Abstract

Presents case studies for, and implementations of, synchronized block join point that together augment the capabilities of join points for synchronized methods in intercepting and modifying synchronization actions in distributed, Java-based, aspect oriented software. The models are applicable in any aspect oriented environment, but emphasis is placed on compatibility with AspectJ.

The power of such a join point is first illustrated by analysing some case studies. While some such examples can be handled using aspects that intercept synchronized method calls, a fully general scheme needs to deal with Java synchronized blocks. The approach for recognising the synchronized block uses context exposure to provide full control of the thread behaviour when many threads compete to be executed. The proposed join point model is enhanced with a mechanism for removal of unnecessary synchronization, which is vital for reducing overheads associated with the lock. There is also a facility for re-introducing necessary synchronization that has previously been removed.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features

General Terms Design, Language.

Keywords Aspect-Oriented Programming, Synchronized Block Join Point, Synchronization, AspectJ.

1. Introduction

The landscape of parallel computing has drastically changed in the last years due to wide availability of parallel processing capabilities in every desktop machine. Moreover, grid systems connect world-wide computer resources that offer an almost unlimited computing power. This demands an important shift in the way that software is designed and developed. To take advantage of recent and future hardware, applications must be re-designed to be structured along parallel activities that can leverage these intrinsically parallel platforms. In addition, many decades of research on parallelizing compilers did not produce an approach competitive to hand-coded parallelization.

Synchronization is a concern that developers of parallel computing must deal with whenever guarded access to a shared resource is required. Access to devices, files and shared memory are all situations that typically require synchronization, and they also often require careful management of multiple threads and synchronization devices such as locks. Avoiding tangling of the code responsible for these various concerns is difficult, as is encapsulating them for reuse in diverse situations. Moreover, parallel programming is often difficult simply due to the complexity of dealing with lock-based synchronization. As a result, there have been proposals to simplify parallel programming [17, 8] by using various forms of transactional memory [14] to replace lock-based synchronization in existing parallel Java programs.

Aspect-oriented Programming (AOP) [13] has the potential to modularise such synchronizations so that user code can become an oblivious pattern to the distributed environment. Indeed, a join point based on the synchronized method has been discussed [5]. This paper foremost shows how to design and implement models for a synchronized block join point (SBJP) and a synchronized block body join point (SBBJP) to encapsulate crosscutting synchronization concerns into logical units using the concepts of AOP. It provides just one join point for a synchronized block, precisely, associated with a synchronized block body join point for the inside part of a synchronization block. The models achieve reusability of synchronized code and thread control management in Java to such an extent that concurrency can be fully handled by a single aspect.

The paper is organised as follows. Section 2 presents the SBJP and SBBJP models and the kinds of behaviour they aim to recognise. Section 3 presents case studies and shows how to write aspects for distributed synchronization and thread control using the proposed join points. Section 4 discusses the performance of different schemes for re-entrant checking, and also compares the work with an alternative approach [5]. Conclusions are drawn in Section 5.

2. Synchronized block join point model

This section describes the objective of the SBJP and SBBJP models. The behaviour that the models aim to recognise is defined, as well as their dynamic characteristics, and they are compared to their counterpart, the synchronized method. The proposed join points are provided with an execution context. The models could in principle be applied to diverse aspect-oriented systems, but the presentation focuses on AspectJ.

A synchronized method locks the monitor associated with the instance of the class (or the class itself, if the method is static). A synchronized block can lock any monitor and can have a scope smaller than that of the enclosing method. Whereas a synchronized
method can be picked out by the existing method call pointcut\(^1\) in AspectJ using the `synchronized` keyword; selection of a synchronized block has not been supported in AspectJ to date. A synchronized block is recognised at bytecode level as a matched (or not) pair of `entermonitor` and `exitmonitor` bytecode instructions. Because an `entermonitor` can be statically (but not dynamically) paired with more-than-one `exitmonitor`, identifying the appropriate matched pair is non-trivial.

A new mechanism, `rm_proceed()`, is provided to allow the programmer to remove unwanted synchronization from synchronized blocks; this helps to eliminate unnecessary synchronization instructions, thereby reducing locking overheads. The exposed monitor instructions could also help in analysing the correctness of synchronization. According to the different weaving requirements, it is desirable to be able to recognise either the synchronized block as a whole or the synchronized block body (the code inside the synchronized block brackets) alone. Thus a supporting synchronized block body join point (SBBJP) is also provided.

The pointcut definitions of the two join point models are as follows:

- `synchronized() && args(object)` matches all executions of synchronized blocks with the named object as their synchronizing object;
- `synchronized_body() && args(object)` matches all executions of synchronized block bodies with the named object as their synchronizing object;

The implementation of these two models uses the pointcut designators (PCDs), that are formal descriptions for `synchronized` and `synchronized_body`, in conjunction with the `args` construct of AspectJ to expose the context of the selected SBJP or SBBJP. For both of the join points, the associated locked object is the argument of the `args` construct of AspectJ. The extra mechanism used to remove unwanted synchronization is bound to the `rm_proceed()` method, similar to `proceed()`, inside the `around` advice.

