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

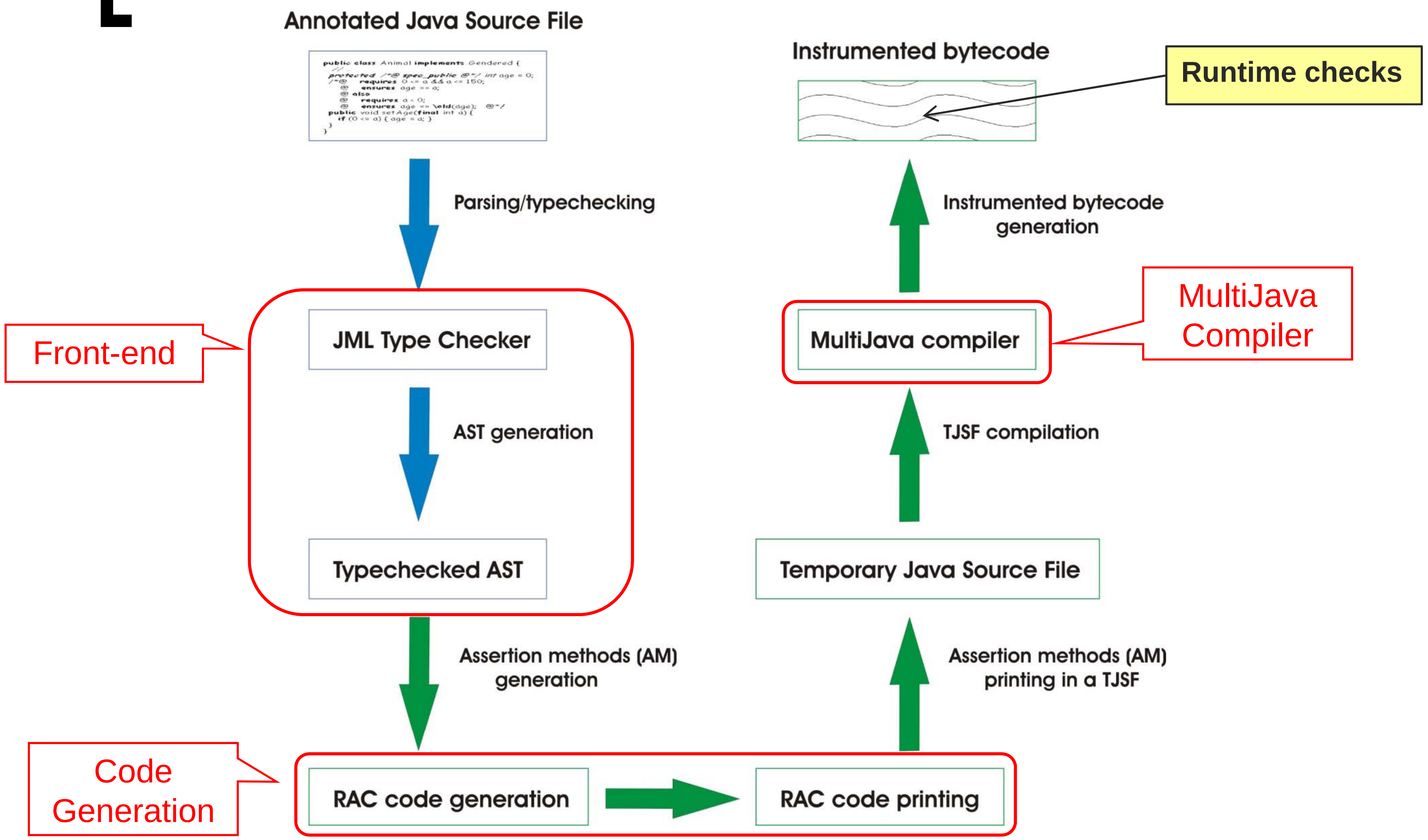
Henrique Rebêlo
Sérgio Soares
Ricardo Lima
Paulo Borba
Márcio Cornélio

