Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms

Carlos A. Cunha
Escola Superior de Tecnologia
Instit. Politécnico de Viseu
Campus de Repeses
3504-510 Viseu
PORTUGAL
cacunha@di.estv.ipv.pt

João L. Sobral
Departamento de Informática
Universidade do Minho
Campus de Gualtar
4710-057 Braga
PORTUGAL
jls@di.uminho.pt

Miguel P. Monteiro
Escola Superior de Tecnologia
Instit. Politécnico de Castelo Branco
Avenida do Empresário
6000-767 Castelo Branco
PORTUGAL
mmonteiro@di.uminho.pt

ABSTRACT

In this paper, we present a collection of well-known high-level concurrency patterns and mechanisms, coded in AspectJ. We discuss benefits of these implementations relative to plain Java implementations of the same concerns. We detect benefits from using AspectJ in all the cases presented, in the form of higher modularity, reuse, understandability and unpluggability. For most of the implementations, two alternatives can be used: one based on traditional pointcut interfaces and one based on annotations.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *Concurrent programming structures, Patterns.*

Keywords

Concurrency Mechanisms and Patterns, Object-Oriented Concurrent Programming, Aspect-Oriented Programming.

1. INTRODUCTION

Programming with concurrency using traditional languages is a complex task, usually left to experts. Examples of concurrency-related concerns are the definition of sections of behaviour that must be subject to synchronised access, in order to avoid race conditions, applying the right scheduling policies to those parts, placing barriers where threads must synchronise, and identifying tasks that can run in a separate thread. Such concerns do not align well with the class decomposition, and therefore source code related to such concerns suffers from the well-known negative phenomena of code scattering and tangling [14]. Debugging concurrent applications is equally complex, due to the execution unpredictability introduced by concurrency. It is often hard to trace incorrect behaviour to its underlying cause (e.g., trace the defect to concurrent or core behaviour). Concurrent programming is gaining importance, has built-in support in recent object-oriented (OO) languages, such as C# and Java [17][25] and is essential to leverage the fast growing multi-core CPU market.

Various efforts have been carried out to improve development of concurrent applications. Specialised libraries such as those provided by Java 1.5 [25] help to reduce the number of lines of code needed to add concurrent behaviour to applications. However, they fail to eliminate the problems associated with crosscutting. New languages have been proposed that provide alternative abstractions, which incorporate high-level concurrency constructs. ABCL [30] is an early example using active objects, one-way calls and futures to model concurrency.

Aspect-Oriented Programming (AOP) was proposed to deal with crosscutting concerns in OO systems [14]. AOP promises to bring to concurrency-related concerns the usual benefits of modularisation, namely improved code readability and analysability, a greater level of reusability and (un)pluggability and more independent development, testing and configurability. However, these promises were not yet fully put to the test. We set ourselves to do that, by developing an aspect-oriented (AO) collection of high-level concurrency patterns and mechanisms. The purpose of this paper is to present such a collection, which includes one-way calls, futures, waiting guards, readers/writers (RW) locks, barriers and active objects [17][27][29]. The collection is coded in AspectJ [13] and built on top of Java's concurrency mechanisms. It does not include classic low-level mechanisms [5] such as semaphores and monitors. This paper does not address the design of concurrent applications in order to separate concurrency from core functionality [23][6][26].

Developing the collection gives rise to the following questions – in this paper, we provide a contribution to answering them:

1. What are the main benefits and drawbacks from going from a modern OO implementation to an AO implementation?
2. Can we replace concurrent OO approaches using this collection?

The rest of the paper is structured as follows. Section 2 presents the collection. Section 3 presents illustrative examples using the collection. Section 4 discusses the implementations. Section 5 briefly surveys related work. Section 6 presents directions for future work and section 7 concludes the paper.

2. CONCURRENCY PATTERNS AND MECHANISMS

The mechanisms whose implementations[1] we present in this section are often used in the development of concurrent applications to introduce flexibility, though with an added cost to complexity and analysability [24]: pattern code is scattered across many classes tangled with code not related to the concern, making

[1] All implementations, code samples and benchmarks are available from http://gec.di.uminho.pt/ppc-vm/conccollection/

it hard to reuse pattern implementations. AspectJ enables the development of reusable implementations of such patterns and mechanisms, by moving each reusable part to a separate module, independent of any case-specific code [10]. The structure of AspectJ implementations follows the *template advice* idiom [9], which entails creating an abstract aspect declaring reusable abstractions and a concrete aspect tailored to a case-specific code base that defines the case-specific joinpoints to be captured in the logic declared by the abstract aspect (see Figure 1).

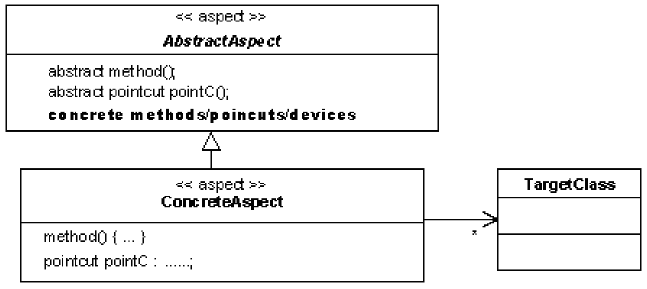

Figure 1: Structure of reusable AO implementations

2.1 One-way calls

One-way [17] is a mechanism that applies to void methods only – when the method does return a value, a future (see 2.2) should be used. One-way methods run on a thread of their own: the client never blocks, waiting for some result. One-way calls can improve throughput in cases in which parallel tasks can run faster than the pure sequential counterparts. Our implementation is based on abstract aspect OnewayProtocol (see Figure 2) that can be applied through the concretisation of pointcut *onewayMethodExecution* and optional definition of *join*, *interrupt* and *interruptAll*. Pointcut *onewayMethodExecution* specifies the events that are intended to run concurrently. The aspect creates a new thread per captured joinpoint, which will run the method call. In the around advice, *proceed()* runs inside a *Runnable* anonymous class. In Figure 3, pointcut *onewayMethodExecution* specifies the method invocations that should run in a new thread. Threads created by the aspect can be spawned with specific thread group other than the current (*getThreadGroupName* method).

The concrete aspect may optionally define a blocking point where the calling thread waits for the termination of all spawned threads**.** Pointcut *join* specifies the joinpoints where the main thread should block, waiting for the spawned threads to terminate (by calling *Thread.join* per each thread started before). All started threads are registered along with the thread that started them, enabling the aspect to relate the current thread to the threads spawned by it. Pointcuts *interrupt* and *interruptAll* use the registration data structure to interrupt threads spawned by the aspect: *interrupt* specifies the joinpoints where all threads created by the current thread should be interrupted and *interruptAll* does the same for all threads created by the aspect.

