Formalizing Reusable Aspect-Oriented Concurrency Control

Neelam Soundarajan, Derek Bronish, Raffi Khatchadourian
Computer Science and Engineering
Ohio State University
{neelam,bronish,khatchad}@cse.ohio-state.edu

Abstract

Java and its library provide powerful concurrency control mechanisms. However, their use can lead to similar synchronization code being scattered *across multiple classes, synchronization and functional code being* tangled *together, and similar code being duplicated in many applications. Aspect-oriented (AO) programming makes it possible to address these problems. The precise behavior of systems built using AO techniques can, however, be difficult to understand. We propose a specification approach to precisely express key concurrency and synchronization properties of such systems. We illustrate the approach with a simple example.*

1 Introduction

Java provides a number of concurrency control mechanisms that allow a designer to specify that particular methods of a class are *synchronized*. When a *thread* invokes such a method on an object, it is suspended until it acquires the *lock* associated with the object, then proceeds to execute the method, releasing the lock when the method finishes. A finer-grained version allows an individual statement (or block) of a method to synchronize on the *this* object or, possibly, a different object. However, use of these mechanisms can reduce the degree of concurrency considerably. To alleviate this, the Java library provides the `Lock` interface implemented by classes such as `ReentrantLock`. By using *lock* objects appropriately, one can regain concurrency. But, at the same time, their use introduces some problems. First, if concurrency was not foreseen or its degree not anticipated when a class was designed, the approach requires invasive changes to the class. Second, synchronization and functional code of a class are often *tangled* together. Third, similar synchronization code may be *scattered* across multiple classes since similar synchronization concerns may arise in each one.

Aspect-oriented (AO) techniques help address such problems. The synchronization code can be contained in an *aspect*, separating it from the functional code. Moreover, by defining the *pointcuts* in the aspect suitably, the aspect's *advice* can apply to methods of multiple classes, thereby meeting the synchronization needs of each class. Synchronization code that is common to different applications can be defined in *abstract* aspects with *subaspects* for each application specializing it for that application, thereby eliminating code duplication.

The precise behavior of systems built in this manner can, however, be difficult to understand and formal specifications can help. We develop an approach that allows us specify, in the form of *contracts*, important behaviors, especially concurrency behaviors, of abstract aspects; and, in the form of *subcontracts*, specify how subaspects specialize the contracts to achieve behavior specific to the given system. We illustrate the approach by applying it to a simple example based on one in [7].

In Section 2, we consider related work. In Section 3, we sketch the model underlying our specifications. In Section 4, we present our specification approach via a simple case-study. Section 5 concludes the paper.

2 Background

Several approaches have been developed for reasoning about sequential AO systems. Dantas and Walker [3] consider *harmless* advice that does not modify the behavior of *base-code*, i.e., the underlying program. In [6], we consider how to provide information to arrive at the "richer" behavior caused by the advice without reanalyzing base-code. Katz and Katz [5] consider similar *rely-guarantee* specs. Aldrich [1], and Griswold *et al.* [4] propose ways to minimize the effects of aspects on base-code. Zhao and Rinard [12] consider abstract aspects but the the types of abstractness they allow are limited. Xu *et al.* [10] review approaches to reasoning about concurrency. Long & Long [8] present a formal specification of Java concurrency. Yang & Poppleton [11] present a model checker for concurrent Java.

3 Model Of Multi-threaded Computation

Fig. 1 outlines, in RESOLVE [9] style, the main components of a model of multi-threaded computation in a language with reference semantics as in Java. A *shared heap state*, lines 1–3, represents the *objects* in the sys-

