The ESC/Java 2 Tool

Néstor Cataño

ncatano@puj.edu.co

Faculty of Engineering

Pontificia Universidad Javeriana

ESC/JAVA

- Extended static checker developed by R. Leino *et al.* at Compaq Research Labs
- Tries to prove correctness of specifications at compile-time
- Fully automated

Limitations of ESC/JAVA

- Not sound, not complete but finds lots of potential bugs efficiently
- Good at proving simple properties
- Good at proving absence of runtime exceptions, *e.g.*, Null dereferencing, ArrayIndexOutOfBounds

Limitations of ESC/JAVA

- Not sound: It may issue a warning about a correct specifications
- Not complete: It may fail to warn about an incorrect specification
- Compromise between soundness-completeness and efficiency
  - ESC/JAVA points out a lot of potential bugs efficiently, and automatically

Static Checking Vs. Runtime Checking

- ESC/JAVA checks specification at compile-time
  ESC/JAVA proves correctness of specifications

Static Checking Vs. Runtime Checking

- ESC/JAVA checks specification at compile-time
  ESC/JAVA proves correctness of specifications
- The JML tools check specifications at run-time
  The JML tools test correctness of specifications

Static Checking Vs. Runtime Checking

- For runtime assertion checking, we choose what we specify
- For ESC/JAVA to accept a specification, one have to specify all pre- and postconditions of methods, and invariants (needed for modular verification)

Static Checking Vs. Runtime Checking

One of these assertions is wrong:

```
if (i <= 0 || j < 0) {
  ...
}
else if (j < 5) {
//@ assert i > 0 && 0 < j && j < 5;
  ...
}
else {
//@ assert i > 0 && j > 5;
  ...
}
```

JML may detect this with a comprehensive test suite.

ESC/JAVA 2 will detect this at compile-time.

Running ESC/JAVA

- Parsing phase
- Type-checking phase
- Static checking phase
  - `Simplify` prover
  - Potential bugs

Supported Platforms

- Linux
- MacOSX
- Cygwin on Windows
- Windows

Supported Platforms

- Linux
- MacOSX
- Cygwin on Windows
- Windows

Java 1.5 is not supported by ESC/JAVA 2

Specification Files

- `.java` file with Java and ESC/JAVA specifications
- `.refines-java` file with Java and ESC/JAVA specifications
- Files must be on the path

Modular Reasoning

The checking of methods in ESC/JAVA is done in a modular way on a method per method basis

```
public class C {
  int[] a;

  //@ ensures a.length >= 4;
  public void m() { p(); a[0] = 5; }


  public void p() { a = new int[20]; }
}
```

Modular Reasoning

The checking of methods in ESC/JAVA is done in a modular way on a method per method basis

```
public class C {
  int[] a;

  //@ ensures a.length >= 4;
  public void m() { p(); a[0] = 5; }


  public void p() { a = new int[20]; }
}
```

ESC/JAVA complains about a possible null dereference in `a[0] = 5`.

Modular Reasoning

The checking of methods in ESC/JAVA is done in a modular way on a method per method basis

```
public class C {
  int[] a;

  //@ ensures a.length >= 4;
  public void m() { p(); a[0] = 5; }

  //@ ensures a != null;
  public void p() { a = new int[20]; }
}
```

To stop ESC/JAVA complaining we add a postcondition to `p()`.

But then ESC/JAVA complains on an "`Array index possibly too large`".

Modular Reasoning

The checking of methods in ESC/JAVA is done in a modular way on a method per method basis

```
public class C {
  int[] a;

  //@ ensures a.length >= 4;
  public void m() { p(); a[0] = 5; }

  //@ ensures a != null && a.length == 20;
  public void p() { a = new int[20]; }
}
```

We thus strengthen `p()`'s postcondition to stop ESC/JAVA complaining.

Modular Reasoning

```
public class C {
  int[] a;


  public void m() { a[0] = 5; }

  public void C() { a = new int[20]; }
}
```

Once again ESC/JAVA complains on a null dereference.

Modular Reasoning: Continuation

```
public class C {
  int[] a;
   //@ invariant a != null && a.length == 20;

  public void m() { a[0] = 5; }

  public void C() { a = new int[20]; }
}
```

This time, one needs to add an invariant to stop ESC/JAVA 2 complaining.

Modular Reasoning: Continuation

```
public class C {
  int[] a;
   //@ invariant a != null && a.length == 20;

  public void m() { a[0] = 5; }

  public void C() { a = new int[20]; }
}
```

This time, one needs to add an invariant to stop ESC/JAVA 2 complaining.

Modular Reasoning: `assume`

```
public class C {
   ...
  public void m() {
   //@ assume a != null && a.length == 20;
   a[0] = 5; }

  public void C() { a = new int[20]; }
}
```

Alternatively one can use an **`assume`** specifications. But this is an unsound specification practice, mostly needed for debugging purposes.

Modular Reasoning: `assignable`

```
public class C {
  int[] a;

  //@ ensures a.length >= 4;
  public void m(){ a = new int[5]; p(); }


  public void p(){ ... }
}
```

What does ESC/JAVA need for checking `m()`'s postcondition?

Modular Reasoning: `assignable`

```
public class C {
  int[] a;

  //@ ensures a.length >= 4;
  public void m(){ a = new int[5]; p(); }


  public void p(){ ... }
}
```

If no postcondition is provided for `p()`, an **`assignable`** specification stating that `p()` does not modifies a is needed.

Bag Demo

```
class Bag {
  int[] a;
  int n;

  Bag(int[] input) {
    n = input.length;
    a = new int[n];
    System.arraycopy(input, 0, a, 0, n);
  }

  int extractMin() {
    int m = Integer.MAX_VALUE;
    int mindex = 0;
    for (int i = 0; i < n; i++)
     if (a[i] < m) { mindex = i; m = a[i]; }
    n--;
    a[mindex] = a[n];
    return m;
  }
}
```