2.1.1 Annotations

The annotations mechanism of Java 1.5 [15] provides an alternative way to intercept method calls. All methods annotated with the *@Oneway* annotation are intercepted by the aspect OnewayProtocol. Pointcut *onewayMethodExecution* is no longer needed. Definitions of *join*, *interrupt* and *interruptAll* pointcuts can similarly be replaced by annotations.

```
public abstract aspect OnewayProtocol {
   //pointcuts presented before
   //data structure for thread registration

   void around():  onewayMethodExecution(){
      Thread t = new Thread(new ThreadGroup(
         getThreadGroupName()), new Runnable(){
         public void run(){
            //...
            proceed();
            // exception handler logic...
         }
      });
      registerThread(t);
      t.start();
   }
   after() : join() {
      waitForAllSpawnedThreads();
   }
   // Definition of other advice...
}
```

Figure 2: Reusable AO implementation of Oneway

```
public aspect aspect_name extends OnewayProtocol {
   protected pointcut onewayMethodExecution () :
                   <pointcut definition>;
   protected pointcut join() :
                   <pointcut definition>;
   protected pointcut interrupt();
   protected pointcut interruptAll();
   protected String getThreadGroupName();
}
```

Figure 3: Example of the use of One-way

2.2 Futures

Futures are join-based mechanisms based on data objects that automatically block when clients try to use their values before the corresponding computation is complete [17]. Futures allow two-way asynchronous invocations that return a value to the client. In typical situations, a variable stores the result of a computation, which will be used later. Instead of blocking at the computation phase, the thread blocks when the variable is actually accessed. Figure 4 shows the synopsis for the use of futures.

```
public aspect aspect_name
extends FutureReflectProtocol {
  protected pointcut futureMethodExecution(Object
servant):
      <pointcut definition>;
  protected pointcut useOfFuture(Object servant):
      <pointcut definition>;
}
```

Figure 4: Composition of Future behaviour

Pointcut *futureMethodExecution* defines the points where the computation methods are invoked and pointcut *useOfFuture* defines the joinpoints where the result is needed. The thread will block on the joinpoints captured by *useOfFuture*, in case the methods defined in *futureMethodExecution* have not returned.

Figure 5 shows the relevant parts of aspect *FutureReflect Protocol*. The first around advice intercepts invocations of methods and the second around advice intercepts accesses to the returned object. The first advice starts by creating a fake object and an object of type Future where the returned value will be stored. It subsequently creates a new thread to execute the method. The thread stores the returned object in the associated Future object after execution. After the method invocation, the fake object is used in place of the genuine one until an attempt to use it takes place. Method *mapFake2Future* associates fake objects to futures.

```
public abstract aspect FutureReflectProtocol {
   //abstract pointcuts presented before
   //data structure for fake and future registration
   Object around(final Object server) :
         futureMethodExecution (server) {
      fake = //create fake using introspection
      final Future future = new Future();
      Thread t = new Thread(new Runnable() {
         public void run() {
            //...
            future.setValue(proceed(server));
            //exception handler logic
         }
      });
      mapFake2Future(fake, future);
      t.start();
      return fake;
   }
   Object around(Object server): useOfFuture(server){
      Object s = removeFakeFromMap(server);
      if(s != null) server = s; //if server is a fake
      return proceed(server);
   }
}
```

Figure 5: Reusable AO implementation of Future

In the around advice acting on pointcut *useOfFuture*, the fake value is replaced by the real one. If it is not yet available, the client thread blocks until the server thread stores the value in the future. Method *removeFakeFromMap* removes the fake from the map and returns the real one. Method *removeFakeFromMap* also blocks when the real value is not available.

The use of introspection to instantiate fake objects can be avoided by using the aspect *FutureProtocol*. In this case, method *getFakeObject* should be defined in concrete aspects. This solution entails creating a case-specific aspect for each method return type. Types without argumentless constructors must use this solution.

2.3 Barrier

Barrier [17] is a mechanism to set blocking points for a set of threads. Threads reaching such points block until a specified threshold number of blocking threads is reached. Aspect BarrierProtocol (see Figure 6) is subclassed by a concrete aspect, defining inherited pointcuts capturing the joinpoints where threads must synchronise and the threshold number of blocking threads. Two pointcuts may be defined: (1) *barrierAfterExecution* defines events where threads must block, immediately after the execution of methods associated with those events; (2) *barrierBeforeExecution* has a similar purpose, but in this case threads block before the method execution. In many situations, barriers should apply only to a specific set of threads. For that purpose, method *getThreadGroupName* can be redefined to specify the thread group name of the threads where barrier should apply. Concrete aspects must implement method *getNumber Threads* to set the threshold number of blocking threads.

The implementation of Barrier presented in Figure 7 is based on a cyclic barrier [17]. It intercepts the joinpoints defined by the concrete subaspect and blocks all the threads that reach one of the specified joinpoints, until the number of blocking threads reaches the threshold.

```
public aspect aspect_name extends BarrierProtocol {
   //barrier number of threads can be specified by:
   protected int getNumberThreads() {
      return numberOfThreads;
   }

   protected String getThreadGroupName() {
      return threadGroup;
   }

   //and define one of the following:
   protected pointcut barrierAfterExecution():
      <pointcut definition>;
   protected pointcut barrierBeforeExecution () :
      <pointcut definition>;
}
```

Figure 6: Outline of the use of Barrier

```
public abstract aspect BarrierProtocol {
   after() : barrierAfterExecution() {
      //...
      applyBarrier(thisJoinPoint, parameters);
      //exception handling logic
   }
   before() : barrierBeforeExecution() {
      //...
      applyBarrier(parameters);
      //exception handling logic
   }
   protected void applyBarrier(parameters) {
      State s = mapJoinPoint2State(parameters)
      synchronized(s) {
         barrier(s); // barrier implementation
      }
      //...
   }
}
```

Figure 7: Reusable AO implementation of Barrier

2.3.1 Annotations

Element-pair values in annotations allow definition of values that can be used to store the number of blocking threads. Element-pair values named *nThreads* and *threadGroup* should be assigned with the number of threads and thread group where the barrier should apply. For instance, for five threads blocking after method execution associated to thread group *calculus*, the annotation is

```
   @BarrierAfterExecution
      (nThreads = 5, threadGroup = "calculus")
```

2.4 Active Object

Active object decouples the invocation of methods from their execution [17]. Each object runs into its own thread of control. Whenever a client object invokes a method from an active object, the thread associated with the active object carries out the execution. Traditional implementations of active objects are structured into three layers. The first layer includes the object that makes the call, the second layer comprises the mechanisms that forward the call to the target object and in the third layer the target object running in its dedicated thread is continuously waiting for method invocations. The implementation of active objects with aspects moves the second and third layer to an aspect and makes participant classes oblivious of their roles in the pattern. Thus, we can make traditional method invocations on active objects and plug the pattern through the introduction of a marker interface [8] into active object classes, using the intertype declaration mechanism of AspectJ.

