JML and Aspects: The Benefits of Instrumenting JML Features with AspectJ

Henrique Rebêlo
Sérgio Soares
Ricardo Lima
Paulo Borba
Márcio Cornélio
Java Modeling Language

- **Formal specification language** for Java
  - behavioral specification of Java modules
  - sequential Java
- Adopts design by contract based on Hoare-style with **assertions**
  - pre-, postconditions and invariants
- Main goal → **Improve functional software correctness** of Java programs
jmlc: compilation passes

Annotated Java Source File

- Parsing/typechecking
- JML Type Checker
  - AST generation
  - Typechecked AST
  - Assertion methods (AM) generation

MultiJava compiler

- Instrumented bytecode generation
- TJSF compilation
- Temporary Java Source File
  - Assertion methods (AM) printing in a TJSF

RAC code generation

- RAC code printing

Runtime checks

Front-end

Code Generation

MultiJava Compiler
Problem

- JML limitation
  - The JML compiler does not work properly when applied to other Java platforms
  - Example: Java ME platform
    - Data structures (e.g. HashSet)
    - Java reflection mechanism
Our approach: previous work

- Verify JavaME/SE programs with JML
  - AOP/AspectJ

We use the AspectJ to
- translate JML features into aspects
- generate bytecodes compliant with Java ME/SE
- verify if bytecode respects the JML features during runtime
Aspect-oriented programming… is Quantification + Obliviousness

(Filman, Elrad, Clarke, and Aksit 2005)
Aspect-oriented languages are quite popular... due to the promise of modularizing crosscutting concerns
ajmlc: implementation Strategy

Front-end

Annotated Java Source File

Parsing/typechecking

JML Type Checker

AST generation

Typechecked AST

Aspect Code Generation

Aspect Assertion methods (AAM) generation

Runtime checks

AspectJ Compiler

Rebêlo and Colleagues’ approach

Temporary Aspect Source File

TASF compilation

Instrumented bytecode generation

AspectJ compiler

Aspect Assertion methods (AAM) printing in a TJSF

Aspect RAC code generation

Aspect RAC code printing
generated code: jmlc VS ajmlc

**jmlc**

- Java source file
- MultiJava
- System (.class)

**ajmlc**

- Java source file
- Weaver
- Woven System (.class)
- AspectJ source file
Research questions

- Does AOP represent the JML features?
- What is the order and relationship between the generated aspects?
- How to check Java ME apps using ajmlc (with aspects)?
- When is it beneficial to aspectize JML features?
- …
Contributions

- Answering the mentioned research questions
- Supporting new assertion semantics
- Generating instrumented bytecode when necessary
- Study — code size
- Guidelines for ajm1c
- …
The analogy between JML and Aspects

- AspectJ — An AOP extension for Java
  - dynamic crosscutting (e.g., before advice)
  - static crosscutting — ITD (e.g., new fields)
  - quantification
  - property-based crosscutting — wildcarding (*)

**execution** (* T.*(..))

Identifies executions to any method, with any return and parameters type, defined on type T.
The invariants analogy

class T {
    int i = 10;
    //@ invariant i == 10;
    void m() {...}
    void n() {...}
    void o() {...}
}

aspect T {
    before(T object) :
        exec(!static * T.*(..)) &&
        within(T+) &&
        this(object){
            if( !(object.i==10)){
                throw new Error("";
            }
        }
        after(T object) :
        exec(!static * T.*(..)) &&
        within(T+) &&
        this(object){
            if( !(object.i==10)){
                throw new Error("";
            }
        }
}

(→) JML feature as an aspect

(←) An aspect feature as JML spec

Both JML spec and aspect quantify the same points on type T
Behavioral subtyping analogy

Both JML spec and aspect quantify the same points on type T and its subtypes
Other analogies

- Not limited to:
  - constraint specifications
  - refinement
  - model-programs
  - ...

Other quantification points in JML that can be implemented using AspectJ
Ordering of advice executions into an aspect

- Before advices to check invariants
- Before advice to check preconditions
- After or around advices to check postconditions
- After advices to check invariants
Expression evaluation with new assertion semantics

We restructured the ajmlc compiler to deal with the new assertion semantics proposed by Chalin’s work. With this semantics, a JML clause can be entirely executable or not.

