Modular Typestate Checking in Concurrent Java Programs

Nels E. Beckman
Carnegie Mellon University
nbeckman@cs.cmu.edu

Abstract: In previous work [Beckman et al. (2008)] we described a modular static analysis based on access permission annotations, which describe the ways in which a reference can be aliased, for preventing the improper use of object protocols in concurrent programs. That system was based on atomic blocks, a mutual exclusion primitive not yet in wide use. Now we extend that system to programs written using synchronized blocks, which are in wide use today. This system can verify concurrent programs without any concurrency-specific annotations.

1 Introduction and Motivation

This work describes an approach for verifying that object protocols are obeyed in concurrent, object-oriented programs that use synchronized blocks as a means of mutual exclusion. Many APIs define object protocols, which are rules about the order in which methods can be called. Much recent work has focused on checking, either statically or dynamically, that clients are using these object protocols correctly. (Static protocol checking is also known as typestate checking.) While most of this work has focused on single-threaded programs, object protocols also exist in concurrent programs.

In recent work we developed a modular static analysis for Java for ensuring that object protocols were used correctly even in the face of concurrent modification by multiple threads [Beckman et al. (2008)]. One limitation of this work was its assumption that atomic blocks would be used as the mutual exclusion primitive. Atomic blocks are a mutual exclusion primitive provided by transactional memory systems that, while showing much promise, are not currently in wide use. Currently, using our analysis on code that uses other forms of mutual exclusion will result in false positives, as our analysis will assume that mutual exclusion does not exist.

Therefore we have extended our approach to work with programs that use synchronized blocks as the primary means of mutual exclusion. While our approach does in some ways limit the flexibility of the locking discipline allowed, in return it does not require any additional annotations describing synchronization procedure.

2 Approach

Our approach is similar to our earlier work in that we require programmers to annotate methods with pre- and post-conditions as well as *access permissions*, static typing annotations which describe the ways in which a method parameter can be aliased. These permissions will act as an approximation of the thread sharedness of the object. However, where our earlier approach used atomic blocks to enable the tracking of possibly thread-shared objects, we will use synchronized blocks.

In our approach, every program reference is associated with a permission *kind*. The permission kind forms a part of the type of that reference, and it tells our analysis whether or not that reference points to an object reachable through other references, and whether this and those other references can be used to read or read and modify the object. Permissions have two important properties: First, permissions are always conservative. If a permission says that a reference is the only reference to an object (which we call, unique permission) then we are guaranteed that this is true. Second, permissions can act as a sound approximation of thread sharing when we assume that other references are held by other threads. For example, if the permission associated with a reference `r` indicates that `r` can be used to modify the object to which it points, but also that other modifying references may exist (which we call share), our analysis assumes that this object could be concurrently modified. If, on the other hand, the permission is unique, meaning we have the only reference to the object, we take this to mean the object is thread-local.

The way that the analysis works, at a high level, is to associate an abstract state with each reference as it flows through a method's body. The abstract state of that reference may change depending on the pre- and post-conditions of the methods called on that reference. Most importantly, if the permission associated with a reference indicates that other modifying references exist, we must conservatively assume that the object could be concurrently modified, and therefore we actively discard knowledge about the abstract state of that object. Specifically, we do this for references associated with share (meaning that the reference may modify, but that other modifying references exist) and pure (meaning that the reference may only read but that other modifying references exist) permission. The only way that we do not have to discard the abstract state of a pure or share reference is if we know statically that we have previously synchronized on that reference. As we will see later, this will be enough to ensure that the object is not being concurrently modified.

```
 1 interface BlockingQueue {
 2   @Full(requires="OPEN", ensures="OPEN")
 3   void enqueue(int i);
 4
 5   @Full(requires="OPEN", ensures="CLOSED")
 6   void close();
 7
 8   @TrueIndicates("OPEN")
 9   @Pure boolean isOpen();
10
11   @Pure(requires="OPEN", ensures="OPEN")
12   int dequeue();
13 }
14
15 void enqueWaitClose(@Full(requires="OPEN"
16     ensures="CLOSED") ) BlockingQueue q) {
17   for(int i=0; i<5; i++) {
18     // full(q) in OPEN
19     q.enque(i);
20   }
21   Thread.sleep(2000);
22   q.close();
23   // full(q) in CLOSED
24 }
25
26 int sumFromQueue(@Pure BlockingQueue q) {
27   int sum = 0;
28   while(true) {
29   //synchronized(q) {
30     if(q.isOpen()) {
31       // pure(q) in ?
32       sum += q.dequeue(); // ERROR
33     }
34     else
35       return sum;
36   //}
37   }
38 }
```

Figure 1: A blocking queue interface, and two methods that use it, each to be called by a different thread. Because the producer has full permission, it can maintain the knowledge that the queue remains open, while the consumer must synchronize on the queue to maintain that knowledge.

