An Integrated Verification Environment for JML: Architecture and Early Results

Patrice Chalin, Perry R. James, and George Karabotsos

Dependable Software Research Group
Computer Science and Software Engineering Dept.
Concordia University, Montreal, Canada
{chalin, perry, george}@dsrg.org
Road map

• JML: language & tools
• Requirements for Next-generation
• JML4 features
• JML4 architecture
• Early benefits
Java Modelling Language

- A language for describing behavior of code
- Tools to ensure they match
Java Modelling Language

• A language for describing behavior of code
  – DbC with lightweight specs
  – Full BISL with heavyweight specs

• Tools to ensure they match
Java Modelling Language

- A language for describing behavior of code
- Tools to ensure they match

- RAC: JML compiler (`jmlc`)
- ESC: ESC/Java2
- FSPV: LOOP, JACK
- testing: JmlUnit
- doc: JmlDoc
- autogen: JmlSpec, Daikon, Houdini
Current State of affairs: Limitations of current tools

Lots of good tools... but

• Not interoperable
• Own parsers, desugarers, etc.
• Out of date
  – Java 5 released in September 2004
  – Still no support for generics
• Mostly command-line driven
Current State of affairs: What worked well in JML 2

• Common JML tool suite
  – Checker, RAC, JmlUnit
• Built on MultiJava compiler (MJ)
  – MJ mostly independent of JML
  – JML subclasses MJ classes & overrides methods
  – Extension points
    • Calls to empty methods

This idea used in JML 4
Requirements for any Next-generation JML tools

• Remove duplication of effort
  – Tool developers
  – Analysis

• No maintenance of a Java compiler

• Integrated (development and) Verification Environment (IVE)
  – Support RAC, ESC, and FSPV

JML4 achieves these
JML4

• Built atop Eclipse, integrated with the JDT
• Currently supports
  – Processes annotations in .java and .jml files
  – Non-null type system
    • Static enforcement
    • RAC generation (desugared)
  – Initial Design by Contract
    • Initial integration with ESC/Java 2
    • RAC generation
High-level Package view

JML 4 replacement for JDT plug-in
+ additional UI plug-in

JML 4 changes / introduces packages in **bold**

Everything’s a plug-in! (except this small bit)
Compilation Phases

Inline & external specs processed

Static verification before code generation so it can influence runtime checking
Eclipse JDT: Lexical analysis

- Hand crafted
- Tedious to modify keywords
- JML in special comments
  - Easy to switch to augmented keywords
Eclipse JDT: Parsing – 2 kinds

- **Diet parsing**
  - Method bodies skipped
  - Only signature information
- **Full parsing**
  - Method bodies processed
  - All info available
- **For memory efficiency**
  - All diet parsed
  - Full parsed individually, then discarded
Eclipse JDT: Parsing

+ Parser generated using JikesPG
+ Grammar follows Java Language Specification
+ One semantic action per reduction
- Little support for token stacks
- Replaced calls to ASTNode constructors with JML-specific versions
- Documentation only in German
Eclipse JDT: Customizing the lexer and parser

replaces manual process
Eclipse JDT: Part of the AST hierarchy

- No copy & change of code
- Only overriding & hooks

-Jml types shadow originals
Eclipse JDT:
Type checking & Flow analysis

- Changed to support non-null type system
- Extended with hooks (calls to empty methods) added in original `resolve` and `analyseCode` methods
**Eclipse JDT:**

**Static verification**

- Originally delegated to ESC/Java2
- Now working to use
  - Eclipse as a front end
  - ESC/Java2 back end
- Later steps are to
  - Optionally remove RAC for proved properties
  - Add interface for FSPV
public static void generateNullityTest(
    CodeStream codeStream,
    String exceptionType,
    String msg) {
    BranchLabel nonnullLabel =
        new BranchLabel(codeStream);
    codeStream.dup();
    codeStream.ifnonnullnonnullLabel;
    codeStream.newClassFromName(exceptionType,
        codeStream.athrow();
    nonnullLabel.place();
}
**JML 4 Validation**

- Compiler is kept up to date with new features
  - JDT already supports Java 6
- No copy & change of JDT code
  - use subclassing and method extension points
  - bracketing our changes with special comments
- CVS vendor branches
- Merging in weekly updates is painless
  - takes on average < 10 min.
JML 4: Early benefits

• Ran JML 4 on ESC/Java2
• New problems found in Main class
JML 4: Early benefits

VcGenerator vcg = null; ...
try {
    ... // possible assignment to vcg
} // multiple catch blocks
catch (Exception e) {
    ... 
}
...
fw.write(vcg.old2Dot()); // possible NPE
JML 4: Early benefits

Potential null pointer access: The variable vcg may be null at this location.
In a superclass of Main

```java
static public Options options = null;
```

In Main

```java
public static Options options() {
    return (Options)options;
}
```

250+ occurrences of

```java
options().someField
or options().someMethod()
```
JML4: Next steps

- Continue adding support JML level 0 (and above)
- Enhance ESC support
- Include interface for FSPV
- ...

P.R. James, p. 25
Related work

• JML 3
  – A proper plug-in → doesn’t use non-API classes
  – Needs its own parser, type checker, etc.
• JML 5
  – Specifications in ‘@’ annotations
  – Can’t put annotations everywhere we want
  – Needs its own parser, type checker, etc.
## Related work

<table>
<thead>
<tr>
<th>Base Compiler / IDE</th>
<th>Name</th>
<th>JML2</th>
<th>JML3</th>
<th>JML4</th>
<th>JML5</th>
<th>ESC/Java2 Plug-in</th>
<th>JACT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintained (supports Java &gt;5)</td>
<td>MJ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓†</td>
<td>✓</td>
</tr>
</tbody>
</table>

| Reuse/extension of base (e.g. parser, AST) vs. copy-and-change | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

<table>
<thead>
<tr>
<th>Tool Support</th>
<th>RAC</th>
<th>ESC</th>
<th>FSPV</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

P.R. James, p. 27
Conclusion

• Integrated (development and) Verification Environment (IVE)
• Support RAC, ESC, and FSPV
• No need to maintain a Java compiler
• Unify support to remove duplication of effort
An Integrated Verification Environment for JML

Thank you!