ESC/Java2

Use and Features

David Cok, Erik Poll, Joe Kiniry

Eastman Kodak Company; University of Nijmegen

June 2004

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();

    ....
```

ESC/Java2's static checking warnings

Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- warnings about possible runtime exceptions: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)

- **These are the most common runtime exceptions caused by coding problems (that is, not by explicitly throwing an exception)**
- **They do not include nearly all of the possible runtime exceptions**
- **Most of the others are explicitly thrown by various library methods**

Cast Warning

The Cast warning occurs when ESC/Java2 cannot verify that a ClassCastException will not be thrown:

```
public class CastWarning {
  public void m(Object o) {
    String s = (String)o;
  }
}
```

results in

```
------------------------------------------------------------------------
CastWarning.java:3: Warning: Possible type cast error (Cast)
    String s = (String)o;
               ^

------------------------------------------------------------------------
```

But this is OK:

```
public class CastWarningOK {
  public void m(Object o) {
    if (o instanceof String) { String s = (String)o; }
  }
}
```

Cast Warning

So is this:

```
public class CastWarningOK2 {
  //@ requires o instanceof String;
  public void m(Object o) {
    String s = (String)o;
  }
}
```

Null Warning

The Null warning occurs when ESC/Java2 cannot verify that a NullPointerException will not be thrown:

```
public class NullWarning {
  public void m(Object o) {
    int i = o.hashCode();
  }
}
```

results in

```
----------------------------------------------------------------------
NullWarning.java:3: Warning: Possible null dereference (Null)
    int i = o.hashCode();
            ^

----------------------------------------------------------------------
```

But this is OK:

```
public class NullWarningOK {
  public void m(/*@ non_null */ Object o) {
    int i = o.hashCode();
  }
}
```

ArrayStore Warning

The ArrayStore warning occurs when ESC/Java2 cannot verify that the assignment of an object to an array element will not result in an ArrayStoreException:

```
public class ArrayStoreWarning {
  public void m(Object o) {
    Object[] s = new String[10];
    s[0] = o;
  }
}
```

results in

```
----------------------------------------------------------------------
ArrayStoreWarning.java:4: Warning: Type of right-hand side possibly not
a subtype of array element type (ArrayStore)
    s[0] = o;
         ^

----------------------------------------------------------------------
```

But this is OK:

```
public class ArrayStoreWarningOK {
  public void m(Object o) {
    Object[] s = new String[10];
    if (o instanceof String) s[0] = o;
  }
}
```

ZeroDiv, index Warnings

- **ZeroDiv - issued when a denominator (integer division) may be 0**
- **NegSize - issued when the array size in an array allocation expression may be negative**
- **IndexNegative - issued when an array index may be negative**
- **IndexTooBig - issued when an array index may be greater than or equal to the array length**

```
public class Index {
  void m() {
    int i = 0;
    int j = 8/i;  // Causes a ZeroDiv warning
    Object[] oo = new Object[i-1]; // NegSize warning
    oo = new Object[10];
    i = oo[-1].hashCode(); // IndexNegative warning
    i = oo[20].hashCode(); // IndexTooBig warning
  }
}
```

Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- **warnings about possible runtime exceptions: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)**
- **warnings about possible method specification violations: (Precondition, Postcondition, Modifies)**
  - **These are all caused by violations of explicit user-written method specifications**

Pre, Post warnings

These warnings occur in response to user-written preconditions (requires), postconditions (ensures, signals), or assert statements.

```
public class PrePost {
  //@ requires i >= 0;
  //@ ensures \result == i;
  public int m(int i);

  //@ ensures \result > 0;
  public int mm() {
    int j = m(-1); // Pre warning - argument must be >= 0
  }

  //@ ensures \result > 0;
  public int mmm() {
    int j = m(0);
    return j;
  }  // Post warning - result is 0 and should be > 0
}
```

Frame conditions

- To reason (modularly) about a call of a method, one must know what that method might modify: this is specified by
  - assignable clauses

