A Join Point for Synchronized Block in AspectJ

Chenchen Xi  Bruno Harbulot  John R. Gurd
Centre for Novel Computing, School of Computer Science,
The University of Manchester, Oxford Road, Manchester M13 9PL, United Kingdom
xic AT cs.man.ac.uk  bruno.harbulot AT manchester.ac.uk  jgurd AT cs.man.ac.uk

Abstract
Synchronization is a key software concern that could be advised not only for synchronization in a multi-threaded program but also for treating the distributed environment as the crosscutting concern. However, the synchronized block in Java has not yet been treated in AspectJ or in any other Aspect-Oriented Programming framework. This paper develops a synchronized block join point model which allows AspectJ to interact directly with synchronization actions.

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. Further, the model could also be used to turn lock based synchronization actions into atomic transactions with software transactional memory. SynAJ, the extensions for the abc compiler that provide AspectJ with a synchronized block join is presented and shown to meet the requirement of different applications.

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, Transactional Memory.

1. Introduction
Synchronization is a concern that developers deal with whenever guarded access to a shared resource is required. Access to devices, files and memory are all things that may require synchronization. Synchronization is also the ideal place to deal with thread control and lock management, but the way to invoke the interaction with synchronization suffer from the tangling problem as it is not easy to express this crosscutting concern and encapsulate it for reuse in both multi-threaded and distributed environments. There are three levels of synchronization according to the problem solving concern, namely, sharing the data among multiple threads, among clusters of Java virtual machine JVMs and among clusters of physical computers. The latter two are defined as ‘distributed environment’ in this paper. The problem is how to migrate the Java language specification across a distributed environment.

Further, sometimes parallel programming is difficult due to the complexity of dealing with conventional lock-based synchronization. To simplify parallel programming, there have been a number of proposals [20, 9] directly in investigating the implications of using software transactional memory [15] to replace lock-based synchronization in existing parallel Java programs. [9] shows that once JVM issues have been addressed, the conversion of a lock-based synchronization into transactions is largely straightforward. However how to facilitate such conversion to avoid code tangling remains a problem.

Aspect-oriented Programming (AOP) [14] has the potential to modularize such synchronization so that user code can become oblivious to the distributed environment. However, although synchronized method has been addressed in [7], the synchronized block has not yet been treated in AspectJ [4] or in any other AOP framework [6]. This paper thus focuses on the AOP part of the implementation which shows how to implement a model for a synchronized block join point (SBJP) to encapsulate crosscutting concerns into logical units using the concepts from AOP, and how this SBJP could help such modularization to plug into the Java Memory Model (JMM) [12] and maintain its semantics along with the semantics defined in the Java Language Specification. 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.

Section 2 presents the SBJP model and the kind of behaviour, in particular which form of blocks, it aims to recognise. Section 3 shows several direct applications: how to
write aspects for distributed synchronization and thread control using the proposed join point. Section 4 presents the SBJP model from the point of view of an aspect compiler, and explains the join point shadow and how to capture the correct semantics in a suitable language design in order to meet the differing requirements of applications. Section 5 enhances the join point model with a contextual link to the data and synchronizations handled by the synchronized block. Section 6 introduces the implementation of a weaver capable of handling the SBJP model, based on abe [5]. Section 7 discusses several issues related to the model and gives a comparison of the work to [7]. Conclusions are drawn in Section 8.

2. Synchronized Block Join Point Model

This section presents the objective of the SBJP model. This consists of defining the behaviour the SBJP aims to recognise, as well as its dynamic characteristics, compared to its counterpart, the synchronized method. That is, the join point is provided with an execution context. This model could be applied to various aspect-oriented systems, but the presentation focuses on AspectJ.

Synchronization is not simply a shorthand for “critical section” or “mutex”. While mutual exclusion is one element of the semantics of synchronization, there are two additional elements: visibility and ordering of each thread. Fortunately, this situation, in which there is a locked object inside a critical section, is similar to the data shared among computational nodes in a distributed environment. This common characteristic makes migration possible.

