Observational Purity in JML

David R. Cok, Eastman Kodak
Gary Leavens, UCF
9 November 2008
SAVCBS08 workshop

Kodak

Method calls in specifications

```
class List<E> {

   //@ ensures last( ).equals(element);
   void add(/*@ non_null*/ E element);

   //@ pure
   //@ non_null
   E last( );

}
```

Runtime checking:
How do we know last( ) and equals( ) do not change something?

Static checking:
What does a method call mean in a specification?

Basic questions

Under what conditions will using a method call in an assertion not affect the execution of a program in a way that invalidates its correctness?

- in runtime checking, will evaluating an assertion change the behavior of the program at all?

- what semantics should be used for method calls in assertions in static checking?

Methods may have side effects

- **fields written to**
  - **values computed and cached, singleton objects**
- **elapsed time**
- **garbage collection**
- **stack space consumed and released**
- **new objects allocated**
- **monitors locked**
- **log files (or the standard output stream) written to**
- **file system changes**

We would like to ignore side effects in specifications if we 'know' that the program does not depend on them

Kodak

Purity

- Strong purity
  - time, stack changes, garbage collection
  - in practice: file system changes, output
- Weak purity (JML’s @Pure)
  - allocation and modification of new objects
- Observational purity
  - modifying fields that are ‘secret’

[locking ignored for now – single threaded JML]

Importance

- Object.equals is used ubiquitously in specifications; implementations in subclasses are not pure – some use caching
- Plenty of examples in user code
- No practical solution implemented as yet

Classic example – a cache

```
class Cache {
   //@ public invariant isCached -> (cachedValue == expensive( ) );
   //@ public JMLDatagroup value;
   private boolean isCached = false; //@ in value;
   private int cachedValue; //@ in value;

   //@ modifies value;
   //@ ensures \result == expensive( );
   public int value( ) {
      if (!isCached) {
         cachedValue = expensive( );
         isCached = true;
      }
      return cachedValue;
   }

   boolean isCached( ) { return isCached; }

   public int expensive( ) { ... }
}
```


Kodak

Other examples

- **caching in a shared database**
- **reading from a structure (e.g. hash table) that reorganizes itself for better performance**

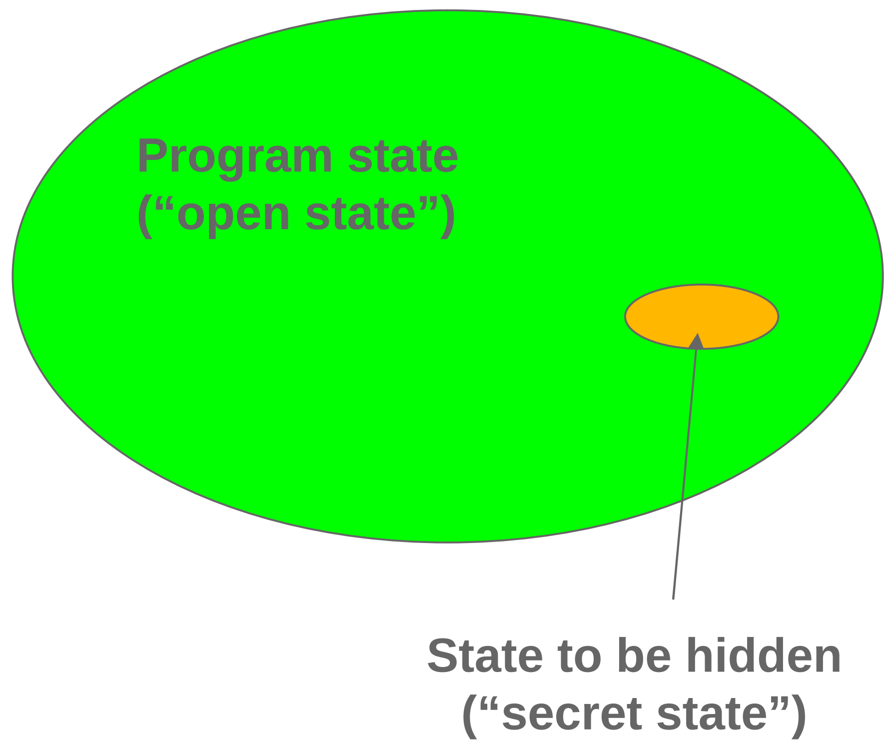
Previous theoretical work

- Problem has been noted and discussed informally

- Theoretical treatment in
  - D. A. Naumann, Observational Purity and Encapsulation, Theoretical Computer Science, 2007
  - Barnett, Naumann, Schulte, Sun. Allowing state changes in specifications, ETRICS, 2006.

