ESC/Java2
Use and Features

David Cok, Joe Kiniry, Erik Poll

Eastman Kodak Company, University College Dublin,
and Radboud University Nijmegen
The ESC/Java2 tool
ESC/Java2 consists of a

- parsing phase (syntax checks),
- typechecking phase (type and usage checks),
- static checking phase (reasoning to find potential bugs) - runs a behind-the-scenes prover called Simplify

Parsing and typechecking produce cautions or errors. Static checking produces warnings.

The focus of ESC/Java2 is on static checking, but reports of bugs, unreported errors, confusing messages, documentation or behavior, and even just email about your application and degree of success are Very Welcome. [and Caution: this is still an alpha release]
Running ESC/Java2

- Download the binary distribution from http://secure.ucd.ie/products/opensource/ESCJava2
- Untar the distribution and follow the instructions in README.release about setting environment variables.
- Run the tool by doing one of the following:
  - Run a script in the release: escjava2 or escj.bat
  - Run the tool directly with java -cp esctools2.jar escjava.Main, but then you need to be sure to provide values for the -simplify and -specs options.
  - Run a GUI version of the tool by double-clicking the release version of esctools2.jar
  - Run a GUI version of the tool by executing it with java -jar esctools2.jar (in which case you can add options).
ESC/Java2 is supported on

- Linux
- MacOSX
- Cygwin on Windows
- Windows (but there are some environment issues still to be resolved)
- Solaris (in principle - we are not testing there)

Note that the tool itself is relatively portable Java, but the underlying prover is a Modula-3 application that must be compiled and supplied for each platform.

Help with platform-dependence issues is welcome.
The application relies on the environment having

- a Simplify executable (such as Simplify-1.5.4.macosx) for your platform, typically in the same directory as the application’s jar file;

- the `SIMPLIFY` environment variable set to the name of the executable for this platform;

- a set of specifications for Java system files - by default these are bundled into the application jar file, but they are also in `jmlspecs.jar`.

- The scripts prefer that the variable `ESCTOOLS_RELEASE` be set to the directory containing the release.
The items on the command-line are either options and their arguments or input entries. Some commonly used options (see the documentation for more):

- **-help** - prints a usage message
- **-quiet** - turns off informational messages (e.g. progress messages)
- **-nowarn** - turns off a warning
- **-classpath** - sets the path to find referenced classes [best if it contains ‘.’]
- **-specs** - sets the path to library specification files
- **-simplify** - provides the path to the simplify executable
- **-f** - the argument is a file containing command-line arguments
- **-nocheck** - parse and typecheck but no verification
- **-routine** - restricts checking to a single routine
- **-eajava, -eajml** - enables checking of Java assertions
- **-counterexample** - gives detailed information about a warning
Input entries

The input entries on the command-line are those classes that are actually checked. Many other classes may be referenced for class definitions or specifications - these are found on the classpath (or sourcepath or specspath).

- **file names** - of java or specification files (relative to the current directory)
- **directories** - processes all java or specification files (relative to the current directory)
- **package** - (fully qualified name) - found on the classpath
- **class** - (fully qualified name) - found on the classpath
- **list** - (prefaced by `-list`) - a file containing input entries
Specifications may be added directly to .java files

Specifications may alternatively be added to specification files.
  - No method bodies
  - No field initializers
  - Recommended suffix: .refines-java
  - Recommend a `refines` annotation (see documentation)
  - Must also be on the classpath
package java.lang;
import java.lang.reflect.*;
import java.io.InputStream;

public final class Class implements java.io.Serializable {

    private Class();

    /** also public normal_behavior 
     * @ ensures \result != null & & !\result.equals("")
     * @ & & (\* \result is the name of this class object \*)
     * @*/
    public /**@ pure @*/ String toString();

    ....
ESC/Java2 reasons about every method individually. So in

```java
class A {
    byte[] b;
    public void n() {
        b = new byte[20];
    }
    public void m() {
        n();
        b[0] = 2;
        ...
    }
}
```