To specify active object behaviour, simply add the *ActiveObject* interface to the list of the object class implemented interfaces (see Figure 8).

```
public aspect aspect_name
extends ActiveObjectProtocol {
   declare parents :
      <case-specific class> implements ActiveObject;
}
```

Figure 8: Composition of Active object

Whenever an instance of *ActiveObject* is created, the aspect associates the object to a scheduler object that assumes the role of a communication channel between client threads and active object threads (see Figure 9). Each invocation of an active object method is intercepted by the aspect, wrapped within a *concurrentlib.activeobject.Callable* object and stored in the active object scheduler queue. The active object thread continuously picks requests inserted in the queue and executes them.

```
before(ObjectActive s) : create(s) {
   MQScheduler mqs = new MQScheduler(50);
   synchronized(this){
      hash.put(s, mqs);
   }
   (new Thread(mqs)).start();
}
```

Figure 9: Interception of Active object instantiation

Figure 10 presents an overview of the implementation of active object. The aspect wraps each method invocation into an instance of a *Callable* anonymous class and puts it in the queue through the scheduler (Figure 11). The client thread controls execution.

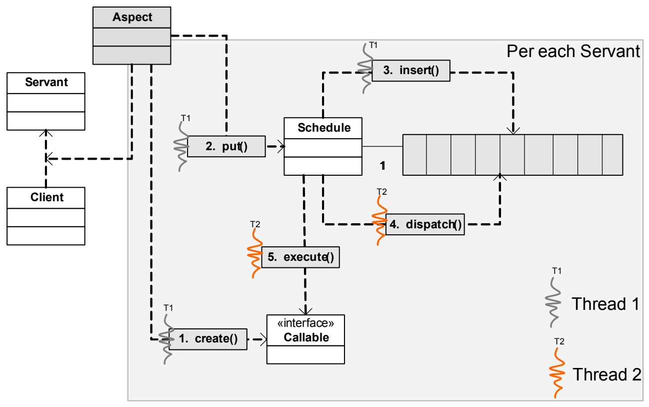

Figure 10: Active object dynamics using AOP

2.4.1 Annotations

Using the *@ActiveObject* annotation in active object classes is equivalent to the *declare parents* clause. Annotated classes are intercepted by the aspect as if they implement the ActiveObject interface.

2.5 Synchronised mechanism

The synchronised mechanism with aspects is straightforward to implement. The synopsis is presented in Figure 12.

```
Object around(final ObjectActive s): getMeth(s){
   MethodRequest request = new MethodRequest(
      new Callable(){
         private Object msg = null;
         public void call(){ msg = proceed(s); }
         public Object getValue(){ return msg; }
      });
   sendToQueue(request,s);
   Object ret = null;
   //exception handler logic
   ret = request.getResult().getValue();
   // exception handler logic
   return ret;
}
```

Figure 11: Method call interception of Active object

```
public aspect aspect_name
extends SynchroniseProtocol {
  protected pointcut
  synchronisedUsingCapturedLock
      (Object targetObject): <pointcut definition>;
  protected pointcut synchronisedUsingSharedLock():
      <pointcut definition>;
}
```

Figure 12: Introduction of the synchronisation mechanism

The *SynchroniseProtocol* aspect wraps the intercepted method execution or variable access into the Java *synchronized* mechanism (Figure 13). *synchronisedUsingCapturedLock* uses the monitor of the target object while *synchronisedUsingSharedLock* uses the aspect monitor to control the access to all captured joinpoints. The second alternative is useful to associate a single lock to multiple type-unrelated objects.

```
public abstract aspect SynchroniseProtocol {
   //Aspect variables...
   //pointcuts...

   Object around(Object targetObject) :
      synchronisedUsingCapturedLock(targetObject){
      synchronized(targetObject) {
         return proceed(targetObject);
      }
   }
   Object around() : synchronisedUsingSharedLock() {
      synchronized(this){
         return proceed();
      }
   }
   //definition of methods and other advices
}
```

Figure 13: AO implementation of the synchronised mechanism

2.5.1 Annotations

The style of quantification shown in Figure 12 can be replaced by *@Synchronised* annotations on method to synchronise. Synchronisation using a shared aspect lock uses an *id* value, which identifies a particular lock in the application, e.g.

```
   @Synchronised(id = "clientTransaction")
```

If *id* is not set, the captured target object lock is used to control the access to annotated methods. Implementation of locks specified through annotation *ids* requires an additional map to associate shared locks to their associated ids.

2.6 Waiting guards

Execution of a particular method may depend on the state of the object. When some precondition is not satisfied, two situations can occur: (1) raise an exception (also known as balking), (2) waiting until the precondition is satisfied. Waiting guards implements the latter policy. When a precondition is not satisfied, the client thread blocks until some action that changes the state unblocks the thread and triggers a precondition re-evaluation. The process is repeated until the precondition is valid. A timeout value can be optionally defined in order to set a waiting time for precondition revalidation. Figure 14 presents the synopsis for the usage of waiting guards.

```
public aspect aspect_name
extends WaitingGuardsProtocol {
   protected pointcut
      deblockingOperation(Object targetObject) :
      <pointcut definition>;
   protected pointcut blockingOperation
   (Object targetObject): <pointcut definition>;

   protected boolean preCondition
      (Object ob, Object[] args) {
         return <precondition validity>;
   }

   //and optionally override method getWaitingTime
   protected long getWaitingTime(){
      return <time in milliseconds>;
   }
}
```

Figure 14: Synopsis of the use of Waiting guards

Pointcut *blockingOperation* specifies joinpoints associated with methods where a precondition is checked. Pointcut *deblockingOperation* specifies methods that may change precondition validity, forcing a re-evaluation. Method *precondition* returns the logical value representing the precondition validity. This method receives two parameters that can be used to retrieve information from the context. Optionally, time over value (representing the maximum waiting time in milliseconds before the precondition revalidation) can be defined by overriding method *getWaitingTime*. The implementation of Waiting Guards (Figure 15) notifies all blocked threads in *deblockingOperations* joinpoints and checks precondition validity before allowing *blockingOperations* joinpoints to proceed.

2.7 Reader/Writer Lock

The synchronisation mechanism allows a single thread to enter in a critical section of code for reading or writing. RW lock differentiates accesses that change object state from the ones that just read state, allowing multiple simultaneous readers but just one writer. Access for reading is allowed when no writers are executing or waiting to access object state whereas writers can access when there are no writing or reading operations executing. One typical use of RW lock is when there are many reads and just few writes.