```
//@ assignable x, o.x, this.*, o.*, a[*], a[3], a[4..5];
```

  - modifies clauses (a synonym)
  - pure modifier

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

- Assignable clauses state what fields may be assigned within a method - this is the set of what might be modified
- The default assignable clause is assignable \everything; (but it is better to be explicit because \everything; is not fully implemented and ESC/Java2 can reason better with more explicit frame conditions)
- A pure method is assignable \nothing;

Frame conditions

- A Modifies warning indicates an attempt to assign to an object field that is not in a modifies clause
- Note: Some violations of modifies clauses can be detected at typecheck time.
- Note also: Handling of frame conditions is an active area of research.

Modifies warnings

For example, in

```
public class ModifiesWarning {
  int i;

  //@ assignable i;
  void m(/*@ non_null */ ModifiesWarning o) {
    i = 1;
    o.i = 2; // Modifies warning
  }
}
```

we don't know if o equals this; since only this.i may be assigned, ESC/Java2 produces

```
------------------------------------------------------------------------
ModifiesWarning.java:7: Warning: Possible violation of modifies clause (Mo
    o.i = 2; // Modifies warning
    ^
Associated declaration is "ModifiesWarning.java", line 4, col 6:
  //@ assignable i;
      ^
------------------------------------------------------------------------
```

Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- **warnings about possible runtime exceptions: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)**
- **warnings about possible specification violations: (Precondition, Postcondition, Modifies)**
- **non null violations (NonNull, NonNullInit)**
  - **These warnings relate to explicit non_null field or parameter specifications**

NonNullInit warning

Class fields declared non_null must be initialized to values that are not null in each constructor, else a NonNullInit warning is produced.

```
public class NonNullInit {
  /*@ non_null */ Object o;

  public NonNullInit() { }
}
```

produces

```
------------------------------------------------------------------------
NonNullInit.java:4: Warning: Field declared non_null possibly
not initialized (NonNullInit)
  public NonNullInit() { }
                       ^
Associated declaration is "NonNullInit.java", line 2, col 6:
  /*@ non_null */ Object o;
      ^
------------------------------------------------------------------------
```

NonNull warning

A NonNull warning is produced whenever an assignment is made to a field or variable that has been declared non_null but ESC/Java2 cannot determine that the right-hand-side value is not null.

```
public class NonNull {
  /*@ non_null */ Object o;

  public void m(Object oo) { o = oo; } // NonNull warning
}
```

produces

```
------------------------------------------------------------------------
NonNull.java:4: Warning: Possible assignment of null to variable
declared non_null (NonNull)
  public void m(Object oo) { o = oo; } // NonNull warning
                             ^

Associated declaration is "NonNull.java", line 2, col 6:
  /*@ non_null */ Object o;
      ^

------------------------------------------------------------------------
```

NonNull warning

But this is OK

```
public class NonNull {
  /*@ non_null */ Object o;
  public void m(/*@ non_null */ Object oo) { o = oo; }
}
```

So is this

```
public class NonNull {
  /*@ non_null */ Object o;
  public void m(Object oo) {
    if (oo != null) o = oo;
  }
}
```

So is this

```
public class NonNull {
  /*@ non_null */ Object o;
  public void m() {
    o = new Object();
  }
}
```

non_null can be applied to

- a field
- a formal parameter
- a return value
- a local variable
- ghost and model variables

Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- warnings about possible runtime exceptions: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)
- warnings about possible method specification violations: (Precondition, Postcondition, Modifies)
- non null violations (NonNull, NonNullInit)
- loop and flow specifications (Assert, Reachable, LoopInv, DecreasesBound)
  - These are caused by violations of specifications in a routine body

Body assertions

- **Assert: warning occurs when an assert annotation may not be satisfied**
- **Reachable: not in JML, only in ESC/Java2; occurs with the //@ unreachable; annotation, which is equivalent to //@ assert false;**

Example:

```
public class AssertWarning {
  //@ requires i >= 0;
  public void m(int i) {
    //@ assert i >= 0; // OK
    --i;
    //@ assert i >= 0; // FAILS
  }
  public void n(int i) {
    switch (i) {
      case 0,1,2: break;
      default:   //@ unreachable; // FAILS
    }
  }
}
```

Loop assertions

- A loop_invariant assertion just before a loop asserts a predicate that is true prior to each iteration and at the termination of the loop (or a LoopInv warning is issued).
- A decreases assertion just before a loop asserts a (int) quantity that is non-negative and decreases with each iteration (or a DecreasesBound warning is issued).
- Caution: Loops are checked by unrolling a few times.

Example:

```
public class LoopInvWarning {
  public int max(/*@ non_null */ int[] a) {
    int m=Integer.MAX_VALUE;
    //@ loop_invariant (\forall int j; 0<=j && j<i; a[j] <= m);
    //@ decreases a.length - i - 1;
    for (int i=0; i<a.length; ++i) {
      if (m < a[i]) m = a[i];
    }
    return m;
  }
}
```

In the scope of the loop variable

Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- warnings about possible runtime exceptions: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)
- warnings about possible method specification violations: (Precondition, Postcondition, Modifies)
- non null violations (NonNull, NonNullInit)
- loop and flow specifications (Assert, Reachable, LoopInv, DecreasesBound)
- **warnings about possible class specification violations: (Invariant, Constraint, Initially)**

class invariant warnings

Invariant and constraint clauses generate additional postconditions for every method. If they do not hold, appropriate warnings are generated:

```
public class Invariant {
  public int i,j;
  //@ invariant i > 0;
  //@ constraint j > \old(j);

  public void m() {
    i = -1; // will provoke an Invariant error
    j = j-1; // will provoke a Constraint error
  }
}
```

Initially warning

An initially clause is a postcondition for every constructor:

```
public class Initially {

  public int i; //@ initially i == 1;

  public Initially() { } // does not set i - Initially warning
}
```

produces

```
------------------------------------------------------------------------
Initially.java:5: Warning: Possible violation of initially condition
at constructor exit (Initially)
  public Initially() { } // does not set i - Initially warning
                       ^
Associated declaration is "Initially.java", line 3, col 20:
  public int i; //@ initially i == 1;
                    ^
------------------------------------------------------------------------
```

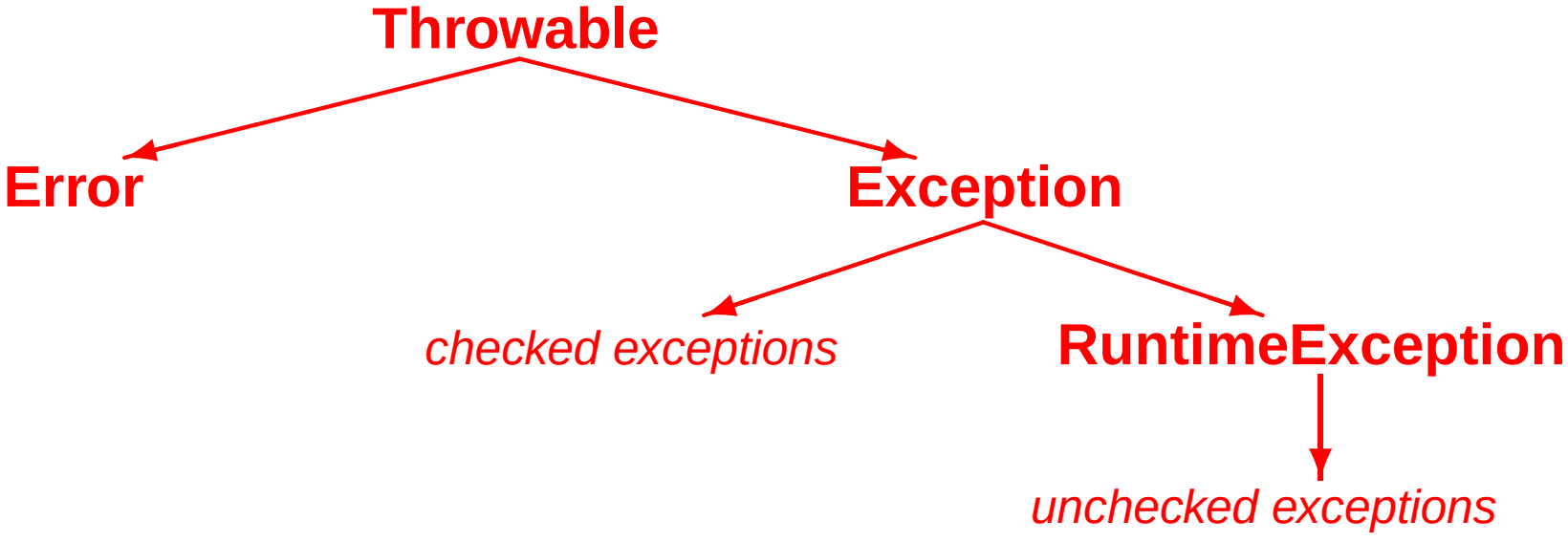
Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- warnings about possible runtime exceptions: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)
- warnings about possible method specification violations: (Precondition, Postcondition, Modifies)
- non null violations (NonNull, NonNullInit)
- loop and flow specifications (Assert, Reachable, LoopInv, DecreasesBound)
- warnings about possible class specification violations: (Invariant, Constraint, Initially)
- **exception problems (Exception)**
  - **These warnings are caused by undeclared exceptions**

Exceptions - Errors

- Java Errors (e.g. OutOfMemoryError) can be thrown at any time
  - No declarations are needed in throws clauses
  - No semantics are implied by JML
  - No checking is performed by ESC/Java2



Checked Exceptions

- **Java checked exceptions (e.g. FileNotFoundException) are Exceptions that are not RuntimeExceptions:**
  - **Declarations of exceptions mentioned in the body are required in throws clauses**
