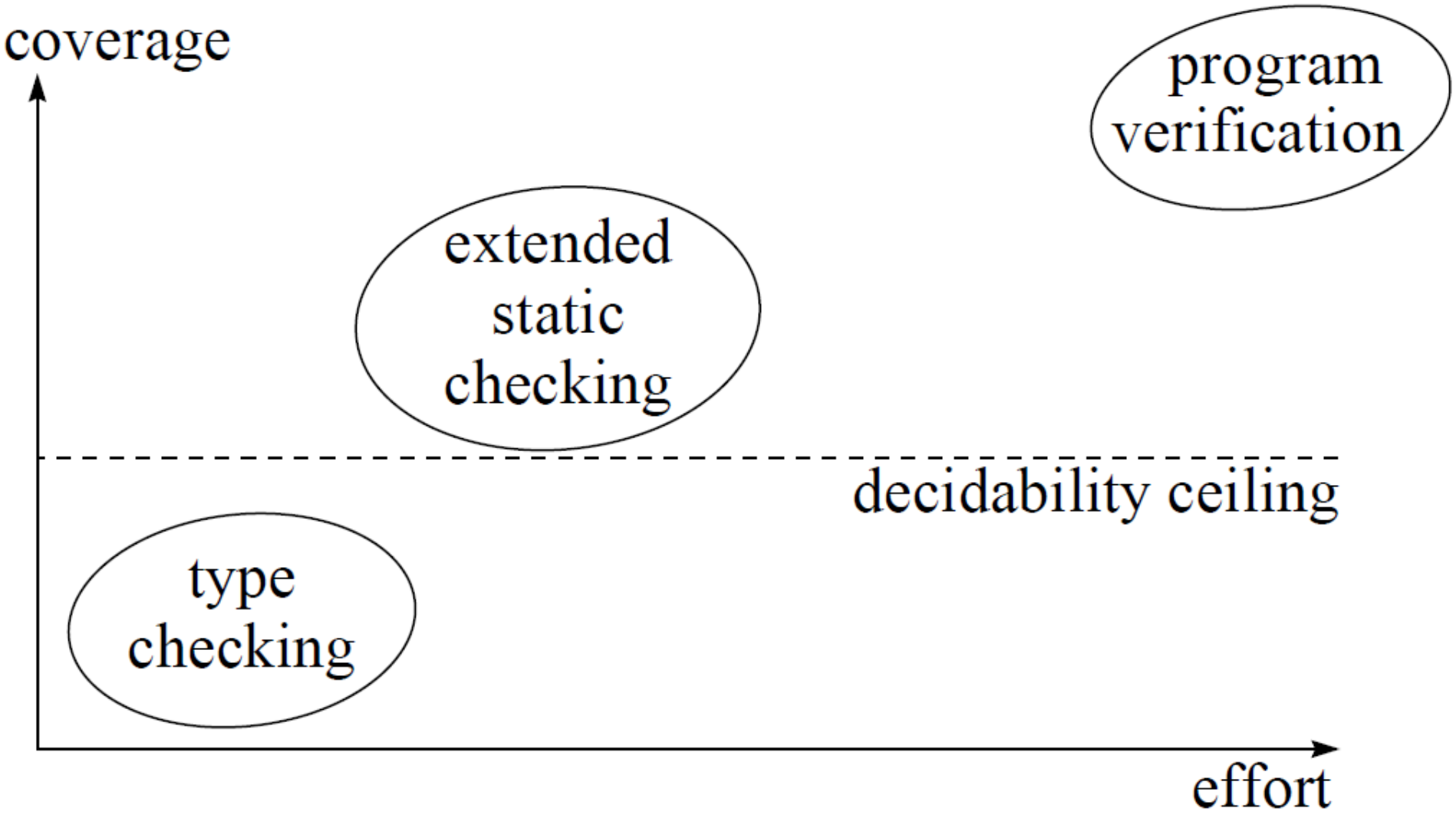
Extended Static Checking

based on:

C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, Extended Static Checking for Java, *Programming Language Design and Implementation (PLDI'02), pp.* 234-245, 2002

Roderick Bloem

IAIK – Graz University of Technology
firstname.lastname@iaik.tugraz.at


coverage
program verification
extended static checking
decidability ceiling
type checking
effort

Dynamic or Static?

