Early Detection of JML Specification Errors using ESC/Java2

Patrice Chalin

Dependable Software Research Group (DSRG)

Computer Science and Software Engineering Department

Concordia University

Montreal, Canada

Specification and Verification of Component-Based Systems (SAVCBS’06)

November 10-11, 2006 held in conjunction with FSE-14, Portland, Oregon, USA

Static Program Verifiers (SPV)

- Significant advances in technology.
- Integration with modern IDEs:
  - Spec#
  - JML (well, soon)

Static Program Verifiers (SPV)

- Two *kinds of error* detected by SPV:
  - Precondition violations (for methods or operators).
  - Correctness violations (of method bodies).
- Order.

SPV: Detection of Errors in …

- Can be routinely used to detect **errors** in **code** relative to **specifications**.

- Unfortunately, **no error detection in specs**.
  - … beyond conventional type checking.
  - (maybe because specifiers do not make mistakes?)

Failing to Detect Errors in Spec’s

Is a serious problem because such errors

- More difficult to detect.
- More costly to fix (… when identified later).

Motivating example …

[Motivating example]---------------------

-

Example: MyUtil Class

```
public class MyUtil {

  public static int minLen(int[] a1, int[] a2);

  public static int sumUpTo(int[] a, int n);
}
```

Example: MyUtil Class + Specs

```
public class MyUtil {
 //@ ensures \result ==
 //@   java.lang.Math.min(a1.length, a2.length);
 public static int minLen(int[] a1, int[] a2);

 //@ requires n <= a.length;
 //@ ensures \result ==
 //@   (\sum int i; 0 <= i && i < n; a[i]);
 public static int sumUpTo(int[] a, int n);
}
```

PairSum Class and Method

```
public class PairSum {
  public static int pairSum(int[] a, int[] b)
  {
  /* returns (a[0] + b[0]) + … + (a[n] + b[n]);
   *     where n is the min length of a and b.
  */


  }
```

PairSum code

```
public class PairSum {
  public static int pairSum(int[] a, int[] b)
  {
    int n = MyUtil.minLen(a, b);
    return MyUtil.sumUpTo(a, n) +
               MyUtil.sumUpTo(b, n);
   // by commutativity.
  }
```

Using PairSum

```
int[] a = readFromFile(...);
int[] b = readFromFile(...);
int sum = pairSum(a, b);    ...
```

- readFromFile is declared to return null on *read error*.
- JML tools reports no errors for this code, yet …

Simple test case: PairSum read error

- Call trace

  pairSum(null,null)

  MyUtil.sumUpTo(null,null) → NullPointerEx

PairSum called with null

```
public static int pairSum(int[] a, int[] b)
{
  int n = MyUtil.minLen(~~a~~ null, ~~b~~ null);
  return ...
}
```

ESC/Java

- Why did it fail to report a problem?
- Examine the code annotated with assertions …

PairSum code + assertions

```
public static int pairSum(int[] a, int[] b) {
  int n = MyUtil.minLen(a, b);
  //@ assume (* postcondition of minLen *)
  //@ assert (* precondition of sumUpTo *)
  return MyUtil.sumUpTo(a, n) +
             MyUtil.sumUpTo(b, n);
}
```

Example: MyUtil Class + Specs

```
public class MyUtil {
 //@ ensures \result ==
 //@   java.lang.Math.min(a1.length, a2.length);
 public static int minLen(int[] a1, int[] a2);

 //@ requires n <= a.length;
 //@ ensures \result ==
 //@   (\sum int i; 0 <= i && i < n; a[i]);
 public static int sumUpTo(int[] a, int n);
}
```

PairSum code + assertions

```
public static int pairSum(int[] a, int[] b) {
  int n = MyUtil.minLen(a, b);
  //@ assume n = min(a.length, b.length);
  //@ assert   n <= a.length && ...;
  return MyUtil.sumUpTo(a, n) +
            MyUtil.sumUpTo(b, n);
}
```

PairSum assertions

assume n = min(a.length,   b.length);

assert n <=   a.length && ...;

PairSum assertions, trace null …

assume n = min(null.length, null.length);
assert n <= null.length && ...;

