Formal Methods for Java
Lecture 9: Extended Static Checking

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

May 24, 2017
Runtime vs. Static Checking

Runtime Checking
- finds bugs at run-time,
- tests for violation during execution,
- can check most of the JML,
- is done by openjml.jar -rac.

Static Checking
- finds bugs at compile-time,
- proves that there is no violation,
- can check only parts of the JML,
- is done by openjml.jar -esc.
Developed by the DEC Software Research Center (now HP Research),
Extended by David Cok and Joe Kiniry (Kind Software)
Rewritten in OpenJML by David Cok
Proves correctness of specification,
Is neither sound nor complete (but this will improve),
Is useful to find bugs.
Consider the following code:

```java
Object[] a;
void m(int i) {
    a[i] = "Hello";
}
```

- Is `a` a null-pointer? *(NullPointerException)*
- Is `i` nonnegative? *(ArrayIndexOutOfBoundsException)*
- Is `i` smaller than the array length? *(ArrayIndexOutOfBoundsException)*
- Is `a` an array of `Object` or `String`? *(ArrayStoreException)*

openjml (esc) warns about these issues. *(Demo)*
ESC and run-time exceptions

ESC checks that no undeclared run-time exceptions occur.

- NullPointerException
- ClassCastException
- ArrayIndexOutOfBoundsException
- ArrayStoreException
- ArithmeticException
- NegativeArraySizeException
- other run-time exception, e.g., when calling library functions.
ESC and specification

ESC also checks the JML specification:

- **ensures** in method contract,
- **requires** in called methods,
- **assert** statements,
- **signals** clause,
- **invariant** (loop invariant and class invariant).

ESC assumes that some formulae hold:

- **requires** in method contract,
- **ensures** in called methods,
- **assume** statements,
- **invariant** (loop invariant and class invariant).
public void put(Object o) {
    int hash = o.hashCode();
    ...
}

results in Possible null dereference.

Solutions:

- Declare o as non_null.
- Add o != null to precondition.
- Add throws NullPointerException.
  (Also add signals (NullPointerException) o == null)
- Add Java code that handles null pointers.
  
  int hash = (o == null ? 0 : o.hashCode());
ClassCastException

class Priority implements Comparable {
    public int compareTo(Object other) {
        Priority o = (Priority) other;
        ...
    }
}

results in **Possible type cast error.**
Solutions:

- Add throws ClassCastException.
  (Also add signals (ClassCastException) !(other instanceof Priority))
- Add Java code that handles differently typed objects:
  ```java
  if (!(other instanceof Priority))
      return -other.compareTo(this)
  Priority o = ...
  ```

This results in a **Possible null dereference.**
void write(/*@non_null@*/ byte[] what, int offset, int len) {
    for (int i = 0; i < len; i++) {
        write(what[offset+i]);
    }
}

results in Possible negative array index

Solution:

- Add offset >= 0 to pre-condition, this results in Array index possibly too large.
- Add offset + len <= what.length.
- Still results in possibly negative array index.
- Add a loop invariant.
- ESC does not complain but there is still a problem. If offset and len are very large numbers, then offset + len can be negative. The code would throw an ArrayIndexOutOfBoundsException at run-time.
/*@ requires offset >= 0 && offset + len <= what.length; @*/

void write(/*@non_null@*/ byte[] what, int offset, int len) {
   /*@ loop_invariant i >= 0; @*/
    for (int i = 0; i < len; i++) {
        write(what[offset+i]);
    }
}

- $i \geq 0$ and $offset \geq 0$ proof that array index is not negative.
- $i \geq 0$ holds initially.
- If $i \geq 0$ holds before the loop, it holds after the loop.
public class Stack {
    /*@ non_null @*/ Object[] elems;
    int top;
    /*@ invariant 0 <= top && top <= elems.length; @*/

    /*@ requires top < elems.length; @*/
    void add(Object o) {
        elems[top++] = o;
    }
}

results in Type of right-hand side possibly not a subtype of array element type (ArrayStore).
Solutions:

- Add an invariant $\text{typeof}(\text{elems}) == \text{typeof}(\text{Object[]})$.
- Add a precondition $\text{typeof}(o) <: \text{elemtype}(\text{typeof}(\text{elems}))$. 
Types in assertions

- `typeof` gets the run-time type of an expression
  
  `typeof(obj) \sim obj.getClass()`.

- `elemtype` gets the base type from an array type.
  
  `elemtype(t1) \sim t1.getComponentType()`.

- `type` gets the type representing the given Java type.
  
  `type(Foo) \sim Foo.class`

- `<:` means is sub-type of.
  
  `t1 <: t2 \sim t2.isAssignableFrom(t1)`