Dynamic

- Run time

Eraser, locktree, purify, tests,…

Static

- Compile time

Extended Static Checking, Model Checking, …

JML

```
public class BankingExample {
  public static final int MAX_BALANCE = 1000;
  //@ public invariant balance >= 0 && balance <= MAX_BALANCE;
  private int balance;

  //@ assignable balance;
  //@ ensures balance == 0;
  public BankingExample() {
    this.balance = 0;
  }

  //@ requires 0 < amount && amount < MAX_BALANCE - balance;
  //@ assignable balance;
  //@ ensures balance == \old(balance) + amount;
  public void credit(final int amount) {
    this.balance += amount;
  }
 …
```

What is checked when?

Adapted from wikipedia article on JML,
http://en.wikipedia.org/wiki/Java_Modeling_Language

Checking JML

Dynamic

During runtime

OpenJML

What is the point in runtime verification?

Static

During compile time

ESC/Java (or OpenJML?)

Finds your bugs during compile time

Well, some bugs.

What is Static Checking?

Static Checking

- Checks during compile time, not run time
- Localizes proofs to methods, not entire programs
- Help from user: annotations per method

Examples of properties

- Null pointer dereferences
- Array bounds

Pro & Con

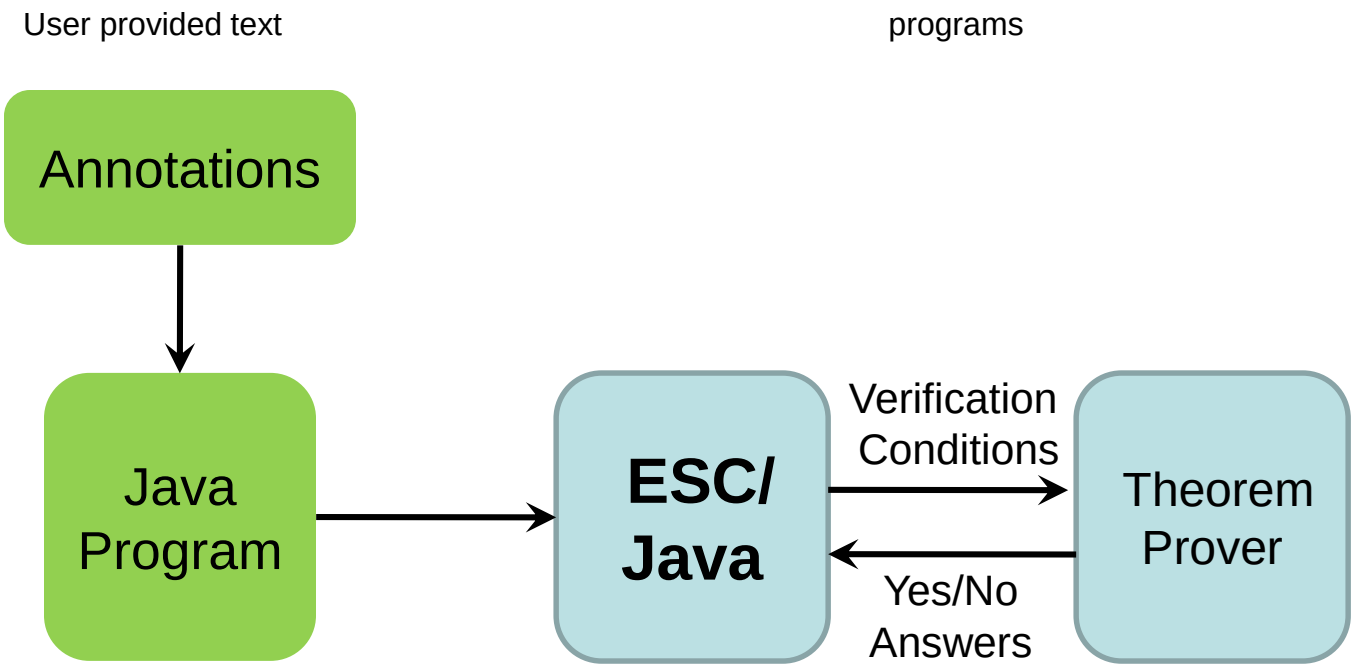
Pro

- Independent of program size
- Systematically excludes bugs
  - Caveat: may not be perfect
- Handles more types of bugs than type checking