A synchronized method locks the monitor associated with the instance of the class (or the class, if it is a static method) and prevents others from doing so until the return from the method. A synchronized block can lock any monitor and can have a scope smaller than that of the enclosing method. A synchronized method can be picked out by the existing method call pointcut in AspectJ using the synchronized keyword. But selection of a synchronized block has not been supported in AspectJ to date. At bytecode level, unlike a synchronized method, a synchronized block is represented as a matched pair of entermonitor and exitmonitor bytecode instructions. Because one entermonitor can be statically (but not dynamically) paired with more-than-one exitmonitor, identifying the appropriate matched pair is non-trivial.

As a result, a synchronized block could have more flexibility in using the given monitor instructions than its counterpart synchronized method in dealing with synchronizations in AspectJ. A mechanism rm_proceed() is utilised to remove unwanted synchronization from synchronized blocks, which helps to eliminate unnecessary synchronization instructions and thereby reduce locking overheads in a multi-threaded program or even in distributed environment. The exposed monitor instructions could also help in analysing the correctness of synchronization. An overview of the removal of synchronization is presented in Section 5.

According to the different weaving requirements, it is necessary not only to recognise the synchronized block, but also recognise the synchronized block body, namely the code inside the synchronized block brackets. Thus a supporting synchronized block body join point (SBBJP) is also provided. Further details about recognising the two join points are presented in Section 4.

The pointcut definitions of the two models are:

- \( \text{synchronized() \&\& \text{args(object)} \) matches all the executions of synchronized blocks with the named object as the synchronizing object;
- \( \text{synchronized_body() \&\& \text{args(object)} \) matches all the synchronized block body with the named object as the synchronizing object;
- \( \text{proceed()} \) inside \( \text{around} \) advice for \( \text{synchronized() \&\& \text{args(object)} \) pointcut, which processes the executions of synchronized block, including its synchronization;
- \( \text{rm_proceed (...) \&\& \text{around} \) advice for \( \text{synchronized() \&\& \text{args(object)} \) pointcut, which processes the synchronized block but without synchronization; and
- \( \text{proceed(...) \&\& \text{around} \) advice for \( \text{synchronized_body() \&\& \text{args(object)} \) pointcut, which processes the synchronized block body.

The implementation of these two models, described in more detail in Section 6, uses the newly developed pointcut designators (PCD), namely synchronized and synchronized_body, in conjunction with the \text{args} construct of AspectJ to expose the context of the synchronized block join point and its body join point. For both of the join points, the locked object is taken as the argument of the \text{args} construct of AspectJ. The extra mechanism used to remove unwanted synchronization is bound to the \text{rm_proceed()} construct, similar to \text{proceed()} inside the \text{around} advice.

For example, the following \text{around} advice would recognize the synchronized block shown in Figure 1, and process the code without synchronization.

\begin{verbatim}
proceed(...) inside around advice for
synchronized() \&\& \text{args(object)}
\end{verbatim}

\begin{verbatim}
\text{rm_proceed(...) inside around advice for}
synchronized_body() \&\& \text{args(object)}
\end{verbatim}

\begin{verbatim}
proceed(...) inside around advice for
synchronized_body() \&\& \text{args(object)}
\end{verbatim}

Note to reviewers: we could add more description here if this would be helpful to the average reader – we are unsure whether they would be likely to know about the structure of the bytecode.

---

1. This fragment of code can be found in AspectJ tutorial at http://dev.eclipse.org/viewcvs/index tech.cgi/aspectj-home

2. Defined above as clusters of JVMs or physical computers.
3. Aspects for Synchronization

This section describes several applications proposed for the SBJP and the supporting SBBJP model, covering the areas of distributed environment [6, 13, 7], IT security [3], performance monitoring [6] and thread control [17] and showing the weaving capability of the model.

3.1 Applications for before and after advice

SBJP and SBBJP are first proposed to capture various times associated with the lock; the concrete use cases in [6] can all be implemented by before and after advice using these two models. These are not realistic examples, but have been chosen for illustrative purposes because they do show the capabilities of the two models for intercepting at specific join points. It is assumed that high precision timer methods are available. For example, to obtain the time spent by the lock, the inserted code needs to count the time via two pieces of advice, one before the lock is acquired and one just after the lock is released. Alternatively, one around advice can be used as shown in Figure 2.