To better illustrate the entire approach, consider the example shown in Figure 1. In this contrived example, the interface for a thread-shared queue is defined. This queue is meant to be used by a producer thread, with full (exclusive modifying but other readers may exist) permission to the queue. That thread will call the `enqueue` and `close` methods, as illustrated in the `enqueWaitClose` method. The consumer thread, who will have a pure (read-only) permission, will only be able to call the `isOpen` and `dequeue` methods.[1]

Our approach works within a method and follows permissions to references as they flow from method pre-conditions to post-conditions. In the `enqueWaitClose` method this means that we track a full permission to the `q` reference that is initially in the OPEN state. This satisfies the pre-condition to both the `enqueue` and `close` methods, so our analysis signals no errors. After the call to `close`, the post-condition of that method indicates that the queue will be transferred to the CLOSED abstract state, which satisfies the calling method's post-condition.

The important thing to note here is that because the thread executing the `enqueWaitClose` method has the only modifying reference to the queue (i.e. full) our analysis does not have to assume the queue can be concurrently modified, and we maintain knowledge of the queue's state from one line to the next.

Things are different in the `sumFromQueue` method where the pure permission to the queue tells our analysis there may be other modifying references held by other threads. In the example as given, our analysis will discard the abstract state implied by the true branch of the conditional on line 30. This means that the pre-condition of the `dequeue` method cannot be satisfied, and our analysis will signal an error. If instead we were to uncomment the *synchronized* block on line 29, verification of this method would succeed, since the queue would no longer subject to concurrent modification. Our approach statically tracks the references on which the current thread is known to have synchronized.

There is one more essential aspect of our approach. We also check that protocols are *implemented* correctly with respect to their specification. Abstract states are defined by concrete predicates over the fields of an object. Our approach verifies that these predicates are valid whenever a method claims to transition the receiver to a particular state.

When checking a method, access to the these predicates is only available inside a block synchronizing on `this`, if the receiver is associated with share, pure, or full permission[2]. In effect this forces field reads and stores to occur within critical regions. The result is that clients of a thread-shared object are allowed to actually remember which state the object is in when it holds the lock associated with that object, since the object could not be modified without acquiring the lock. This requirement also ensures that objects referenced from a unique field of a thread-shared object can still be treated as thread local, since the outer object will be itself be required to provide mutual exclusion.

The fact that programmers are required to synchronize on the receiver object does prohibit programmers from using private objects for the purposes of mutual exclusion. However, it has the benefit of being implicit in our analysis, and therefore we do not require programmers to specify which locks protect which pieces of memory, a common source of specification burden in other modular static analyses. Our approach requires just the aliasing annotations that would be required in a single-threaded analyses.

We formalized this static analysis as a core, Java-like language whose type system enforces correct use of typestate in concurrent programs. We then proved the analysis sound with respect to a formal operational semantics. The proof of soundness guarantees that no thread can ever assume an object is in an abstract state at runtime that it is not in, even if that object is thread-shared. This approach has been implemented as a static analysis called *Sync-or-Swim* for Java programs. The analysis is open-source, based on the Eclipse platform, and has been used to verify numerous open-source programs.

2.1 Related Work

We believe our approach is the first to statically and modularly check object protocols in concurrent Java programs. While similar behavioral checkers exist (e.g., [Jacobs et al. (2005)]), they generally require specifications describing which locks protect which objects, or method specifications describing whether or not locks must be held when called, and do not focus on typestate.

References

[Beckman (2009)]
N. E. Beckman. Verifying concurrent software using atomic blocks and alias control. Thesis Proposal, January 2009.

[Beckman et al. (2008)]
N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *The 2008 Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM Press, 2008.

[Jacobs et al. (2005)]
B. Jacobs, F. Piessens, K. R. M. Leino, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 137–147, Washington, DC, USA, 2005. IEEE Computer Society.

1
Readers may wonder how the `dequeue` method can be implemented if the receiver has non-modifying permission but will have to modify the underlying structure of the queue in some way to remove the item. In our complete solution [Beckman (2009)], this is achieved through dimensions, which allow clients to have different permissions to different logical parts of an object.

2
These permissions are the only ones that indicate that an object may be thread-shared and can be modified by at least one thread.