Con

- Requires significant user effort (depending on program size)
- Can only check relatively simple properties

Architecture



Soundness & Completenes

- **Soundness**: no incorrect programs are deemed correct by the tool (all errors are found)
- **Completeness**: A correct program is deemed correct by the tool (no spurious errors)

ESC/Java is neither sound nor complete

Example

```
 1: class Bag{
 2:   int size;
 3:   int[] elts; //valid:elts[0..size-1]
 4:
 5:   Bag(int[ ] input){
 6:     size = input.length;
 7:     elts = new int[size];
 8:     System.arraycopy(input,0,elts,0,
    size);
 9:   }
10:
11:   int extractMin(){
12:     int min = Integer.MAX_VALUE;
13:     int minIndex = 0;
14:     for (int i= 0; i <= size ; i++){
15:       if (elts[i] < min){
16:         min = elts[i];
17:         minIndex = i;
18:       }
19:     }
20:     size--;
21:     elts[minIndex]= elts[size];
22:     return min;
23:   }
24: }
```

```
 1: class Bag{
 2:   int size;
 3:   int[] elts; //valid:elts[0..size-1]
 4:
 5:   Bag(int[ ] input){
 6:     size = input.length;
 7:     elts = new int[size];
 8:     System.arraycopy(input,0,elts,0,
   size);
 9:   }
10:
11:   int extractMin(){
12:     int min = Integer.MAX VALUE;
13:     int minIndex = 0;
14:     for (int i= 0; i <= size ; i++){
15:       if (elts[i] < min){
16:         min = elts[i];
17:         minIndex = i;
18:       }
19:     }
20:     size--;
21:     elts[minIndex]= elts[size];
22:     return min;
23:   }
24: }
```

Example

```
Bag.java:6: Warning: Possible null
dereference (Null)
size = input.length;
            ^
Bag.java:15: Warning: Possible null
dereference (Null)
if (elts[i] < min) {
        ^
Bag.java:21: Warning: Possible null
dereference (Null)
elts[minIndex] = elts[size];
    ^
Bag.java:15: Warning: Array index
possibly too large (...
if (elts[i] < min) {
        ^
Bag.java:21: Warning: Possible
negative array index (...
elts[minIndex] = elts[size];
    ^
```

```
 1: class Bag{
 2:   int size;
 3:   int[] elts; //valid:elts[0..size-1]
 4:
 5:   Bag(int[ ] input){
 6:     size = input.length;
 7:     elts = new int[size];
 8:     System.arraycopy(input,0,elts,0,
   size);
 9:   }
10:
11:   int extractMin(){
12:     int min = Integer.MAX VALUE;
13:     int minIndex = 0;
14:     for (int i= 0; i <= size ; i++){
15:       if (elts[i] < min){
16:         min = elts[i];
17:         minIndex = i;
18:       }
19:     }
20:     size--;
21:     elts[minIndex]= elts[size];
22:     return min;
23:   }
24: }
```

Example

```
Bag.java:6: Warning: Possible null
dereference (Null)
size = input.length;
            ^
```

```
 1: class Bag{
 2:   int size;
 3:   int[] elts; //valid:elts[0..size-1]
 4:
4a:   //@requires input != null
 5:   Bag(int[ ] input){
 6:     size = input.length;
 7:     elts = new int[size];
 8:     System.arraycopy(input,0,elts,0,
   size);
 9:   }
10:
11:   int extractMin(){
12:     int min = Integer.MAX VALUE;
13:     int minIndex = 0;
14:     for (int i= 0; i <= size ; i++){
15:       if (elts[i] < min){
16:         min = elts[i];
17:         minIndex = i;
18:       }
19:     }
20:     size--;
21:     elts[minIndex]= elts[size];
22:     return min;
23:   }
24: }
```