In order to specify which methods change object state and which ones reads it, four pointcuts may be specified in concrete aspects (Figure 16): pointcuts *readMethodObjectLock* and *readMethodSharedLock* capture executions of reader methods; pointcuts *writeMethodObjectLock* and *writeMethodSharedLock* capture the execution of writer methods. **SharedLock* pointcuts quantify over methods synchronised by a single shared lock. **ObjectLock* pointcuts perform synchronisation using one lock per target.

```
public abstract aspect WaitingGuardsProtocol {
   //...
   protected long getWaitingTime(){ return 0; }

   after(Object ob) : deblockingOperation(ob) {
      synchronized(ob) {
            ob.notifyAll();
      }
   }
   before(Object ob, Object[] parameters) :
         blockingOperation(ob) && args(parameters){
      //...
      synchronized(ob) {
         while(! preCondition(ob, parameters)) {
            ob.wait(getWaitingTime());
         }
      }
      //exception handling logic
   }
}
```

Figure 15: AO reusable implementation of Waiting guards

```
public aspect aspect_name extends RWLockProtocol {
   protected pointcut
      readMethodObjectLock(Object targetObject) :
      pointcut_definition;
   protected pointcut
      writeMethodObjectLock(Object targetObject) :
      pointcut_definition;
   protected pointcut readMethodSharedLock() :
      pointcut_definition;
   protected pointcut writeMethodSharedLock() :
      pointcut_definition;
}
```

Figure 16: Synopsis of the use of RW lock

Implementation of RW lock is shown in Figure 17. Lock management functionality is implemented in class *RWLock*. The appropriate lock type (i.e. Reader lock or Writer lock) is acquired before joinpoint execution and released after execution.

```
public abstract aspect RWLockProtocol {
   // variables and pointcuts referred before
   before(Object targetObject) :
   readMethodObjectLock(targetObject) {
      RWLock lock = mapObjectCaptured2Lock(
                     targetObject);
      readLockAcquire(lock);
   }
   after(Object targetObject) :
   readMethodObjectLock(targetObject) {
      RWLock lock = mapObjectCaptured2Lock(
                     targetObject);
      lock.readLock().release();
   }
   before(Object targetObject) :
   writeMethodObjectLock(targetObject) {
      RWLock lock = mapObjectCaptured2Lock(
               targetObject);
      writeLockAcquire(lock);
   }
   after(Object targetObject) :
   writeMethodObjectLock(targetObject) {
      RWLock lock = mapObjectCaptured2Lock(
                     targetObject);
      lock.writeLock().release();
   }
   //other advice
}
```

Figure 17: AO reusable implementation of RW lock

2.7.1 *Annotations*

@Reader and @Writer can be used to annotate reader and writer methods. Shared RW locks are specified with the id attributed, in a way similar to the synchronised mechanism.

2.8 Scheduler

Synchronisation of Java is inflexible due to its implicitness. Each monitor associated to an object restricts scheduling of threads in waiting state to monitor implementation. *Scheduler* (Figure 18) allows specification of a scheduling order. Each thread attempting to execute a scheduled method blocks until the mechanism wakes it. By default, scheduling order is FIFO. An order can be specified or other scheduling policy can be implemented (e.g. based on state) by redefining method *selectRunningThread*. Pointcut *scheduledMethodExecution* specifies joinpoints where scheduler synchronisation must apply (Figure 19).

```
public aspect aspect_name extends SchedulerProtocol{
   protected pointcut
   scheduledMethodExecution(Object targetObject) :
      pointcut_definition;
   protected int selectRunningThread (ArrayList th){
      return <next element position>;
   }
}
```

Figure 18: Synopsis of the use of Scheduler

```
public abstract aspect SchedulerProtocol {
   //variables and pointcuts referred before
   Object around(Object targetObject) :
         scheduledMethodExecution(targetObject) {
      //...
      try{
         enter(thisJoinPoint);
         return (proceed(targetObject));
      } catch(InterruptedException e) {
         //exception handler logic
      } finally{
         done(thisJoinPoint, targetObject);
      }
   }
   public void enter(String textJoinPoint) /*...*/ {
      Thread thisThread = Thread.currentThread();
      mapJoinPoint2ThreadSet(/*...*/);
      synchronized(thisThread){
         while(thisThread != runningThread)
            thisThread.wait();
      }
      removeRequest(textJoinPoint);
   }
   public synchronized void
   done(String textJoinPoint, Object targetObject) {
      //...
      ArrayList<Thread> arr =
            waitingThreads.get(textJoinPoint);
      if(arr.size()==0) runningThread = null;
      else { //else determines next request
         runningThread= arr.get(selectRunningThread(
                              arr, targetObject));
         synchronized(runningThread) {
            runningThread.notifyAll();
         }
      }
   }
   // Returns the position of next running thread
   protected int selectRunningThread(parameters)
      //thread execution order logic
   }
   //definition of other methods and advices
}
```

Figure 19: AO reusable implementation of Scheduler

Method *selectRunningThread* gets the reference for the Thread queue and the reference of the intercepted object as parameters. Like other synchronisation mechanisms previously presented, each Scheduler instance can be used to synchronise accesses to joinpoints from many classes using a single lock.

2.8.1 *Annotations*

Scheduler can be applied with annotations, but only FIFO and LIFO scheduling orders are allowed. More application-specific situations require the redefinition of *selectRunningThread*.

3. ILLUSTRATIVE EXAMPLES

This section illustrates the usage of several mechanisms presented in previous section.

3.1 Water Tank

WaterTank[17] illustrates the use of Active Object, RW lock and Waiting Guards. It shows how to apply several refinements of the same aspect to a particular tank instance, as well as applying reuses of different aspects. This subject is further discussed in section 4.1.

Instances of WaterTank (Figure 20) have a preset capacity and volume, as well as public operations *addWater* and *removeWater*. WaterTank methods can be classified as *readers* or *writers*. RW Lock is plugged to WaterTank by way of aspect RWLockWaterTank, which subclasses *RWLockProtocol* (Figure 21). Plugging the aspect allows simultaneous reads (e.g., get* calls), but just one writer (either *addWater* or *removeWater*).

```
public class WaterTank {
   private float capacity, currentVolume;
   //constructors
   public void addWater(float amount) {
      //add the water to tank
   }
   public void removeWater(float amount) {
      //remove the water from the tank
   }
   //getter/setter methods
}
```

Figure 20: WaterTank class

```
public aspect RWLockWaterTank
extends RWLockProtocol {
   protected pointcut readMethodObjectLock(/*...*/):
      execution(* *.get*(..)) &&
      this(targetObject);
   protected pointcut writeMethodObjectLock(/*..*/):
      (execution(* *.addWater(..)) ||
      execution(* *.removeWater(..))) &&
      this(targetObject);
}
```

Figure 21: Water tank with RW lock

Aspects *OverFlowWaitingGuard* and *UnderFlowWaitingGuard* are used to avoid WaterTank overflow or underflow. The aspects ensure that threads executing methods that lead to an invalid state are blocked until a state change enables each blocked thread to re-evaluate its condition. The former is shown in Figure 22; the latter is similar. The aspects reuse waiting guards logic implemented in *WaitingGuardsProtocol*. Both aspects rely on parameter *amount* used by modifiers *addWater* and *removeWater* and on getters *getCurrentVolume* and *getCapacity*.