Summary

- **Allow a portion of the program state to be modified in assertions – but then not accessed by the rest of the program**



Proof idea

- Proof is carried out by simulation:
  - showing that assert Q
    is equivalent to skip
    even if Q contains query calls
- Requires
  - that open methods are restricted in accessing secret state
  - that query methods, which can access secret state, may not use query methods in specs
  - that the values returned by query methods could be calculated from open state

- **Cannot mix access to secret fields and calling of query methods**

- **Query method might modify secret fields in unknown ways**

```
@Query
int m() {

  ... isCached ...
   //@ assert value() == 0;
   ... isCached ...


}
```


Kodak

Summary – current theory

Java:

Open methods

- read/write open state
- call open methods
- call query methods
- NOT read/write hidden state

Query methods

- read(only) open state
- call pure methods
- call query methods
- read/write hidden state
- query methods must maintain hidden state invariants

JML (assertions):

Open methods

- read open state
- call pure open methods
- call query methods
- NOT read/write hidden state

Query methods

- read open state
- call pure open methods
- NOT call query methods
- may read hidden state
- since method specs are visible to open methods, they do not reference secret state

Practical Issues

- Encapsulation boundary is a class
- Real programs have multiple independent pieces of secret state
- Not calling query methods within assertions in query methods is too restrictive (e.g. in the specifications of query methods)
- No semantics for static checking is defined
- Need methods to manipulate secret state

Encapsulation boundary

- Straightforward to use a smaller unit than class
- We use JML datagroups and a @Secret annotation to define content of secret state
- Datagroups enable the secret state to be open to extension in subclasses
- Associating secret state with datagroups allows distinguishing multiple subsets of the secret state

Kodak

Encapsulating secret state

- Group secret fields using a datagroup
- Associate query methods with a secret datagroup

```
class X {
	@Secret private JMLDatagroup cacheGroup;
	@Secret private boolean isCached; //@ in cacheGroup;
	@Secret private int cachedValue; //@ in cacheGroup;

	@Query(“cacheGroup”)
	public int value() {
		if (!isCached) {
			cachedValue = expensive();
			isCached = true;
		}
		return cachedValue;
	}
}
```

Encapsulating secret state - defaults

```
class X {
    //////  @Secret protected JMLDatagroup value; - implicitly defined
    @Secret private boolean isCached; //@ in value;
    @Secret private int cachedValue; //@ in value;

    @Query
    public int value() {
            if (!isCached) {
                    cachedValue = expensive();
                    isCached = true;
            }
            return cachedValue;
    }
}
```


Kodak

Encapsulating secret state - defaults

```
class Object {
      //////  @Secret protected JMLDatagroup equals; - implicitly defined

      @Query
      public boolean equals(Object o);
}
```

Do need to plan ahead: in super classes, methods which might not be pure but are wanted to be used in assertions must be declared @Query

Kodak

Secret methods

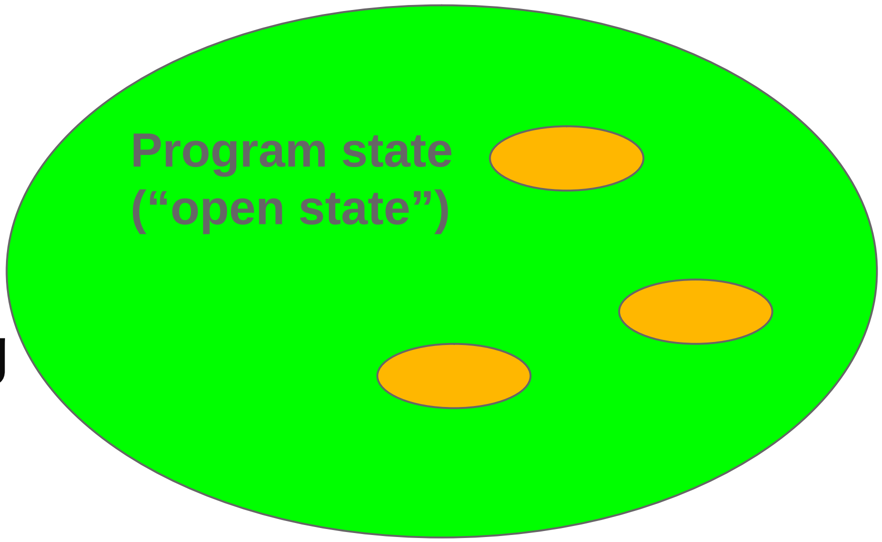
- Also does not invalidate theory/proofs to have secret methods:
  - may manipulate secret state
  - never called by open methods
  - may be used as helpers by query methods
  - conceptually private