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



```
public class Animal implements Gendered {
  //...
  protected /*@ spec_public @*/ int age = 0;
  /*@   requires 0 <= a && a <= 150;
    @   ensures age == a;
    @ also
    @   requires a < 0;
    @   ensures age == \old(age);   @*/
  public void setAge(final int a) {
    if (0 <= a) { age = a; }
  }
}
```

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)

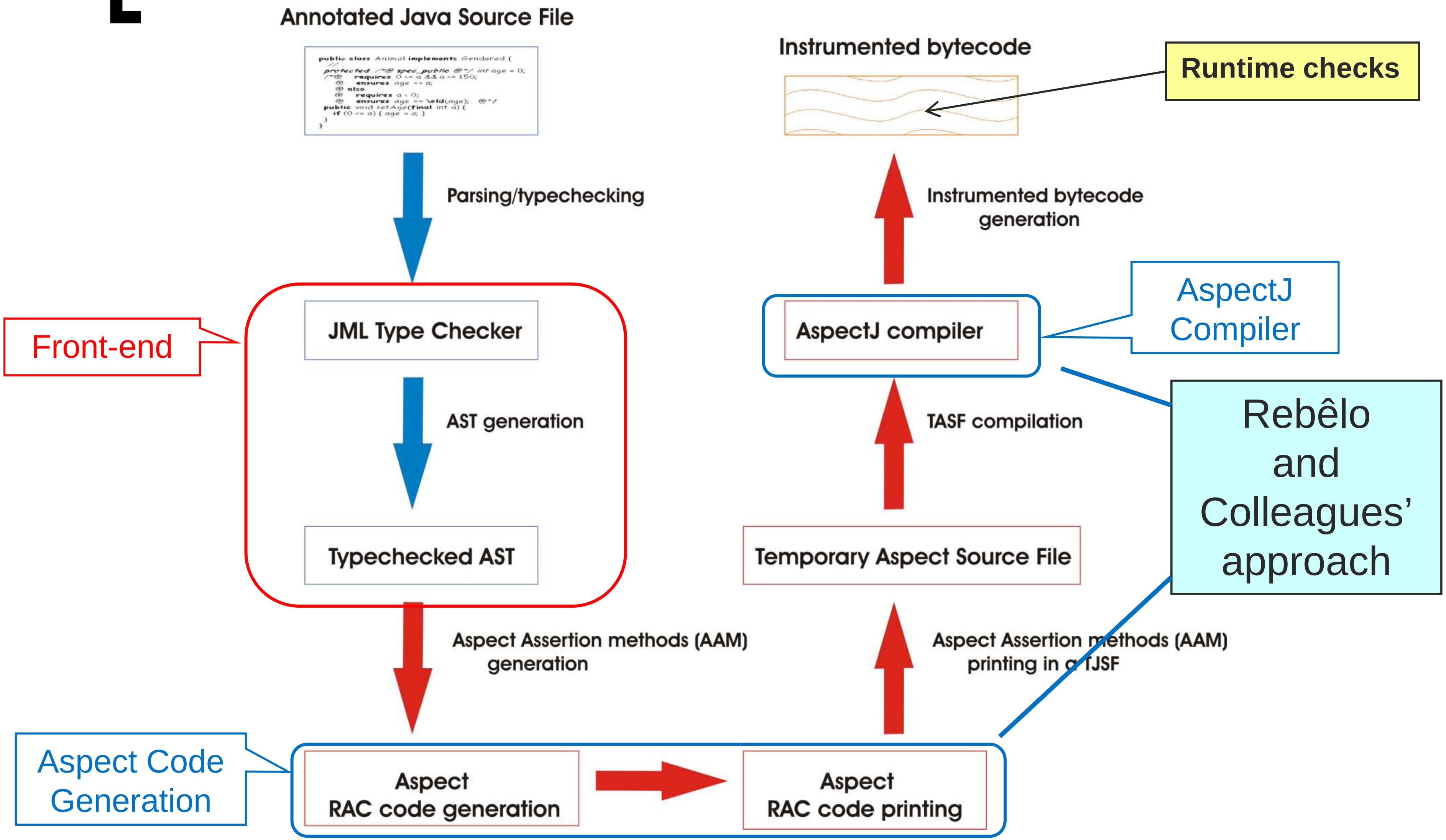
Aspect-oriented languages are quite popular...

