Annotation-Based Diagrams for Shared-Data Concurrency

Elissa Newman* Aaron Greenhouse* William L. Scherlis*

Abstract

In languages such as Java and Ada, there are a number of significant concurrency-related design decisions that may not be manifest locally in code. These include, for example, the identification of shared state, how that state is protected, which threads may visit that state, and so on. These decisions relate threads, code, and shared data. We describe new UML-style diagrams that can help make explicit some useful aspects of these relationships. In addition, by using lightweight annotations on source code, we can document how these decisions are realized. By combining diagrams and annotations, we can support both forward and reverse engineering, using tools and analyses to keep the diagrams consistent with the annotated code.

1 Introduction

Multithreaded systems using shared-data have data-flow properties that must be understood to avoid errors resulting from the misuse of shared data. Although UML can model threads via sequence and collaboration diagrams, it does not facilitate describing the ways threads interact through shared segments of state. Such thread interactions are difficult to understand because they are non-local—they are distributed among the cooperating threads and the data they share. The representation invariants of shared data are rarely expressed sufficiently to facilitate the semantics-based reasoning necessary to differentiate "good" concurrency (efficient use of resources) from "bad" concurrency (race conditions). We propose, therefore, to augment the UML with diagrams that simply and visually represent surrogate design information that, in present practice, is difficult or infeasible to extract from other design representations or source code. Our examples are based in Java, although the ideas extend to similar shared-memory concurrency models.

Our diagrammatic approach to managing and understanding multithreaded systems is combined with assurance that the design information is consistent with code. This assurance is provided through the use of formal annotations in the code relating to critical "mechanical" properties [2]. The annotations represent pre- and post-conditions, and enable incremental analyses to be used both to validate the assertions and to assure consistency of the diagrams, annotations, and source code.

We employ a number of different kinds of annotations, relating to properties such as aliasing [1], effects [3], lock–region associations and locking behavior [4], and usage of condition variables. Most of the annotations reference state in named aggregates called *regions*. Regions enable better management of private information in encapsulations.

Our diagrams and underlying annotations are intended to answer key questions a programmer would ask about a program's concurrent implementation, including *What is the shared state and how is it protected*? *What locks must be held before a particular method may be invoked*? *What are the conditions that may be waited for*? *Which methods may block*? *What is the locking order*? *What locks might a method acquire*? Two diagrams statically depict the structure of lock–state associations and the interactions of methods through shared state:

- The *regional state hierarchy diagram* extends the UML class diagram and identifies the shared state within a set of classes and describes how the state is protected.
- The *method concurrency diagram* is a new diagram incorporating a call graph that describes how methods from many classes may interact through lock acquisitions, condition variables, and method calls.

We suggest that these diagrams are useful both for guiding implementation and for informing later code maintainers of the intended concurrency-related properties of the system. Actual annotations generally relate one-to-one with particular visual abstractions in the diagrams. Region information, central to our techniques for understanding shared state, is represented in both kinds of diagrams. Annotations act as a bridge between diagrams and code enabling validation of implementation–design consistency and automated design–code consistency management, for example, the regeneration of diagrams (annotations) as annotated code segments (design diagrams) evolve.

2 Related Work

Mehner and Wagner [7] present UML collaboration diagrams extended with new UML stereotypes to describe synchronization of Java threads. A new "temporary active" stereotype manages complexity by eliminating uninteresting method call chains. Their diagrams are useful for capturing specific counterexamples to claims of thread-safety, but they are object-based and thus not as useful for expressing details of concurrent designs. In particular, our approach is data-centric and is intended to expose method interactions over shared data at the class level, thus providing a system-wide view of intended synchronization behavior that may be used for both design and understanding purposes.

Artisan Software proposes a task-level concurrency diagram [6] intended for real-time embedded systems design

*Affiliation: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. E-mail: {elissa, aarong, wls}@cs.cmu.edu.

in which tasks, compositions of one or more objects, coordinate via various mechanisms. The concurrency diagram decomposes system functionality into event-driven intertask communication and coordination. While coordination mechanisms are explicitly represented, the classes and methods responsible for performing the coordination within the tasks are not. Although shared state is explicitly identified, aggregations and hierarchies of shared state, immediately visible in our approach, cannot be represented.

3 Understanding Shared State

Uncoordinated access to shared mutable state—violations of lock–state conventions—may lead to race conditions. We propose the explicit declaration of both shared state and how it is protected. We use *regions* to create an abstract hierarchy of state both to mitigate the loss of encapsulation inherent in the explicit identification of state and to support object-oriented features such as subclassing and method overriding.

