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 jmlrac.

Static Checking
- finds bugs at compile-time,
- proves that there is no violation,
- can check only parts of the JML,
- is done by ESC/Java.
**ESC/Java 2**

- Developed by the DEC Software Research Center (now HP Research),
- Extended by David Cok and Joe Kiniry (Kind Software)
- **Proves** correctness of specification,
- Is neither **sound** nor **complete** (but this will improve),
- Is useful to find bugs.

- **Homepage:**
  http://kind.ucd.ie/products/opensource/ESCJava2
- **Download link:** ESCJava2.0.5
- Works with Java-1.5.0 (1.6.0 does not work).
Example

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)

ESC/Java warns about these issues. (Demo)
ESC/Java 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/Java and specification

ESC/Java also checks the JML specification:

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

ESC/Java assumes that some formulae hold:

- **requires** in method contract,
- **ensures** in called methods,
- **assume** statements,
- **invariant** (loop invariant and class invariant).
NullPointerException

```
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.
  ```java
  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.
ArrayIndexOutOfBoundsException

```java
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 \( offset \geq 0 \) to pre-condition, this results in Array index possibly too large.
- Add \( offset + len \leq what.length \).
- ESC/Java 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.
- The correct pre-condition is:
  ```java
  /*@ requires offset >= 0 && offset + len >= offset @
  @ offset + len <= what.length; @*/
  ```
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})).