A second implementation of WaterTank illustrates the use of active objects. Each instance of WaterTank class is an active object through the inclusion of a concrete aspect that subclasses *ActiveObjectProtocol* (Figure 23). Each class instance which implements interface ActiveObject is intercepted by the aspect *ActiveObjectProtocol* and all the inherited structure is created per each object. Clients of that object are oblivious of their participation in the active object role.

```
public aspect OverFlowWaitingGuard
extends WaitingGuardsProtocol {
   protected pointcut blockingOperation(/*...*/):
      call(* WaterTank.addWater(..)) &&
      target(targetObject);

   protected pointcut deblockingOperation(/*...*/):
      call(* WaterTank.removeWater(..)) &&
      target(targetObject);

   protected boolean preCondition(arguments) {
      WaterTank wt = (WaterTank) targetObject;
      Float amount = (Float) args[0];
      return (wt.getCurrentVolume() + amount) <=
             wt.getCapacity();
   }
}
```

Figure 22: Water tank overflow waiting guard

```
public aspect ActiveObject
extends ActiveObjectProtocol {
   declare parents :
      WaterTank implements ActiveObject;
}
```

Figure 23: Water tank enhancement with Active Object behaviour

3.2 Fibonacci

The well-know recursive Fibonacci function illustrates the use of future calls to introduce concurrency into fork/join applications. Figure 24 presents a Java implementation of a sequential Fibonacci class. Figure 25 presents the concrete future aspect for Fibonacci. It includes the conditional pointcut designator *if*, to limit the number of parallel calls.

```
public class Fibonacci {
   public long value;
   Fibonacci(long val) {  value = val; }
   public Long compute() {
      if (value <=1) return(value);
      else {
         Fibonacci f1 = new Fibonacci(value-1);
         Fibonacci f2 = new Fibonacci(value-2);
         Long r1 = f1.compute();
         Long r2 = f2.compute();
         return (r1+r2);
      }
   }
   public static void main(String args[]) {
      Fibonacci fibo = new Fibonacci(12);
      Long result = fibo.compute();
      System.out.println("Fibonacci result :" +
                           result.longValue());
   }
}
```

Figure 24: A Java implementation of Fibonacci

3.3 Particle applet

Particle applet [17] is a toy example based on movable bodies controlled by threads that randomly change their locations. Class *Particle* (Figure 26) maintains the state of each particle (fields *x* and *y*) and provides managing behaviour. Class *ParticleApplet* (Figure 27) shows the movement of particles inside an applet viewer: a set of squares, each one representing one particle of the set. Class *ParticleCanvas* contains references to all particles. Whenever method *paint* is invoked, it calls *draw* on every particle.

```
public aspect FutureFibonacci
extends FutureReflectProtocol {
   protected pointcut futureMethodExecution (
              Object servant) :
      call(Long Fibonacci.compute()) &&
      if(((Fibonacci) servant).value>8) &&
      target(servant);
   protected pointcut useOfFuture(Object servant) :
      call(* Long.longValue()) &&
      target(servant);
}
```

Figure 25: Implementation of Fibonacci with futures

```
public class Particle {
   //fields x and y, constructors

   public synchronized void move() {
      x += rng.nextInt(10) - 5;
      y += rng.nextInt(20) - 10;
   }
   public void draw(Graphics g){ draw rectangle }
}
```

Figure 26: Particle class

```
public class ParticleApplet extends Applet {
   // null when not running
   protected Thread[] threads = null;

   //...
   protected Thread makeThread(final Particle p) {
      //utility
      Runnable runloop = new Runnable() {
      public void run() {
         //...
            for(;;) {
               p.move();
               canvas.repaint();
               //...
            }
         ...// exception handling
      }};
      return new Thread(runloop);
   }
   public synchronized void start() {
      int n = 10; // just for demo
      if (threads == null) {
         Particle[ ] particles = new Particle[n];
         for (int i = 0; i < n; ++i)
            particles[i] = new Particle(50, 50);
         canvas.setParticles(particles);
         threads = new Thread[n];
         for (int i = 0; i < n; ++i) {
            threads[i] = makeThread(particles[i]);
            threads[i].start();
         }
      }
   }
   public synchronized void stop() {
      if (threads != null) {
      for (int i = 0; i < threads.length; ++i)
         threads[i].interrupt();
         threads = null;
      }
   }
}
```

Figure 27: Particle Applet

In its original form [17], Particle contains concurrency code unrelated to the core logic (presented shaded). We can remove the various occurrences of the *synchronized* keyword by reusing aspect Synchronisation (Figure 28) and localise thread management (including spawning) in *OnewayProtocol* (Figure 29). *onewayMethodExecution* requires extracting method *makeThread* to *moveParticle* (Figure 31). Pointcut *interruptAll* is defined to interrupt all spawned threads when method *stop* is executed.

```
public aspect Synchronisation
extends SynchroniseProtocol {
   protected pointcut synchronisedUsingCapturedLock(
                  Object capturedLock):
      (execution(* ParticleApplet.start(..)) ||
      execution(* Particle.move(..))) &&
      this(capturedLock);
}
```

Figure 28: Synchronisation particles aspect

```
public aspect Oneway extends OnewayProtocol{
   //...
   protected pointcut onewayMethodExecution() :
      execution(* ParticleApplet.moveParticle(..));
   protected pointcut interruptAll() :
      execution(* ParticleApplet.stop(..));
}
```

Figure 29: Oneway particles aspect

To implement a step-wise movement among particles a barrier can be introduced after each movement (see Figure 30). Threads are unblocked when the last thread reaches the intended joinpoint.

```
public aspect Barrier extends BarrierProtocol{
   //...
   protected pointcut barrierAfterExecution() :
      execution(* Particle.move());
   protected int getNumberThreads(){ return 10; }
   protected String getThreadGroupName() {
      return "ParticleMover";
   }
}
```

Figure 30: Example of the use of Barrier

This application was selected for its intrinsically concurrent behaviour. There are situations where sequential code alone does not make sense by itself. For instance, if we move elsewhere the concurrency code shown in Figure 27, method *makeThread* does not make sense unless the aspect that intercepts the method is taken into consideration as well. If we use annotations to tag that method, we can modularise concurrency code. Figure 31 shows method *makeThread* devoid of concurrency (the method name was changed due to semantics). By tagging the method with annotation @Oneway, the aspect will intercept it and create a new thread per each method invocation, thus avoiding the need to define pointcut *onewayMethodExecution*. The annotation makes it clear that the method will be executed asynchronously in another thread. A similar situation occurs in method *stop*, since it becomes empty when we remove concurrency-related code.

```
@Oneway
protected void moveParticle(final Particle p) {
   for(;;) {
      p.move();
      canvas.repaint();
   }
}
```

Figure 31: Use of annotations to represent the intention of a given method and to simplify quantification by aspects

4. DISCUSSION

This section discusses benefits, limitations and performance trade-offs of using AOP-based implementations to develop concurrent applications.

4.1 Reusability

Table 1 presents a summary of presented mechanisms, referring to the granularity of the quantification used, the possible use of annotations, composition transparency [10] and the possibility to intercept multiple participant classes with a single aspect instance. All mechanisms except Active Object support method-level quantification. Active Object is restricted to class-level quantification as it represents a class role with an associated behaviour. Field-level quantification can be used whenever we want to bind some implementation mechanism to the access to a field, e.g. Scheduler, Synchronisation and Barrier.