- We add two try-catch blocks
  - one for non-executable exceptions
  - another to handle all other exceptions
public class T {
    private int x, y;
    //@ requires b && (x < y);
    public void m(boolean b) {
        ...
    }
}
Add before-execution with new assertion semantics capability

```java
before (T object, boolean b) :
    execution(void T.m(boolean)) && {...{
    boolean rac$b = true;
    try{
        ...
        if (!rac$b) {...}
    }
    } catch (JMLNonExecutableException rac$nonExec){
        rac$b = true;
    } catch (Throwable rac$cause){
        if(...) {...}
        else {throw new JMLEvaluationError("\""};
    }
}
```
To verify Java ME applications, our compiler only generates aspects that avoid AspectJ constructs that are not supported by Java ME.

- Avoids AspectJ constructs such as…
  - cflow pointcut
  - cflow below pointcut
  - thisJoinPoint, …
Compiling empty classes

- **ajmlc** generates no code
- **jmlc**
  - generates 11.0 KB (source code instrumentation)
  - generates 5.93 KB (bytecode instrumentation)

```java
public class T {
}
```
## Jmlc VS ajmlc

<table>
<thead>
<tr>
<th>JML clauses</th>
<th>jmlc generates</th>
<th>ajmlc generates</th>
</tr>
</thead>
<tbody>
<tr>
<td>requires</td>
<td>yes ✓</td>
<td>no</td>
</tr>
<tr>
<td>ensures</td>
<td>yes ✓</td>
<td>no</td>
</tr>
<tr>
<td>signals</td>
<td>yes ✓</td>
<td>no</td>
</tr>
<tr>
<td>invariant</td>
<td>yes ✓</td>
<td>no</td>
</tr>
</tbody>
</table>
Study

- 3 Java SE applications
  - annotated with JML
  - extracted from JML literature
- We have compiled these programs
  - using `ajmlc` with **two different weavers**
    - `ajc`
    - `abc`
  - using `jmlc` (classical JML compiler)
Considered metric

- **Code size**
  - instrumented *source code* size
  - instrumented *bytecode* size
  - Jar size (bytecode size + JML lib)
## Results

### Source code instrumentation

<table>
<thead>
<tr>
<th></th>
<th>jmlc (KB)</th>
<th>ajmlc (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animal</td>
<td>28.8</td>
<td>4.8</td>
</tr>
<tr>
<td>Person</td>
<td>27.4</td>
<td>0.5</td>
</tr>
<tr>
<td>Patient</td>
<td>26.2</td>
<td>9.6</td>
</tr>
<tr>
<td>IntMathOps</td>
<td>18.2</td>
<td>2.0</td>
</tr>
<tr>
<td>StackAsArray</td>
<td>55.7</td>
<td>9.2</td>
</tr>
</tbody>
</table>

### Jar size

<table>
<thead>
<tr>
<th></th>
<th>jmlc (KB)</th>
<th>ajmlc (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hierarchy classes</td>
<td>33.6</td>
<td>18.7</td>
</tr>
<tr>
<td>IntMathOps</td>
<td>20.6</td>
<td>7.5</td>
</tr>
<tr>
<td>StackAsArray</td>
<td>25.2</td>
<td>11.7</td>
</tr>
</tbody>
</table>

### Bytecode instrumentation

<table>
<thead>
<tr>
<th></th>
<th>jmlc (KB)</th>
<th>ajmlc (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animal</td>
<td>13.3</td>
<td>17.0</td>
</tr>
<tr>
<td>Person</td>
<td>11.7</td>
<td>2.3</td>
</tr>
<tr>
<td>Patient</td>
<td>12.7</td>
<td>25.3</td>
</tr>
<tr>
<td>IntMathOps</td>
<td>9.39</td>
<td>5.4</td>
</tr>
<tr>
<td>StackAsArray</td>
<td>21.7</td>
<td>23.2</td>
</tr>
</tbody>
</table>
Guidelines

1. If the application is not to be fully compiled with the JML compiler — ajmlc can be used with either ajc or abc weaver, otherwise is better to use only abc weaver

2. If the user needs to take maximum of code optimization — ajmlc always combined with abc weaver

These guidelines are helpful when Java ME applications are considered
Conclusion

- Benefits to use AOP to instrument JML
  - suitability
  - flexibility
  - evidence to be less complex
- Answers to research questions
- New assertion semantics capability
- ajmlc optimizations
- Study + guidelines to use ajmlc
Future Work

- To extend our compiler to treat other JML features (e.g., model programs)
- To support assertion checking in a concurrent environment
- More case studies (including performance comparison)
JML and Aspects: The Benefits of Instrumenting JML Features with AspectJ

Henrique Rebêlo
Sérgio Soares
Ricardo Lima
Paulo Borba
Márcio Cornélio
1. If the application which you want to compile using the JML compiler refers to JML features not available in ajmlc, you can use only the classical JML compiler (jmlc), which does not generate code to run on Java ME platform.