  - **ESC/Java2 checks during typechecking that throws declarations are correct (as a Java compiler does)**
  - **Typically specified in signals clauses in JML**
  - **ESC/Java2 checks via reasoning that signals conditions hold**
  - **Default specification is that *declared* exceptions may occur: signals (Exception) true;**
  - **ESC/Java2 presumes that checked exceptions not declared in a throws clause will not occur.**

Unchecked Exceptions

- **Java unchecked exceptions (e.g. NoSuchElementException) are RuntimeExceptions:**
  - **Java does not require these to be declared in throws clauses**
  - **ESC/Java2 is stricter than Java - it will issue an Exception warning if an unchecked exception might be *explicitly* thrown but is not declared in a throws declaration**
  - **Caution: currently ESC/Java2 will assume that an undeclared unchecked exception will not be thrown, even if it is specified in a signals clause -**
    Declare all unchecked exceptions that might be thrown!
    (e.g. especially when there is no implementation to check).

So this

Exception warning

```
public class Ex {
  public void m(Object o) {
    if (!(o instanceof String)) throw new ClassCastException();
  }
}
```

produces

```
----------------------------------------------------------------------
Ex.java:4: Warning: Possible unexpected exception (Exception)
  }
  ^
Execution trace information:
    Executed then branch in "Ex.java", line 3, col 32.
    Executed throw in "Ex.java", line 3, col 32.

----------------------------------------------------------------------
```

Turn off this warning by

- declaring the exception in a throws clause
- using //@ nowarn Exception; on the offending line
- using a -nowarn Exception command-line option

Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- warnings about possible runtime exceptions: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)
- warnings about possible method specification violations: (Precondition, Postcondition, Modifies)
- non null violations (NonNull, NonNullInit)
- loop and flow specifications (Assert, Reachable, LoopInv, DecreasesBound)
- warnings about possible class specification violations: (Invariant, Constraint, Initially)
- exception problems (Exception)
- **multithreading (Race, RaceAllNull, Deadlock)**
  - **These warnings are caused by potential problems with monitors**
  - **Multithreading problems caused by the absence of any synchronization are not detected.**

Race conditions

- Java defines monitors associated with any object and allows critical sections to be guarded by synchronization statements.
- ESC/Java permits fields to be declared as monitored by one or more objects.
- To read a monitored field, at least one monitor must be held (or a Race warning is issued).
- To write a monitored field, all non-null monitors must be held (or a Race warning is issued).
- To write a monitored field, at least one of its monitors must be non-null (or a RaceAllNull warning is issued).

Race warnings

For example,