3.1 Regions Identify State

Regions are hierarchical extensible named groups of fields. Fields are leaf regions in their own right. A new abstract region named *region* can be declared using the annotation

`/*@ visibility region region in parent */`

where *visibility* is a standard visibility modifier (*e.g.*, `public`, *etc.*) and the clause "`in` *parent*" is optional.[1] Fields may be assigned a non-default parent region using the annotation:

`/*@ in parent */`

Regions are inherited by subclasses (subject to visibility constraints), which may add newly defined regions to existing regions, but may not otherwise alter their structure.

Programmers are expected to locate fields associated with the same abstract concept within the same region, producing an abstraction hierarchy. For example, a 2-D point class might place its `x` and `y` fields within a `Location` region, a 3-D point subclass could extend that region with a new `z` field, and a color point subclass could locate a new `color` field within a new `Appearance` region.

There are several predefined regions, including: `Instance`, the default parent of instance regions, and `All`, the root of the region hierarchy. An *instance target* identifies in its full generality a region of a particular object. For example, if variable `p2d` refers to a 2-D point, then instance target `p2d.Location` identifies the `x` and `y` fields of that object; but if `p2d` refers to a 3-D point, then that same target identifies the `x`, `y`, *and* `z` fields. (We have omitted descriptions of other kinds of targets, target aliasing, and our object-oriented effects system.)

3.2 Protecting Shared State

Shared state is identified by associating a region with a lock. In Java, any object may be used as a lock. Canonical practice is to use the object itself to protect its own state, or to use objects referred to by immutable `final` fields as locks [5]. A region is associated with a lock using the "lock/protects" annotation:

`/*@ lock lock is [this|field] protects region */`

This annotation both identifies *region* as being shared state and identifies the lock—either the object itself or the object referenced by immutable field *field*—that must be held before the region may be accessed. The abstract name *lock* is used to identify the lock object in other annotations.

Methods are assumed to acquire the locks they require. It is common, however, for private methods to assume instead that their callers will have already acquired particular locks. This expectation is recorded using the method annotation:

`/*@ requires lock */`

Methods may also return locks (for example, `getTreeLock` in `java.awt.Component`). The "returns lock" annotation is used to record that a method returns a particular lock:

`/*@ returns lock lock */`

3.3 Aggregating State

State identification is enhanced through *target aggregation* in which a single target refers to state within multiple objects. One technique for aggregating state is to parameterize a class by a target, as in done in the example in Figure 1[2] (further described below). Annotations in the source class define the parameterization (the *target parameter* `Backbone` in class `CachedThread`), while annotations in the destination classes provide actual targets (the target `this.Threads`, abbreviated as `Threads`, in various uses in `ThreadCache`), binding the target parameters when instances of the source class are created (as in method `createThread` in `ThreadCache`). Aggregation is achieved in the source class by using the target parameter as a parent within a region declaration (as with fields `next` and `prev` in `CachedThread`).

3.4 Regional State Hierarchy Diagram

This diagram, an extension of the UML class diagram, depicts the hierarchy of regions, as well as the locks that protect various regions. Figure 1 shows a pair of classes from the Jigsaw web server with formal annotations; Figure 2 shows the diagram for this example. `ThreadCache`—a pool of active `CachedThreads` to which it dispatches tasks—maintains a linked list of idle `CachedThreads`, referred to by the field `freelist`. The aggregated backbone of this list—field `freelist` and the `next` and `prev` fields of list's `CachedThreads`—is treated as a single abstract entity: the region `Threads`. The diagram makes it is easy to see that the `freelist` and the `next` and `prev` fields are in the same region even though they are in different classes. In the diagram, the local regions are circumscribed within their parent region, `ThreadCache.Instance`, that is shown to

[1] Following Java convention, our annotations are inserted within comments that begin with the at-sign "@."

[2] Jigsaw is Copyright ©1995–2001 World Wide Web Consortium (MIT, INRIA, Keio Univ.). All Rights Reserved.

```
public class ThreadCache {
  //@ public region Threads in Instance
  //@ public region CacheInfo in Instance
  protected int threadcount, usedthreads; //@ in CacheInfo
  //          >> code omitted <<
  protected CachedThread /*@<Threads>*/
    freelist, freetail; //@ in Threads

  //@ lock CacheLock is this protects Instance

  private synchronized
  CachedThread /*@<Threads>*/ createThread() { ...
    return new CachedThread /*@<Threads>*/ (this, ...);
  }

  synchronized boolean
  isFree(CachedThread /*@<Threads>*/ t, ...) {
    if(!t.isTerminated()) { ... }
    else { ...
      t.prev = freetail;
      if(freetail != null) freetail.next = t;
      freetail = t;
      if(freelist == null) freelist = t;
      usedthreads--; ...
    } ...
  } //      >> rest of class omitted <<
}

class CachedThread /*@<target Backbone>*/ ex-
tends Thread {
  //@ public region ThreadInfo
  private final ThreadCache cache;
  private boolean alive;   //@ in ThreadInfo
  private Runnable runner; //@ in ThreadInfo
  //          >> code omitted <<
  CachedThread /*@<Backbone>*/ next, prev; //@ in Backbone

  //@ lock ThreadLock is this protects ThreadInfo

  synchronized boolean isTerminated() { ... }
  //    >> Additional synchronized methods follow <<
}
```

Figure 1: Annotated versions of ThreadCache and CachedThread.

