Java Instrumentation Engine
version 1.01
IMPLEMENTATION AND USAGE NOTES

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

This document provides an overview of the implementation of Java Instrumentation Engine version 1.0. It notes the capabilities of this version, documents its usage and provides various details that may be of interest during use or extension.

This document, as well as JIE itself, is based on *"Java Instrumentation Engine: Revised Design Document"*. Familiarity with the concepts and terminology presented in that document is assumed. In case of discrepancy, this document takes precedence.

1 Package Content

The JIE v1.0 package contains the following:

`JIE.jar`	Compiled JAR file (requires `lib/*`)
`lib/*`	External libraries used by `JIE.jar`
`org/tromer/jie/*`	JIE sourcecode
`JavaDoc/*`	JavaDoc documentation of JIE sourcecode
`doc/*`	Various documents
`autogen/*`	Programs and data for creating the machine-generated parts of JIE
`test/*`	Test data

Each directory contains an informative `README.txt` file.
Additional details are provided below.

2 Capabilities

JIE v1.0 has the following capabilities:

- Full support for Java syntax, including JDK 1.2 extensions and undocumented features supported by `javac`.
- Reading of XML-based configuration files using a specialized XML data type
- Instrumentation opportunities:
  - Method (Entry/Exit)
  - Nominal Method (Entry/Exit)
  - Class (Entry/Exit)
  - Class Instantiation
  - Nominal Basic Block
- Logging: produces lists of basic blocks, methods, classes, etc.
- Generic static filters, both global and per-rule:
  - Package
  - Class
  - Method
- Generic context parameters
- Command-line based activation

All features scheduled for phase 1 of JIE are fully implemented, as well as three additional features (logging, context parameters and XML configuration).

3 Usage

Command line:

```
jie configfile
```

or

```
jie configfile input output [-relative]
```

Where `jie` is the command used to run JIE. For instance, using JDK 1.2:

```
java -jar JIE.jar other-parameters
```

or, using *Kaffe* 1.04b:

```
kaffe -addclasspath lib/xml4j.jar -jar JIE.jar other-parameters
```

The configuration file is an XML file containing an instrumentation configuration using the JIE XML data type (see below).

Input and output paths may be specified in the configuration file, in the command line, or both. If they are specified in both, the command line takes precedence.

The `-relative` switch means input and output paths are relative to the configuration file, instead of the current directory. Note that this is always the case for paths that appear in the configuration file.

If the input path is a file, JIE will create an instrumented copy of that file and name it according to the output path. If the input file is a directory, JIE will recursively traverse this directory. The output path will then be a directory (newly created, if needed) containing instrumented versions of all the `.java` files found in the input directory. Existing files in the output directory will *not* be deleted. The input path may not be a subdirectory of the output path, or vice versa.

All input files must be valid Java 1.2 sourcecode.

If logging is used (see below), additional files will be created for each instrumented source files. These files will be of the form *source*`.`*opportunity*`.log`. For instance., the list of basic blocks found in `moose.java` will be called `moose.java.NominalBasicBlock.log`, in the same (output) directory.

Note that `JIE.jar` uses `lib/xml4j.jar` (see below).. This is reflected in the manifest file of `JIE.jar`, so JDK 1.2 handles the dependency automatically, but other JVMs require explicit `-classpath` or `-addclasspath` arguments. Also, if you copy `JIE.jar` you'll need to copy `xml4j.jar` along with it, and consider consulting the IBM *XML Parser for Java* license agreement.

Here is a transcript of a successful JIE run:

```
% java -jar JIE.jar test/ClassOpp/conf.jie in out -relative
ClassOpp/in/HelloWorld.java ~=> ClassOpp/out/HelloWorld.java
ClassOpp/in/Assert.java ~=> ClassOpp/out/Assert.java

Done.
No errors encountered.

2 file(s), 2.284 seconds, average 1.142 seconds per file.

12 transformations total, average 6.0 per file. Breakdown by rule:
    Class: 12  (logged)
```

where `test/ClassOpp/conf.jie` contains:

```
<JIE>
<RULE opportunity="Class" log="1">
<ENTRY_ACTION>
```

```
{ System.err.println("Starting init code of ${className}"); }
</ENTRY_ACTION>
<EXIT_ACTION>
{ System.err.println("Finishing init code of ${classIdent}"); }
</EXIT_ACTION>
</RULE>
</JIE>
```

4 XML Configuration File

JIE accepts the instrumentation configuration as an XML file. The structure of the XML file is as follows:

Tag	Attributes	Content (optional)