```
public class RaceWarning {
  //@ monitored
  int i;

  void m() {
    i = 0; // should have a synchronization guard
  }
}
```

produces

```
----------------------------------------------------------------------
RaceWarning.java:6: Warning: Possible race condition (Race)
    i = 0; // should have a synchronization guard
      ^
Associated declaration is "RaceWarning.java", line 2, col 6:
  //@ monitored
      ^
----------------------------------------------------------------------
```

Deadlocks

- Deadlocks occur when each thread of a group of threads needs monitors held by another thread in the group.
- One way to avoid this is to always acquire monitors in a specific order.
- This requires
  - the user state a (partial) order for monitors (typically using an axiom)
  - that it be clear before acquiring a monitor that the thread does not hold any ‘larger’ monitors (typically a precondition)
- Checking for Deadlock warnings is off by default but can be turned on with **-warn Deadlock**.

Deadlock warnings

For example:

```
public class DeadlockWarning {
  /*@ non_null */ final static Object o = new Object();
  /*@ non_null */ final static Object oo = new Object();

  //@ axiom o < oo;

  //@ requires \max(\lockset) < o;
  public void m() {
      synchronized(o) { synchronized(oo) { }}
  }

  //@ requires \max(\lockset) < o;
  public void mm() {
      synchronized(oo) { synchronized(o) { }} // Deadlock warning
  }
}
```

Types of ESC/Java2 warnings

ESC/Java2 warnings fall into various categories:

- warnings about possible runtime exceptions: (Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore)
- warnings about possible method specification violations: (Precondition, Postcondition, Modifies)
- non null violations (NonNull, NonNullInit)
- loop and flow specifications (Assert, Reachable, LoopInv, DecreasesBound)
- warnings about possible class specification violations: (Invariant, Constraint, Initially)
- exception problems (Exception)
- multithreading (Race, RaceAllNull, Deadlock)
- a few others (OwnerNull, Uninit, Unreadable, Writable)

Other warnings

- **Uninit: used with the uninitialized annotation**
- **OwnerNull: see the ESC/Java User Manual for a description**
- **Unreadable: occurs with the readable_if annotation on shared variables. [JML's change of syntax from readable_if to readable is not complete in ESC/Java2.]**
- **Writable: occurs with the writable_if annotation on shared variables. [JML's change of syntax from writable_if to writable is not complete in ESC/Java2.]**

trace information

For complicated bodies, the warning messages give some information about which if-then-else branches caused the warning:

```
public class Trace {
  //@ ensures \result > 0;
  int m(int i) {
    if (i == 0) return 1;
    if (i == 2) return 0;
    return 4;
  }
}
```

produces

```
Trace.java:8: Warning: Postcondition possibly not established (Post)
  }
  ^

Associated declaration is "Trace.java", line 2, col 6:
  //@ ensures \result > 0;
      ^

Execution trace information:
    Executed else branch in "Trace.java", line 4, col 4.
    Executed then branch in "Trace.java", line 5, col 16.
    Executed return in "Trace.java", line 5, col 16.
```

Counterexamples

- **Sometimes when a specification is found to be invalid, ESC/Java2 will produce a *counterexample context*.**
- **A full context will be produced with the -counterexample option**
- **These are difficult to read, but can give information about the reason for failure.**
- **They state formulae that the prover believes to be true; if there is something you think should not be true, that is a hint about the problem.**

- **Note also: Typically only one warning will be issued in a given run.**

Specification tips and pitfalls

#1: Inherited specifications

- **Base class specifications apply to derived classes**
  - **that is, ESC/Java2 enforces** *behavioral subtyping*
  - **Specs from implemented interfaces also must hold for implementing classes**
- **Be thoughtful about how strict the base class specs should be**
- **Guard them with \typeof(this) == \type(...) if need be**
- **Restrictions on exceptions such as normal_behavior or signals (E e) false; will apply to derived classes as well.**

#1: Inherited specifications

For example, in the code below

- **Parent.m(i) satisfies** $i \geq 0 \Rightarrow \backslash result \geq i$
- **Child.m(i) satisfies** $i \geq 0 \Rightarrow \backslash result \geq i$
  and $i \leq 0 \Rightarrow \backslash result \leq i$
  so Child.m(0) must be 0.

