Reasoning with specifications containing method calls

David R. Cok
Eastman Kodak R&D Laboratories
FTfJP – June 2004

Outline

- Background
    - on Abstraction in specification
    - on JML
    - on ESC/Java & ESC/Java2
    - on Simplify
- Implementing method calls
- Exceptional behavior in annotations
- Other applications

Abstraction in specifications

- Using (pure) methods in specs
- Model fields
- Model classes
  - e.g. JML's mathematical classes

Advantages

- Abbreviation (readability)
- Simplifies mental models (e.g. can make use of functions on mathematical constructs)
- Allows specification in terms of abstract concepts instead of (or in the absence of) concrete implementations
- Inheritance
- Simplifies automated reasoning

Java Modeling Language

- JML is
  - a specification language (a BISL)
  - for Java
  - uses Java-like syntax and semantics
  - embeds annotations in formatted Java comments (either in a source file or in a specification file)

Examples of JML

```
class C {

//@ requires i != 0;
//@ ensures i < 0 ==> \result > 0;
//@ signals (Exception e) i == 0;
//@ diverges i > 1000000000;
public int m(int i ) {
    ...
    //@ assert i < 10;
     ...
}

}
```

precondition

(calling method is required to satisfy this pre-state condition; implementation may presume it)

normal postcondition

(If method terminates normally, then this post-state expression is true)

exceptional postcondition

(If method throws exception of the given type, then the post-state expression must be true)

non-termination

(if method does not terminate, then this pre-state expression is true)

Specially formatted comment

In-body logical assertion

ESC/Java

- A static analysis tool that
  - efficiently checks for bugs in low-level code constructs (e.g. NullPointerException) by applying a (hidden) prover to generated verification conditions
  - had reasonably good performance
  - annotation language close to a subset of JML
  - no manual proving required
- But
  - no abstraction
  - not consistent with JML
  - not maintained