PairSum, trace null & simplifying (1)

assume n = null.length;

assert n <= null.length && ...;

PairSum, trace null & simplifying (2)

```
assume n = null.length;
assert null.length <= null.length && ...;
```

PairSum, trace null & simplifying (3)

```
assume n = null.length;
assert true && true;
```

Cause: precond. violation in contract

```
public class MyUtil {
 //@ ensures \result ==
 //@   java.lang.Math.min(a1.length, a2.length);
 public static int minLen(int[] a1, int[] a2);

 //@ requires n <= a.length;
 //@ ensures \result ==
 //@   (\sum int i; 0 <= i && i < n; a[i]);
 public static int sumUpTo(int[] a, int n);
}
```

SPV Detection of Errors In Specs

- Two *kinds of* coding *error* detected by SPV:
  - Precondition violations (for methods or operators).
  - ~~Correctness~~ violations (of method bodies).

- New ESC/Java2 feature:
  - definedness checking
  - "Is-defined checks", or IDC.

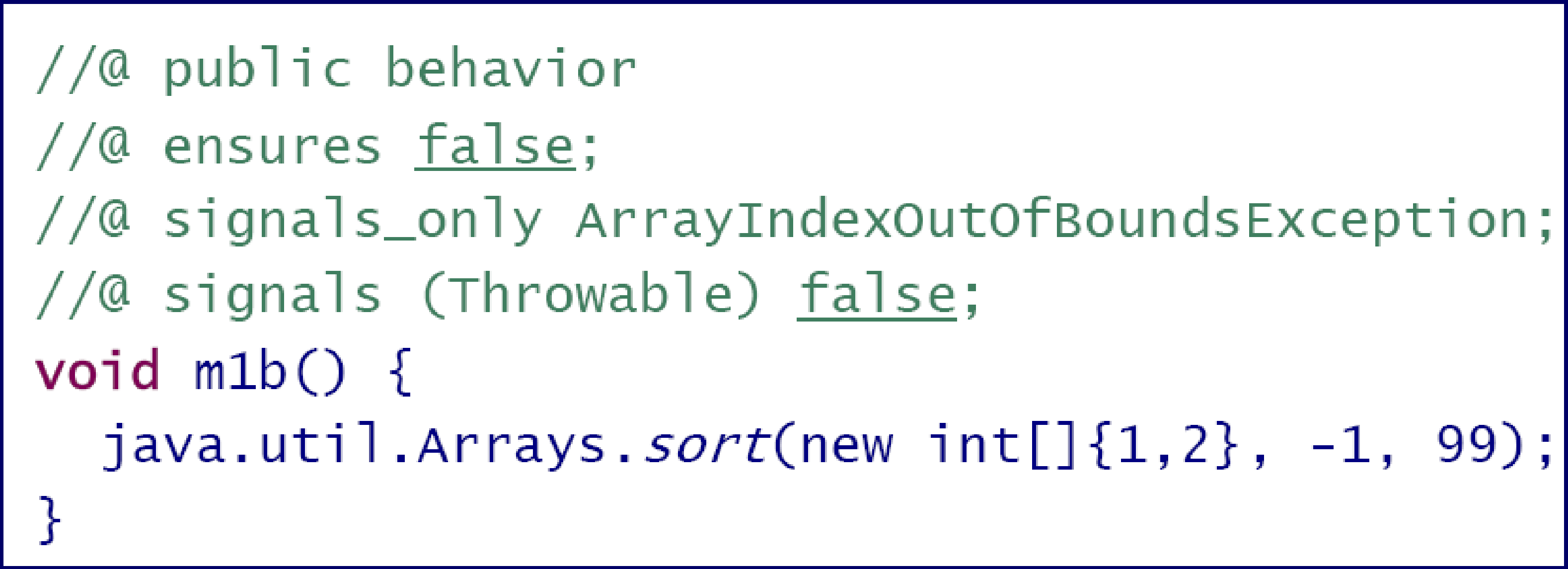
ESC/Java2 run with IDC

```
MyUtil: minLen(int[], int[]) ...
------------------------------------------
MyUtil.jml:3: Warning: Possible null deref…
  //@   java.lang.Math.min(a1.length,…);
                                ^
------------------------------------------
[0.062 s 12135232 bytes]  failed
```