Example

```
Bag.java:6: Warning: Possible null
dereference (Null)
size = input.length;
               ^
```

```
 1: class Bag{
 2:   int size;
 3:   int[] elts; //valid:elts[0..size-1]
 4:
4a:   //@requires input != null
 5:   Bag(int[ ] input){
 6:     size = input.length;
 7:     elts = new int[size];
 8:     System.arraycopy(input,0,elts,0,
   size);
 9:   }
10:
11:   int extractMin(){
12:     int min = Integer.MAX VALUE;
13:     int minIndex = 0;
14:     for (int i= 0; i <= size ; i++){
15:       if (elts[i] < min){
16:         min = elts[i];
17:         minIndex = i;
18:       }
19:     }
20:     size--;
21:     elts[minIndex]= elts[size];
22:     return min;
23:   }
24: }
```

Example

```
Bag.java:15: Warning: Possible null
dereference (Null)
if (elts[i] < min) {
        ^
Bag.java:21: Warning: Possible null
dereference (Null)
elts[minIndex] = elts[size];
                 ^
```

```
 1: class Bag{
 2:   int size;
 3:   /*@non_null*/ int[] elts;
   //valid:elts[0..size-1]
 4:
4a:   //@requires input != null
 5:   Bag(int[ ] input){
 6:     size = input.length;
 7:     elts = new int[size];
 8:     System.arraycopy(input,0,elts,0,
   size);
 9:   }
10:
11:   int extractMin(){
12:     int min = Integer.MAX VALUE;
13:     int minIndex = 0;
14:     for (int i= 0; i <= size ; i++){
15:       if (elts[i] < min){
16:         min = elts[i];
17:         minIndex = i;
18:       }
19:     }
20:     size--;
21:     elts[minIndex]= elts[size];
22:     return min;
23:   }
24: }
```

Example

Bag.java:15: Warning: Possible null dereference (Null)

```
if (elts[i] < min) {
      ^
```

Bag.java:21: Warning: Possible null dereference (Null)

```
elements[minIndex] = elts[size];
                     ^
```

```
 1: class Bag{
 2:   int size;
 3:   /*@non_null*/ int[] elts;
   //valid:elts[0..size-1]
 4:
4a:   //@requires input != null
 5:   Bag(int[ ] input){
 6:     size = input.length;
 7:     elts = new int[size];
 8:     System.arraycopy(input,0,elts,0,
   size);
 9:   }
10:
11:   int extractMin(){
12:     int min = Integer.MAX VALUE;
13:     int minIndex = 0;
14:     for (int i= 0; i <= size ; i++){
15:       if (elts[i] < min){
16:         min = elts[i];
17:         minIndex = i;
18:       }
19:     }
20:     size--;
21:     elts[minIndex]= elts[size];
22:     return min;
23:   }
24: }
```

Example

```
Bag.java:15: Warning: Array index
possibly too large (...
if (elts[i] < min) {
        ^
Bag.java:21: Warning: Possible
negative array index (...
elts[minIndex] = elts[size];
                      ^
```

```
 1: class Bag{
 2:   int size;
2a:   //@ invariant 0<=size &&
    size<=elts.length
 3:   /*@non_null*/ int[] elts;
    //valid:elts[0..size-1]
 4:
4a:   //@requires input != null
 5:   Bag(int[ ] input){
 6:     size = input.length;
 7:     elts = new int[size];
 8:     System.arraycopy(input,0,elts,0,
    size);
 9:   }
10:
11:   int extractMin(){
12:     int min = Integer.MAX VALUE;
13:     int minIndex = 0;
14:     for (int i= 0; i <= size ; i++){
15:       if (elts[i] < min){
16:         min = elts[i];
17:         minIndex = i;
18:       }
19:     }
20:     size--;
21:     elts[minIndex]= elts[size];
22:     return min;
23:   }
24: }
```

Example

```
Bag.java:15: Warning: Array index
possibly too large (...
if (elts[i] < min) {
        ^

Bag.java:21: Warning: Possible
negative array index (...
elts[minIndex] = elts[size];
                      ^
```

```
 1: class Bag{
 2:   int size;
2a:   //@ invariant 0<=size &&
   size<=elts.length
 3:   /*@non_null*/ int[] elts;
   //valid:elts[0..size-1]
 4:
4a:   //@requires input != null
 5:   Bag(int[ ] input){
 6:     size = input.length;
 7:     elts = new int[size];
 8:     System.arraycopy(input,0,elts,0,
   size);
 9:   }
10:
11:   int extractMin(){
12:     int min = Integer.MAX VALUE;
13:     int minIndex = 0;
14:     for (int i= 0; i <= size ; i++){
15:       if (elts[i] < min){
16:         min = elts[i];
17:         minIndex = i;
18:       }
19:     }
20:     size--;
21:     elts[minIndex]= elts[size];
22:     return min;
23:   }
24: }
```