Table 1. Analysis of mechanisms/patterns using joinpoint granularity (method, field and class levels), quantification of annotations (QA), composition transparency (CT) and multiple participant classes interception (MPC) criteria

		One-way	Futures	Barrier	Active Object	Synchronisation	Waiting guards	RW lock	Scheduler
Granu-larity	Class	--	--	--	X	--	--	--	--
	Method	X	X	X	--	X	X	X	X
	Field	-	--	X	--	X	--	--	X
Analisys Criteria	QA	X	--	X	X	X	--	X	X*
	CT	--	--	X	--	X	X	X	X
	MPC	--	--	X	--	X	X	X	X

* Possible but with restrictions

Some problems may require multiple instances of a mechanism or pattern capturing the same joinpoints. Such situations do not cause problems when aspect instances do not interfere with each other. The implementations presented here enable composition transparency whenever compositions are valid in equivalent Java implementations. However, discussion of issues related to composition of mechanisms is out of the scope of this paper.

Barrier, Synchronisation, RW lock and Scheduler are able to include instances of multiple, type-unrelated classes in the context of a single mechanism. This can be useful in many situations, e.g. creating locks involving many classes or implementing a barrier with multiple, optional synchronisation points. In many situations, waiting guards depend on context in order to validate a precondition. In such situations, it may not be straightforward to define a precondition that uses state from multiple instances of various unrelated classes. Even when possible, handling multiple instances of multiple classes can be tricky, involving identification of intercepted objects according to their type and using different code to access their state.

4.1.1 Use of Annotations

The use of annotations can be useful and sometimes essential in situations where concurrency is intrinsic to the situation at hand.

In that case, if we modularise concurrency the functional code may be misunderstood. Therefore, annotations have two basic roles: describe the concurrent behaviour of a method and provide a hook for aspects to compose. By using annotations as attributes describing some property or some role of an element (i.e. class, method or class fields), aspects become more independent from element syntax. Consequently, changes on class names, field names or method signatures do not cause changes on aspects that intercept such properties.

Some pointcuts require the implementation of methods (e.g. precondition method from waiting guards). Those mechanisms cannot be fully implemented with annotations, as aspect concretisation entails more than simply intercepting annotated elements. Element-value pairs can be used with annotations to define configuration values. Such values are typically returned by anchor methods implemented in the concrete aspect, e.g. method *getNumberThreads* implemented in Barrier passes to the aspect the number of threads that must synchronise at a given point. Such information can be defined at annotation level.

Though annotations reduce aspect decoupling from intercepted classes, they suffer from non-locality, which is one of the problems that aspects are supposed to solve. Thus, capture of what methods are consumers of the implemented concern is explicit but scattered across multiple modules. This problem is analysed in [15].

4.1.2 Known limitations

Most limitations of our collection relate to limitations of AspectJ in obtaining local joinpoint context information. This is partly due to the fact that abstract pointcuts typically preset the context information that is captured. This can be a problem for aspects that manage multiple instances of the mechanism they implement, because it can be tricky to distinguish between instances. Several of the implementations overcome the problem by resorting to reflection to obtain the needed information. This solution yields maximum reusability but pays a price in performance (see section 4.2). In cases in which performance degradation is not acceptable, the alternative is to define a separate concrete aspect for each instance of the mechanism. The latter solution yields better performance at the expense of reusability.

Reusable aspects are global entities that cannot distinguish among specific class instances. For instance, it is not possible to apply a barrier to specific threads. We partially overcame this limitation by enabling a barrier to apply to a specific thread group. To do that, each thread spawned by one-way mechanism can be associated to a specific thread group.

Futures can only be applied to methods with non-primitive return types. This is required to allow the aspect to return a fake object instead of a real one. Furthermore, it intercepts all accesses, defined in pointcut *useOfFuture*, not just accesses to instances returned by future calls. This is required to verify if the value is a future value and may have an impact on performance. However, it can be minimised by limiting the scope of the pointcut.

4.2 Performance

We classify overheads in our implementation into three classes:

1. retrieve of joinpoint context;
2. management of global joinpoint history;
3. object and aspect overheads.

Some of reusable aspects must be able to capture specific joinpoint context information (e.g., the reusable barrier can be used to simultaneously manage several barriers). Joinpoint context can be retrieved by using *thisJoinPoint** values from the AspectJ API and *Thread.currentThread*, or can be passed explicitly in pointcut designators *this* or *target*. The aspect must manage history of each joinpoint context, maintaining data structures for each joinpoint context. This can be performed through a collection that associates each joinpoint context to particular data structures. Moving code to an aspect also introduces overheads related to aspect instantiation and management, additional objects and method calls. This is a cost due to higher modularisation.

To measure the impact of these overheads we updated the Java Grande Forum (JGF) multithreaded benchmarks [28]. These provide low-level benchmarks to measure the overhead of barriers, synchronised methods and fork/join of threads, using concurrent programming Java mechanisms (CPJ). We developed similar benchmarks for each pattern/mechanism presented in the paper and not included in the original JGF benchmark, and for the AO versions using our collection. Thread spawning benchmarks (e.g., Oneway, Futures and Active Objects) perform a short non-trivial calculation on a separate thread. Synchronisation benchmarks spawn several threads accessing a single shared counter, which implements the benchmarked synchronisation policy. We followed the JGF benchmark style as close as possible, even if sometimes it was not the most natural AO way.

Table 2 presents the overheard percentage, calculated through the formula (CPJ–AOP)/CPJ. CPJ is concurrent Java style implementations and AOP our AO implementations. We collected results using 1, 4, 16 and 64 threads. Presented values are median of 5 executions and were collected on two unloaded machines, both with Sun JDK 1.5.0_3 and AJDT 1.3.0: a AMD Athlon XP 1800+, 512 MB RAM DDR 266, running Windows XP, and a Intel Dual Xeon 3.2GHz (with Hyper-thread enabled), 1MB L2, 1 GB RAM DDR2 400, running CentOS 4.1. For reference, we provide the CPJ number of operations per second using 4 threads on both machines. Table 3 presents the relative cost, in percentage of the total cost, for each implementation. The results were obtained by developing specialised aspects for each benchmark, removing each cost component, and represent the average on these two machines (measurements among machines are close enough to just be relevant to present average values).

Table 2. Overhead of reusable AO implementations relative to CPJ implementations, using 1, 4, 16 and 64 threads, on an Athlon XP and on a Dual Xeon.

	Athlon XP				**Dual Xeon**				**Operations /s**	
	1	**4**	**16**	**64**	**1**	**4**	**16**	**64**	**XP**	**Xeon**
One-way	7	20	17	12	15	15	12	10	6K	12K
Futures	20	31	28	29	13	12	6	4	6K	12K
Barrier	93	45	39	41	88	24	24	27	200K	100K
Active Object	64	44	46	43	9	6	2	6	100K	100K
Synchronisation	2	17	6	13	9	9	8	1	600K	2M
Waiting Guards	19	26	16	18	31	47	44	47	400K	700K
Readers/Writers	0	36	33	31	4	33	29	29	1M	300K
Scheduler	78	63	61	55	8	13	13	15	100K	70K

The implementation of Barrier, Waitings guards and Scheduler use *thisJoinPoint** variables to retrieve jointpoint context information. In these implementations, context retrieval introduces a significant part of overheads (see Table 3). Barrier execution with a single thread betrays an unusually high overhead because the thread never blocks in the barrier. Waiting guards also have significant object/aspect overheads, since the AO version introduces additional locks and notifications. We can say that this cost is due to higher modularity and not a cost of moving to reusable implementations.