JIE	*none*	`INPUT`
		`OUTPUT`
		`RULE`
		`FILTER`

	Top-level tag. All other tags are descendants of the `JIE` tag. There must be exactly one `JIE` element in the XML file, but it does not have to be the root.	
INPUT	`url` (string)	*none*
	Input path (relative to configuration file)	
OUTPUT	`url` (string)	*none*
	Output path (relative to configuration file)	
RULE	`name` (string, optional) `opportunity` (see below) `log` (`"0"` or `"1"`, optional, default `"0"`)	FILTER ACTION ENTRY_ACTION EXIT_ACTION
	Defines a new rule with the given name and instrumentation opportunity. Currently the name is used only for presentation of statistics. The `opportunity` attribute is one of those listed below this table. If the opportunity is "Entry/Exit", then there must be either an `ENTRY_ACTION` element, an `EXIT_ACTION` element, or both. For other opportunities, there must be an `ACTION` element. If `log` is "1", a log file will be created for every source file, containing details on the transformations carried out. The content is opportunity-specific: e.g., *Nominal Basic Block* will produce a list of basic blocks.	
ACTION	*none*	*CDATA*
	Contains free text (subject to XML escape codes). Defines the action template for rules using a single-action opportunity.	
ENTRY-ACTION	*none*	*CDATA*
	Contains free text (subject to XML escape codes). Defines the action template for an entry action, in Entry/Exit rules.	
EXIT-ACTION	*none*	*CDATA*
	Contains free text (subject to XML escape codes). Defines the action template for an exit action, in Entry/Exit rules.	
FILTER	`type` (`"package"`, `"class"` or `"method"`) `value` (string)	*none*

	Defines a filter. If directly nested under a `JIE` tag, this is a global filter. If placed under a `RULE` tag, this is a local rule filter. Filters are combined using logical AND. Only packages, classes and methods matching the pattern defined by `value` will be instrumented. A *pattern* contains zero or more *simple patterns* separated by "|". The string matches the patten it matches at least one of the simple patterns, or if the pattern is empty. A simple pattern may contain at most one "`*`" wildcard, which matches any string. If a simple pattern is of the form "`*.xxx`", then as a special case it also matches "`xxx`" (useful for the unnamed package). If it of the form "`xxx.*`", then as a special case it also matches "`xxx`" (useful for filtering by method or class name regardless of package). Leading and trailing spaces are ignored.

Possible values for the `opportunity` attribute of the RULE tag are:

- `Class` (Entry/Exit)
- `ClassInstantiation` (Entry/Exit)
- `NominalBasicBlock` (Entry/Exit)
- `NominalMethod` (Entry/Exit)
- `Method` (single action)

The above list is automatically produced by JIE when ran with the `--help` argument (or no argument). For details about the meaning of each opportunity, see "*Java Instrumentation Engine: Revised Design Document*".

Action templates may contain the following context parameters, which are expanded according to the context of the transformation:

Macro and type	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[1]
`${classIdent}` identifier	`Demo`	Unqualified class name[2]
`${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}`	`show`	Unqualified method name[3]

[1] For inner classes, after the package name '`$`' is used as a separator instead of '`.`'.

[2] 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`*. For anonymous classes, same as the above using "`_anonymous_`" as the class name.

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

identifier		
`${methodIdentSig}` method header	`void show(String)`	Unqualified method name with type signature
`${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.

Numerous examples of JIE configuration files can be found in the test suit included in this package; each subdirectory of `test` includes a `conf.jie` file used for running the test.

5 Machine-generated sourcecode

Some portions of JIE are machine-generated. Specifically, all files in the following packages are machine-generated and should not be edited directly:

- `org.tromer.jie.syntaxtree`
- `org.tromer.jie.visitor`
- `org.tromer.jie.parser`

It is perfectly possible to make significant changes to JIE without the need to re-generate the machine generated portions. However, it is not recommended to manually edit these portions. Instead, the generation scripts should be modified to reflect the required changes.

The autogeneration process is rather complex, and involves numerous external tools and custom scripts. All scripts, input and output files are included in the `autogen` directory. See `autogen/README.txt` for more details. The autogenerated process requires Unix system utilities, Perl 5.004 or later and a Java Virtual Machine. It was tested under RedHat Linux 6.0, and should run with few modifications on other Unix systems, but is unlikely to run on non-Unix systems without major changes.

The following is transcript of a successful autogeneration run (not including JTB and JavaCC messages), which should provide a general impression of the tasks carried out.

```
JIE root directory is "/phor/e/Data/jie".
Java runtime found, using "-addclasspath" for classpath arguments.
Preparing JTB output directory "jtb_out".

Running JTB...
---JTB output here---
Done.

Fixing bug in JTB's TreeDumper visitor.
patching file `TreeDumper.java'
Removing position fields from NodeToken.

Creating additional visitor classes...
Generating NullObjectVisitor
```

For instance initializers, the illegal identifier "`*instance-initializer@`*`line`*`#`*`column`*`*`".
For field initializers, the illegal identifier "`*field-initializer@`*`line`*`#`*`column`*`*`".

```
Generating GreedyDepthFirst
Generating CallingDepthFirst
Generating PrePostDepthFirst
Generating PrePostTreeDumper
Done.

Preparing JavaCC output directory "javacc_out".

Running JavaCC... (this may take a while)
---JavaCC output here---
Done.

Adding 'package' declarations to parser source files.
Creating and evacuating source tree directories at "org/tromer/jie".
Copying generated source files into source tree...

Autogeneration tasks completed successfully.
```

6 Testing and Profiling

A comprehensive test suit was used during development, and is available as a part of the JIE package for regression testing of future versions. The tests consist of manually constructed input designed to test atypical, complex or edge cases, as well as some randomly chosen "typical" input. The test suit covers the filter mechanism and all instrumentation opportunities, but not the command-line application.

