Java Modeling Language

- **Formal specification language** for Java
  - behavioral specification of Java modules
- Adopts design by contract and Hoare-style with **assertions**
  - pre-, postconditions and invariants

- Main goal → **Improve functional software correctness** of Java programs
Java Modeling Language

- **Assertions** are added as comments in .java files
  - between /*@ ... @*/ or //@

```java
public class foo{
    //= invariant S;
    //= requires P;
    //= ensures Q;
    //= signals (FooException) R;
    //=
    public void foo() throw FooException  {...}
}
```
Meaning of Postconditions

- **normal (return)**
  - ensures $Q$

- **exceptional (throw)**
  - signals (...) $R$
JML compiler: compilation passes

Front-end

Annotated Java Source File

Parsing/typechecking

JML Type Checker

AST generation

Typechecked AST

Assertion methods (AM) generation

RAC code generation

Runtime checks

Instrumented bytecode

Instrumented bytecode generation

MultiJava compiler

TJSF compilation

Temporary Java Source File

Assertion methods (AM) printing in a TJSF

RAC code printing
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

- Verify Java ME/SE programs with JML
  - AspectJ - AOP extension to Java

We use the AspectJ to
- translate JML contracts into aspects
- generate bytecodes compliant with Java ME/SE
- verify if bytecode respects the JML contracts during runtime
Aspect Oriented Programming with AspectJ

- AspectJ
  - Crosscutting concern modularization
    - persistence
    - distribution
    - ...

“[…] there are many other concerns that, in specific system, have crosscutting structure. Aspects can be used to maintain internal consistency among several methods of a class. They are well suited to enforcing a Design by Contract style programming.”

Gregor Kiczales

http://www.theserverside.com/talks/videos/GregorKiczalesText/interview.tss
Aspect Oriented Programming with AspectJ

- **Dynamic crosscutting**
  - Define additional code that should be executed
    - before
    - after
    - around

- **Static crosscutting**
  - ...
  - Add new members to types
  - ...
Contributions

- A novel JML compiler compliant with
  - Java ME/SE applications

- The use of AspectJ to implement JML contracts
  - Mapping JML contracts into AspectJ aspects
  - Checking the contracts during runtime
Outline

- Implementation Strategy
- Subset of JML: pre, post, invariant
- Mapping Contracts into Aspects
- Comparative Study
- Conclusion
- Future Work
Implementation Strategy

Annotated Java Source File

- Parsing/typechecking
  - JML Type Checker
    - AST generation
  - Typechecked AST

- Aspect Assertion methods (AAM) generation

Instrumented bytecode

- Instrumented bytecode generation
  - AspectJ compiler
    - TASF compilation

- Aspect Assertion methods (AAM) printing in a TJSF

Aspect Code Generation

- Aspect RAC code generation
- Aspect RAC code printing

Runtime checks

Front-end

AspectJ Compiler
Subset of JML: pre, post, invariant

- **Preconditions** (*requires* keyword)
  Properties that must hold before method calls.

- **Normal Postconditions** (*ensures* keyword)
  Properties that must hold after method calls.

- **Invariants** (*invariant* keyword)
  Properties that must be maintained by all methods.
Subset of JML: pre, post, invariant

- Inheritance of JML specifications
  - Leavens’ definition

If \( T' \triangleright (pre', post') \), \( T \triangleright (pre, post) \), \( S \leq T' \), \( S \leq T \), then

\[
(pre', post') \sqcup^S (pre, post) = (p, q)
\]

where \( p = pre' \parallel \ parallel \ pre \)

and \( q = (\text{old}(pre') \implies post') \&\& (\text{old}(pre) \implies post) \)

and \( S \triangleright (p, q) \).

*preconditions* are combined by *disjunction*

*normal postconditions* are combined by *conjunction*

*invariants* are combined by *conjunction*
Mapping contracts into aspects

- Assume the following generic JML specification

```java
public class S extends T{
  //@ invariant inv;
  //@ also
  //@ requires pre;
  //@ ensures post;
  //@+
  public void m() {...}
}
```

Class S inherits JML specifications from class T
Mapping contracts into aspects

- AspectJ mapping for precondition

**Inter Type declaration:** `checkPre$m()`

```java
public boolean S.checkPre$m()
{
    return pre || super.checkPre$m();
}
```

**AspectJ advice:** `before`

```java
before(S current):
    execution(void S.m()) && within(S) && this(curren){
        if(!current.checkPre$m()){
            throw new JMLInternalPreconditionError();
        }
    }
```
Mapping contracts into aspects

- AspectJ mapping for normal postcondition

Inter Type declaration: `checkPost$m$C()`

```java
public boolean S.checkPost$m$S(){
    return !pre || post;
}
```

AspectJ advice: `around`

```java
void around(S current) :
    execution(void S.m()) && this(current){
        // saving all old values (pre-state)
        proceed();
        // (post-state)
        if(!current.checkPost$m$S()){ 
            throw new JMLInternalNormalPostconditionError();
        }
    }
```
Mapping contracts into aspects