One-way uses *Thread.currentThread* to retrieve the running thread and a *Hashmap* of *Hashmaps* to maintain relationships among creator and created threads. These are relatively low-cost functions when compared to thread management cost. Futures have higher overheads: they rely on reflection to identify and instantiate method return values (joinpoint context cost) and use a *Hashmap* to manage futures.

Table 3. Relative cost, in percentage of total cost, in AspectJ implementations.

	One-way	Futures	Barrier	Active Object	Synchronise	Waiting Guards	Readers/Writers	Scheduler
joinpoint Context	8	48	83	--	--	40	--	88
Global Execution History	27	52*	14	47	--	--	92	9
Aspect/Object Overheads	65		3	53	100	60	8	3

* It was not possible to separately measure global execution history and aspect/object overheads

Active object requires a global history to associate each intercepted object to the corresponding scheduler (described as part of the active object pattern). As the pattern is applied to the object (captured with pointcut designator *target*) it does not need joinpoint-specific information. Readers/Writers uses a *WeakHashMap* to associate each object to a RW lock, which accounts for most of overheads on these machines. We did not used *pertarget* aspects in Readers/Writers because the AJDT *pertarget* implementation is not thread safe. Synchronisation does not require context information and history as the aspect uses the target object mutex and monitor, which is a pointcut parameter.

It should be noted that the above are worst-case scenarios (i.e., most real cases are likely to yield better performance) and that no attempts were yet made to optimise the implementations presented.

Access to the joinpoint history must be synchronised (this overhead was also included in global history execution overhead), which seems to be the main constraint to scalability. Joinpoint context and execution history costs can be reduced by creating case-specific aspects for each context, or whenever the mechanism applies to a single object, *pertarget/perthis* can be applied to create an aspect instance per each intercepted object. However, this is not feasible for most of these implementations. As an example, the barrier implementation would require a *perjoinpoint* association (not currently supported in AspectJ).

In all benchmarks, AO implementations enable unpluggability of concurrency mechanisms. This helps to validate the benchmark code itself, by comparing execution times with and without concurrency, ensuring that they are measuring the concurrency mechanisms overhead.

4.3 High-level OO concurrent approaches

Our collection provides an AO approach to develop concurrent applications. Use of this collection combined with plain Java (without Java concurrency constructs) is a replacement for a high-level OO concurrent programming language. As an example, ABCL [30] provides active objects, one-way calls and futures. These high-level abstractions for concurrency can be replaced by our equivalent AO implementations. Even express messages can be implemented using the scheduler pattern to provide higher priority to specific message types.

Our approach is to treat Java as a core language and concurrency issues as additional language features. This approach has two main benefits. First, traditional support to concurrency features degrades performance even when the features are not used, e.g., in Java, each object implements a monitor even when concurrency features are not used in the program. A similar benefit was observed in middleware systems, where a performance improvement was attained by extracting non-core features to aspects [31]. Second, treating concurrency issues as additional features – modelled by aspects – helps to develop concurrent applications where concurrency issues are more modular and can be unplugged. This approach eases application development, since the concurrency can be added in a later stage of development and can be unplugged for debugging purposes.

Implementing equivalent high-level concurrency constructs using traditional OO approaches requires a significant amount of effort. One example is the transparent implementation of futures (e.g., without following a library approach of Java 5), in a similar way to several high-level OO concurrent languages. Traditional approaches require analysis and transformation of base code, which entails parsing of method calls, return values and consequent use of these return values. An AO implementation significantly reduces implementation effort. Furthermore, the programmer can use this mechanism as almost if it were built in the core language.

5. RELATED WORK

In [10], Hannemann and Kiczales present implementations in Java and AspectJ of the 23 Gang-of-Four patterns and provide an analysis. In their concluding remarks, they mention several directions for further experimentation, including applying AspectJ to more patterns. In this paper, we provide a contribution in that direction.

Some of concurrency mechanisms presented here were previously implemented with aspects. JBoss AOP [2] and AspectJ 5 Developer's Notebook [1] use the *@Oneway* annotation in void methods to specify execution in separate threads. However, additional thread functionality such as *join*, *interrupt*, *sleep* and definition of some parameters (e.g. thread group) is not implemented. An aspect-based implementation of Future is presented in [3] that modularises thread spawning code within an aspect. However, management of futures must be written in the classes that use them. AO implementations of RW lock are presented in [7][16]. These implementations only support a lock per object and do not support annotations. Our work provides a collection including the most well known mechanisms, modularises the implementation of such mechanisms. In addition, we support both annotation and traditional model whenever possible and provide several parameters to configure each mechanism (e.g. specify the thread group in Barrier and Oneway).

OpenMP [4] model uses annotations to express concurrency issues, in a way similar to our annotations. Both approaches support the development of concurrent applications where concurrency is specified through code annotations that can be ignored when concurrency is unplugged. As an example, OpenMP's *parallel for* annotations resemble *@Oneway* annotations and OpenMP's *critical* annotations are similar to *@Synchronised* annotations. Our annotations are OO based, providing less flexibility and we do not provide some features of OpenMP, such as loop scheduling. On the other hand, we provide a more high-level way to structure applications, by leveraging OO annotations. In addition, we include high level mechanisms not present in OpenMP, namely futures and RW locks.

In [6], Bergmans extends the Composition Filters (CF) model specifically to deal with concurrency and synchronisation issues. Bergmans introduces the wait filter, which either accepts a message and forwards it to the next filter or stores it in a queue where it remains blocked until the message can be accepted (they work like a waiting guard for methods). Wait filters are specified at the interface level and implementations of core logic are oblivious to them. The CF model with wait filters achieves a modularisation level for the concurrency concern similar to that of our collection. Synchronisation granularity is limited to that of messages and is therefore more limited. On the other hand, it seems to be more expressive, as it allows composition of synchronisation constraints. In the general case, that is not the case with the aspects presented in this paper.

Lopes proposes the use of aspect-specific languages to deal with issues related to thread synchronisation and application-level data transfers over remote method invocations [19]. Use of specialised languages has the advantage of minimising the gap between the intentions of the programmer and the representation in source code of those intentions. However, it also requires a significant upfront investment in designing and implementing the aspect-specific language, as well as developing a specialised weaver for the aspect language. By Lopes' own admission, the approach is not scalable [18]. By contrast, our approach leverages the capabilities of a relatively mature AOP general-purpose language, thus avoiding that particular disadvantage.

Reflective systems (or meta-level architectures) allow the programmer to change the system behaviour by providing access to the meta-architecture. Examples of such concurrent systems are ABCL/R3 [21] and MPC++ [12]. In [20], several parallel constructors were implemented at a meta-level, using ABCL/R3. These included object replication, pre-fetch and method scheduling. Object replication and pre-fetch rely on annotations in the base program, similar to AspectJ annotations. The other mechanisms are completely and transparently implemented at meta-level. Reflective systems were the roots of AOP; however, instead of using run-time reification, AO languages perform compile-time weaving, which lead to higher efficiency. AspectJ does not seem to allow the same flexibility as meta-level approaches to change the system behaviour.