tem. A *thread*, lines 7–8, consists of its id, a boolean indicating if it is active, and its control-flow-state which, lines 4–6, represents the sequence of method calls made in the thread that have not yet completed. A *synchronized lock*, lines 9–11, consists of the id of the associated object, whether it is currenly locked and, if so, the id of the holding object, and two sets of threads waiting to execute, respectively, synchronized and unsynchronized methods on the object. The other type of lock (not for a specific object), lines 12–13, consists of a boolean indicating whether it is locked, if so the id of the object holding it, and the set of ids of the objects waiting for it. A *lock*, lines 14–15, is one of these two types; note that this model is inadequate for dealing with *read-write* locks which we do not consider. A *system*, lines 16–17, consists of a heap, its current threads, and its locks.

```
1 math type Heap_State is
2  partial function from ObjectId
3         to ObjectValue
4 math type Control_Flow_State is
5  sequence of tuple of
6   (m: method sig, obj: ObjectId)
7 math type Thread is tuple of
8   (id, active, cfs)
9 math type Synch_Lock is tuple of
10   (object, locked, holder,
11    waitersForSynch, waitersForUnsynch)
12 math type Other_Lock is tuple of
13   (locked, holder, waiters)
14 math type Lock is
15   Synch_Lock or Other_Lock
16 math type Threaded_System is tuple of
17   (heap, threads, locks)
18 constraint: l: Synch_Lock
19  l.locked ⇒ [l.holder.active
20  ∧ head(l.holder.cfs).obj=lId
21  ∧ head(l.holder.cfs).m is synchronized
22  ∧ ∀ t ∈ waitersForSynch:
23    not t.active ∧ head(t.cfs).obj=lId
24    ∧ head(t.cfs).m is synchronized]
```

Figure 1. Model of threaded system

The model must satisfy certain constraints, one of which, lines 18–27, is that if a `synch_Lock` is locked, the thread holding it must be `active`, must, as indicated by its `cfs`, be currently executing a synchronized method on the corresponding object, and threads whose id's are in `waitersForSynch` must be inactive and waiting to execute a method on the object.

4 Case Study

Due to lack of space, we present our specification approach via a simple case-study, in Fig. 2, based on the `Shape` example of [7]. Instances of the class represent shapes with `x`–, `y`–coordinates, and `height` and `width` which may be accessed using `get` methods; `move` methods allow us to move the shape; `magnify()` and `shrink()` to expand and contract it.

```
1 class Shape
2 {protected int x = 0, y = 0,
3   height = 5, width = 10;
4  public int getX() { return x; }
5  //getY(), getHeight() etc. similar
6  public void moveNorth() { y++; }
7  //moveSouth(), moveEast(), etc. similar
8  public void magnify() {
9   height=height+5; width=width+10;}
10 public void shrink() {
11  height=height-5; if(height<=0)height=5;
12  width=width-10;if(width<=0)width=10;}}
13
14 protected final Lock locLock =
15  new ReentrantLock();// lock for location
16 protected final Lock dimLock =
17  new ReentrantLock();// and dimension
18 public int getX() { int tx;
19  locLock.lock(); tx = x;
20  locLock.unlock(); return tx; }
21 //getY() similar; getHeight(),getWidth()
22 // similar but using dimLock
23 public void magnify() {
24  dimLock.lock();...;dimLock.unlock();}
25 //move methods similar but using locLock
```

Figure 2. Shape class and Split locks

Suppose the application has many threads accessing a `shape`. Making each method *synchronized* would prevent interference between the threads but minimize concurrency. Instead, we use *two locks*, lines 14–17, one for location, the other for dimension. The methods, lines 18–25, which access/modify the location, respectively dimension, acquire the former, respectively the latter, perform their work, and release the lock.

4.1 Split Lock Aspect

In the AO approach in Fig. 3, `_lock1`, `_lock2` serve the same purpose as `locLock`, `dimLock`; `fstSetOps()`/`secondSetOps()` pointcuts correspond to calls that should use `_lock1`/`_lock2`. The advice, lines 7-10, is the synchronization code. Fig. 4 defines the subaspect for *Shape*, achieving the same level of concurrency as before but without scattering/tangling.

Suppose now `shrink()` was changed as follows: if the object was shrunk below a certain size, move it to a more visible place by changing `x` and `y`; and suppose we left the subaspect as in Fig. 4. This system may behave acceptably in all test cases. Indeed, the only time it

```
1 public abstract aspect SplitLockAspect
2  perthis(fstSetOps() || scndSetOps()){
3  protected abstract pointcut fstSetOps();
4  protected abstract pointcut scndSetOps();
5  private Lock _lock1 = new ReentractLock();
6  private Lock _lock2 = new ReentractLock();
7  before():fstSetOps(){ _lock1.lock();}
8  after() :fstSetOps(){ _lock1.unlock();}
9  before():scndSetOps(){ _lock2.lock();}
10 after() :scndSetOps(){ _lock2.unlock();}
```

Figure 3. SplitLock Aspect

would misbehave is if one thread invoked, say, `getX()` on a `shape` at the same time another was executing `shrink()` on it, *and* the size of the `shape` was below our minimum, *and* the timing of the two executions resulted in a strange value being returned for `x`.

```
1 aspect ShapeSplitLockAspect
2      extends SplitLockAspect {
3  pointcut fstSetOps() :
4     execution(Shape.getX())
5   ||execution(Shape.getY())
6   ||execution(Shape.moveNorth())||...
7  protected pointcut scndSetOps() :
8   execution(Shape.getHeight())||...
9   ||execution(Shape.shrink())||...
```