Example

```
Bag.java:15: Warning: Array index
possibly too large (...
if (elts[i] < min) {
        ^
Bag.java:21: Warning: Possible
negative array index (...
elts[minIndex] = elts[size];
                      ^
```

```
 1: class Bag{
 2:   int size;
2a:   //@ invariant 0<=size &&
   size<=elts.length
 3:   /*@non_null*/ int[] elts;
   //valid:elts[0..size-1]
 4:
4a:   //@requires input != null
 5:   Bag(int[ ] input){
 6:     size = input.length;
 7:     elts = new int[size];
 8:     System.arraycopy(input,0,elts,0,
   size);
 9:   }
10:
11:   int extractMin(){
12:     int min = Integer.MAX VALUE;
13:     int minIndex = 0;
14:     for (int i= 0; i < size ; i++){
15:       if (elts[i] < min){
16:         min = elts[i];
17:         minIndex = i;
18:       }
19:     }
20:     size--;
21:     elts[minIndex]= elts[size];
22:     return min;
23:   }
24: }
```

Example

```
Bag.java:15: Warning: Array index
possibly too large (...
if (elts[i] < min) {
        ^

Bag.java:21: Warning: Possible
negative array index (...
elts[minIndex] = elts[size];
                       ^
```

```
 1: class Bag{
 2:   int size;
2a:   //@ invariant 0<=size &&
   size<=elts.length
 3:   /*@non_null*/ int[] elts;
   //valid:elts[0..size-1]
 4:
4a:   //@requires input != null
 5:   Bag(int[ ] input){
 6:     size = input.length;
 7:     elts = new int[size];
 8:     System.arraycopy(input,0,elts,0,
   size);
 9:   }
10:
11:   int extractMin(){
12:     int min = Integer.MAX VALUE;
13:     int minIndex = 0;
14:     for (int i= 0; i < size ; i++){
15:       if (elts[i] < min){
16:         min = elts[i];
17:         minIndex = i;
18:       }
19:     }
20:     size--;
20a:    if(size>=0)
21:       elts[minIndex]= elts[size];
22:     return min;
23:   }
24: }
```

Example

```
Bag.java:15: Warning: Array index
possibly too large (...
if (elts[i] < min) {
            ^

Bag.java:21: Warning: Possible
negative array index (...
elts[minIndex] = elts[size];
                          ^
```

```
 1: class Bag{
 2:   int size;
2a:   //@ invariant 0<=size &&
   size<=elts.length
 3:   /*@non_null*/ int[] elts;
   //valid:elts[0..size-1]
 4:
4a:   //@requires input != null
 5:   Bag(int[ ] input){
 6:     size = input.length;
 7:     elts = new int[size];
 8:     System.arraycopy(input,0,elts,0,
   size);
 9:   }
10:
11:   int extractMin(){
12:     int min = Integer.MAX VALUE;
13:     int minIndex = 0;
14:     for (int i= 0; i < size ; i++){
15:       if (elts[i] < min){
16:         min = elts[i];
17:         minIndex = i;
18:       }
19:     }
20:     size--;
20a:    if(size>=0)
21:       elts[minIndex]= elts[size];
22:     return min;
23:   }
24: }
```