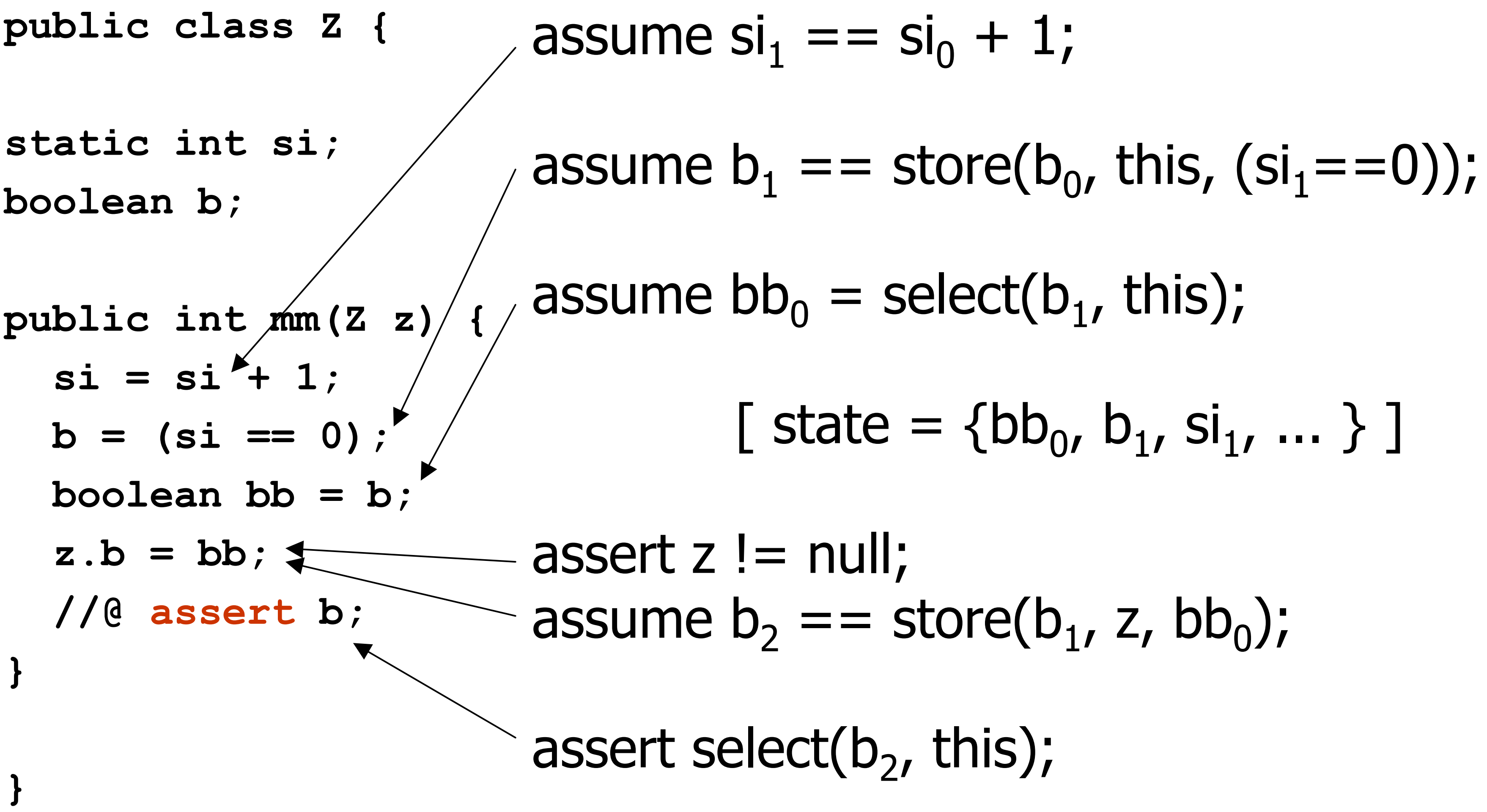
ESC/Java2

- Project begun by Cok & Kiniry to evolve ESC/Java
  - bring ESC/Java to Java 1.4
  - bring ESC/Java to current JML
  - extend the set of checked constructs, while maintaining the original design philosophy
  - improve the overall packaging as needed
  - provide some ongoing support
- Enable evalution of this style of verification on sets of Java code with more extensive and abstract specifications

Simplify

- ESC/Java(2) uses a back-end prover named Simplify
- It accepts expressions in an untyped first-order logic with quantifiers
- Decides validity, invalidity, sometimes produces counterexamples, sometimes runs out of resources
- Has built-in knowledge of term equality, simple arithmetic (using Simplex algorithm), + axioms for arrays, type relationships
- fully automatic (no access for manual intervention)

Translation – implicit state



```
public class Z {

static int si;
boolean b;

public int mm(Z z) {
   si = si + 1;
   b = (si == 0);
   boolean bb = b;
   z.b = bb;
   //@ assert b;
}

}
```

assume $si_1 == si_0 + 1$;

assume $b_1 == store(b_0, this, (si_1==0))$;

assume $bb_0 = select(b_1, this)$;

[state = $\{bb_0, b_1, si_1, \ldots\}$]

assert z != null;

assume $b_2 == store(b_1, z, bb_0)$;

assert $select(b_2, this)$;

-Instance fields are represented as arrays indexed by object ids.

-The ‘state’ is the set of current variables.

Translation – explicit state

```
public class Z {

static int si;
boolean b;

public int mm(Z z) {
  si = si + 1;
  b = (si == 0);
  boolean bb = b;
  z.b = bb;
  //@ assert b;
}

}
```

assume state_1 ==
store(state_0,si,select(state_0,si) + 1);

assume state_2 == store(state_1,b,this,
(select(state_1,si) == 0));

…

-arrays representing field values now have an additional dimension
-using arrays builds in the axioms about the values of fields that do not change

Translation – explicit state II

```
public class Z {

static int si;
boolean b;

public int mm(Z z) {
  si = si + 1;
  b = (si == 0);
  boolean bb = b;
  z.b = bb;
  //@ assert b;
}

}
```

assume $si(state_1) == si(state_0) + 1$;

assume $b(state_2, this) ==$
$(si(state_1) == 0)$;

assume $bb(state_2) == b(state_2)$;

assume $b(state_3, z) == bb(state_2)$;

assert $b(state_3, this)$;

-fields are functions on a state variable and object ids
-What about $b(state_2, x)$ for x != this? *Needs an axiom*
-What about $f(state_2, x)$ for a different field f?

Translation of method calls

- Bad choices:
  - inlining the specification
    - e.g. if the spec is ensures \result == ...;
    - Not always a suitable expression to inline
    - Might be more than one
    - May be recursive calls
    - Can get huge verification conditions
  - inlining the implementation
    - There may not be an implementation
    - There may be recursive calls
    - Messy – mixing logical with imperative statements
    - Loses benefits of abstraction
- For some methods (e.g. getters and setters), inlining might be a good optimization

Translation of method calls

- Convert each method call into a function term with appropriate arguments.
- Use a state argument to distinguish calls in different state contexts.
- Include the specifications of the method as assumptions (in the appropriate state context).