due to the promise of

modularizing

crosscutting concerns

ajmlc: implementation Strategy



Annotated Java Source File

```
public class Animal implements Gendered {
  //...
  protected /*@ spec_public @*/ int age = 0;
  /*@   requires 0 <= a && a <= 150;
    @   ensures age == a;
    @ also
    @   requires a < 0;
    @   ensures age == \old(age);  @*/
  public void setAge(final int a) {
    if (0 <= a) { age = a; }
  }
}
```

generated code: jmlc VS ajmlc

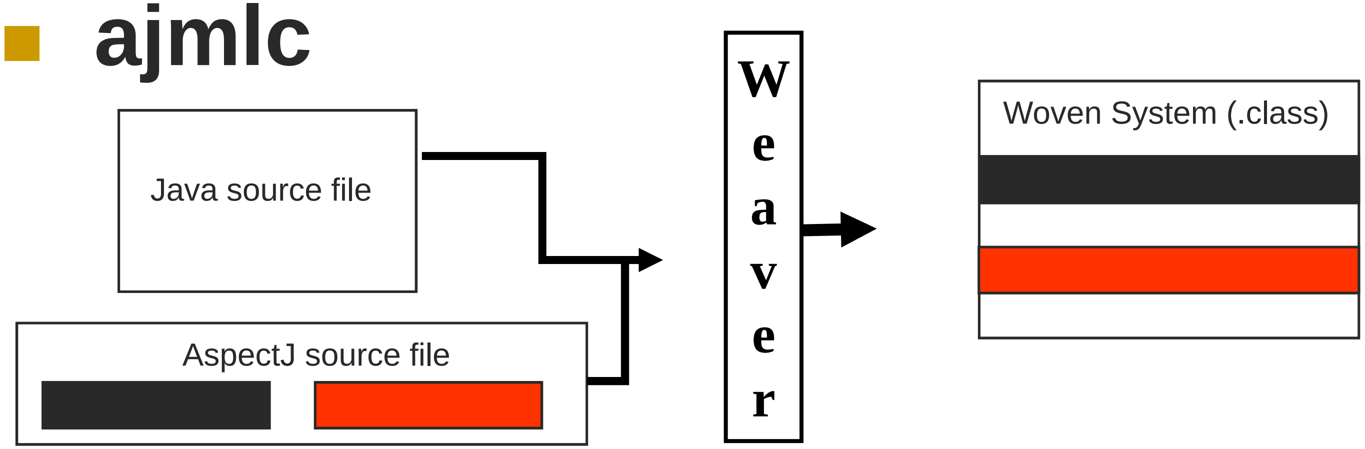
- **jmlc**

Java source file

MultiJava

System (.class)

- **ajmlc**



Java source file

AspectJ source file

Weaver

Woven System (.class)