Example

```
Bag.java:26: Warning: Possible
violation of object invariant
}
^
Associated declaration is
"Bag.java", line 3, col 6:
//@ invariant 0 <= size && size <=
elts.length
^
Possibly relevant items from the
counterexample context:
brokenObj == this
(brokenObj* refers to the object for
which the invariant
is broken.)
```

```
 1: class Bag{
 2:   int size;
2a:   //@ invariant 0<=size &&
   size<=elts.length
 3:   /*@non_null*/ int[] elts;
   //valid:elts[0..size-1]
 4:
4a:   //@requires input != null
 5:   Bag(int[ ] input){
 6:     size = input.length;
 7:     elts = new int[size];
 8:     System.arraycopy(input,0,elts,0,
   size);
 9:   }
10:
11:   int extractMin(){
12:     int min = Integer.MAX VALUE;
13:     int minIndex = 0;
14:     for (int i= 0; i < size ; i++){
15:       if (elts[i] < min){
16:         min = elts[i];
17:         minIndex = i;
18:       }
19:     }
19a:    if(size>0) {
20:       size--;
21:       elts[minIndex]= elts[size];
21a:    }
22:     return min;
23:   }
```

Example

```
Bag.java:26: Warning: Possible
violation of object invariant
}
^
Associated declaration is
"Bag.java", line 3, col 6:
//@ invariant 0 <= size && size <=
elts.length
^
Possibly relevant items from the
counterexample context:
brokenObj == this
(brokenObj* refers to the object for
which the invariant
is broken.)
```

No more warnings

Specification & Check

Specification:

```
class XXX {
 //@ INVARIANT

 //@requires PRE1
 //@ensures POST1
 method1(){

 }
}
```

1. Prove:

{pre1 ∧ invariant} ← **assumption**

method1

{post1 ∧ invariant} ← **guarantee**

2. Prove:

{pre1 ∧ invariant}

method1

{no_null_pointer_dereferences ∧
no_array_out_of_bounds}

(For constructors, the invariant is not part of the assumptions)

```
 1: class Bag{
 2:   int size;
2a:   //@ invariant 0<=size &&
    size<=elts.length
 3:   /*@non_null*/ int[] elts;
    //valid:elts[0..size-1]
 4:
4a:   //@requires input != null
 5:   Bag(int[ ] input){
 6:     size = input.length;
 7:     elts = new int[size];
 8:     System.arraycopy(input,0,elts,0,
    size);
 9:   }
10:
11:   int extractMin(){
12:     int min = Integer.MAX VALUE;
13:     int minIndex = 0;
14:     for (int i= 0; i < size ; i++){
15:       if (elts[i] < min){
16:         min = elts[i];
17:         minIndex = i;
18:       }
19:     }
19a:    if(size>0) {
20:       size--;
21:       elts[minIndex]= elts[size];
21a:    }
22:     return min;
23:   }
24: }
```

Example

For line 6, prove:

{input !=null}

→

{input != null}

For constructor, prove:

{input !=null}

```
size = input.length;
elts = new int[size];
System.arraycopy(input,0,elt
   s,0,size);
```

{elts != null && 0<=size && size <= elts.length}

Specification & Check

Specification:

```
class XXX {
 //@ INVARIANT

 //@requires PRE1
 //@ensures POST1
 method1(){
   body11
   method2();
   body12;
 }

 //@requires PRE2
 //@ensures POST2
 method2(){

 }

}
```

For nested functions you must prove

- PRE2 ∧ INVARIANT using PRE1 ∧ INVARIANT
- POST1 ∧ INVARIANT using PRE1 ∧ POST2 ∧ INVARIANT

```
1.  //@requires a != null
2.  //@requires 0<=i && i < a.size-1
3.  f(int[] a; int i){
4.    j = g(i);
5.    a[j] = 1;
6.  }
7.
8.  //@requires true
9.  //@ensures true
10. g(int i){
11.   return i+1;
12. }
```

Example: Modular Proof

For f, prove:

{a != null && 0 ≤ i < a.size - 1}

implies

{true}

and then

This part stays because g does not touch a or i

{a != null && 0 ≤ i < a.size - 1}

implies

{j < a.size}

This fails.

```
1.  //@ requires a != null
2.  //@requires 0<=i && i < a.size-1
3.  f(int[] a; int i){
4.    j = g(i);
5.    a[j] = 1;
6.  }
7.
8.  //@requires true
9.  //@ ensures \result = \old i + 1
10. int g(int i){
11.   return i+1;
12. }
```

Example: Modular Proof

For f, prove:

{a != null && 0 ≤ i < a.size - 1}

implies

{true}

and then

{a != null && 0 ≤ i < a.size – 1 && j = i + 1}

implies

{j < a.size}

This succeeds, which shows the need for annotations.