Example

```
//@ pure
public boolean m(M o);

static public M make(int i);

//@ requires o != null;
//@ requires m(o);
//@ ensures m(o);
public int mm(M o) {
  //@ assert m(o);
  o.i = 1;
  //@ assert m(o);
  o = make(0);
  //@ assert m(o);
}
```

assume ZZ.m(state$_0$,this,o$_0$);

assert ZZ.m(state$_0$,this,o$_0$);

assume i$_1$ == store(i$_0$,o$_0$,1);
assert ZZ.m(state$_1$,this,o$_0$);

assume o$_1$ == ...;
assert ZZ.m(state$_2$,this,o$_1$);

assert ZZ.m(state$_0$,this,o$_0$) ==>
ZZ.m(state$_2$,this,o$_0$);

Example

```
//@ pure
public boolean m(M o);

static public M make(int i);

static public M o;

//@ requires o != null;
//@ requires m(o);
//@ ensures m(o);
public int mm() {
  //@ assert m(o);
  o.i = 1;
  //@ assert m(o);
  o = make(0);
  //@ assert m(o);
}
```

assume ZZ.m($state_0$,this,o_0);

assert ZZ.m($state_0$,this,o_0);

assume i_1 == store(i_0,o_0,1);
assert ZZ.m($state_1$,this,o_0);

assume o_1 == ...;
assert ZZ.m($state_2$,this,o_1);

assert ZZ.m($state_0$,this,o_0) ==>
ZZ.m($state_2$,this,o_1);

Example – adding specs

```
//@ ensures \result ==
                  (o.i==0);
//@ pure
public boolean m(M o);

static public M make(int i);

//@ requires o != null;
//@ requires m(o);
//@ ensures m(o);
public int mm(M o) {
  //@ assert m(o);
  o.i = 1;
  //@ assert m(o);
  o = make(0);
  //@ assert m(o);
}
```

assume (forall t,o; ZZ.m(state$_0$,t,o) == (i$_0$[o] == 0));
assume ZZ.m(state$_0$,this,o$_0$);

assume (forall t,o; ZZ.m(state$_0$,t,o) == (i$_0$[o] == 0));
assert ZZ.m(state$_0$,this,o$_0$); // OK

assume i$_1$ == store(i$_0$,o$_0$,1);
assume (forall t,o; ZZ.m(state$_1$,t,o) == (i$_1$[o] == 0));
assert ZZ.m(state$_1$,this,o$_0$); // FAILS

assume o$_1$ == ...;
assume (forall t,o; ZZ.m(state$_2$,t,o) == (i$_1$[o] == 0));
assert ZZ.m(state$_2$,this,o$_1$); // DEPENDS

assume (forall t,o; ZZ.m(state$_2$,t,o) == (i$_1$[o] == 0));
assert ZZ.m(state$_0$,this,o$_0$) ==>
    ZZ.m(state$_2$,this,o$_0$); // FAILS

Example – Java vs. spec

```
//@ pure
public boolean m(M o);

public int mm(M o) {
   ...
   b = m(o);
   //@ assert b == m(o);
   ...
}
```

assume b_1 == store(b_0,this,RES);

assert b_1 == ZZ.m(state$_1$,this,o$_0$);

No logical connection between these values that enables the assertion to be proved!

Need a connection between m in the code and m in the assertion.

Example – Java vs. spec

```
//@ pure
public boolean m(M o);

public int mm(M o) {
   ...
   b = m(o);
   //@ assert b == m(o);
   ...
}
```

assume RES == ZZ.m($state_0$,this,o_0);

assume b_1 == store(b_0,this,RES);

assert b_1 == ZZ.m($state_1$,this,o_0);

Need to add an assumption when an annotation method is used in the source code.

But still cannot prove the assertion because the state has changed.

Example – Java vs. spec

```
//@ ensures \result ==
                 (o.i==0);
//@ pure
public boolean m(M o);

public int mm(M o) {
   ...
   b = m(o);
   //@ assert b == m(o);
   ...
}
```

assume (forall t,o; ZZ.m(state$_0$,t,o)
== (i$_0$[o] == 0));
assume RES == ZZ.m(state$_0$,this,o$_0$);
assume b$_1$ == store(b$_0$,this,RES);

assume (forall t,o; ZZ.m(state$_1$,t,o)
== (i$_0$[o] == 0));
assert b$_1$ == ZZ.m(state$_1$,this,o$_0$);

Now the assertion is provable.