[Example 2]-----------------------

Another Motivating Example

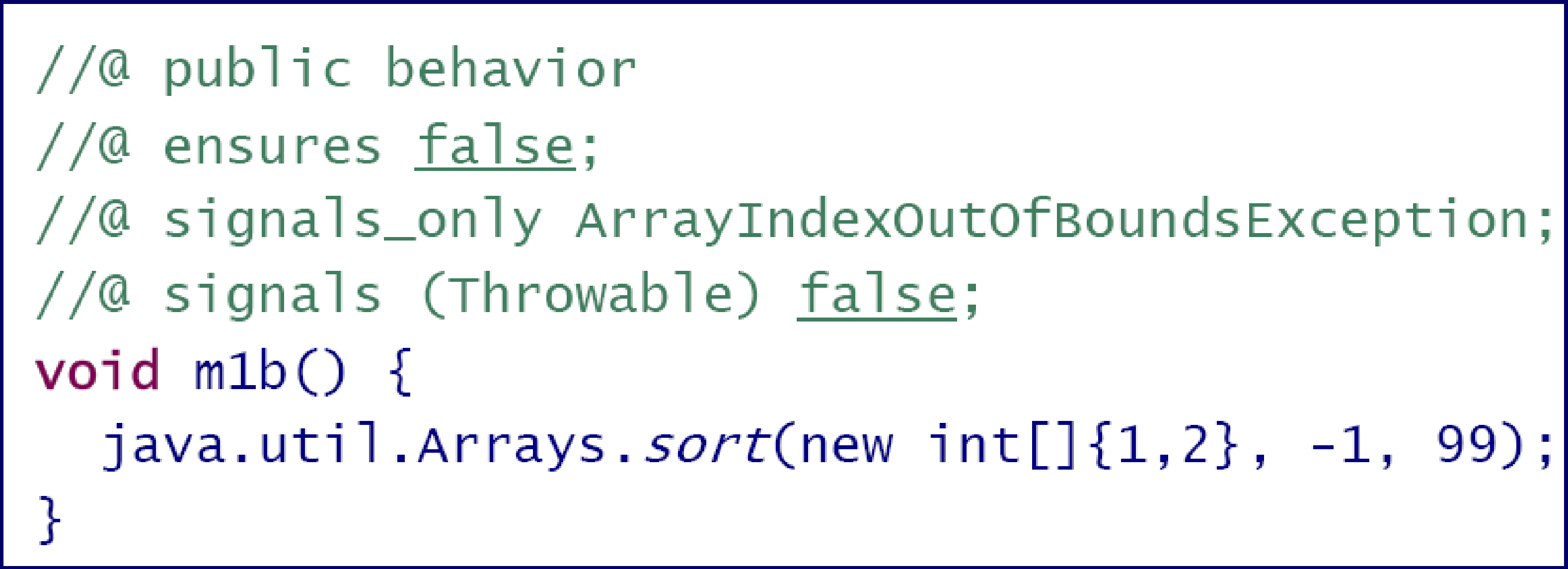
- Consider the following method + contract:



```
//@ public behavior
//@ ensures false;
//@ signals_only ArrayIndexOutOfBoundsException;
//@ signals (Throwable) false;
void m1b() {
  java.util.Arrays.sort(new int[]{1,2}, -1, 99);
}
```