In [11], Harbulot and Gurd use AspectJ in an attempt to separate the core functionality from parallelisation issues. Several experiments were made, on the basis of various parallel benchmarks, to move thread-related code and message-passing code into aspects. Harbulot and Gurd conclude that most of parallel applications require refactorings to take advantage of AO approaches, as parallel code is not generally developed in an OO manner. Harbulot and Gurd place all code related to concurrency issues in a single aspect, without further structuring. Our work achieves a similar goal through a collection of aspects dealing with similar issues, which lead to a higher level of modularity and reuse.

6. FUTURE WORK

An obvious development of our work is the update of our collection to leverage the new Java 5 concurrency mechanisms, providing optimised implementations and additional mechanisms, such as executors.

We intend to test this collection using concurrent applications of non-trivial dimensions. Such experiments are expected to provide directions for further improvements and hints on how to refactor current concurrent applications to use this type of reusable libraries. We also plan to test how effectively the collection addresses the inheritance anomaly problem [22].

A longer-term goal includes finding new ways to increase context information available to abstract pointcuts and to overcome limitations of our collection.

7. CONCLUSION

This paper presents a collection of AO implementations of several of the most well known high-level concurrency mechanisms and patterns, namely one-way calls, futures, waiting guards, readers/writers, barriers and active objects. Use of AspectJ enables the development of reusable implementations of the aforementioned mechanisms. In the cases when it makes sense to (un)plug the mechanism, the implementations enable such unpluggability and do not introduce additional overhead when aspects are not included in the build. Moving concurrency code to aspects can contribute to making the core logic of applications more understandable.

AspectJ has limitations in obtaining local joinpoint context information, partly due to the fact that abstract pointcuts typically preset the context information that is captured. We identify this as the main restriction imposed by AspectJ to build a collection of reusable aspect for concurrent programming.

Most of the implementations seem to benefit from the use of annotations. Annotations make the intentions of the programmer more explicit and have the potential to make the code base more amenable to quantification by aspects.

ACKNOWLEDGEMENTS

We thank Hidehiko Masuhara for numerous suggestions and feedback. Three anonymous referees made valuable suggestions that helped to improve the paper. This work was partially supported by PPC-VM project POSI/CHS/47158/2002, funded by Portuguese *Fundação para a Ciência e Tecnologia* (FCT) and by

European funds (FEDER). Miguel Monteiro is partially supported by FCT under project SOFTAS (POSI/EIA/60189/2004).

REFERENCES

[1] The AspectJ 5 Development Kit Developer's Notebook, http://eclipse.org/aspectj/doc/next/adk15notebook/

[2] JBoss AOP web site. http://www.jboss.org/products/aop.

[3] Colyer A., Making concurrency a little bit easier, blog entry, January 2005, www.aspectprogrammer.org/blogs/adrian/2005/01/making_concurre.html

[4] OpenMP architecture review board, OpenMP Application Program Interface, Version 2.5, May 2005, www.openmp.org

[5] Andrews, G., Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley, 2000.

[6] Bergmans, L. M. J., Composing Concurrent Objects: Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs, Ph.D. thesis, University of Twente, Netherlands, June 1994.

[7] Colyer A., Clement A., Harley G., Webster M., Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools, Addison Wesley 2004.

[8] Grand, M., Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, 2nd Edition, Wiley, 1998.

[9] Hanenberg, S., Schmidmeier, A., Unland, R., AspectJ Idioms for Aspect-Oriented Software Construction, 8th EuroPLoP, Irsee, Germany, June 2003.

[10] Hannemann, J., Kiczales, G., Design Pattern implementation in Java and in AspectJ, OOPSLA 2002, Seattle, USA, November 2002.

[11] Harbulot, B., Gurd, J., Using AspectJ to Separate Concerns in Parallel Scientific Java Code, AOSD 2004, Lancaster, UK, March 2004.

[12] Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte J., Tezuka, H., Konaka, H., Maeda, M., Kubota, K., Design and Implementation of Metalevel Architecture in C++ - MPC++ Approach, Workshop on Reflection and Metalevel Architecture (Reflection'96), San Francisco, CA, April 1996.

[13] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G., An Overview of AspectJ. ECOOP 2001, Budapest, Hungary, Springer Verlag LNCS vol. 2072, pp. 327-353, June 2001.

[14] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J. Aspect Oriented Programming, ECOOP'97, Jyväskylä, Finnland, June 1997.

[15] Kiczales, G., Mezini, M., Separation of Concerns with Procedures, Annotations, Advice and Pointcuts, ECOOP'05, Glasgow, UK, July 2005.

[16] Laddad, R., AspectJ in Action – Practical Aspect-Oriented Programming, Manning 2003.

[17] Lea, D., Concurrent Programming in Java, Second edition, Addison-Wesley, 1999.

[18] Lopes, C. V., AOP: A Historical Perspective (What's in a Name?). In Aspect-Oriented Software Development, pages 97–122. Addison-Wesley, 2005.

[19] Lopes C. V., D: A Language Framework for Distributed Computing, Ph.D. thesis, College of Computer Science, Northeastern University, Boston, USA, November 1997.

[20] Masuhara, H., Matsuoka, S., Yonezawa, A., Implementing Parallel Language Constructors Using a Reflexive Object Oriented Language, Workshop on Reflection and Metalevel Architecture (Reflection'96), San Francisco, USA, April 1996.

[21] Masuhara, H., Yonezawa, A., Design and Partial Evaluation of Meta-Objects for a Concurrent Reflective Language, ECOOP'98, Brussels, July 1998.

[22] Matsuoka S., Yonezawa A., Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In Research Directions in Concurrent Object-Oriented Programming (Agha G., Wegner P., et al., editors), pp. 107-150, MIT press, 1993.

[23] McHale C., Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance. Ph.D. thesis, Department of Computer Science, Trinity College, Dublin, Ireland, October 1994.

[24] Nordberg III, M. E., Aspect-Oriented Dependency Management, In Aspect-Oriented Software Development, pages 557–584. Addison-Wesley, 2005.

[25] Oaks, S., Wong, H., Java Threads, 3rd edition, O'Reilly 2004.

[26] Silva A. R., Concurrent Object-Oriented Programming: Separation and Composition of Concerns using Design Patterns, Pattern Languages, and Object-Oriented Frameworks, Ph.D. thesis, Technical University of Lisbon, March 1999.

[27] Schmidt, D. C., Stal, M., Rohnert, H., Buschmann, F. (eds), Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley & Sons 2000.

[28] Smith, A., Bull, J., Obdrzálek, J. A Parallel Java Grande Benchmark Suite, Supercomputing 2001, Denver, November 2001.

[29] Yonezawa, A. Tokoro, M. (ed). Object-Oriented Concurrent Programming, MIT Press, 1987.

[30] Yonezawa. A., ABCL: an Object-Oriented Concurrent System, MIT Press, 1990.

[31] Zhang, C., Jacobsen, H., Resolving Feature Convolution in Middleware Systems, OOPSLA'04, Vancouver, Canada, October 2004.