Multiple pieces of secret state

- Can treat each piece of secret state independently

- Datagroups allow naming and identifying each piece

- But, need to be sure that the various query methods do not interfere with each other



Kodak

Multiple pieces of secret state – use restrictions

- Could treat the union of all of the pieces as one glob of secret state:
  - would restrict assertions in one query method from calling query methods for unrelated secret state
- Better to treat them as distinct – so long as the pieces of secret state are disjoint
- KEY INGREDIENT: associate secret state with object instances, not with classes

Interference more closely

- The presence of the assert statement alters the subsequent control flow in runtime checking.

- What semantics should we use for static checking?

```
//@ invariant isCached ->
  (cachedValue == expensive( ));

//@ ensures isCached; //??????
//@ ensures \result == expensive( );
public int value( ) {
  ...
  //@ assert value2( ) == 0;
  if (!isCached) {
       cachedValue = expensive( );
       isCached = true;
  } else {
       ...
  }
}
```

Static checking

- Only assume the invariant remains valid.
- No further assumptions
  - corresponds to a weakly pure semantics
  - soundly approximates (via underspecification) the runtime semantics
- Static check checks all permissible runtime paths

```
//@ invariant isCached ->
  (cachedValue == expensive( ));

//@ ensures isCached; //? NO
//@ ensures \result == expensive( );
public int value( ) {
  ...
  //@ assert value2( ) == 0;
  if (!isCached) {
      cachedValue = expensive( );
      isCached = true;
  } else {
      ...
  }
}
```

Summary

Java:

Open methods
- read/write open state
- call open methods
- call query methods
- NOT read/write hidden state directly

Query methods
- read(only) open state
- call pure methods
- call query methods
- read/write own hidden state
- NOT read/write other hidden state
- query methods must maintain hidden state invariants

JML (assertions):

Open methods
- read open state
- call pure open methods
- call query methods
- NOT read/write hidden state directly

Query methods
- read open state
- call pure open methods
- call query methods, but these calls 'havoc' the secret state
- may read/write own hidden state directly, but not for other datagroups

Summary - caveat

Using query methods in specs or assertions within query methods:

– query methods may be used in method specs and in in-line assertions
– do affect the runtime control flow
– in static checking:
  » Do not allow pre- and post-conditions to depend on secret state (other than invariant)
  » equivalent to loss of knowledge (a havoc) about secret state, other than invariant
  » soundly approximates the runtime behavior
  » would be helpful to compartmentalize query methods for different secret state

Kodak

Issue – frame conditions

The issue of interference has an analogy in frame conditions:

- What frame condition should be used for a query method?

```
//@ assignable value; // for the appropriate datagroup
@Query
public int value( ) { ... }
```

- But what about callers of value()?
  - datagroup abstracts the implementation
  - does every caller have to list the secret datagroups of every query method (recursively) that it calls???

Kodak

```
class X {
    //////  @Secret protected JMLDatagroup value; - implicitly defined
    @Secret private boolean isCached; //@ in value;
    @Secret private int cachedValue; //@ in value;

    @Query
    //@ assignable value; // implicitly defined?
    public int value() {
        if (!isCached) {
            cachedValue = expensive();
            isCached = true;
        }
        return cachedValue;
    }
}
```


Kodak

Issue – frame conditions

- Suppose we allow omitting references to secret state in frame conditions?
  - then any query method call might change any secret state (including your own)
  - workable for disjoint bits of secret state
  - unclear whether this is workable for nested, hierarchical information hiding

Conclusions

- **Integration of an initial design for observational purity in JML**
- **Extension to accommodate multiple disjoint islands of secret state, inheritance, invariants and frame conditions**
- **Relaxation of the restrictions on obs. purity to allow query methods within the specs of query methods**
- **Work to be done:**
  - **formalization**
  - **usability of frame conditions in complex designs**