Another Motivating Example

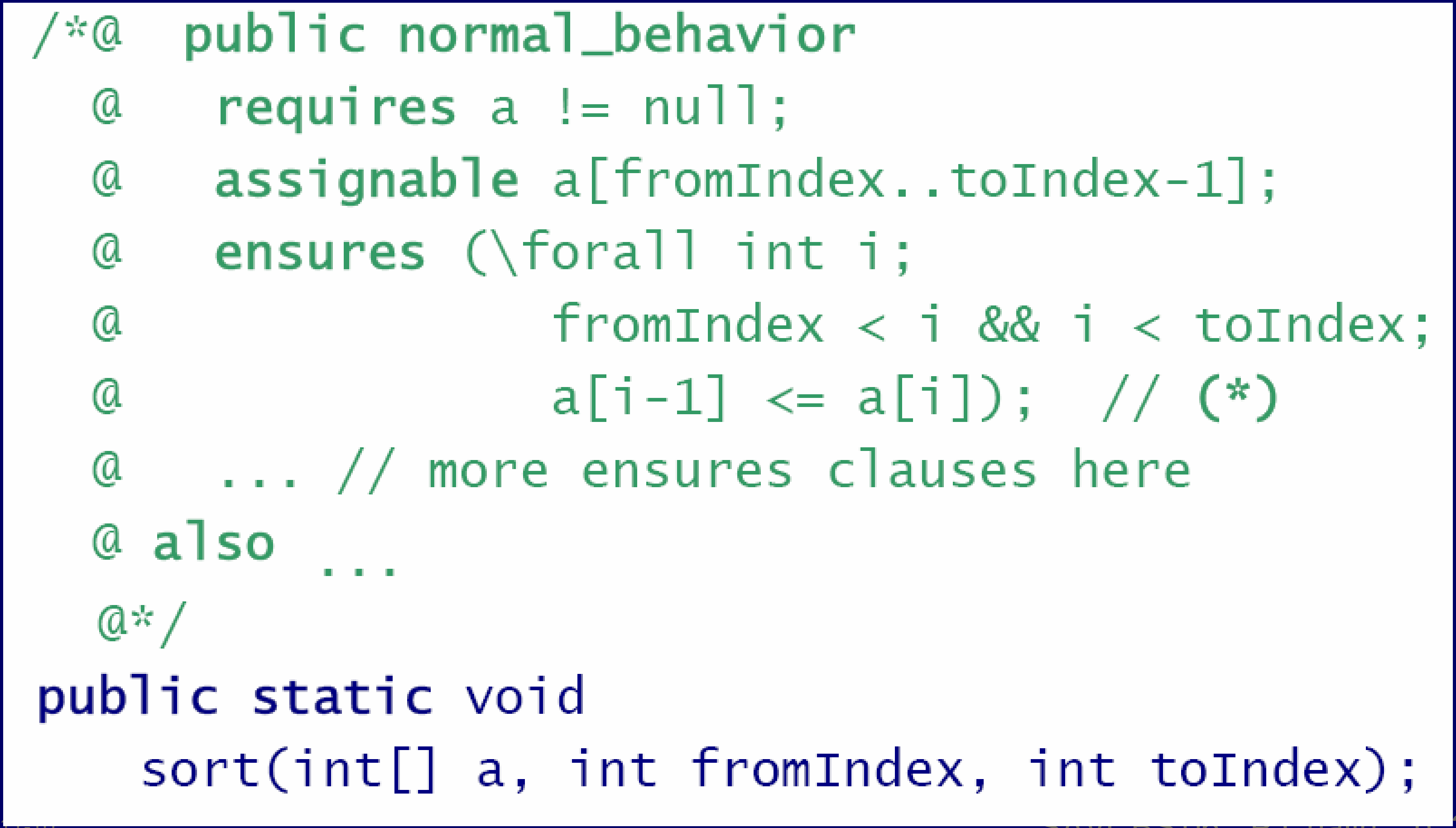
- Postconditions are false, hence contract is unimplementable and yet ... ESC/Java2 proves method “correct”.



```
//@ public behavior
//@ ensures false;
//@ signals_only ArrayIndexOutOfBoundsException;
//@ signals (Throwable) false;
void m1b() {
  java.util.Arrays.sort(new int[]{1,2}, -1, 99);
}
```

Inconsistency in Arrays.sort contract

- ESC/Java2 IDC points out …



```
/*@  public normal_behavior
  @   requires a != null;
  @   assignable a[fromIndex..toIndex-1];
  @   ensures (\forall int i;
  @                fromIndex < i && i < toIndex;
  @                a[i-1] <= a[i]);  // (*)
  @   ... // more ensures clauses here
  @ also ...
  @*/
public static void
   sort(int[] a, int fromIndex, int toIndex);
```

[ESC redesign]-------------------------

Supporting Definedness Checking

- JML's current assertion semantics
  - based on classical logic
  - Partial functions modeled by underspecified total functions

Newly proposed semantics

- Based on Strong Validity:
  an *assertion expression* is taken to hold iff it is
  - Defined and
  - True.