Simplification: Loop Unrolling

- Loops are hard: unrolling
- Proper unrolling: The following three programs are identical:

```
1.  while(c){ s1 } s2
```

```
2.  if(c){
      s1;
      while(c){
        s1
      }
    }
    s2
```

```
1.  if(c){
      s1;
      if(c){
        s1;
        while(c){
          s1
        }
      }
    }
    s2
```

Proper unrolling gives an equivalent program (but does not make it simpler)

Simplification: Loop Unrolling

- Loops are hard: unrolling
- Limited unrolling (ESCJava)

```
while(c){ s1 }; s2;
```

- Example: depth 1

```
if(c){
  s1;
  if(c){
    assume(false)
  }
}
s2
```

  - depth 2

```
while(c){ s1 }; s2;
```

becomes

```
if(c){
  s1;
  if(c){
    assume(false)
  }
}
s2
```

assume false means “all futher assertions can be assumed true”

Correctness is proven if loop is traversed 0, 1, or 2 times (set unrolling using –loop i)

No checks for runs that go through loop more often!

Loop Unrolling Effects

This program is deemed **correct** by ESCJava!

```
class Loop {
  Loop() {
    int i =  0;

    while(i < 100){
      i++;
    }
    //@ assert 4<3;
  }
}
```

This program is deemed correct by ESCJava (unles you cal ESCJava using –loop 2 or bigger)

```
import java.util.Random;

class Loop {
  Loop() {
    int i =  0;
    Random r = new Random();

    while(i < 100){
      i++;
      if(r.nextInt(2)>0) break;
    }
    //@ assert i < 2;
  }
}
```

Loop Unrolling: Effects

This program is deemed correct by ESCJava

```
import java.util.Random;

class Loop {
  Loop() {
    int i =  0;
    Random r = new Random();

    while(i < 100){
      i++;
      //if(r.nextInt(2)>0) break;
    }
    //@ assert 4<3;
  }
}
```

This program is deemed correct by ESCJava

```
import java.util.Random;

class Loop {
  Loop() {
    int i =  0;
    Random r = new Random();

    while(i < 100){
      i++;
      //if(r.nextInt(2)>0) break;
    }
    //@ assert 4<3;
  }
}
```

Loop Unrolling: Effects

This program is deemed correct by ESCJava

```
import java.util.Random;

class Loop {
  Loop() {
    int i =  0;
    Random r = new Random();

    while(i < 100){
      i++;
      //if(r.nextInt(2)>0) break;
    }
    //@ assert 4<3;
  }
}
```

This program is deemed correct by ESCJava

```
import java.util.Random;

class Loop {
  Loop() {
    int i =  0;
    Random r = new Random();

    while(i < 100){
      i++;
      //if(r.nextInt(2)>0) break;
    }
    //@ assert 4<3;
  }
}
```

```
 1: class Bag{
 2:   int size;
2a:   //@ invariant 0<=size &&
    size<=elts.length
 3:   /*@non_null*/ int[] elts;
    //valid:elts[0..size-1]
 4:
4a:   //@requires input != null
 5:   Bag(int[ ] input){
 6:     size = input.length;
 7:     elts = new int[size];
 8:     System.arraycopy(input,0,elts,0,
    size);
 9:   }
10:
11:   int extractMin(){
12:     int min = Integer.MAX VALUE;
13:     int minIndex = 0;
14:     for (int i= 0; i < size ; i++){
15:       if (elts[i] < min){
16:         min = elts[i];
17:         minIndex = i;
18:       }
19:     }
19a:    if(size>0) {
20:       size--;
21:       elts[minIndex]= elts[size];
21a:    }
22:     return min;
23:   }
24: }
```

Example: Loop unrolling

For line 21, prove:

{0<=size && size <= elts.length}

```
min = Integer.MAX VALUE;
minIndex = 0;
i = 0;
if(i < size){
  if (elts[i] < min){
    min = elts[i];
    minIndex = i;
  }
  if(elts[i] < min)
    assume(false);
}
size--;
{0<=minIndex<=elts.length &&
   0<=size<=elts.length}
```

This proof fails!