Specification tips and pitfalls

David Cok, Joe Kiniry, and Erik Poll

Eastman Kodak Company, University College Dublin,
and Radboud University Nijmegen

Specifications tips and pitfalls

1. Inherited specifications
2. Aliasing
3. Object invariants
4. Inconsistent assumptions
5. Exposed references
6. \old
7. How to write specs

#1: Specification inheritance and behavioural subtyping

Behavioural subtyping

Suppose `Child extends Parent`.

- Behavioural subtyping = objects from subclass `Child` "behave like" objects from superclass `Parent`
- Principle of substitutivity [Liskov]:
  code will behave "as expected" if we provide an `Child` object where a `Parent` object was expected.

Behavioural subtyping

Behavioural subtyping usually enforced by insisting that

- invariant in subclass is stronger than invariant in superclass
- for every method,
  - precondition in subclass is weaker (!) than precondition is superclass
  - postcondition in subclass is stronger than postcondition is superclass

JML achieves behavioural subtyping by specification inheritance: any child class inherits the specification of its parent.

Specification inheritance for invariants

Invariants are inherited in subclasses. Eg.

```
class Parent {
    ...
    //@ invariant invParent;
    ...  }

 class Child extends Parent {
    ...
    //@ invariant invChild;
    ...  }
```

the invariant for `Child` is `invChild && invParent`

Specification inheritance for method specs

```
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){ ... }
  }
```

Keyword `also` indicates there are inherited specs.

Specification inheritance for method specs

Method `m` in `Child` also has to meet the spec given in `Parent` class. So the complete spec for `Child` is

```
class Child extends Parent {

 /*@   requires i >= 0;
   @   ensures  \result >= i;
   @ also
   @   requires i <= 0
   @   ensures  \result <= i;
   @*/
 int m(int i){ ... }
}
```

What can result of `m(0)` be?

Specification inheritance for method specs

This spec for `Child` is equivalent with

```
class Child extends Parent {

  /*@   requires i <= 0 || i >= 0;
    @   ensures  \old(i) >= 0 ==> \result >= i;
    @   ensures  \old(i) <= 0 ==> \result <= i;
    @*/
  int m(int i){ ... }
}
```

Inherited specifications: trick

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

Inherited specifications

So

- **Base class specifications apply to subclasses**
  - **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.**

#2: Aliasing

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

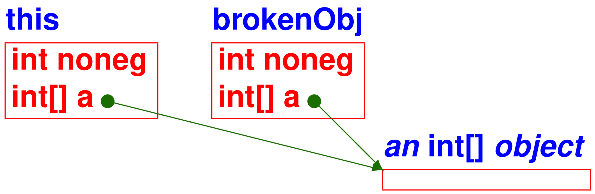
```
Alias.java:12: Warning: Possible violation of object invariant (Invariant)
  }
  ^
Associated declaration is "Alias.java", line 5, col 6:
  /*@ invariant (\forall int i; 0<=i && i < a.length;
```

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.



this

int noneg
int[] a

brokenObj

int noneg
int[] a

an int[] *object*

Aliasing

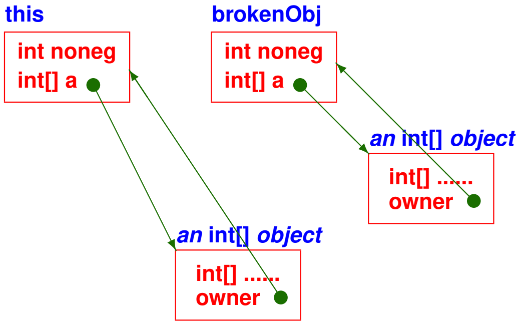
To fix this, declare that a is owned only by its parent object: (owner is a ghost field of java.lang.Object)

```
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;
  }
}
```



this

int noneg
int[] a

brokenObj

int noneg
int[] a

an int[] *object*

int[] ......
owner

an int[] *object*

int[] ......
owner

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();
}
```

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.

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
- There is a proposal to add so-called universes to JML to provide a more advanced form of alias control.

#3: 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.

#4: 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
  }
}
```

#4: 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!

#5: 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.**

#6: \old

\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 l
```

Try:

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

\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!

\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 l
```

Try:

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

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]);
```

\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]);
```

\old

In postcondition

```
ensures (\forall int i; 0<=i && i<length;
                            dest[destPos+i] == \old(src[srcPos+i]);
public static native void arraycopy(Object[] src, int srcPos,
                                    Object[] dest, int destPos, int l
```

shouldn’t we write `\old(length)` instead of `length`?

\old

In postcondition

```
ensures (\forall int i; 0<=i && i<length;
                             dest[destPos+i] == \old(src[srcPos+i]));
public static native void arraycopy(Object[] src, int srcPos,
                                    Object[] dest, int destPos, int length);
```

shouldn't we write `\old(length)` instead of `length`?
And `\old(dest)[...]` instead of `dest[...]`?

\old

In postcondition

```
ensures (\forall int i; 0<=i && i<length;
                             dest[destPos+i] == \old(src[srcPos+i]));
public static native void arraycopy(Object[] src, int srcPos,
                                    Object[] dest, int destPos, int l
```

shouldn't we write `\old(length)` instead of `length`?
And `\old(dest)[...]` instead of `dest[destPos+i]`?

Strictly speaking: yes. But because this is so easy to get forget, any mention of an argument x in postcondition means `\old(x)`.

\old

In postcondition

```
ensures (\forall int i; 0<=i && i<length;
                             dest[destPos+i] == \old(src[srcPos+i]));
public static native void arraycopy(Object[] src, int srcPos,
                                    Object[] dest, int destPos, int length);
```

shouldn't we write `\old(length)` instead of `length`?
And `\old(dest)[...]` instead of `dest[destPos+i]`?

Strictly speaking: yes. But because this is so easy to get forget, any mention of an argument x in postcondition means `\old(x)`.

This means it's impossible to refer to the new value of `length` in postcondition of `arraycopy`. But this value is unobservable for clients anyway.

#7: How to write specs

Getting started

- Start with foundation and library routines

Getting started

- Start with foundation and library routines
- For each field: is there an invariant for this field?

Getting started

- Start with foundation and library routines
- For each field: is there an invariant for this field?
- For each reference field: should it be non_null?

Getting started

- Start with foundation and library routines
- For each field: is there an invariant for this field?
- For each reference field: should it be non_null?
- For each reference field: should an owner field be set for it?

Getting started

- Start with foundation and library routines
- For each field: is there an invariant for this field?
- For each reference 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?

Getting started

- Start with foundation and library routines
- For each field: is there an invariant for this field?
- For each reference 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?

Getting started

- Start with foundation and library routines
- For each field: is there an invariant for this field?
- For each reference 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.

Getting started

- Start with foundation and library routines
- For each field: is there an invariant for this field?
- For each reference 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.

Getting started

- Start with foundation and library routines
- For each field: is there an invariant for this field?
- For each reference 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.

Getting started

- 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.

Finally

- 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)