:`
	
	`return EXPRESSION;`
		`RESULT METHOD-DECLARATOR`
		`{`
		`  ENTRY-ACTION`
		`  MODIFIED-STATEMENTS`
		`  EXIT-ACTION`
		`}`
		
		inside `MODIFIED-STATEMENTS:`
		
		`return exit_METHOD(result);`
		
		and at class scope, in the class that contains the method:
		
		`RESULT exit_METHOD(RESULT val)`
		`{`
		`  EXIT-ACTION`
		`  return val;`
		`}`

Name	**Class (Entry/Exit)**

[11] Here and in other opportunities that deal with methods, constructors and static initializers are handled similarly.

Description	Entry: At the beginning of a class body (before the first *ClassBodyDeclaration*). Exit: At the end of a class body (after the last *ClassBodyDeclaration*). This opportunity is useful for insertion of new class members or object initialization code. This is the only opportunity that places instrumentation points outside methods.	
Filters	none	
Context	none	

Template	`CLASS-HEADER {`
	`  DECLARATION-1`
	`  ...`
	`  DECLARATION-N`
	`}`
		`CLASS-HEADER {`
		`  ENTRY-ACTION`
		`  DECLARATION-2`
		`  ...`
		`  DECLARATION-N`
		`  EXIT-ACTION`
		`}`

Name	**Class Instantiation (Entry/Exit)**		
Description	Entry: Just before a class is instantiated Exit: Right after the newly built class instance is returned.		
Filters	Instantiated Class	text pattern	Limit to classes that match the pattern
Context	`${instanceClass}`	name	The name of the instantiated class, as specified in the instantiation expression.
	`${instanceClassName}`	identifier	The unqualified class name of the instantiated class
	`${object}`	identifier	The newly created object (or array of objects)

Template[12]	`new CLASS(ARGUMENTS)`		`afterCreation_CLASS (`
			`  beforeCreation_CLASS()?null:`
			`  new CLASS(ARGUMENTS)`
			`)`
			
			and at class scope, in the class that contains the instantiation expression[13]*:*
			
			`private CLASS`
			`afterCreation_CLASS(CLASS obj)`
			`{`
			`  EXIT-ACTION(CLASS,obj)`
			`  return obj;`
			`}`
			
			`private boolean`
			`beforeCreation_CLASS()`
			`{`
			`  ENTRY-ACTION(CLASS)`
			`  return false;`
			`}`

[12] Instantiation of an array of classes is handled in the same fashion, with obvious syntactical adaptation.

[13] The "`?:`" conditional expressed is used to guarantee that **`beforeCreation`**`_CLASS()` is called before the instantiation. The call is made during the actual expression evaluation. This is necessary to make sure it's called at the correct time (just before and after creation), and only if the `new` expression is actually evaluated (e.g., the `new` expression may itself be inside a "`?:`" conditional operator).

Name	**Wait Set Operation (Entry/Exit)**		
Description	Entry: Just before a wait set operation. Exit: Just after a wait set operation. A *wait set operation* is a method call that directly affects an object's wait set (as specified in [JAVA-LANG] section 17.14). These are the methods `wait`, `notify` and `notifyAll` of `java.lang.Object`.[14]		
Filters	none		
Context	`${object}`	identifier	The object whose method was called.

Template	`EXPRESSION.METHOD(ARGS);`
	
	where `METHOD` is one of these:
	
	`wait`
	`notify`
	`notifyAll`
		`{`
		`  java.lang.Object obj =`
		`  EXPRESSION;`
		`  ENTRY-ACTION(obj)`
		`  obj.METHOD(ARGS);`
		`  EXIT-ACTION(obj)`
		`}`	

Name	**Field Read**		
Description	Just before a field is read. Only access via the `this` keyword (e.g., "`this.color`") is supported. Identifying fields of other objects is not generally possible in JIE, since that requires global type information. It is assumed that most fields are private or protected, so this limitation should not be significant.		
Filters	Field	text pattern	Limit to fields that match the pattern
Context	`${field}`	name	The fully qualified field name
	`${fieldName}`	identifier	The unqualified field name

Template	**`this`**`.IDENTIFIER`
	
	Except when in the operators "++" and "--", or LHS of operators "=" and "op=".
		`get``_IDENTIFIER()`
		
		and at class scope:
		
