Advances in Programming Languages
APL5: ESC/Java2 — The Java Extended Static Checker

David Aspinall
(including slides by Ian Stark and material adapted from ESC/Java2 tutorial by David Cok, Joe Kiniry and Erik Poll)

School of Informatics
The University of Edinburgh

Monday 25th January 2010
Semester 2 Week 3
This is the third of four lectures about some techniques and tools for formal verification, specifically:

- Hoare logic
- JML: The Java Modeling Language
- ESC/Java2: The Extended Static Checker for Java
- Certifying correctness: approaches and examples
The *Java Modeling Language*, JML, combines model-based and contract approaches to specification.

Some design features:

**The specification lives close to the code**

Within the Java source, in *annotation comments* /*@...@*/

**Uses Java syntax and expressions**

Rather than a separate specification language.

**Common language for many tools and analysis**

Tools add their own extensions, and ignore those of others.

Web site: jmlspecs.org
Outline

1. ESC/Java 2
2. Common idioms
3. Behavioural subtyping
4. Frame conditions
5. Future of JML and ESC/Java
6. Summary
“The Extended Static Checker for Java version 2 (ESC/Java2) is a programming tool that attempts to find common run-time errors in JML-annotated Java programs by static analysis of the program code and its formal annotations.”

http://kind.ucd.ie/products/opensource/ESCJava2

It is available both as a command-line tool and a plugin for the Eclipse development environment.

ESC/Java performs different kinds of static check:
- checks based on types, flow of data, existing Java declarations;
- JML annotation checking that can be carried out directly;
- logical assertions that need an external proof tool.

These last ones are passed to the Simplify automated theorem prover.

Recent versions of ESC/Java also support other provers.
History

ESC/Modula-3  DEC Systems Research Center (SRC) 1991–1996

ESC/Java  Compaq SRC, then Hewlett-Packard 1997–2002

ESC/Java2  University of Nijmegen, University College Dublin 2004–2009

emerging JML+ESC successors
  University of Central Florida,
  Kansas State University,
  Concordia University, ... 

Many different checks

ESC/Java2 checks for very many things. These include:

- Null pointer dereference
- Negative array index
- Array index too large
- Invalid type casts
- Array storage type mismatch
- Divide by zero
- Negative array size
- Unreachable code
- Deadlock in concurrent code
- Race condition
- Race condition
- Unchecked exception
- Object invariant broken
- Loop invariant broken
- Precondition not satisfied
- Postcondition not satisfied
- Assertion not satisfied

JML annotations and assertions can help with all of these.
As a practical tool ESC/Java makes some compromises: it is not perfect.
- Not sound: it may approve an incorrect program.
- Not complete: it may complain about a correct program.

However, it reliably checks straightforward specifications, and automatically points out many potential bugs.

In particular:
- Distinguishes between errors (definitely bad), warnings (could be bad) and cautions (can’t be sure it’s good).
- Sources of unsoundness and incompleteness are documented.
Soundness and Completeness

As a practical tool ESC/Java makes some compromises: it is not perfect.

- Not sound: it may approve an incorrect program.
- Not complete: it may complain about a correct program.

However, it reliably checks straightforward specifications, and automatically points out many potential bugs.

In particular:

- Distinguishes between *errors* (definitely bad), *warnings* (could be bad) and *cautions* (can’t be sure it’s good).
- Sources of unsoundness and incompleteness are documented.

...as we know, there are “known knowns”; there are things we know we know. We also know there are “known unknowns”; that is to say we know there are some things we do not know.

But there are also “unknown unknowns” — the ones we don’t know we don’t know.

(Donald Rumsfeld, 2002)
Alternatively: try the command line tools. Here is a pseudo-demo.
1. ESC/Java 2
2. Common idioms
3. Behavioural subtyping
4. Frame conditions
5. Future of JML and ESC/Java
6. Summary
JML and ESC/Java2 introduce keywords for common specifications.

One of the most common specification requirements in Java is that objects be non-null. That’s because one of the most common Java programming errors is `NullPointerException`.

```java
//@ non_null
Object o;
```

Now every method invocation on `o` is known to not cause an exception, *but* every assignment to `o` must be checked to be non-null.

This is so important that it is about to enter the Java language as an official annotation `@NonNull`, to be exploited by ordinary compilers.
Common specification idioms: non null

JML and ESC/Java2 introduce keywords for common specifications.