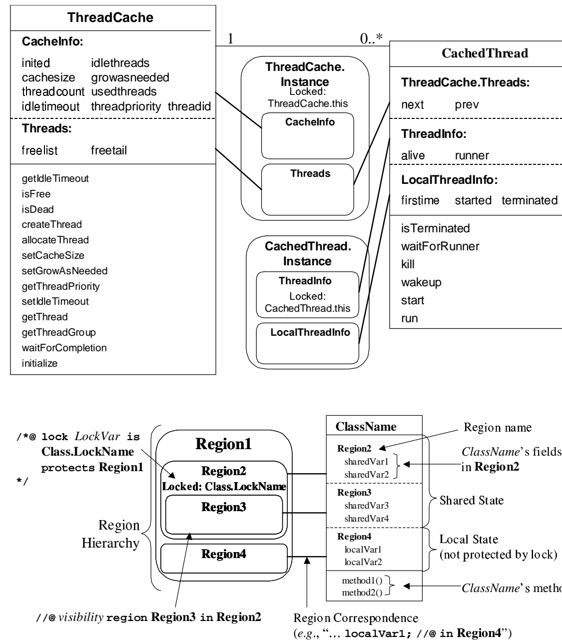

Figure 2: Regional State Hierarchy Diagram for ThreadCache and CachedThread.

be protected by the `ThreadCache` object itself: `ThreadCache.this`.

4 Blocking and Deadlock

Our concurrency annotations can be used to assure freedom from deadlock arising from the use of multiple shared regions, and from irregularities in the use of condition variables.

4.1 Condition Variables

Any object in Java may be used as a signal-and-continue condition variable via the methods `wait`, `notify`, or `notifyAll` with the caveat that the object must already be acquired as a lock. Typical usage requires `wait`ing within a `while` loop that checks the condition, which can obscure the actual condition being waiting for. We therefore use "condition" annotations to define named conditions—boolean expressions without write effects—within a class:

`/*@ condition` *cond* `is` *boolexp* `*/`

To assist with the identification of communication-related errors, *e.g.*, nested monitor lockouts and missed notifications, methods are annotated with their blocking effects:

`/*@ awaits` $cond_1$ `, ...,` $cond_n$ `*/`
`/*@ satisfies` $cond_1$ `, ...,` $cond_n$ `*/`

4.2 Lock Order

It is well known that annotations specifying partial orders of locks can assist in assuring deadlock freedom. The relationship between locks is defined by "acquire/before" annotations:

`/*@ acquire` *lock* `before` $lock_1$ `, ...,` $lock_n$ `*/`

declares that *lock* must be acquired before locks $lock_{1...n}$. To support modular checking, methods are annotated with the least lock (*i.e.*, no lock ordered *before* that lock will be acquired) that may be acquired during the execution of a method:

`/*@ acquires` *lock* `*/`

4.3 Method Concurrency Diagram

This diagram is an extension to the UML to assist in the detection of deadlocks and signaling irregularities. Methods with the same concurrency properties and method calls (within the scope of the diagram) are combined into *method groupings*. Figure 3 shows class `WakeupManager` from Sun's Jini[3] threads package; Figure 4 shows the corresponding diagram. `WakeupManager` is a queue of time-stamped tasks sorted by deadline. Nested class `Kicker` implements a thread responsible for executing tasks in the queue and requires the locks for both itself and the queue so lock ordering is used: `ContentsLock` must be acquired before the `KickerLock`. `Kicker` has two condition variables (stop signs in the diagram): one to abort, and one to signal the addition of a new task. From the diagram it is easy to see,

