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

Structure of ESC/Java2

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://www.cs.kun.nl/sos/research/escjava
- 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).

Supported platforms

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.

Environment

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.**

Command-line options

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**

Specification files

- 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

Specification file example

```
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();

    ....
```

Bag demo

modular reasoning

ESC/Java2 reasons about every method individually. So in

```
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.

modular reasoning

To stop ESC/Java2 complaining: add a postcondition

```
class A{
 byte[] b;
 //@ ensures b != null && b.length = 20;
 public void n() { a = 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.

modular reasoning

Similarly, ESC/Java will complain about `b[0] = 2` in

```
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.

modular reasoning

To stop ESC/Java2 complaining here: add an invariant

```
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.

assume

Alternative to stop ESC/Java2 complaining: add an assumption:

```
...
//@ 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`.)

need for assignable clauses

```
class A{
 byte[] b;
 ...
 public void m() { ...
                   b = new byte[3];
                   //@ assert b != null; // ok!
                   o.n(...);
                   //@ assert b != null; // ok?
                   ...
 }
```

What does ESC/Java need to know about `o.n` to check the second assert ?

need for assignable clauses

```
class A{
 byte[] b;
 ...
 public void m() { ...
                   b = new byte[3];
                   //@ assert b != null; // ok!
                   o.n(b);
                   //@ assert b != null; // ok?
                   ...
 }
```

A detailed spec for `o.n` might give a postcondition saying that `b` is not null.

need for assignable clauses

```
class A{
 byte[] b;
 ...
 public void m() { ...
                   b = new byte[3];
                   //@ assert b != null; // ok!
                   o.n();
                   //@ assert b != null; // 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`.

Declaring `o.n` as pure would solve the problem.