ESC/Java2 warns that `b[0]` may be a null dereference here, even though you can see that it won’t be.
To stop ESC/Java2 complaining: add a postcondition

```java
class A{
    byte[] b;
    //@ ensures b != null && b.length = 20;
    public void n() {
        b = new byte[20];
    }
    public void m() {
        n();
        b[0] = 2;
        ...
    }
}
```

So: property of method that is relied on has to be made explicit.
Also: subclasses that override methods have to preserve these.
Similarly, ESC/Java will complain about $b[0] = 2$ in

```java
class A{
    byte[] b;
    public void A() { b = new byte[20]; }
    public void m() { b[0] = 2;
        ...
    }
}
```

Maybe you can see that this is a spurious warning, though this will be harder than in the previous example: you’ll have to inspect all constructors and all methods.
To stop ESC/Java2 complaining here: add an invariant

```java
class A{
    byte[] b;
    //@ invariant b != null && b.length == 20;
    // or weaker property for b.length ?
    public void A() { b = new byte[20]; }
    public void m() { b[0] = 2;
        ...
    }
}
```

So again: properties you rely on have to be made explicit.
And again: subclasses have to preserve these properties.
Alternative to stop ESC/Java2 complaining: add an assumption:

```java
...  
//@ assume b != null && b.length > 0;
    b[0] = 2;
...  
```

Especially useful during development, when you’re still trying to discover hidden assumptions, or when ESC/Java2’s reasoning power is too weak.

*(requires can be understood as a form of assume.)*
What does ESC/Java need to know about `o.n` to check the second `assert`?
A detailed spec for `o.n` might give a postcondition saying that `b[0]` is still 0.
class A{
    byte[] b;
    ...
    public void m() {
        b = new byte[3];
        //@ assert b[0] == 0; // ok!
        o.n();
        //@ assert b[0] == 0; // ok?
        ...
    }
}

If the postcondition of o.n doesn’t tell us b won’t be not null – and can’t be expected to – we need the assignable clause to tell us that o.n won’t affect b[0].

Declaring o.n as pure would solve the problem.
ESC/Java is not complete

ESC/Java may produce warnings about correct programs.

```java
/*@ requires 0 < n;
@ ensures \result ==
@ @ (\exists int x,y,z;
@ @ \quad pow(x,n)+pow(y,n) == pow(z,n));
@*/

public static boolean fermat(double n) {
    return (n==2);
}
```

Warning: *postcondition possibly not satisfied*  
(Typically, the theorem prover times out in complicated cases.)
ESC/Java is not sound

ESC/Java may fail to produce warning about incorrect program.

```java
public class Positive{
    private int n = 1;    //@ invariant n > 0;

    public void increase(){ n++; }
}
```

ESC/Java(2) produces no warning, but `increase` may break the invariant, namely if `n` is $2^{32} - 1$.

This can be fixed by improved model of Java arithmetic, but this does come at a price (both in specs and in code).
ESC/Java is not sound

More fundamental problem: sound modular verification for OO programs with invariants.

```java
public class A{
    B b;
    int x;
    //@ invariant x <= b.y;
    void decr_x(){
        x--;
    }
}

public class B{
    int y;
    void decr_y(){
        y--;
    }
}
```

How can we know that invoking `decr_y` on some `B` won’t break the invariant of some `A`, or some object whose invariant depends on a `B` object.
public class A{
    B b;
    int x;
    //@ invariant x <= b.y;
    void decr_x(){x++;}
}

public class B{
    int y;
    void incr_y(){y++;}
}

public class D{
    B b;
    void decr_y(){
        b.y--;    }
}

How can $D$ know it might be breaking $A$’s invariant?
Modular verification for (open) OO programs with invariants is a big & fundamental problem. Most verification tools fail here. Root causes:

1. invariants talking about another object’s fields
2. object modifying another object’s field
3. possibility of aliasing

NB 1 & 2 are unavoidable, eg. think of an object modifying – or its invariant mentioning – the contents of an array field

Alias control and ownership might provide solutions, eg. universes by Peter Müller & co or explicit pack/unpack operations by Rustan Leino & co.