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 ajmlc
- …

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 crosscuting — 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() {...}
}
```

=

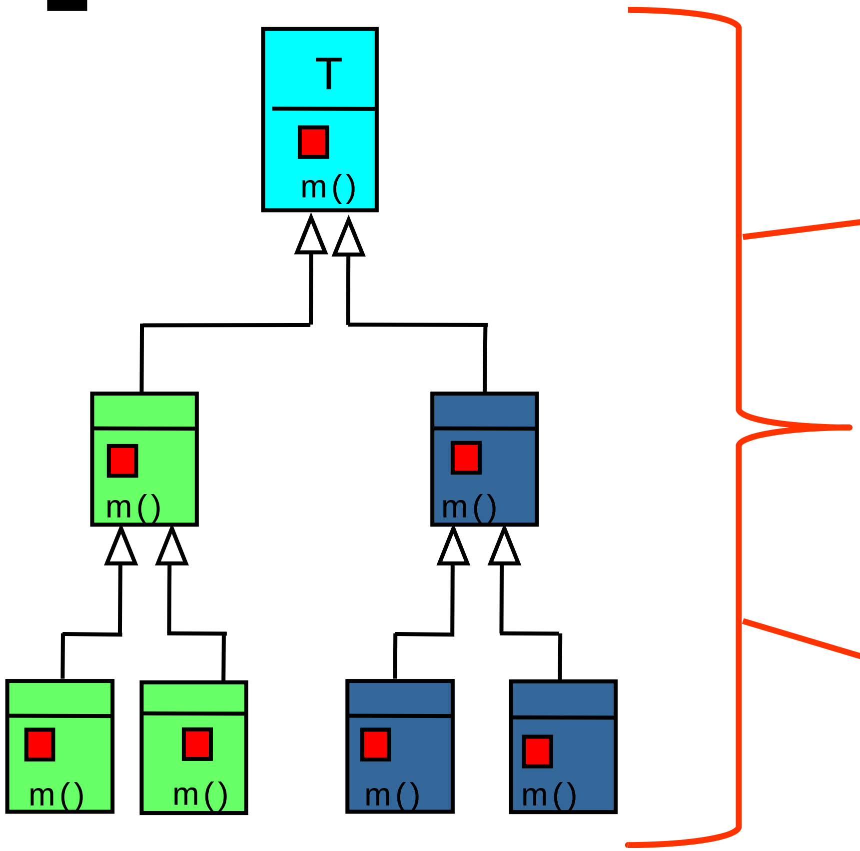
```
aspect T {
  before(T object) :
      exec(!static * T.*(..)) &&
      within(T+) &&
      this(object){
        if( !(object.i==10)){
           throw new Error(“”);
        }
      }
  after(T object) :
      exec(!static * T.*(..)) &&
      within(T+) &&
      this(object){
        if( !(object.i==10)){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



```
class T {
  int i = 10;

  //@ post i >= 10;
  void m() {...}
}
```

```
aspect T {
  after(T object) :
      exec(void T.m() &&
      within(T+) &&
      this(object){
        if( !(object.i>=10)){
           throw new Error("");
        }
      }
}
```

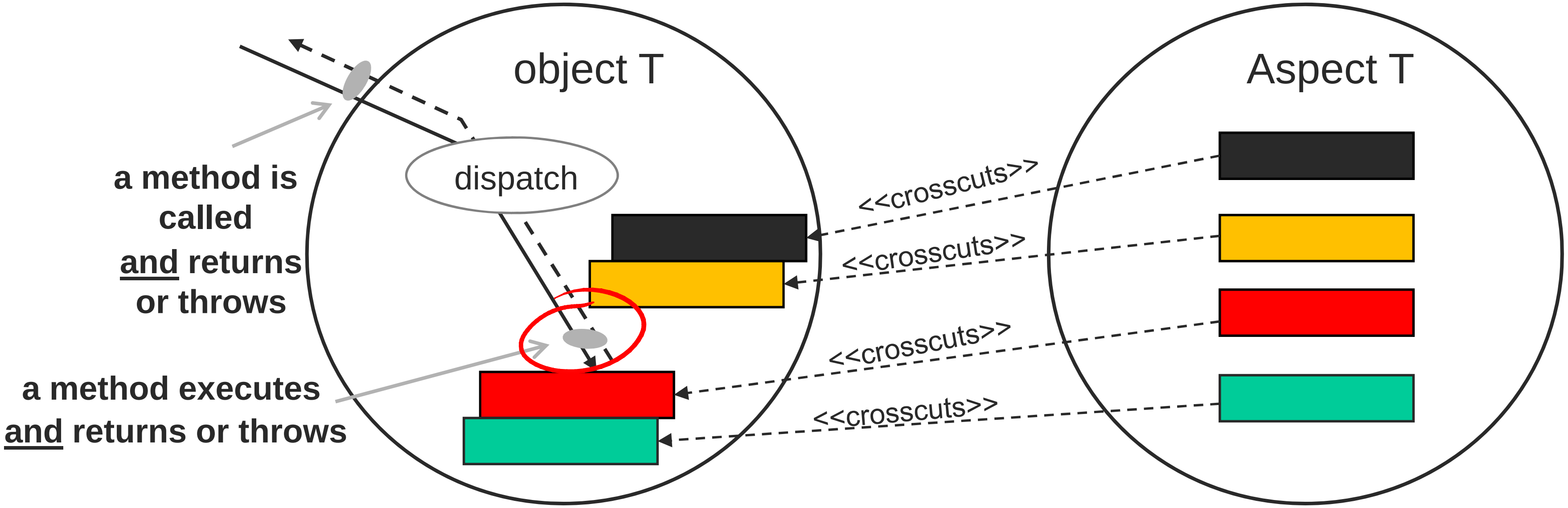
Both JML spec and aspect quantify the same points on type T and its subtypes