		`private TYPE-OF-IDENTIFIER`
		`get``_IDENTIFIER()`
		`{`
		`  ACTION(IDENTIFIER)`
		`  `**`return this`**`.IDENTIFIER;`
		`}`	

	`EXPRESSION`
	
	Where `EXPRESSION` is one of the above operators applied to an expression of the following form (modulus parenthesis):
	
	`this``.IDENTIFIER`
	
	Example: "`this.date++`"
		`get_modified``_IDENTIFIER(`
		`EXPRESSION)`
		
		and at class scope:
		
		`private TYPE-OF-IDENTIFIER`
		`get_modified``_IDENTIFIER`
		`(TYPE-OF-IDENTIFIER `**`val`**`)`
		`{`
		`  ACTION(IDENTIFIER)`
		`  `**`return val`**`;`
		`}`	

[14] Method identification is done by method name alone. It is assumed that no class introduces additional overloaded versions of these methods that differ only by parameter type.

Name	**Field Write**		
Description	Just after a field is modified The same limitations as in *Field Read* apply.		
Filters	Field	text pattern	Limit to fields that match the pattern
Context	`${field}`	name	The fully qualified field name
	`${fieldName}`	identifier	The unqualified field name

Template	`EXPRESSION`
	
	where EXPRESSION is one of the following (modulus parenthesis)
	
	`this.IDENTIFIER = VALUE`
	
	`this.IDENTIFIER op= VALUE`
	
	`this.IDENTIFIER++`
	`this.IDENTIFIER--`
	`++this.IDENTIFIER`
	`--this.IDENTIFIER`
			`modified_IDENTIFIER(EXPRESSION)`
			
			and at class scope:
			
			`private TYPE-OF-IDENTIFIER modified_IDENTIFIER( TYPE-OF-IDENTIFIER val) { ACTION(IDENTIFIER) return val; }`

Name	**Exception Throw**		
Description	Just before the expression of a throw statement is evaluated and thrown. Note that the throw expression may be complex and have side effects, yet the action occurs *before* it is evaluated. Also, the thrown object is not available as a context parameter. These limitations cannot in general be relieved without full global type information.		
Filters	Exception class	text pattern	Limit to `throw new CLASS(...);` where CLASS matches the pattern.
Context	none		
Template	`throw EXPRESSION;`		`{ ACTION throw EXPRESSION; }`

Name	**Exception Catch**		
Description	Just before an exception is thrown.		
Filters	Exception class	text pattern	Limit to classes that match the pattern
Context	`${object}`	identifier	The exception object that was caught
Template	`catch (TYPE OBJECT) { STATEMENTS }`		`catch (TYPE OBJECT) { ACTION(OBJECT) STATEMENTS }`

<table>
<tr><td>Name</td><td colspan="2">Nominal Basic Block (Entry/Exit)</td></tr>
<tr><td>Description</td><td colspan="2">Entry: Method entry and just after any control branching
Exit: Method exit and just before any control branching

Control branching occurs in <code>while</code>, <code>do</code>, <code>for</code>, <code>if</code>, <code>switch</code>, <code>try</code>, <code>break</code>, <code>continue</code> and <code>throw</code> statements, and in expressions using the conditional operators “<code>&&</code>", "<code>||</code>" and "<code>?:</code>".

This is not the traditional meaning of “basic block”, since jumps that don’t involve control branching (e.g., unconditional method invocation) do not break the basic block. Subsequently, “nominal basic blocks” are nested at each method call. The instrumentation action (or a processor of a resulting trace) may need to take this into account.

Note that some simple expressions (e.g., field access, array access and method invocation) in principle incur additional basic blocks even in the above sense, since they may throw an exception. However, adding instrumentation actions for each such expression would incur a dramatic explosion in code size, and is of questionable usefulness. It is also very hard to achieve in the general case, since global type information is required. This opportunity does not handle these cases, and may fail to perform the exit action in case of exception throw. That’s what the “Nominal” in “Nominal Basic Block” stands for.</td></tr>
<tr><td>Filters</td><td colspan="2">none</td></tr>
<tr><td>Context</td><td colspan="2">none</td></tr>
<tr><td>Template</td><td><pre>METHOD-HEADER {
    STATEMENTS
}</pre></td><td>The transformation is identical to Nominal Method.</td></tr>
<tr><td></td><td><pre>if (EXPRESSION)
    STATEMENT1
else
    STATEMENT2</pre></td><td><pre>{
    boolean cond = EXPRESSION;
    EXIT-ACTION
    if (cond) {
        ENTRY-ACTION
        STATEMENTS1
        EXIT-ACTION
    } else {
        ENTRY-ACTION
        STATEMENTS2
        EXIT-ACTION
    }
    ENTRY-ACTION
}</pre></td></tr>
</table>