```
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);
}
```

Note: In
Parent p = new Parent();
Parent pc = new Child();
Child c = new Child();
p.m(i); // i must be >= 0
pc.m(i); // i must be >= 0
c.m(i); // i must be >= 0 or <=

Indicates there are inherited specs

#1: Inherited specifications

Another example: two Objects that are == are always also equals. But the converse is not necessarily true. But it is true for objects whose dynamic type is Object.

```
public class Object {
  //@ ensures (this == o) ==> \result;
  /*@ ensures \typeof(this) == \type(Object)
              ==> (\result == (this==o));
  */
  public boolean equals(Object o);
}
```

True for all Objects

Not necessarily true for subtypes

#2: Specifying exceptions

- **A signals clause such as**
  signals (FileNotFoundException e) true;
  states what must be true if an exception of the stated type is thrown.
- **It does not say what other exception types may or may not be thrown.**
- **To forbid a particular exception, omit it from the Java throws clause or write**
  signals (FileNotFoundException e) false;
- **To limit the set of allowed exceptions, use a postcondition such as**

```
/*@ signals (Exception e) e instanceof E1
                          || e instanceof E2
                          || ... ;
  */
```

- **To forbid all exceptions use a normal_behavior or signals (Exception e) false; specification - be careful not to overly restrict derived classes**

#3: Aliasing

A common but non-obvious problem that causes violated invariants is aliasing.

```
public class Alias {
  /*@ non_null */ int[] a = new int[10];
  boolean noneg = true;

  /*@ invariant noneg ==>
                (\forall int i; 0<=i && i < a.length;  a[i]>=0); */

  //@ requires 0<=i && i < a.length;
  public void insert(int i, int v) {
    a[i] = v;
    if (v < 0) noneg = false;
  }
}
```

produces

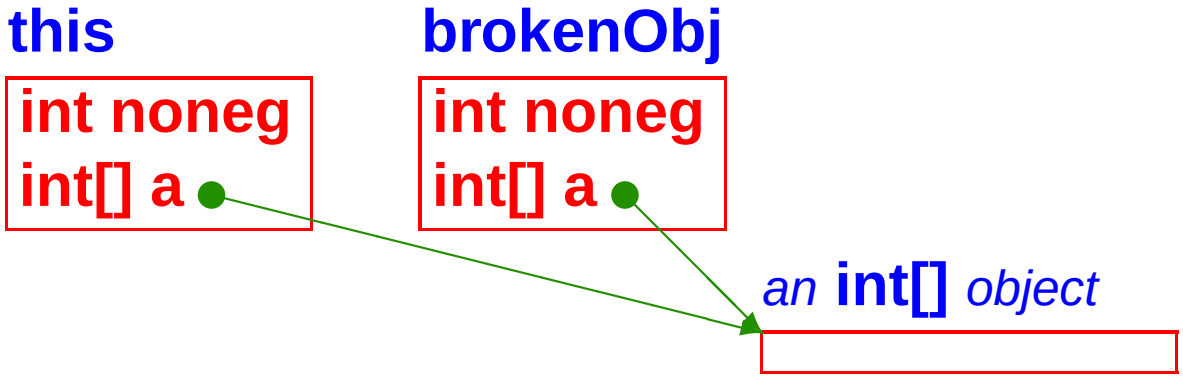
```
Alias.java:12: Warning: Possible violation of object invariant (Invariant)
  }
  ^
Associated declaration is "Alias.java", line 5, col 6:
  /*@ invariant (\forall int i; 0<=i && i < a.length;
```

#3: Aliasing

A full counterexample context (-counterexample option) produces, among lots of other information:

```
brokenObj%0 != this
(brokenObj%0).(a@pre:2.24) == tmp0!a:10.4
this.(a@pre:2.24) == tmp0!a:10.4
```

that is, this and some different object (brokenObj) share the same a object.



#3: Aliasing

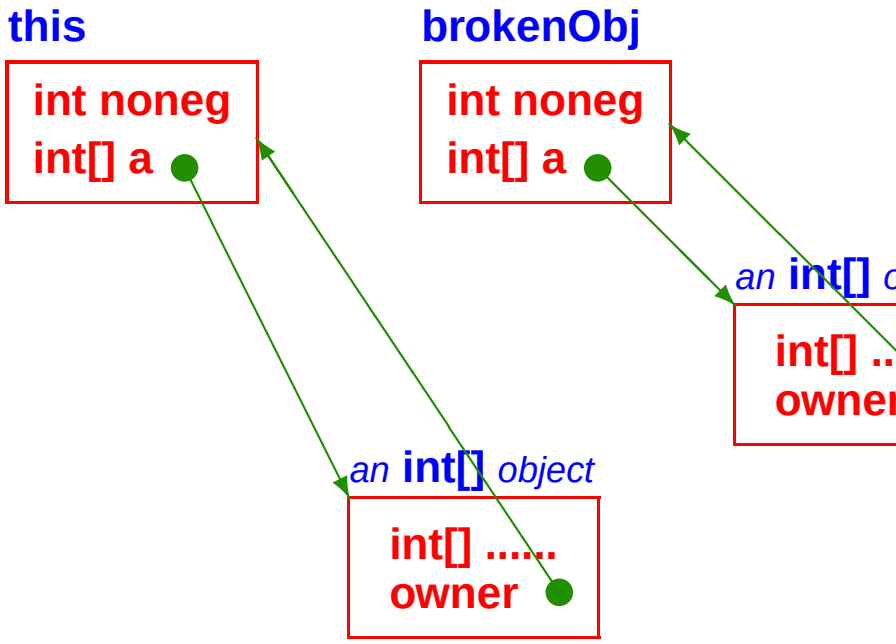
To fix this, declare that a is owned only by its parent object: (owner is a ghost field of java.lang.Object)

```
public class Alias {
  /*@ non_null */ int[] a = new int[10];
  boolean noneg = true;

  /*@ invariant (\forall int i; 0<=i && i < a.length;
                          noneg ==> (a[i]>=0)); */
  //@ invariant a.owner == this;

  //@ requires 0<=i && i < a.length;
  public void insert(int i, int v) {
    a[i] = v;
    if (v < 0) noneg = false;
  }

  public Alias() {
    //@ set a.owner = this;
  }
}
```



#3: Aliasing

Another example. This one fails on the postcondition.

```
public class Alias2 {
  /*@ non_null */ Inner n = new Inner();
  /*@ non_null */ Inner nn = new Inner();
  //@ invariant n.owner == this;
  //@ invariant nn.owner == this;

  //@ ensures n.i == \old(n.i + 1);
  public void add() {
    n.i++;
    nn.i++;
  }

  Alias2();
}

class Inner {
  public int i;
  //@ ensures i == 0;
  Inner();
}
```

#3: Aliasing

- **The counterexample context shows**

```
this.(nn:3.24) == tmp0!n:10.4
tmp2!nn:11.4 == tmp0!n:10.4
```

- **These hint that n and nn are references to the same object.**
- **If we add the invariant //@ invariant n != nn; to forbid aliasing between these two fields, then all is well.**

#3: Aliasing

- Aliasing is a serious difficulty in verification
- Handling aliasing is an active area of research, related to handling frame conditions
- It is all about knowing what is modified and what is not
- These owner fields or the equivalent create a form of encapsulation that can be checked by ESC/Java to control what might be modified by a given operation

#4: Write object invariants

- **Be sure that class invariants are about the object at hand.**
- **Statements about all objects of a class may indeed be true, but they are difficult to prove, especially for automated provers.**
- **For example, if a predicate P is supposed to hold for objects of type T, then do not write**

```
//@ invariant (\forall T t; P(t));
```

- **Instead, write**

```
//@ invariant P(this);
```

- **The latter will make a more provable postcondition at the end of a constructor.**

#5: Inconsistent assumptions

If you have inconsistent specifications you can prove anything:

```
public class Inconsistent {
  public void m() {
    int a,b,c,d;
    //@ assume a == b;
    //@ assume b == c;
    //@ assume a != c;
    //@ assert a == d; // Passes, but inconsistent
    //@ assert false;  // Passes, but inconsistent
  }
}
```

#5: Inconsistent assumptions

Another example:

```
public class Inconsistent2 {
  public int a,b,c,d;
  //@ invariant a == b;
  //@ invariant b == c;
  //@ invariant a != c;

  public void m() {
    //@ assert a == d; // Passes, but inconsistent
    //@ assert false;  // Passes, but inconsistent
  }
}
```

We hope to put in checks for this someday!

#6: Exposed references

Problems can arise when a reference to an internal object is exported from a class:

```
public class Exposed {
    /*@ non_null */ private int[] a = new int[10];
    //@ invariant a.length > 0 && a[0] >= 0;

    //@ ensures \result != null;
    //@ ensures \result.length > 0;
    //@ pure
    public int[] getArray() { return a; }
}
class X {
    void m(/*@ non_null */ Exposed e) {
      e.getArray()[0] = -1; // unchecked invariant violation
    }
}
```

- **ESC/Java does not check that *every* allocated object still satisfies its invariants.**
- **Similar hidden problems can result if public fields are modified directly.**

#7: \old

\old is used to indicate evaluation in the pre-state in a postcondition expression.

Consider specifying

```
public static native void arraycopy(Object[] src, int srcPos,
                                    Object[] dest, int destPos, int length
```

Try:

```
ensures (\forall int i; 0<=i && i<length; dest[destPos+i] == src[srcPos+i]
```

#7: \old

\old is used to indicate evaluation in the pre-state in a postcondition expression.

Consider specifying

```
public static native void arraycopy(Object[] src, int srcPos,
                                    Object[] dest, int destPos, int length
```

Try:

```
ensures (\forall int i; 0<=i && i<length; dest[destPos+i] == src[srcPos+i]
```

Wrong!

#7: \old

\old is used to indicate evaluation in the pre-state in a postcondition expression.

Consider specifying

```
public static native void arraycopy(Object[] src, int srcPos,
                                    Object[] dest, int destPos, int length
```

Try:

```
ensures (\forall int i; 0<=i && i<length; dest[destPos+i] == src[srcPos+i]
```

Wrong!

Besides exceptions and invalid arguments, don't forget aliasing - dest and src may be the same array:

```
ensures (\forall int i; 0<=i && i<length;
                            dest[destPos+i] == \old(src[srcPos+i]);
```

#7: \old

\old is used to indicate evaluation in the pre-state in a postcondition expression.

Consider specifying

```
public static native void arraycopy(Object[] src, int srcPos,
                                    Object[] dest, int destPos, int length
```

Try:

```
ensures (\forall int i; 0<=i && i<length; dest[destPos+i] == src[srcPos+i]
```

Wrong!

Besides exceptions and invalid arguments, don't forget aliasing - dest and src may be the same array:

```
ensures (\forall int i; 0<=i && i<length;
                              dest[destPos+i] == \old(src[srcPos+i]);
```

And don't forget the other elements:

```
ensures (\forall int i; (0<=i && i<destPos) ||
                                (destPos+length <= i && i < destPos.length);
                              dest[i] == \old(dest[i]);
```

Getting started

- Start with foundation and library routines
- For each field: should it be non_null?
- For each reference field: should an owner field be set for it?
- For each method: should it be pure? Should the arguments or the result be non_null?
- For each class: what invariant expresses the self-consistency of the internal data?
- Add pre- and post-conditions to limit the inputs and outputs of each method.
- Add possible unchecked exceptions to throws clauses.
- Start with simple specifications; proceed to complex ones as they have value.

- **Separate conjunctions to get information about which conjunct is violated. Use**

```
requires A;
requires B;
```

not

```
requires A && B;
```

- **Use assert statements to find out what is going wrong. Use assume statements** *that you KNOW are correct* **to help the prover along.**

- **Specification is tricky - getting it right is hard, even with tools**
- **Try it - a substantial research gap is experience on industrial-scale sets of code**
- **Communicate - we are willing to offer advice**
- **Share your experience - tools will get better and we will all learn better techniques for successful specification (use JML and ESC/Java mailing lists)**