Figure 4. Subaspect for Shape System

4.2 Contracts and Subcontracts

Consider an abstract aspect *AB*. We specify *AB* in an *aspect contract*. For *abstract* portions of *AB*, we will use *abstraction concepts* in the contract. These will not be *defined* in the contract but will be *used* in it. The contract will also impose certain constraints that definitions, provided in subaspects, for the concepts must satisfy.

Part of the `SplitLockAspect` contract appears in Fig. 5. Line 5 requires that the definition, in a subaspect, of `firstSetOps()` must be such that all methods whose execution join points match it must be from a *single* class. This represents that the lock is not intended to prevent simultaneous execution of methods from different classes. The next clause, for `scndSetOps()`, is similar. The next clause, line 6, requires that these two classes be the same. The next clause requires a given method to be mapped to (at most) one of these pointcuts. The subaspect in Fig. 4 can be easily seen to meet these requirements.

The constraint in lines 10–11 requires that the (union of the) set of objects accessed by the methods mapped to `firstSetOps` be disjoint from the set accessed by methods mapped to `secondSetOps`. But the information needed to check this constraint is *not* part of the subaspect in Fig. 4. The *abstraction concept*,

```
1  abstraction concepts:
2  set AccessedObjects(set mthds)
3   //objects accessed by methods in mthds
4  constraints:
5  [(|fstSetOps.Class|=|scndSetOps.Class|=1)
6  ∧(fstSetOps.Class=scndSetOps.Class)
7  ∧(firstSetOps.Mthds∩scndSetOps.Mthds=∅)]
8  AccessedObjects(S1 ∪ S2) =
9   AccessedObjects(S1)∪AccessedObjects(S2)
10 [(AccessedObjects(fstSetOps.Mthds)∩
11   AccessedObjects(scndSetOps.Mthds) = ∅)]
12 results:
13 [(m1∈fstSetOps.Mthds)∧(m2∈scndSetOps.Mthds)]
14       ⇒ [ nosuspension(m1,m2)
15           ∧ nosuspension(m2,m1)]
16 [(m1∈fstSetOps.Mthds)∧(m4∈ fstSetOps.Mthds)]
17       ⇒ noconcurrency(m1,m4)
18 [(m3∈scndSetOps.Mthds)∧(m2∈scndSetOps.Mthds)]
19       ⇒ noconcurrency(m3,m2)
```

Figure 5. Contract for SplitLock Aspect

`AccessedObjects` (line 2) (and its definition in the subcontract), help address this. The contract does not define it but the name (and associated comment) is suggestive: the set of objects accessed by the methods in `mthds`. The constraint in lines 10–11 uses it to capture the key requirement that the intersection between the sets of objects accessed respectively by the methods mapped to the two pointcuts be empty. Lines 8–9 impose a simple constraint on the definition, in the subcontract, of `AccessedObjects`: that it be distributive.

Lines (11-16) specify the results of using the aspect: if two methods are mapped to the two pointcuts, their respective executions will not suspend each other; but if they are mapped to the same pointcut, there will be no concurrency between their executions.

The subcontract `ShapeSplitLockAspect` needs to only define `AccessedObjects`. Since it is required, by the aspect contract, to be distributive, we can define it by specifying its value for each method mapped to the two pointcuts. This is easily done for our example; thus, for `getX()`, the value will be the set {`x`}; for `magnify()`, it will be {`width, height`}; etc.

Next we have to check that the contract's constraints are satisfied. It is at *this* point that we can locate the bug considered earlier where `shrink()` may modify `x` and `y`. Given this code, `AccessedObjects(shrink)` should be {`x, y, height, width`}; hence, the constraint in lines (8-9) is violated; thus, for example, `x` is in both `AccessedObjects(shrink)` and `AccessedObjects(getX)` and these two methods are mapped to the two pointcuts and the contract is violated.