<table>
<tr><td>

```
switch (EXPRESSION) {
   SWITCH-LABELS-1
       STATEMENTS1
   SWITCH-LABELS-2
       STATEMENTS2
   ...
}
```

where SWITCH-LABELS-N contains:

```
case CONST-EXPESSION:
```

</td><td>

```
{
int switchVar = EXPRESSION;
EXIT-ACTION
switch (switchVar) {
    NEW-SWITCH-LABELS-1 {
        ENTRY-ACTION
        STATEMENTS1
        EXIT-ACTION
    }
    NEW-SWITCH-LABELS-2 {
        ENTRY-ACTION
        STATEMENTS2
        EXIT-ACTION
    }
    ...
}
ENTRY-ACTION
```

where NEW-SWITCH-LABELS-N contains:

```
case ((int)(CONST-EXPESSION)):
```

[15]

</td></tr>
<tr><td>

```
while (EXPRESSION)
    STATEMENT
```

</td><td>

```
{
    bool cond = EXPRESSION;
    EXIT-ACTION
    if (cond)
       do {
          ENTRY-ACTION
          STATEMENTS
          cond = EXPRESSION;
          EXIT-ACTION
       } while (cond);
    ENTRY-ACTION
}
```

</td></tr>
<tr><td colspan="2">`for` and `do` statements are transformed similarly to `while` statements. For more information about loops see [HOFSTADTER].</td></tr>
<tr><td colspan="2">Labeled statements that aren't loops cause entail starting a new basic block at their end, since a `break` statement can be used to exit them abruptly.</td></tr>
<tr><td>

```
BREAK-STATEMENT
```

</td><td>

```
{
    EXIT-ACTION
    BREAK-STATEMENT
}
```

</td></tr>
<tr><td>

```
CONTINUE-STATEMENT
```

</td><td>

```
{
    EXIT-ACTION
    CONTINUE-STATEMENT
}
```

</td></tr>
<tr><td>

```
throw EXPRESSION;
```

</td><td>

```
{
    EXIT-ACTION
    throw EXPRESSION;
}
```

[16]

</td></tr>
</table>

[15] It's not locally known what's the type of the `switch` expression. However, it so happens that all possible types can be safely converted to `int`. The switch label values require conversion as well.

[16] Note the evaluation of `EXPRESSION` occurs *after* the basic block exit action. Resolving this requires global type information.

catch (FORMAL-PARAM){
  STATEMENTS
}
	catch (FORMAL-PARAM){
	  ENTRY-ACTION
	  STATEMENTS
	  EXIT-ACTION
	}

finally {
  STATEMENTS
}
	finally {
	  ENTRY-ACTION
	  STATEMENTS
	  EXIT-ACTION
	}

EXPRESSION1 **&&**
EXPRESSION2
	conditionResult(EXPRESSION1) **&&**
	  EXPRESSION2
	
	and at class scope:
	private boolean conditionResult(
	  boolean leftResult) {
	  EXIT-ACTION
	  ENTRY-ACTION
	  return leftResult;
	}

The "`||`" conditional operator is transformed similarly to "`&&`". The issue of the "`?:`" conditional operator is more complex, and may require global type information.	