“Is-Defined” Operator

- $D(e)$

“Is-Defined” Operator

- In general (for *strict* functions):

  $D(f(e1, \ldots, en)) =$

  $D(e1) \wedge \ldots \wedge D(en) \wedge p(e1,\ldots,en)$

  e.g.

  - $D(e1 / e2) = D(e1) \wedge D(e2) \wedge e2 \neq 0$

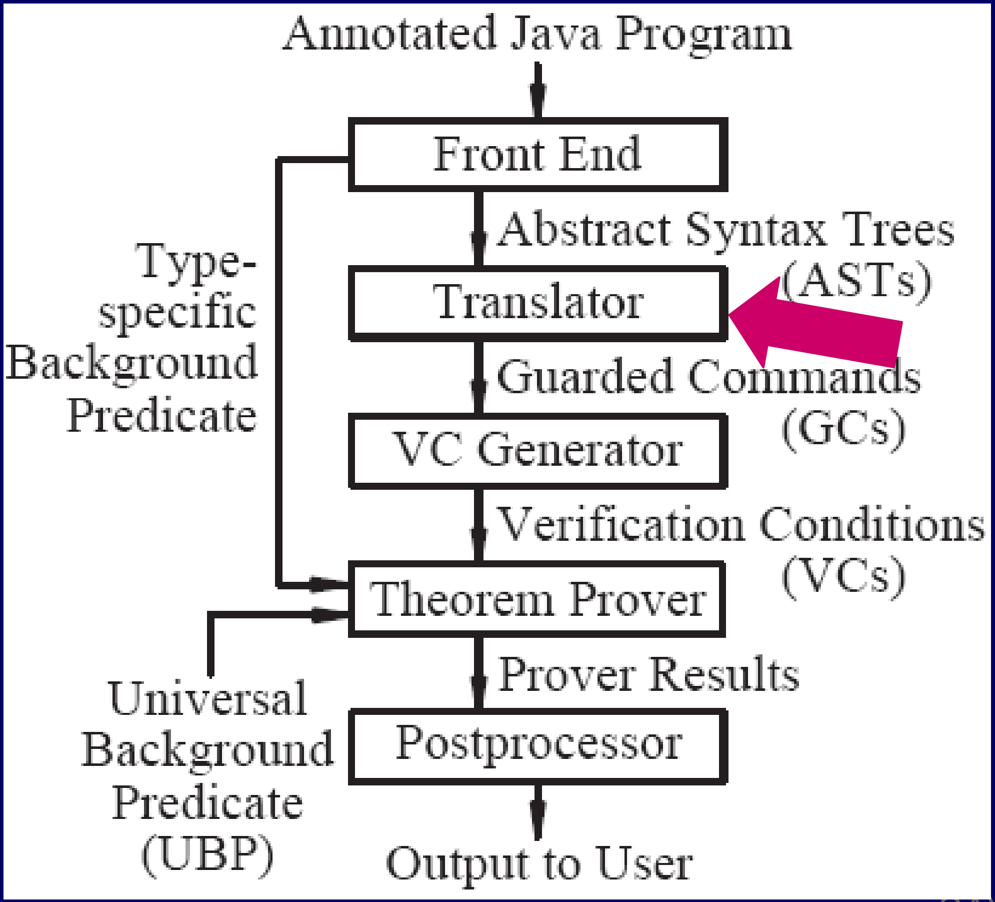
“Is-Defined” Operator

- *Non-strict* operators, e.g.