[3]Jini is Copyright ©1999–2001 Sun Microsystems, Inc. All Rights Reserved.

```
public WakeupManager implements TimeConstants { ...
  //@ public region ContentsRegion in Instance
  private final sortedMap contents; //@ in ContentsRegion
  private Ticket head = null; //@ in ContentsRegion
  private Kicker kicker;

  /*@ lock ContentsLock is contents
           protects ContentsRegion  */

  //@ acquires ContentsLock
  public void cancellAll() {
    synchronized (contents) {
      contents.clear();
      checkHead();
    }
  }

  //@ requires ContentsLock
  //@ acquires kicker.KickerLock
  //@ satisfies newTicket
  private void checkHead() {
    final Ticket oldHead = head;
    //   >> code omitted <<
    kicker.newTime();
  }

  private class Kicker implements Runnable { ...
    private long sleepTime;

    //@ public region KickerInfo
    private boolean sleepTimevalid,
                    die = false; //@ in KickerInfo

    //@ lock KickerLock is this protects Instance
    //@ acquire ContentsLock before KickerLock
    //@ condition isDead is (die == true)
    //@ condition newTicket is (sleepTimeValid == false)

    //@ acquires KickerLock
    //@ awaits isDead, newTicket
    public void run() { ... }

    //@ acquires KickerLock
    //@ satisfies newTicket
    private synchronized void newTime() {
      sleepTimeValid = false;
      notifyAll();
    }

    //@ acquires ContentsLock
    private void doTasks() { ...
        synchronized( contents ){
          synchronized( this ){
            sleepTimeValid = true;
          }
        } ...
    }
  }
}
```

Figure 3: Annotated WakeupManager.

for example, the lock order and that invoking `newTime` may cause a thread blocked in `run` to unblock by satisfying the `newTicket` condition.

5 Conclusion

Our diagrams are intended to provide local views of properties of shared-data concurrent systems that are often inherently non-local. We suggest that our diagrams provide significant information for working with multithreaded systems, including shared state identification and protection, which classes use it and how, and the means used to avoid deadlock. Formal program annotations mediate between diagrams and code, enabling programming tools to maintain their consistency.

We are currently researching additional annotations and diagrammatic techniques for relating threads, shared state, and code segments via "thread coloring" [8] and for describing concurrency policy [4]. These additional annotations and diagrams help provide answers to such additional questions as *Which threads execute this code*? *Which threads access this field*? and *Is it safe to execute these methods concurrently*?

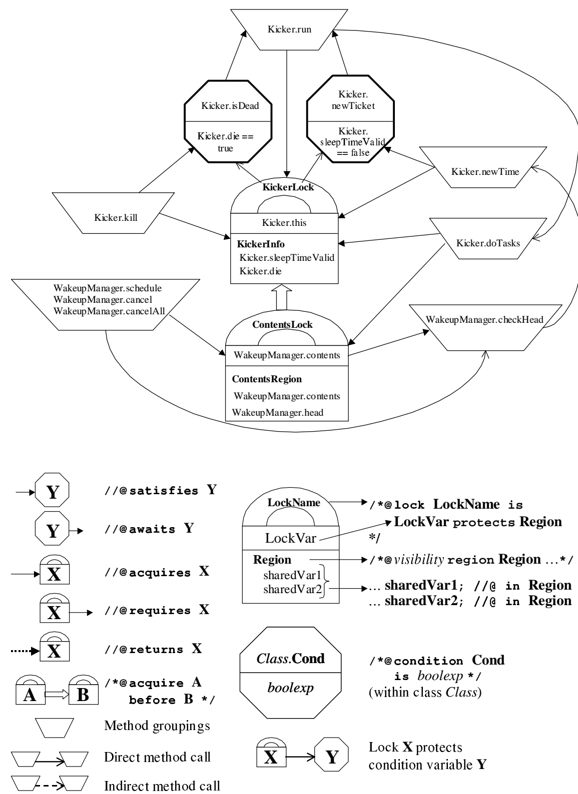

Figure 4: Method Concurrency Diagram for WakeupManager.

References

[1] J. Boyland. Alias burying: Unique variables without destructive reads. *Softw. Pract. and Exper.*, 31(6):533–553, May 2001.

[2] E. C. Chan, J. T. Boyland, and W. L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *ICSE '98*, pages 167–176, Los Alamitos, California, 1998. IEEECS.

[3] A. Greenhouse and J. Boyland. An object-oriented effects system. In R. Guerraoui, editor, *ECOOP '99*, volume 1628 of *LNCS*, pages 205–229, Berlin, 1999. Springer.

[4] A. Greenhouse and W. L. Scherlis. Assuring and evolving concurrent programs: Annotations and policy. Submitted to *Automated Software Engineering* 2001.

[5] D. Lea. *Concurrent Programming in Java*. The Java Series. Addison-Wesley, Reading, MA, second edition, 2000.

[6] M. J. McLaughlin and A. Moore. Real-time extensions to UML. *Dr. Dobb's Journal*, pages 82–93, Dec. 1998.

[7] K. Mehner and A. Wagner. Visualizing the synchronization of Java-threads with UML. In *Int'l Symposium on Visual Lang. 2000*, pages 199–206. IEEECS, Sept. 2000.

[8] D. F. Sutherland. Personal communication: Discussions on thread coloring, 2001.