6 Appendix B: Instrumentation Configuration DTD

This appendix provides an example of what an XML representation of an instrumentation configuration looks like. For readability, **comments** and *CDATA* are emphasized. This representation uses its own DTD, under the namespace "`http://www.tromer.org/jie`". For details see [XML] and [XML-NAMES]. This example is not a formal definition, but suffices in order to provide the framework for further elaboration, to be done on per-need basis.

```
<XML xmlns="http://www.tromer.org/jie">
<JIE>

<!-- Global parameters -->
<INPUT url="~/project/src/" />
<OUTPUT url="~/project/jied/" />
<LOGGING_STATEMENT str='java.lang.System.out.print("${}");' />

<!-- Global filter: limit to own packages -->
<FILTER type="package" value="org.tromer" />

<!-- Log the instrumentation ID each time a basic block in our
package is entered or exited -->

<RULE name="Basic block logging" opportunity="NominalBasicBlock">
     <ENTRY_ACTION>
          <WRITE_TO_LOG>${jiid} entered</WRITE_TO_LOG>
     </ENTRY_ACTION>
     <EXIT_ACTION>
          <WRITE_TO_LOG>${jiid} exited</WRITE_TO_LOG>
     </EXIT_ACTION>
</RULE>

<!-- Write method execution time in milliseconds to stdout, for the
first execution of each method in the BitBucket class.
This rule is shown in the GUI screenshots in chapter 4.
-->

<RULE name ="Method stopwatch" opportunity="Method">
     <ENTRY_ACTION>
          long startTime__ =
               Calendar.getInstance().getTimeInMillis();
     </ENTRY_ACTION>
     <EXIT_ACTION>
          long diff__ = Calendar.getInstance().getTimeInMillis()
                        - startTime__;
          System.out.println("${methodNameSig} took "+diff__+"ms");
     </EXIT_ACTION>
     <FILTER type="class" value="org.tromer.test.BitBucket" />
     <DYNAFILTER type="once" />
</RULE>

</JIE>
</XML>
```

7 Appendix C: Global Type Information

A current design limitation of JIE is lack of global type information – method signatures and modifiers, field types and inheritance relationships for arbitrary classes. JIE processes its input sourcecode one file at a time, so it does not have access to information declared anywhere but in the currently processes file. Furthermore, some useful information may be available only in compiled form (`.class` files), or in compiled and compressed archives (`.jar` or `.zip` files).

This limitation entails some limitations on the offered set of instrumentation opportunities, and on their details of operation. A useful and usable set of opportunities can be provided nonetheless, as described in this document. However, it seems prudent to suggest means of resolving this issue, as it may form a barrier for future extension.

One way to collect global type information is to parse all classes imported by the currently processed files, whether in sourcecode form or in binary form. Clearly this approach is very costly in terms of implementation time, performance in maintainability.

An alternative approach is using the Java Reflection API. This set of classes, added to the Java specification in JDK 1.1, allows Java code to load a compiled Java classes at run-time and make queries about their content: inheritance, modifiers, methods – in fact, all the elements required by JIE.

For instance, the following program (based on [FLANAGAN]) uses the Java reflection API to print the modifiers, name, superclass and implemented interfaces of an arbitrary Java class, whose name is provided as a string.

```
import java.lang.reflect.*;
public class ShowClass {
public static void main(String[] args) throws ClassNotFoundException
{
    // Get class object. Loads the class if needed.
    Class c = Class.forName(args[0]);
    print_class(c);
}

public static void print_class(Class c)
{
    // Print modifiers and name and superclass:
    System.out.print(
        Modifier.toString(c.getModifiers()) +
        " class " + c.getName() +
        " extends " + c.getSuperclass().getName());

    // Print implemented interfaces:
    Class[] interfaces = c.getInterfaces();
    if ((interfaces != null) && (interfaces.length > 0)) {
        System.out.print(" implements ");
        for(int i = 0; i < interfaces.length; i++) {
            if (i > 0) System.out.print(", ");
            System.out.print(interfaces[i].getName());
      }
    }
}
```

This solution has the following implications:

- All classes must be available in compiled form that is up-to-date. This means that the instrumented files need to be compiled twice – once before instrumentation and once after.

- Since the classes will actually be loaded, all Java run-time parameters must be set correctly, as if the instrumented program was about to be executed.
- JIE and the class loaded for reflection exist under the same Java Virtual Machine. This may create unexpected interaction. For instance, library conflict can result if JIE and the instrumented sourcecode rely on different versions of the same external library.
- Instrumentation of even a tiny portion of the program may cause a cascade of class loading that may have a major performance impact.

Furthermore, in order to make use of global type information there's need for very elaborate instrumentation transformations. In particular, expressions need to be taken apart and analyzed for the types of their constituent parts, taking into account scope, nesting and casting rules. This analysis is far from trivial.

