Modular Typestate Checking in Concurrent Java Programs

Nels E. Beckman
nbeckman@cs.cmu.edu

Problem: *Check-then-act* Methods in Multithreaded Programs

Certain classes define check-then-act operations:

```
class OnePlaceBuffer<T> {
  T item = null;
  public synchronized boolean isFull() {...}
  public synchronized void put(T item) {...}

  public synchronized T get() {
    if(item==null) throw new Empty();

    T result = item;
    item = null;

    return result;
  }
}
```

'get' should not be called if buffer is empty!

In the face of concurrent, modifying access, the state of the object may be modified by another thread:

```
void pass() {
  final OnePlaceBuffer<String> b = new OnePlaceBuffer<String>();
  b.put("Welcome to OOPSLA!");

  (new Thread() {
     void run() { if( b.isFull() ) print(b.get()); }}).start();

  if( b.isFull() )
    print(b.get());
}
```

Race condition! On the Empty/Full state of the buffer.

Other more familiar examples:
- java.util.Queue (`remove`/`element` methods)
- java.nio.channels.Channel (`close`/`isOpen` methods)
- javax.sql.rowset.BaseRowSet (`initParams`/`setCommand` methods)
- java.util.Iterator (`next`/`hasNext` methods)

Can we statically, modularly ensure correct protocol usage?

Carnegie Mellon

Solution: Access Permissions Modularly Denote Concurrent Modification

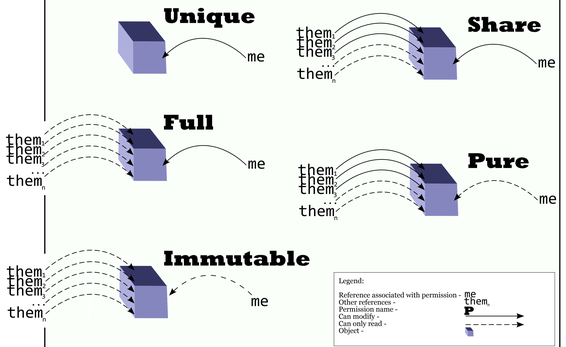
Solution in a Nutshell

1 - Programmer specifications include Access Permissions (which tell analysis if threads will concurrently modify).
2 - Programmers specify protocol transitions (implicitly, using PRE- and POST-conditions).
3 - We can only track the state for an object statically if Access Permission indicates concurrent modification is impossible.
4 - OR if reference is synchronized!

1 - Access Permissions

Programmers specify if/how references may be aliased, and whether modification can occur. There are 5 permission kinds:

Forbid Concurrent Modification: Permit Concurrent Modification:



You may already understand Access Permissions, if you are familiar with:
- Boyland's Fractional Permissions [3]
- Separation Logic [4]
- Ownership Types [5]
- Bierhoff's Access Permissions [2] (which this work extends)

2 - Method Specifications

Receiver must have share permission, buffer must be FULL.

```
@Share(requires="Full",
       ensures="Empty")
T get() { ... }
```

Callers need pure permission to read receiver.

```
@Pure
@TrueIndicates("Full")
@FalseIndicates("Empty")
boolean isFull() { ... }
```

If this method returns true, the buffer is full, otherwise it's empty.

3 - Static State Tracking

- Associate a reference with the state of the object.
- State changes as methods are called on that reference (using PRE & POST).
- Account for thread interference. **Share** & **Pure** references must be associated with the unknown state:

Share(ref) in FULL ✗

Share(ref) in ??? ✓

4 - Accounting for Mutual Exclusion

- Track synchronized references. Those references can be associated with a known state.

Share(ref) in FULL ✓
where ref is synchronized

- Internal object synchronization is also enforced. (Ask me for details.)

Related Work:
[1] N. Beckman, K. Bierhoff, J. Aldrich. Verifying Correct Usage of Atomic Blocks and Typestate. OOPSLA 2008.
[2] K. Bierhoff, J. Aldrich. Modular Typestate Checking of Aliased Objects. OOPSLA 2007.
[3] J. Boyland. Checking Interference with Fractional Permissions. SAS 2003.
[4] P. O'Hearn. Resources, Concurrency and Local Reasoning. Theor. Comput. Sci. 2007.
[5] B. Jacobs, F. Piessens, K. Rustan M. Leino, W. Schulte. Safe Concurrency for Aggregate Objects with Invariants. SEFM 2005.

Example: Finding Race Condition

```
void pass(OnePlaceBuffer<String> b) {
  ... // see above
```

perms = { share(b) in ? } synchronized = {}

```
  if( b.isFull() )
```

perms = { share(b) in ? } synchronized = {}

```
    print(b.get());
}
```

b has Share permission, since other thread also needs to modify.

Share means other modifying refs. exist. Simulate interference.

Error! Pre-condition unsatisfied! b not Full.

Example: Verifying Corrected Code

```
void pass(OnePlaceBuffer<String> b) {
  ... // see above
```

perms = { share(b) in ? } synchronized = {}

```
  synchronized(b) {
```

perms = { share(b) in ? } synchronized = { b }

Track b as a synchronized ref.

```
    if( b.isFull() )
```

perms = { share(b) in Full } synchronized = { b }

```
    print(b.get());
```

perms = { share(b) in Empty } synchronized = { b }

Reference is not subject to concurrent modification!

```
  }
}
```

Download the Eclipse plug-in:
http://code.google.com/p/pluralism/

Image courtesy tkgd2007, CC-SA