Java Program Verification Challenges

Bart Jacobs

bart@cs.kun.nl **http://www.cs.kun.nl/~bart**.

Department of Computer Science
University of Nijmegen, The Netherlands

Joint work with

Joe Kiniry, Martijn Warnier

Outline

I. Motivation

Standard examples

Traditional examples in (sequential) program verification:

- due to founding fathers: Dijkstra, Hoare, Gries, $\ldots$, used since 70s
- of limited size
- in limited programming language with only a few commands
- with limited assertion language
- far from current, real-life programs

Aims

- Lift the level of verification challenges, by using
- a realistic programming language (Java),
- with an expressive assertion language (JML)
- for programs of non-trivial size
  (Currently: 100s of lines of code)

In this talk

- Small fragments from realistic Java + JML examples,
- with emphasis not on verification, but on ***specification***
  (all examples have been verified by the LOOP tool)
- intro to JML by examples.

> Please don't suggest reductions to your favourite "simpler" language.

It shows that you have missed the point!

II. JML

JML: Java Modeling Language

In ***JML*** [Leavens et al.] one may add specifications as special comments in Java code, for:

- Class invariants and constraints
- Method specifications:

```
/*@ behavior
  @ requires   <precondition>
  @ assignable <items that may be modified>
  @ diverges   <precondition for non-termination>
  @ ensures    <postcond for normal termination>
  @ signals    <postcond for exceptional
  @             termination>
  @*/
void method() { ...  }
```

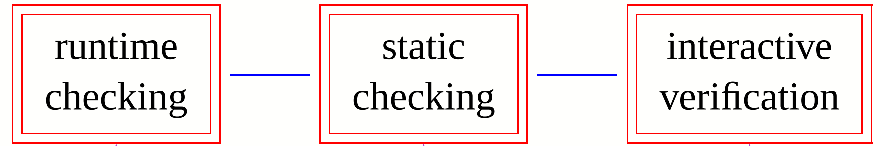
- Model variables (specification only)

JML: difference with javadoc

- Javadoc also uses special comments `/**...*/` to annotate programs, with special tags like **`@param`** and **`@return`**.
- A special compiler recognises these comments and produces html pages for uniform documentation.
- But: JML has a formal semantics and its assertions can be ***formally verified***.
- For this purpose, its language is richer (eg. has class invariants).

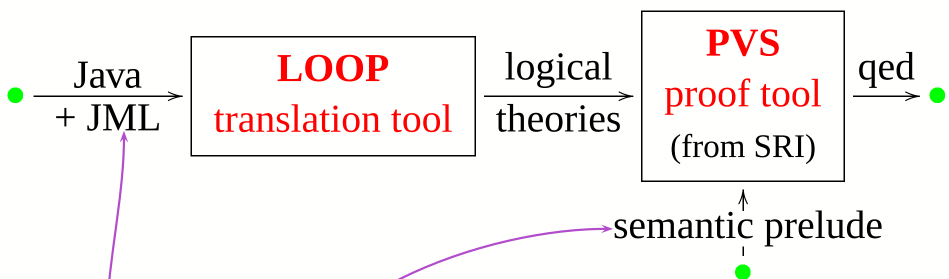
JML: Tool support

Especially appealing: range of options for JML:



- Iowa (Leavens et al.): JML parser, typechecker, inserter of run-time checks
- Compaq (Leino et al): extended static checking: automatic verification of ***simple*** assertions
- MIT (Ernst): ***Daikon*** invariant detection tool
- Nijmegen / INRIA / Gemplus / Hagen theorem proving, via LOOP / Krakatoa / Jack / Jive tool: interactive verification of arbitrary assertions.

LOOP project: overview



- JML annotations become PVS predicates, which should be proved for the (translated) Java code.
- The semantic prelude contains the semantics in PVS of Java language constructs like composition, if-then-else, while, try-catch-finally, . . .