The following are possible ways to express pointcuts and minimum advice codes:

- `proceed(...) inside around advice for a synchronized() && args(object) pointcut, which processes the executions of the synchronized block, including its synchronization;
- `proceed(...) inside around advice for a synchronized() && args(object) pointcut, which processes the synchronized block but without synchronization; and
- `proceed(...) inside around advice for a synchronized_body() && args(object) pointcut, which processes the synchronized block body.

3. Case Studies

This section describes several case studies for the SBJP and SBBJP models, covering the areas of distributed execution [12, 5], IT security [2], thread control, concurrency program testing and debugging, emphasising the broad weaving capability of the two join point models.

3.1 Distributed lock migration

The first example application is also unrealistic, due to the widely different times taken to manipulate the various locks, but it also clearly illustrates the power of the join point model. In the context of Java, the JMM determines what values can be read in multi-threaded environments. It also allows complete prediction of the values that are seen by each thread. Thus a lock acquired by a synchronized block needs to be thought of as a JMM lock.

To migrate a multithreaded program from one JVM to a cluster of JVMs, there would need to be another kind of cluster-wide lock, as seen in [5], for example. This is used to secure competition among the threads, possibly with conditional logic, to bypass the JMM lock and to use an alternative locking implementation, such as distributed locks or cluster locks. Meanwhile, it would need to remove the original JMM lock by calling `rm_proceed()`, which executes the code inside the lock. Pseudo-code for this is shown in Figure 1.

```java
void around(Object o); synchronized() && args(o) {
  if (isDistributed(o)) {
    ClusterManager.acquireDistributedLock(o); 
    rm_proceed();
    ClusterManager.releaseDistributedLock(o);
  }
  else proceed();
}
```

Figure 1. Example to change lock implementation.

This strategy could also be used in a cluster of physical computers. The code inside `rm_proceed()` could even be sent to a remote computational node during the execution of the `around` advice by a distributed dynamic aspect machine, such as that described in [12].

3.2 Re-entrant checking and thread re-scheduling

The second application demonstrates the convenience of using the SBJP for thread management from a security standpoint, as discussed in [2]. Imagine a synchronized block that could launch a denial-of-service attack by containing code that eats CPU cycles in the same way as the code in [15] that implements Ackerman’s function. In order to repel the attack, it is essential to have a join point at the beginning of the synchronized block that limits the CPU usage or the size of free memory. Java provides the Java Native Interface (JNI) library for talking to an operating system via a C layer and providing the resulting data to the Java application [16]. A before advice can use Java assertions to check the CPU usage, as shown in Figure 2.

```java
void around(Object obj): synchronized() && args(obj) {
  double receivedCPUUsage = 100.0 * SystemInformation.
    getProcessCPUUsage(m_prevSnapshot, event);
  // before proceed(), limits the CPU usage */
  assert(receivedCPUUsage < 50) ;
  // before proceed(), check re-entrant locking, handle re-entrant locking if it happens */
  if ( !Thread.holdsLock(obj) )
    proceed(obj) ;
  else if (isLegalLock(obj))
    rm_proceed(obj) ;
  else
    throw new Exception(“Acquired illegal Lock”) ;
}
```

Figure 2. Example to check CPU usage and re-entrant risk.

Another important application emerges when one thread needs to acquire multiple locks. This so-called lock re-entrant behaviour may cause a denial-of-service attack, which is an attempt to make a

\(^1\)The appropriate fragment of code can be found in the AspectJ tutorial at http://dev.eclipse.org/viewcvs/indextech.cgi/aspectj-home
computer resource unavailable to its intended users through quasi-
global synchronization of many Transmission Control Protocol (TCP) flows [18]. Moreover, if the thread that owns the lock ma-
nipulates files, this will prevent users from accessing files to which
they should have access. To counter this, an around advice can be
used to check that a thread has not obtained the same lock before
entering a synchronized block, as shown in Figure 2. If another
thread already holds the same lock, and could also safely release
it, one solution is to let the around advice automatically remove
the synchronization by using \texttt{xm\_proceed()}, meaning that the user
does not need to acquire a new lock if they already hold a legal one.
Otherwise, an ‘acquired illegal lock’ exception is thrown.