  $D(e1 \mathbin{\&\&} e2) \; = \; D(e1) \;\wedge\; (e1 \Rightarrow D(e2))$

ESC/Java2 Redesign

- Current / previous architecture



Guarded Command Language

C ::= *Id* **:**= *Expr*

  | **ASSUME** *Expr*

  | **ASSERT** *Expr*

  | *C* ; *C'*

  | *C*   *C'*

Supporting the New Semantics (IDC)

- Inline assertions

$$\llbracket \textbf{assert } R \rrbracket \;=\; \text{ASSERT } \llbracket D(R) \rrbracket \;; \\ \text{ASSERT } \llbracket R \rrbracket$$

IDC: Basic Method Contracts

Without IDC

$$
\begin{aligned}
[\![\{P\}B\{Q\}]\!] \;=\; & \text{ASSUME } [\![P]\!]\;; \\
& [\![B]\!]\;; \\
& \text{ASSERT } [\![Q]\!]
\end{aligned}
$$

With IDC

$$
\begin{aligned}
[\![\{P\}B\{Q\}]\!] \;=\; & \underline{\text{ASSERT } [\![D(P)]\!]\;;} \\
& \text{ASSUME } [\![P]\!]\;; \\
& [\![B]\!]\;; \\
& \underline{\text{ASSERT } [\![D(Q)]\!]\;;} \\
& \text{ASSERT } [\![Q]\!]
\end{aligned}
$$

Checking Methods Without Bodies

$$
\begin{aligned}
\llbracket \{P\}\_\{Q\} \rrbracket \;=\; & \underline{\text{ASSERT } \llbracket D(I(\text{this})) \rrbracket \;;} \\
& \text{ASSUME } \llbracket \forall\ \text{o}{:}C \,.\, I(\text{o}) \rrbracket \;; \\
& \underline{\text{ASSERT } \llbracket D(P) \rrbracket \;;} \\
& \text{ASSUME } \llbracket P \rrbracket \;; \\
& \llbracket \text{return } \_ \rrbracket \;[\,]\; \llbracket \text{throw } \ldots \rrbracket \;; \\
& \underline{\text{ASSERT } \llbracket D(Q) \rrbracket \;;} \\
& \underline{\text{ASSERT } \llbracket D(I(\text{this})) \rrbracket \;;}
\end{aligned}
$$

If life we this simple, we wouldn't need …

- Unfortunately, previous translation gives poor error reporting.
- ESC will report errors only for GCs:

  ASSERT Label(L, E).
- But this gives coarse grained report:

$$[\![\textbf{assert } R]\!] \; = \; \text{ASSERT Label(I, } [\![D(R)]\!] \text{);} \\ \text{ASSERT } [\![R]\!]$$

- We want ESC to pinpoint the errors in $D(R)$.

Better Diagnostics

- Need to expand
  ASSERT $[\![D(e)]\!]$

Expanded GC for strict functions

- Recall that

  $D(f(e1, \ldots, en)) = D(e1) \wedge \ldots \wedge D(en) \wedge p(e1,\ldots,en)$

- Expanded GC form, $\mathbf{E} [\![D(f(e1, \ldots, en))]\!]$, would be:

  $\mathbf{E} [\![D(e1)]\!]$;

  ... ;

  $\mathbf{E} [\![D(en)]\!]$;

  ASSERT Label(L, $[\![p(e1,\ldots,en)]\!]$)

Expanded GC for non-strict functions

- E.g. for conditional operator

  $D(e1 \mathbin{?} e2 : e3) \;=\; D(e1) \;\wedge\; (e1 \Rightarrow D(e2)) \;\wedge\; (\neg e1 \Rightarrow D(e3))$

- Expanded GC form would be

  $\mathbf{E}\ [\![D(e1)]\!]$;

  { ASSUME $[\![e1]\!]$;

   $\mathbf{E}\ [\![D(e2)]\!]$;

   []

   ASSUME $[\![\neg e1]\!]$;

   $\mathbf{E}\ [\![D(e3)]\!]$;

  }

[IDC results]-------------------------

ESC/Java2 Definedness Checking: Preliminary Results

- Tested on 90+KLOC code + specs.
- 50+ errors detected in java.* API specs.
- Negligible overhead (preliminary).
- Did not overwhelm Simplify.
  - Could prove no less than before.

- Looking forward to using CVC3 backend: offers native support for new semantics.

Questions?