III. Challenges

Aliasing and field access

```
class C {
  C a;
  int i;
  C() { a = null; i = 1; }
}
class D {
  /*@ normal_behavior
    @   requires true;
    @ assignable \nothing;
    @    ensures \result == 4;
    @*/
  int m() {
    C c = new C();
    c.a = c;
    c.i = 2;
    return c.i + c.a.i;
  }
}
```


Expressions with side-effects

```
class C {
  boolean b, result1, result2;
  /*@ normal_behavior
    @   requires true;
    @ assignable b;
    @    ensures b == !\old(b) && \result == b;
    @*/
  boolean f() { b = !b; return b; }
  /*@ normal_behavior
    @   requires true;
    @ assignable b, result1, result2;
    @    ensures result1 == false &&
    @            result2 == true;
    @*/
  void m() { b = true;
    result1 = f() || !f();
    result2 = !f() && f(); }
}
```


Breaking out of a loop

```
class C {
  int[] ia;
  /*@ normal_behavior
    @   requires ia != null;
    @ assignable ia[*];
    @    ensures \forall int i;
    @  0 <= i && i < ia.length ==>
    @   (\old(ia[i]) < 0 && \forall int j;
    @     0 <= j && j < i ==> \old(ia[j]) >= 0))
    @   ?  (ia[i] == -\old(ia[i]))
    @   :  (ia[i] == \old(ia[i]));
    @*/
  void negatefirst() {
    for(int i = 0; i < ia.length; i++) {
      if (ia[i] < 0) { ia[i] = -ia[i]; break; }
    }
  }
}
```


Catching exceptions

```
class C {
  int m;
  /*@ normal_behavior
    @   requires true;
    @ assignable m;
    @    ensures \result == ((d == 0)
    @              ?  \old(m) :  \old(m) / d)
    @            && m == \old(m) + 10;
    @*/
  int m(int d) {
    try { return m / d; }
    catch (Exception e) { return m / (d+1); }
    finally { m += 10; }
  }
}
```

Integral types

```
class C {
  /*@ behavior
    @   requires true;
    @ assignable \nothing
    @    ensures false
    @    signals (Exception e) false
    @   diverges true
    @*/
  void m() {
    for (byte b = Byte.MIN_VALUE;
              b <= Byte.MAX_VALUE; b++)
      ;
  }
}
```


Static initialisation

```
class C1 {   static boolean b1 = C2.d2;
     static boolean d1 = true; }
class C2 {   static boolean d2 = true;
     static boolean b2 = C1.d1; }
class C {
  static boolean r1, r2, r3, r4;
  /*@ normal_behavior
    @   requires !\is_initialized(C) &&
    @            !\is_initialized(C1) &&
    @            !\is_initialized(C2);
    @ assignable \everything;
    @    ensures r1 && !r2 && r3 && r4;
    @*/
  static void m() {
    r1 = C1.b1; r2 = C2.b2;
    r3 = C1.d1; r4 = C2.d2; }
}
```


Inheritance

```
class C {
  void m() throws Exception { m(); }
}
class D extends C {
  void m() throws Exception {
    throw new Exception(); }
  /*@ exceptional_behavior
    @   requires true;
    @ assignable \nothing
    @    signals (Exception) true;
    @*/
  void test() throws Exception { super.m(); }
}
```


Invariants I

```
class C {
  private /*@ spec_public @*/ int k, m;
  /*@ invariant
    @    k + m == 0;
    @*/
  /*@ normal_behavior
    @   requires true;
    @ assignable k,m;
    @    ensures k == \old(k) - 1 &&
    @            m == \old(m) + 1;
    @*/
  void decrementk() { k--; m++; }
  // more on next slide
}
```

Invariants II: callbacks

```
class C {
  // previous slide
  void decrementk() { k--; m++; }
  B b;
    /*@ normal_behavior
      @   requires b != null;
      @ assignable k,m;
      @    ensures true;
      @*/
  void incrementk() { k++; b.go(this); m--; }
}
class B {
  /*@ normal_behavior •••
    @*/
  void go(C arg) { arg.decrementk(); }
}
```

Should not be provable!

Overflow in specification

```
class C {
  /*@ normal_behavior
    @   requires x >= 0&& x <= 2147390966;
    @ assignable \nothing;
    @    ensures \result * \result <= x &&
    @            x < (\result+1) * (\result+1)
    @            && \result < 46340;
    @*/
  int m(int x) {
    int count = 0, sum = 1;
    while (sum <= x) {
      count++; sum += 2 * count + 1; }
    return count;
  }
}
```


IV. Conclusions

Conclusions

- JML is an expressive, easy-to-use specification language for Java,
- supported by a range of tools.
- Real challenge: express and prove "higher level" properties like confidentiality, non-repudiation, service levels, etc.

Paper available at: **`www.cs.kun.nl/~bart/`**