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

Example

```
public class T {
  private int x,y;

  //@ requires b && (x < y);
  public void m(boolean b) {
      ...
  }
}
```

Add before-execution with new assertion semantics capability

```
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(“”);}
     }
}
```

ajmlc and Java ME applications

To verify Java ME applications, our compiler only generates aspects that avoid AspectJ constructs that are not supported by Java ME

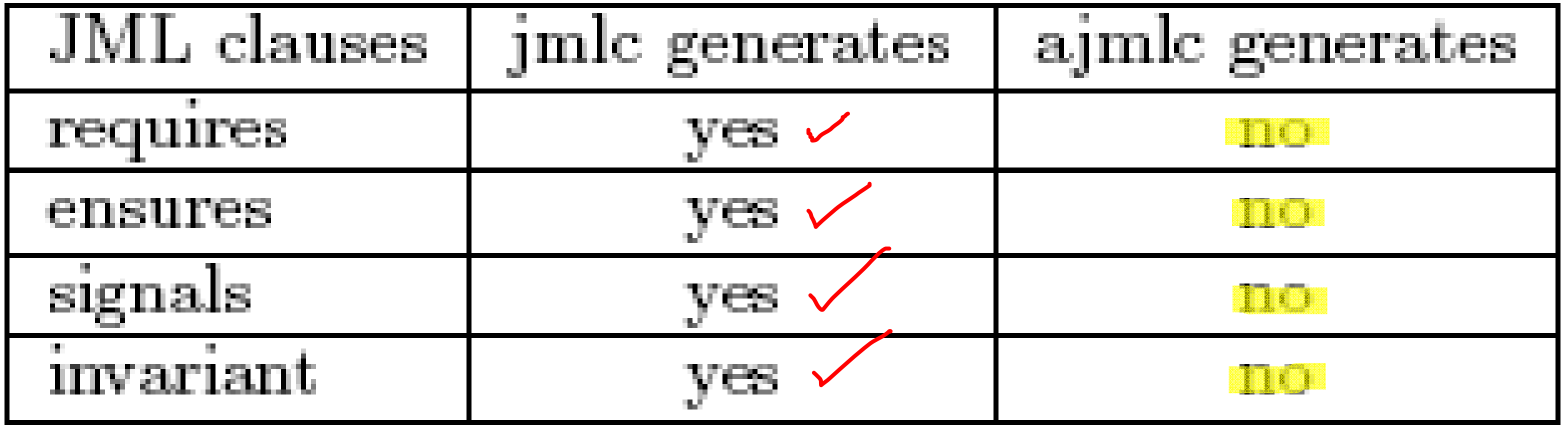
- Avoids AspectJ constructs such as…
  - cflow pointcut
  - cflow below pointcut
  - thisJoinPoint, …

ajmlc optimizations

- Compiling empty classes
  - ajmlc generates no code
  - jmlc
    - generates 11.0 KB (source code instrumentation)
    - generates 5.93 KB (bytecode instrumentation)

```
public class T {
}
```