3.3 Converting synchronized blocks into transactions
The third application involves changing code so that it uses trans-
sactions instead of locks. As discussed in [14, 17, 1], transactions
provide strong atomicity semantics for all referenced objects, pro-
viding a natural replacement for the critical sections defined using
Java synchronized. Optimistic execution of transactions provides
good parallel performance in the common case of non-conflicting
object accesses, without the need for fine-grain locking mecha-
nisms that further complicate correctness and introduce significant
performance overhead. An easy way to provide these benefits for
an existing parallel program is simply to replace each lock with
a new construct, such as atomic(B), that executes the statements
in block B as a transaction [1]. The implementation that provides
atomicity and isolation depends on a transactional memory model
that is different from the JMM, but the general idea of this conver-
sion can be readily implemented using the synchronized block join
point.

Consider the simple string interning example in Figure 3. With
transactional execution, there is no need to use anything other than
the non-locking HashMap since the caller specifies its atomicity re-
quirements, creating a single logical operation out of the \texttt{get()}
and \texttt{put()} operations. Concurrent reads to the map happen in par-
allel due to speculation and the speculation is handled automatically
by the system. The detailed implementation can be found in [14, 17, 1].

```java
public class MyClass {
    myOtherClass myObject = new myOtherClass();
    public void myMethod() {
        synchronized(this) {
            doSomething();
        }
    }

    before(Object obj): synchronized_body(obj) {
        beginRegisterAccess(Thread.currentThread(), obj);
    }

    after(Object obj) returning : synchronized_body(obj) {
        endRegisterAccess(obj);
    }

    after(Object obj) throwing : synchronized_body(obj) {
        endRegisterAccess(obj);
    }
}
```

Figure 3. Converting synchronized blocks into transactions.

3.4 Synchronized block in concurrency program testing
The fifth example involves testing. Testing of concurrent programs
presents additional difficulties that of sequential programs. One of
them is the non-determinism that makes it harder to reproduce the
behaviour of a program execution. Even if the concurrent processes
are deterministic, non-determinism can arise from the possible differ-
ent sequences of process scheduling.

In this use case an approach to reproduce the behaviour of a
Java multi-threaded program is presented. The term “record and
playback”(R&P) [9] has been used to describe techniques and tools
to register the input and output of a program execution and to use
such data to automatically re-execute the same (or other) program
and compare the results. The program under analysis (under testing
or debugging) is instrumented with synchronized block join
points in such a way that each of the synchronization points (where
the program accesses shared objects) are recorded. In a second
execution the instrumented program can use this synchronization
sequence to force the same sequence of accesses to the shared
objects, to produce the same behaviour. In addition, the analysis of
a given synchronization sequence created in a particular execution
can help to create new synchronization sequences and possibly new
behaviours.

As shown in Figure 4, the call to beginRegisterAccess is respon-
sible for inserting an event in the synchronization sequence, i.e., the
fact that a thread has locked an object. Although not necessary for
the characterization of the synchronization sequence, the call to en-
registerAccess inserts the opposite event, i.e., the release of an
object’s lock. Such an event is useful for synchronization sequence
generation.

```java
String intern() {
    synchronized(map) {
        Object o = map.get(this);
        if (o!=null)
            return (String)o;
    }
}
```

Figure 4. Recording with synchronized block body join point.

The replaying phase, shown in Figure 5, requires a synchro-
nized block join point. The first point to note is that before entering
a synchronized code, the thread should consult the synchronization
sequence and check whether the next event is the one to be exec-
cuted.

The call to checkAccess does not use the synchronizing object to
consult the synchronization sequence. This is so because, consid-
ering that the thread is deterministic and followed the synchroniza-
tion sequence until that point, it would not be accessing a different
object. It has only to check if it is its turn to execute. The call to
nextEvent inside the block assures that the object has been locked
before the event is removed from the synchronization sequence,
liberating other threads to follow their execution (if possible).

3.5 Synchronized block in concurrency program debugging
The final example is about debugging. Like concurrency program
testing, concurrency program debugging, such as unintentional race
conditions or deadlocks are difficult and expensive to uncover and

analyze, and such faults often escape to the field. One reason for this difficulty is that the set of possible interleavings is huge, and it is not practical to try all of them. Only a few of the interleavings actually produce concurrent faults; thus, the probability of producing one is very low. Since the scheduler is deterministic, executing the same tests many times will not help, because the same interleaving is usually created. The problem of debugging multi-threaded programs is compounded by the fact that tests that reveal a concurrent fault in the field or in stress test are usually long and run under different environmental conditions.

ConTest [10] is a tool used by more than fifty testing and developer teams in IBM for finding bugs caused by concurrency. It alleviates the need to create a complex testing environment with many processors and applications, and works by instrumenting the bytecode of the application with heuristically controlled conditional sleep and yield instructions. As examined in [7], the possibility of implementing the instrumentation part of a commercial quality multi-threaded testing tool using AOP is high and will get the benefits of going to a higher abstraction level. However, without the ability of instrument synchronization blocks, it is not easy to find some concurrent bugs and therefore impossible to completely reimplement ConTest with AspectJ.