- **AspectJ** mapping for invariant

  **Inter Type declaration:** `checkInv$Instance ()`

  ```java
  public boolean S.checkInv$Instance (){
    return inv && super.checkInv$Instance ();
  }
  ```

  **AspectJ advice:** before, after returning after throwing

  ```java
  execution(!static * *(..)) && within(S)
  ```
Comparative study

- An open source Java ME floating point calculator in three different ways:
  - Using our approach with AOP (our compiler) - CalcAspSol
  - Using the original approach (jmlc compiler) - CalcJmlSol
  - Using a pure one (with no bytecode instrumentation) - CalcPureSol

(→) The open source calculator is available at https://meapplicationdevelopers.dev.java.net/demo box.html
Comparative study

- **Metrics**
  - MiDlet class size
  - Bytecode size (JAR)
  - Library API size

- **Annotations used:** pre, postconditions, and invariant

- **Enforced properties**
  - The calculator yields only positive results
  - The calculator prevents division by zero

The calculator was executed in a real mobile phone!
Comparative study

- Our analysis were extracted from the following results:

<table>
<thead>
<tr>
<th></th>
<th>Midlet class size (KB)</th>
<th>JAR size (KB)</th>
<th>Lib JAR size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CalcAspSol</td>
<td>21.1</td>
<td>11.8</td>
<td>4.6</td>
</tr>
<tr>
<td>CalcJmlSol</td>
<td>39.5</td>
<td>278.0</td>
<td>261.0</td>
</tr>
<tr>
<td>CalcPureSol</td>
<td>4.9</td>
<td>2.7</td>
<td>—</td>
</tr>
</tbody>
</table>

Such results provide indication that our compiler generates a bytecode that requires less memory space than one generated by the JML compiler (jmlc).
Conclusion

- A novel JML compiler
  - Java ME/SE applications
- The use of AspectJ to implement contracts written in JML
- A comparative study
  - Our compiler produces smaller code than the jmlc
- Limitation
  - Subset of JML constructs
Future Work

- To extend our compiler to treat other JML constructs
- Optimization techniques for AspectJ advices
- Case studies
- Formalization of the JML/AspectJ mapping
Implementing Java Modeling Language Contracts with AspectJ

Henrique Rebêlo
Ricardo Lima
Márcio Cornélio
Sérgio Soares
Leopoldo Ferreira
(hemr, ricardo, marcio, sergio, lpf@dsc.upe.br)
Example: pre and post
Example: pre and post

**AspectJ for T**

```
public boolean T.checkPrem(){
    return pre;
}
```

**AspectJ for S**

```
public boolean S.checkPrem(){
    return pre' || super.checkPrem();
}
```

```java
S s = new S();
s.m();
```
Example: pre and post

**AspectJ for T**

```java
public boolean T.checkPost() {
    return !pre || post;
}

void around(T current) {
    execution(void T.m()) && this(current){
        ...// saving all old values (pre-state)
        proceed();
        // (post-state)
        if (!current.checkPost()){
            throw new JMLInternalNormalPostconditionError();
        }
    }
}
```

**AspectJ for S**

```java
public boolean S.checkPost() {
    return !pre' || post';
}

void around(S current) {
    execution(void S.m()) && this(current){
        ...// saving all old values (pre-state)
        proceed();
        // (post-state)
        if (!current.checkPost()){
            throw new JMLInternalNormalPostconditionError();
        }
    }
}
```

```java
S s = new S();
s.m();
```
Complete invariant mapping

```java
public boolean S.checkInv$Instance(){
    return inv && super.checkInv$Instance();
}
```

```java
after(S current) returning(Object o) :
    execution(!static * *(..)) && within(S) &&
    this(current){
        if (!current.checkInv$Instance()){#
            throw new JMLInvariantError("<@post>");
        }
    }
}
```

```java
before(S current) :
    execution(!static * *(..)) && within(S) &&
    this(current){
        if (!current.checkInv$Instance()){#
            throw new JMLInvariantError("<@post>");
        }
    }
}
```

```java
after(S current) throwing(Throwables throwable) :
    execution(!static * *(..)) && within(S) &&
    this(current){
        if (throwable instanceof JMLViolationError){#
            throw (JMLViolationError) throwable;
        }
        if (throwable instanceof otherException){#
            if (!current.checkInv$Instance()){#
                throw new JMLInvariantError("<@post>");
            }
            else{
                throw (otherException) throwable;
            }
        }
    }
}
```
Example: old construct

```java
public int T.old_y;

public boolean T.checkPost$m$T()
{
    return !true || y == (old_y + 10);
}

void around(T current)
{
    execution(void T.m()) && this(current)
    // saving all old values (pre-state)
    old_y = y;
    proceed();
    // (post-state)
    if (!current.checkPost$m$T())
    {
        throw new JMLInternalNormalPostconditionError();
    }
}

//@ ensures y == \old(y + 10);
```