Jmlc VS ajmlc



JML clauses	jmlc generates	ajmlc generates
requires	yes ✓	no
ensures	yes ✓	no
signals	yes ✓	no
invariant	yes ✓	no

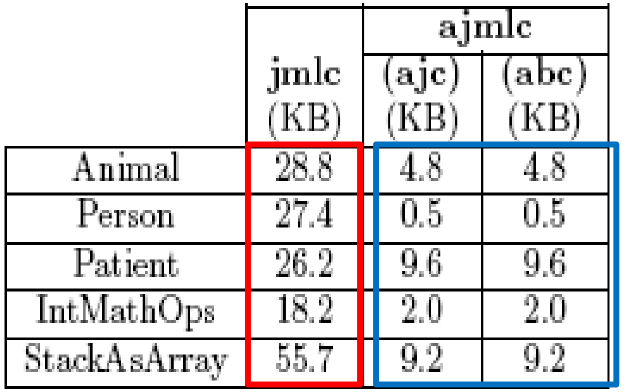
Study

- 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

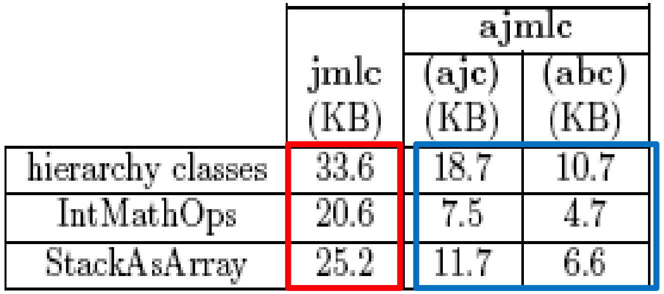
- Code size
    - instrumented source code size
    - instrumented bytecode size
    - Jar size (bytecode size + JML lib)

Results



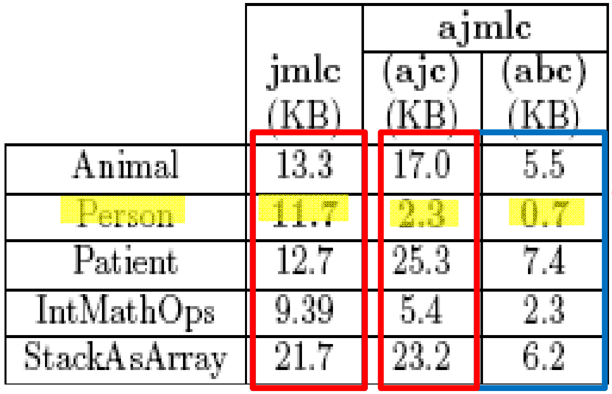
	jmlc (KB)	ajmlc (ajc) (KB)	ajmlc (abc) (KB)
Animal	28.8	4.8	4.8
Person	27.4	0.5	0.5
Patient	26.2	9.6	9.6
IntMathOps	18.2	2.0	2.0
StackAsArray	55.7	9.2	9.2

Source code instrumentation



	jmlc (KB)	ajmlc (ajc) (KB)	ajmlc (abc) (KB)
hierarchy classes	33.6	18.7	10.7
IntMathOps	20.6	7.5	4.7
StackAsArray	25.2	11.7	6.6

Jar size



	jmlc (KB)	ajmlc (ajc) (KB)	ajmlc (abc) (KB)
Animal	13.3	17.0	5.5
Person	11.7	2.3	0.7
Patient	12.7	25.3	7.4
IntMathOps	9.39	5.4	2.3
StackAsArray	21.7	23.2	6.2

Bytecode instrumentation

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

UPE
UNIVERSIDADE
DE PERNAMBUCO

CAPES

FINEP

Centro de Informática
U·F·P·E

Guidelines — no silver bullets

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