As shown in Figure 6, we implemented a few aspects to demonstrate the capabilities of an AOP language with the instrumentation of synchronized block join point. The aspect alters the class files to increase the likelihood of catching concurrent bugs, using ideas already implemented in ConTest. A special emphasis is put on the instrumentation capabilities.

SynchronizeSleepNoise is an aspect based on a single pointcut and a single advice. The pointcut defines where the instrumentation is being done. The advice is a call to sleep() with a random parameter in the range [0,50] with a probability of 1% for invoking the sleep method. This adds noise to the instrumented application as done by ConTest’s instrumentor. The difference, however, is that this aspect inlines the noise, whereas ConTest instruments call back methods, which add some runtime overhead. This example could easily be expanded to instrument special concurrent related methods, such as sleep, yield, notify, notifyAll, and so on. In addition, the type of noise could be altered to other types of noise that affect the interleaving of the program, all creating different kinds of heuristics.

4. Discussion

This section discusses some characteristics of the implemented SBJP and SBBJP models. First, performance of various re-entrant checking approaches is studied. Secondly, experimental results are shown for the reimplementation of ConTest.

4.1 Re-entrant checking

Re-entrant behaviour can be caused by doubly nested synchronized blocks, as described in Section 3.2. Particular attention should be paid to the object on which the inner synchronized block is locked. This subsection analyses and proposes solutions to the problem of writing pointcuts for synchronized block selections. Since synchronized blocks cannot be named, it is impossible to use a name-based pattern to write a pointcut that would select a particular synchronized block. It is thus proposed that selection of synchronized blocks is made to rely on the data being processed. In Figure 7, the pointcuts to select the inner synchronized block are written in three distinct forms using a mixture of the thisJoinPoint, args or cflowbelow constructs in AspectJ.

These three approaches pick out the same inner synchronized block, but the resulting performance differs. The first two turn out to be identical; both rely on the processed data and make a selection based on an instanceof test. They generally perform better than the third approach, which uses counter based selection. Figure 8 shows results obtained with these three pointcuts.
However, performance is not always paramount; the cflowbelow pointcut can be used to pick out the inner blocks without needing to know the processed data, and this is helpful for anonymous selection.

4.2 Evaluation for Reimplementation of ConTest

As discussed in Section 3.5, with the instrumentation of synchronized block join point, the reimplementation of Contest becomes real. We tested the aspect SynchronizeSleepNoise against several programs with documented bugs. These programs received a single parameter: the number of threads running simultaneously, and are categorized as low, medium, or high. We ran the tests 10 times for each category and for each configuration: once as the uninstrumented program, called original in the figure, then using ConTest with simple noise, and finally with the SynchronizeSleepNoise aspect.

The results shown in Figure 9 indicate the benefit in using this type of testing. We see that SynchronizeSleepNoise increased the chance of finding bugs. Note that the heuristic is very simple and was not modified to suit the specific program. By tuning the frequency of adding noise and the type of noise, we can achieve better results.

5. Conclusion

A synchronized block join point and its associated synchronized block body join point have been demonstrated in AspectJ. The model achieves reusability of synchronized code and thread control management in Java, to such an extent that the concurrency can be fully handled by a single aspect. More generally, it has been shown that around advice for join points is not limited to performing the proceed() method with given arguments, but can also address more complex and flexible behaviours, such as the rm_proceed() method to remove synchronization overhead.

The main limitations of the model are due to its reliance on bytecode for recognising the synchronized block. This design decision has been made for the same reasons as in AspectJ, which aims to make the aspect applicable to wider code bases. The information prepared for shadow matching and weaving can only be collected at bytecode level.

Another limitation is the inability to call a specific proceed() outside the scope of a pointcut. A possible solution is to expose the aspect as a closure object, that is, a self-contained executable object (typically implementing the Runnable interface) that will be able to execute proceed with all the required execution context.

References

[17] I. Watson, C. Kirkhan and M. Lujan. A Study of a Transactional model achieves reusability of synchronized code and thread control management in Java, to such an extent that the concurrency can be fully handled by a single aspect. More generally, it has been shown that around advice for join points is not limited to performing the proceed() method with given arguments, but can also address more complex and flexible behaviours, such as the rm_proceed() method to remove synchronization overhead.

The main limitations of the model are due to its reliance on bytecode for recognising the synchronized block. This design decision has been made for the same reasons as in AspectJ, which aims to make the aspect applicable to wider code bases. The information prepared for shadow matching and weaving can only be collected at bytecode level.

Another limitation is the inability to call a specific proceed() outside the scope of a pointcut. A possible solution is to expose the aspect as a closure object, that is, a self-contained executable object (typically implementing the Runnable interface) that will be able to execute proceed with all the required execution context.

References