Next, the definitions in the subaspect and subcon-

tract can be plugged into the aspect contract, in particular, into the *results* clauses of the contract to arrive at the specialized versions of these clauses that apply to this particular system. In the current case this will allow us to conclude, for example, from lines (15–16) of the contract, that if a thread was currently executing `getHeight()` on a shape object and another invoked `magnify()` on the same object, the latter would be suspended until the former finishes. To build another application in which `SplitLockAspect` could be used, we would only have to define the corresponding subaspect and subcontract, defining respectively the pointcuts and `AccessedObjects`; next, verify that, with these definitions, the constraints in Fig. 5 are satisfied; and, finally, arrive at the behavior of the application by plugging in the definitions into the results in Fig. 5.

How do we define **nosuspension**() and **noconcurrency**()? These are primitives provided by the formalism and are defined in terms of the model of Section 3. Thus **noconcurrency**(M), where M is a set of methods, means if `t1` and `t2` are two threads in the `threads` component of a system and `m1, m2` are elements of M, `m1` is in `cfs`, the control-flow-state of `t1`, `m2` is in `cfs` of `t2`, and the corresponding `obj` components are equal to each other, then the `active` component of `t2` must be *false* if that of `t1` is *true*. The precise set of primitives and their definitions is part of our current work. One other point is worth noting. Standard approaches to formalizing aspects typically make use of *pre-* and *post*-conditions. Why doesn't our contract for `SplitLockAspect` involve these? The answer is that our focus is on *concurrency* issues. Thus, for example, the constraint expressed in lines (8–9) of Fig. 5 can be compared with the notion of *interference freedom* [10]. Our clause, by requiring that the intersection of the sets of objects accessed by the two groups of methods to be empty, ensures that no method in the first group will interfere with any method in the second group.

5 Discussion

Our goal was to show how common synchronization patterns may be implemented using abstract aspects that are then specialized, using subaspects, for individual systems; and develop an approach to specify essential properties of the aspects and subaspects. Although the question of and reasoning about AO programs has received much attention, the problem of abstract aspects, especially for concurrency behaviors, and their specializations has not, to our knowledge, been addressed.

Our contracts and subcontracts had to refer to more than just the class variables. Hence we defined a model that provided a view of the threads that exist at runtime. And in our specs, we referred to components of this model, allowing us to specify the concurrency behaviors of interest. Importantly, the use of *abstraction concepts*, in conjunction with the constraints imposed on them, allowed us to represent the *intent* of the abstract aspect.

In Section 4, we relied on intuitive reasoning to show that the subaspect and its subcontract satisfy the aspect contract's requirements. For more complex systems, tool support would be needed. Bodden and Havelund [2] describe an AO algorithm, *Racer*, that uses new *concurrency-related* pointcuts that may be used to identify concurrency bugs. It should be possible to use a *Racer*-like approach to build a tool that will spot violations of our contracts/subcontracts and we plan to pursue this in future work. Along a different line, our model may help identify additional AO primitives that will be of use in writing concurrent programs; for example, primitives that allow the programmer access to information about the threads in the system may be useful. These would be analogous to such primitives as *cflow* but tuned to the needs of concurrency behaviors.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *Proc. of ECOOP*, pages 144–168. Springer, 2005.

[2] E. Bodden and K. Havelund. Racer: Race detection using "AspectJ". In *ISSTA*, pages 155–166. ACM, 2008.

[3] D. Dantas and D. Walker. Harmless advice. In *POPL '06*, pages 383–396. ACM, 2006.

[4] W. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Mod. softw. des. with crosscutting interfaces. *IEEE Softw.*, 23(1):51–60, 2006.

[5] E. Katz and S. Katz. Incremental analysis of interference among aspects. In *Wkshp. on fnds. of AO langs.*, 2008.

[6] R. Khatchadourian, J. Dovland, and N. Soundarajan. Enforcing behavioral constraints in evolving AO programs. In *Wkshp. on fnds. of AO langs.*, 2008.

[7] D. Lea. *Concurrent Programming in Java, Second Edition*. Addison-Wesley, 2000.

[8] B. Long and B. Long. Formal specification of java concurrency to assist software verification. In *Int. Symp. on Par. and Dist. Processing*. IEEE-CS, 2003.

[9] M. Sitaraman and B. Weide. Component-based software using "RESOLVE". *Software Eng. Notes*, 19(4):21–63, 1994.

[10] Q. Xu, W. de Roever, and J. He. Rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

[11] L. Yang and M. Poppleton. Jcsprob: Implementing integrated formal specifications in concurrent java. In *CPA*, pages 67–88, 2007.

[12] J. Zhao and M. Rinard. Pipa: Behavioral interface for "AspectJ". In *FASE*, pages 150–165. Springer, 2003.