One of the most common specification requirements in Java is that objects be non-null. That’s because one of the most common Java programming errors is NullPointerException.

```
//@ non_null
Object o;
```

Now every method invocation on o is known to not cause an exception, but every assignment to o must be checked to be non-null.

This is so important that it is about to enter the Java language as an official annotation `@NonNull`, to be exploited by ordinary compilers.

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. [...] My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn’t resist the temptation to put in a null reference

(Tony Hoare, 2009)
Part of the object-oriented paradigm: an object in a subclass can **behave like** an object in a superclass.

Sometimes known as Liskov’s *principle of substitutivity*:

> properties that can be proved using the specification of an object’s presumed type should hold even though the object is actually a subtype of that type

[`Liskov and Wing, 1994`]

This is captured by requiring, when A **extends** B

- each invariant in subclass A $\implies$ an invariant in B.
- precondition for A.m $\iff$ precondition for B.m
- postcondition for A.m $\implies$ postcondition for B.m
Inherited specifications

Behavioural subtyping is ensured by *inherited specifications*. A child class automatically inherits the specification of its parent.

```java
class Parent {
    //@ requires i >= 0;
    //@ ensures \result >= i;
    int m(int i){ ... }
}

class Child extends Parent {
    //@ also
    //@ requires i <= 0
    //@ ensures \result <= i;
    int m(int i){ ... }
}
```
The specification for Child is short for:

```java
class Child extends Parent {
    /* @ requires i >= 0;
        @ ensures result >= i;
        @ also
        @ requires i <= 0
        @ ensures result <= i;
     */
    int m(int i){ ... }
}
```

What can the result of m(0) be?
Inherited specifications: the answer

This specification is in fact equivalent to:

```java
class Child extends Parent {
    /*@ requires i <= 0 || i >= 0; */
    @ ensures i >= 0 ==> \result >= i;
    @ ensures i <= 0 ==> \result <= i;
    @*/
    int m(int i){ ... }
}
```
Inherited specifications: the answer

This specification is in fact equivalent to:

```java
class Child extends Parent {
    /*
     * @ requires i <= 0 || i >= 0;
     * @ ensures i >= 0 ==> \result >= i;
     * @ ensures i <= 0 ==> \result <= i;
     */
    int m(int i) { ... }
}
```

- moral: take care specifying methods that may be overridden
- complex specifications may use a test

```java
typeof(this)==\type(Parent)
```

to guard properties that are likely to change in child classes.
Outline

1. ESC/Java 2
2. Common idioms
3. Behavioural subtyping
4. Frame conditions
5. Future of JML and ESC/Java
6. Summary
Methods leading to madness

Imperative programs can be very difficult to verify because of *reference escape* and *aliasing*.

```java
class MyClass {
    int i;

    //@ modifies i;
    void m(MyClass o) {
        i = 3;
        o.i = 2; // ESC/Java2 gives a warning
    }
}
```
When verifying, we want to use frame conditions that say what stays the same when a method is executed.

Usually we want to assume that as much as possible is unchanged, but the conservative default in ESC/Java2 is:

```java
//@ modifies \everything
```

Another example where the functional paradigm is very useful:

```java
//@ pure
def public int getX() { return x; }
```

The `pure` annotation implies `modifies \nothing`. 
ESC/Java2 and other JML tools have an old-fashioned *batch mode* architecture

they’re also stuck on Java 1.4

**JML4** proposed an *Integrated Verification Environment*

...integrated with Eclipse JDT

...allowing multi-threaded verification, with per-method and per-class parallelism

Development is now suspended, may be superseded by JMLEclipse and OpenJML.

---

**JML4 compiler phases**

from James, Chalin, Giannas, Karabotsos: *Distributed, Multi-threaded Verification of Java Programs*, SAVCBS 2008.
Summary

ESC/Java 2

- A practical tool combining several analysis techniques (types, dataflow, proof)
- Many checks, but exhibits false positives and missing defects
- Has specialised annotations extending core JML (unreachable)
- Primarily batch mode, Java 1.4
- Some advanced JML aspects handled by ESC/Java2
  - `non_null`, `modifies`, `pure`
  - specification inheritance
- Follow-up projects currently in a state of flux
  - OpenJML
  - JML4 and ESC4
  - JMLEclipse

Watch jmlspecs.org and the JML specs wiki.