The test suit is available in the `test` directory. See `test/README.txt` for technical details.

JIE was profiled for hotspots using Intuitive Systems Inc.'s *OptimizeIt Professional* v3.02. The resulting optimizations, including alternative implementations of several standard JDK classes, produced a 200% performance improvement on typical input.

During development, JIE was tested on a large subset of the JDK 1.1 sourcecode. These tests isolated numerous problematic conditions related to fine points in the language definition, and revealed a few undocumented language features supported by the `javac` compiler. All of these conditions were addressed.

Output validity was tested by compiling the instrumented JDK 1.1 sourcecode using the following compilers:

- Sun JDK 1.1 (`javac`)
- Sun JDK 1.2 (`javac`)
- Symantec *Visual Cafe* 3.0c (`sj`)
- IBM *Jikes* 1.02
- *Pizza* 0.39g

JIE makes rather atypical use of some language features, which causes excessive stress for some compilers. Numerous bugs in the JDK 1.1 and *Visual Cafe* 3.0c compilers were encountered (and acknowledged by Sun and Symantec, respectively). A single error was encountered in JDK 1.2 (anonymous classes in interface field initializers), and is fixed in the current beta version of JDK 1.3. Notably, IBM's *Jikes* v1.02 compiler accepted all valid input.

As a final test, JIE was tested on the full JDK 1.2 Java sourcecode (4,030 files, 43MB), using a configuration which invokes all instrumentation opportunities. The test ran flawlessly (except for 4 invalid input files that were also rejected by Java compilers), with an average throughput of 320 milliseconds per file on a Pentium II 350MHz running Windows NT 4.0 using JDK 1.2.

7 Known Issues

1. The *Nominal Basic Block* instrumentation opportunity does not handle the "`?:`" conditional operator. I've been unable to find any way to implement this without global type information.

2. In the *Nominal Basic Block* instrumentation opportunity: for a basic block that ends with "`throw expression;`", the exit action is executed before the evaluation of the expression. Resolving this requires global type information.

3. For instrumentation at the beginning of methods (e.g., *Method* and *Nominal Basic Block)*: When explicit constructor invocation ("`this()`" or "`super()`") is used, the entry action happens *after* the explicit constructor invocation. In the general case it's impossible to insert instrumentation code before their invocation. For more information, see the relevant item in the Java Language Specification at: http://java.sun.com/docs/books/jls/html/8.doc.html#78435. Note, however, that conditional operators ("`&&`" and "`||`") in the parameter list of an explicit constructor invocation are properly transformed and logged.

4. If there are several rules employing the same instrumentation opportunity (possibly with different actions) and more than one of them has logging enabled, each rule overrides the previous one and only the log file of the last rule is retained. This is unlikely to be an issue under practical use, and could be resolved by changing the log file naming convention.

5. Code that uses a block-local inner class may be transformed incorrectly, since the name of the inner class is known only inside the block while the transformation may require helper methods outside the block. Resolving this may require global type information. This situation was never encountered in real-world tests.

8 External tools

The following tools are used by JIE:

Name: XML Parser for Java
Version: 2.0.15 *(1999/26/8)*
Source: IBM Corp.
Use: Reading instrumentation configuration from XML files, during run-time.
Home page: http://www.alphaWorks.ibm.com/tech/xml4j
Used files: `xml4j.jar`
Location: `lib/`

Name: Java Compiler Compiler (JavaCC)
Version: 1.1 *(1999/08/10)*
Source: Sun Microsystems and Metamata
Use: Parser generator, used during autogeneration.
Home page: http://www.metamata.com/JavaCC/
Used files: `JavaCC.zip`
Location: `autogen/bin/`

Name: Java Tree Builder (JTB)
Version: 1.1 *(1999/01/10)*
Source: Kevin Tao, Purdue University
Use: Tree builder and processor for JavaCC, used during autogeneration.
Home page: http://www.cs.purdue.edu/jtb/
Used files: `jtb.jar`
Location: `autogen/bin/`

Name: Java 1.2 grammar file for JavaCC
Source: David Williams
Use: Input to JavaCC.
Home page: http://www.metamata.com/javacc/java12grammar.html
http://www.metamata.com/javacc/java12grammars.zip
Used files: `Java1.2.jj`
Location: `autogen/input/`

The JIE package includes binaries of all of the above tools, and therefore its distribution is affected by their license agreements. The minimum run-time requirements are the *XML Parser for Java* JAR file.