As can be seen, this solution for the global type information issue is fairly complex and clearly outside the scope of the current design and implementation stages. However, should the need arise in the future, it offers the means for future advancements.

8 Appendix D: Parsing Tool License Agreements

This appendix provides the portions of the tool license agreements that may affect their suitability for this project.

8.1 JavaCC (excerpts)

These are relevant excerpts from the JavaCC license agreement, as provided on the download page:
http://www.suntest.com/JavaCC/javaccdownload.html
Note that the Metamata Parse license agreement is different, and can be found at:
http://www.metamata.com/download/license.html

```
     Java Compiler Compiler Download and License Agreement

SUN MICROSYSTEMS, INC. ACTING BY AND THROUGH ITS SUN TEST GROUP ("SUN") IS WILLING TO
LICENSE THE JAVA(TM) COMPILER COMPILER(TM) SOFTWARE, SAMPLE GRAMMARS, AND THE
ACCOMPANYING DOCUMENTATION ("SOFTWARE") TO YOU ONLY ON THE CONDITION THAT YOU ACCEPT
ALL OF THE TERMS IN THIS AGREEMENT.

...

1. LICENSE

You are permitted to download, install, and use the Software on a single computer or a
network server. You may use the Software as installed on a network server on clients
of that server. You may copy the Software only for backup purposes in support of your
use of the Software, provided that you reproduce all copyright and other proprietary
notices that are on the original copy of the Software.

2. SAMPLE GRAMMARS

You may modify the sample grammars included in the Software to develop derivatives
thereof ("Sample Grammar Derivatives"), and sublicense the Sample Grammar Derivatives
directly or indirectly to your customers.

3. DEVELOPED PRODUCTS

You may use the Software to generate software program(s) ("Developed Programs"). Sun
claims no rights in or to the Developed Programs.

...
```

8.2 JTB

This is the JTB license agreement, as shown during the installation process.

It is not clear from this license whether commercial use of generated code is permitted.

```
All files in the distribution of JTB, The Java Tree Builder are
Copyright 1997, 1998 by the Purdue Research Foundation of Purdue
University.  All rights reserved.

Redistribution and use in source and binary forms are permitted
provided that this entire copyright notice is duplicated in all
such copies, and that any documentation, announcements, and
other materials related to such distribution and use acknowledge
that the software was developed at Purdue University, West Lafayette,
Indiana by Kevin Tao and Jens Palsberg.  No charge may be made
for copies, derivations, or distributions of this material
without the express written consent of the copyright holder.
Neither the name of the University nor the name of the author
may be used to endorse or promote products derived from this
material without specific prior written permission.
```

```
THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR ANY PARTICULAR PURPOSE.
```

8.3 SableCC (excerpts)

This is an excerpt from the GNU Library General Public License (LGPL), under which SableCC is distributed. The LGPL is similar to the GPL, but provides somewhat more liberal provisions for some uses. SableCC includes utility classes that need to be distributed along with the final product, so this is directly relevant.

The LGPL can be found at

http://www.gnu.org/copyleft/lgpl.html

In this excerpt, “Library” refers to SableCC.

```
...

6. As an exception to the Sections above, you may also compile or link a "work that
uses the Library" with the Library to produce a work containing portions of the
Library, and distribute that work under terms of your choice, provided that the terms
permit modification of the work for the customer's own use and reverse engineering for
debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in
it and that the Library and its use are covered by this License. You must supply a
copy of this License. If the work during execution displays copyright notices, you
must include the copyright notice for the Library among them, as well as a reference
directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for
the Library including whatever changes were used in the work (which must be
distributed under Sections 1 and 2 above); and, if the work is an executable linked
with the Library, with the complete machine-readable "work that uses the Library", as
object code and/or source code, so that the user can modify the Library and then
relink to produce a modified executable containing the modified Library. (It is
understood that the user who changes the contents of definitions files in the Library
will not necessarily be able to recompile the application to use the modified
definitions.)

b) Accompany the work with a written offer, valid for at least three years, to give
the same user the materials specified in Subsection 6a, above, for a charge no more
than the cost of performing this distribution.

c) If distribution of the work is made by offering access to copy from a designated
place, offer equivalent access to copy the above specified materials from the same
place.

d) Verify that the user has already received a copy of these materials or that you
have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include
any data and utility programs needed for reproducing the executable from it. However,
as a special exception, the source code distributed need not include anything that is
normally distributed (in either source or binary form) with the major components
(compiler, kernel, and so on) of the operating system on which the executable runs,
unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other
proprietary libraries that do not normally accompany the operating system. Such a
contradiction means you cannot use both them and the Library together in an executable
that you distribute.

...
```

9 Appendix E: Implementation Matrix

This appendix offers estimations of complexity and importance of various aspects of JIE, and suggests a functionality subset to serve as the initial implementation goal.

