Background

• ESC/Java - Original development by Compaq Systems Research Center (1997) as a successor of ESC/Modula-3

• Sources released to open source community (2002)

• ESC/Java2 alpha – University of Nijmegen, Security Systems group, added JML specifications support (2004)

• ESC/Java2 – University College Dublin, KindSoftware Research Group (2004~) support multiple theorem provers, Eclipse plug-in
Basic Operation

- Statically check the code (NO EXECUTION!)
- Take into consideration the JML annotations (pragma-s)
- JML annotations are an option (not a must!) can perform regular static code analysis or combined with model check.
Basic JML Reminder

- Preconditions /*@requires
- Postconditions /*@ensures
- Exceptions /*@signals
- “Writable” fields /*@assignable
- Const method /*@pure
- Class invariant /*@invariant
- Inherit supertype conditions /*@also
- JML assertion /*@assert
Basic JML Reminder

- Return value of the method - \result
- Value of the variable at time of entry to the method - \old(<variable name>)
- Universal quantifier - \forall
- Existential quantifier - \exists
- Logical a implies b - a==>b
- Logical a iff b - a<==>b
- /*@model - Can model entire java constructs (functions etc.) without the true implementations
Spec Files

• ESC/Java supports use of java source files or compiled byte code (without source)

• If we want to add JML annotation we simply add a SPEC file, which is annotation only

• Example

definition 1

package java.lang;
public class UnsupportedOperationException extends RuntimeException {

    /** @ public normal_behavior
        @  ensures standardThrowable(null);
        @*/
    //@ pure
    public UnsupportedOperationException();

    /** @ public normal_behavior
        @  ensures standardThrowable(s);
        @*/
    //@ pure
    public UnsupportedOperationException(String s);
}

Spec Files

• Spec files may have any of the extensions associated with JML (.java, .spec, .jml, .refines-java, .refines-spec, .refines-jml)

• Spec files may “refine” each other by extending the specs, a parsing process seeks them out

• Syntax: //@ refine “filename”

• During parsing – refining file must match a refined file (which in turn might be refining other file), the search stops when first non refining spec is found in the chain
Spec Files

• It is not compulsory that the search should end with a source/class file – the modeling concept takes into account that JML description may/should precede the implementation.

*Note: some of the abilities implementation status is questionable, some are partially implemented, some are just de-facto not used in the package of bundled specs for standard java packages
Annotations Syntax

• Three types of annotations are accepted:
  – Standard JML single line comments: //@
  – Standard multi line comments /*@ ... */ or @*/
  – Javadoc embedded annotations enclosed in any of the tags:
    • <esc></esc> or <ESC></ESC>
    • <jml></jml> or <JML></JML>

• Distinguishing annotations:
  – Known to JML but not to ESC/Java2 //+@ or /*+@
  – Known to ESC/Java2 but not to JML //:@ or /*:@

*Note: -parsePlus forces parse the //+@ tags
A Little about JML v.s. ESC/J2

- `/*@model Import` – problematic, will not affect the compilation (naturally) but will be regarded as a normal java import by the ESC/Java2, hence possible type ambiguities may be reported.

- JML access modifiers - ignored
Underneath

- Step 1: ESC/Java 2 translates code into “guarded commands”
- Step 2: Translation into single assignment representation
- Step 3: Translation into verification conditions (first order logic in our case)
- Step 4: Verification conditions are passed to theorem prover (in our case Simplify)
- Step 5: Theorem prover proves correctness or provides us with counter examples.
Underneath

ESC/Java 2

Java/Class/JML

First Order Logic

Simplify

Results Report
Underneath

- Simplify – theorem prover employed by ESC/Java2 (by Dermot Cochran) (...SAT solver)
- Translate the code into series of predicates and axioms, attempt to prove correctness.
- Example:

```java
void f(T[][] a, T[] b) {
    a[0] = b;
}
```

need to prove that b is a subtype of element type of a
Simplify is given that:
  \( \text{typeof}(a) \text{ <: array(array}(T)) \);
  \( \text{typeof}(b) \text{ <: array}(T) \);

and needs to prove that:
  \( \text{typeof}(b) \text{ <: elemType}(\text{typeof}(a)) \);

From the array axiom triggered on the first antecedent, we have:
  \( \text{typeof}(a) == \text{array}(\text{elemType}(\text{typeof}(a))) \) \&\&
  \( \text{elemType}(\text{typeof}(a)) \text{ <: array}(T) \);

From the array axiom triggered on the last line, we have:
  \( \text{elemType}(\text{typeof}(a)) == \text{array}(\text{elemType}(\text{elemType}(\text{typeof}(a)))) \) \&\&
  \( \text{elemType}(\text{elemType}(\text{typeof}(a))) \text{ <: T} \);

If \( T \) is a final type, then the final type axiom is triggered, and yields that:
  \( \text{elemType}(\text{elemType}(\text{typeof}(a))) == T \);
  \( \text{elemType}(\text{typeof}(a)) == \text{array}(T) \);

and hence the desired consequent holds.
Results

- The output is an annotated warning per analyzed file in a tree structure
- Maximum number of counterexamples found by Simplify controlled by PROVER_CC_LIMIT variable
- Run on a single package (common practice) with a -package option
- Reasons about every method individually
The Tool
The Tool
The Tool

ESC/Java2 (Version: ESCJava-2.0b0)

File  View  Check  Tools  L&F  Help

Reload  Clear  Check  Stop

Clearing all results because the CLASSPATH changed

CLASSPATH:

Specs path:
/usr/opt/ESCJava-2.0b0

Files, directories, classes, packages and lists to process:
examples/0.0/Bag.java
...

31444912 bytes
The Tool

- Examples/0.0
  - Examples/0.0/Bag.java
  - Bag
    - Bag(int[])
    - exactMin()