Example – Java vs. spec

```
//@ pure
public boolean m(M o);

public int mm(M o) {
   ...
    if (b == m(o)) {
   //@ assert b == m(o);
   ...
   }
}
```

// In the then branch…

assume b_0 == ZZ.m(state$_0$,this, o$_0$);

assert b_0 == ZZ.m(state$_0$,this,o$_0$);

Without a state change the assertion is trivially provable, even without a specification.

Example – explicit state

```
//@ ensures \result ==
              (o.i==0);
//@ pure
public boolean m(M o);
public boolean b;

public int mm(M o) {
   ...
   b = m(o);
  //@ assert b == m(o);
   ...
}
```

assume (forall s,t,o; ZZ.m(s,t,o)
      == (select(s,i,o) == 0));

assume $state_1$ == store($state_0$, b, this,
      ZZ.m($state_0$,this,o_0);

assert select($state_1$,b,this) ==
      ZZ.m($state_1$,this, o_0);

-using explicit state reduces the number of introduced assumptions for ZZ.m

Implicit vs. Explicit

- Using explicit state
  - allows more compact representation of method calls
  - complicates reasoning about field access by introducing a new array dimension/function argument
- It would be useful to understand the trade-off experimentally

Exceptional behavior

```
//@ ensures P;
//@ pure
public boolean m(M o);

//@ ensures Q;
public int mm(M o) {
   b = m(o);
  //@ assert b == m(o);
}
```

If m terminates normally, then P holds. Nothing known if m terminates exceptionally.

If mm terminates normally, then Q holds.

Exceptional behavior

```
//@ ensures P;
//@ pure
public boolean m(M o);

//@ ensures m(o);
public int mm(M o) {
   ...;
}
```

If m terminates normally, then P holds.

What if m terminates with an exception in the postcondition?

JML semantics say the result is undefined (more specifically, an arbitrary value). [Spec# says the postcondition fails.]

We can only conclude that if (mm terminates normally AND the various assertions terminate normally) then mm satisfies its specification. *Pretty Weak!*

Exceptional behavior

```
//@ ensures P;
//@ signals (Exception) false;
//@ diverges false;
//@ pure
public boolean m(M o);

//@ ensures m(o);
public int mm(M o) {
   ...;
}
```

For a method that is used in an annotation, we need a spec that guarantees normal termination (under the relevant preconditions)

- This is stronger than most specs are written.

- This puts a significant burden on overriding methods.

- If we presume this behavior, then the default behavior in an annotation is different than the default behavior in code (and a problem for runtime checking)

Other applications

- Pure constructors
- Array constructors
- Model variables
- Quantified expressions
  - No specs – what about exceptional behavior ?
  - We lose guarding conditionals

Immutable values

- Figuring out what does and does not change is a big part of a verifier’s task.
- Knowing which types and values are ***immutable*** could assist reasoning: these objects remain equal despite state changes.
- Requires purity, immutable internal objects, limits on rep exposure, a way to check for immutability, ...

Conclusions

- We have successfully implemented the use of methods in annotations in ESC/Java2.
- Methods used in annotations should preclude exceptional behavior – which puts a burden on specification writers and on derived classes.
- The same techniques can be used for other specification constructs.

For discussion…

- The choice of logical representation is not obvious and needs some comparative work.
- Will a concept of immutability assist in verification?

Translation – explicit state II

```
public class Z {

static int si;
boolean b;

public int mm(Z z) {
  si = si + 1;
  b = (si == 0);
  boolean bb = b;
  z.b = bb;
  //@ assert b;
}

}
```

assume H(si,state_1) == H(si,state_0) + 1;
assume (forall f; f != si ==>
H(f,state_1) == H(f, state_0));

assume H(b,state_2,this) ==
(H(si,state_1) == 0);
assume (forall f,o;
(f != b || o != this) ==>
H(f,state_2,o) == H(f, state_1,o));

Example – explicit state

```
//@ ensures \result ==
                (o.i==0);
//@ pure
public boolean m(M o);

public int mm(M o) {
   ...
   b = m(o);
   //@ assert b == m(o);
   ...
}
```

assume (forall s,t,o; ZZ.m(s,t,o) == (i(s,t,o) == 0));

assume $b(state_1,this) == ZZ.m(state_0,this,o_0)$;

assert $b(state_1,this) == ZZ.m(state_1,this,o_0)$;

-using explicit state reduces the assumptions for ZZ.m

-but requires a number of other assumptions on each assignment noted earlier