This design document is very liberal in the functionality it describes. With few exceptions, all envisioned features were described in detail. As a result, a full implementation of all the features described herein dramatically exceeds the planned scope for an initial implementation. A subset of that functionality should be defined in a way that balances relative priorities with scope of effort.

The following table lists major functionality components of JIE and ranks each in terms of priority and effort. The highlighted rows define the proposed subset.

Priority:[17]
- **A** – Mandatory
- **B** – Desired
- **C** – Nice to Have

Effort:
- **1** – Easy (given the dependencies)
- **2** – Moderate effort
- **3** – Significant effort
- **4** – Lots of coffee and pizza

	Priority	Effort
Basic functionality:		
Detailed design	A	3
Sourcecode transformation mechanism	A	4
Interface with Java parsing tool	A	4
Context parameters	A	2
XML DTD detailed definition	A	2
Interface with XML parser	A	3
Global type information	B	4+
Opportunities:		
Method (Entry/Exit)	A	1
Nominal Method (Entry/Exit)	A	1
Class (Entry/Exit)	B	1
Class Instantiation	A	2
Wait Set Operation	A	2
Field Read	A	2
Field Write	A	2
Exception Throw	C	1
Exception Catch	C	1
Nominal Basic Block	A	3+
Generic static filters:		
Package	A	1
Class	A	1
Method	A	1
Class visibility	C	2
Method visibility	C	2
Excluded JIIDs	B	3
Generic dynamic filters:		
Enable/Disable	B	2
Once Only	B	2
Every N Times	C	2
Minimum Time Interval	C	2
Actions:		
Action Template	A	3
Write to Log	B	1
Application wrappers:		
Command-line application	A	1
GUI application	A	4

[17] Based on discussion and general impressions. This is rather volatile.

10 Bibliograpgy

Note: On-line resources are included in this section only if it was inappropriate, for stylistic reasons, to include them in the main text. It is assumed that on-line resources will be accessed from this document mainly when it is in electronic form, and in that case in-text links are preferable.

[ANTLR-REF] Terence Parr, *"ANTLR Reference Manual"*, http://www.antlr.org/doc/

[CCCT] German National Research Center for Information Technology, *"Catalog of Compiler Construction Tools"*, http://www.first.gmd.de/cogent/catalog/

[CFCI] David Muir Sharnoff et al., *"Catalog of Free Compilers and Interpreters"*, http://www.idiom.com/free-compilers/

[FLANAGAN] David Flanagan, *"Java in a Nutshell"*, O'Reilly, May 1997 http://www.oreilly.com/catalog/javanut2/ (table of contents and general information)

[GAGNON] Étienne Gagnon, *"SableCC, an Object Oriented Compiler Framework"*, School of Computer Science, McGill University, March 1998, http://www.sable.mcgill.ca/sablecc/#Documentation

[GAMMA] Erich Gamma at al., *"Design Patterns, Elements of Reusable Object-Oriented Software"*, Addison Wesley, 1994

[HOFSTADTER] Douglas Hofstadter, *"Gödel, Escher, Bach – an Eternal Golden Braid"*, Basic Books, 1979

[JAVA-LANG] James Gosling et al, *"The Java™ Language Specification"*, Addison Wesley Longman, 1996 http://java.sun.com/docs/books/jls/ (full content available on-line)

[JAVA-INNER] Sun Microsystems, Inc. *"Inner Classes Specifications"*, February 1997 http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc.html

[XML] World Wide Web Consortium, *"Extensible Markup Language (XML) 1.0"*, February 1998, http://www.w3.org/TR/REC-xml

[XML-NAMES] World Wide Web Consortium, *"Namespaces in XML"*, January 1999, http://www.w3.org/TR/REC-xml-names