3.2 Distributed lock migration

This second example is also unrealistic, due to the widely different times taken to manipulate the various locks, but it also illustrates the power of the join point model clearly. In the context of Java, the JMM determines what values can be read in multithreaded environments. It also allows complete prediction of the values that are seen by each thread. Thus the lock acquired by the synchronized block could be named as a JMM lock. To achieve migration from multithreaded program to a cluster of JVMs, there could be another kind of cluster-wide lock (as seen in [7]) to secure the competition among the threads possibly with conditional logic, bypass the JMM lock and use alternative locking implementation, such as distributed locks or cluster locks, and meanwhile remove the original JMM lock by \texttt{rm_proceed()}, which only executes the code inside the lock, as the pseudo-code in Figure 4 shows. This strategy could also be used in a cluster of physical computers. The code inside \texttt{rm_proceed()} could even be sent to a remote computational node during the execution of the around advice by a distributed dynamic aspect machine [13].

3.3 Re-entrant checking and Thread re-scheduling

The importance of the SBJP for thread management could be assessed from a security standpoint, as discussed in [3]. 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 that implements Ackerman’s function in [16]. It is essential to have a join point at the beginning of the synchronized block to limit the CPU usage or the size of free memory, which will repel the attack. Java provides the Java Native Interface (JNI) library for talking to an operating system in a C layer and providing the resulting data to the Java application [18]. A before advice uses Java assertions to check the CPU usage, as shown in Figure 5. Another important application of this join point emerges when the same thread acquires multiple locks. This so-called lock re-entrant behaviour may cause a denial-of-service attack, which is an attempt to make a computer resource unavailable to its intended users through quasi-global syn-

Figure 1. Simple synchronized block in Java.

```java
void around(Object obj): synchronized() & args(obj) 
{ 
  rm_proceed(ArrayList obj); 
}
```

Figure 2. Example to obtain time the lock is locked.

```java
ArrayList list;
void run() {  
  /* before synchronized block */
  synchronized(list) {  
    /* before synchronized block body */
    /* do something with list ... */
    /* after synchronized block body */
  }  
  /* after synchronized block */
}
```

Figure 3. Example to obtain time to acquire the lock.

```java
void around(Object obj): synchronized() & args(obj) {  
  time t_start = getCurrentTime(); 
  proceed(); 
  time t_end = getCurrentTime(); 
  time t_elapsed = t_end - t_start; 
}
```

Figure 4. Example to change lock implementation.

```java
void around(Object obj): synchronized() & args(obj) {  
  if (isDistributed()) |  
    ClusterManager.acquireDistributedLock(obj);  
  else  
    ClusterManager.releaseDistributedLock(obj);  
  
  time t_start = getCurrentTime(); 
  proceed(); 
  time t_end = getCurrentTime(); 
  time t_elapsed = t_end - t_start; 
}
```
Figure 5. Example to check CPU usage and re-entrant risk.

```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 (isIllegalLock(obj))
        rm_proceed(obj);
    else 
        throw new Exception("Acquired illegal Lock") ;
}
```

Figure 6. Example to control and reschedule thread.

```java
ArrayList al = new ArrayList();
public void run() { 
    synchronized(al) { 
        if (al.size() > 0) 
            Object obj = al.get(0);
            // do something with obj...
    }
}
void around(ArrayList list); synchronized() 
    // args(list) 
    if (canBeSplit(list)) { 
        new Thread(new Runnable() { 
            public void run() { 
                ArrayList sublist = list.clone().subList(0);
                proceed(sublist); 
            }
        }).start(); 
        new Thread(new Runnable() { 
            public void run() { 
                list = list.subList(0);
                proceed(list); 
            }
        }).start(); 
    } else 
        proceed(list); 
```

chronization of many Transmission Control Protocol (TCP) flows [21]. Moreover, if the thread that owns the lock manipulates files, this will block users from accessing files to which they should have access.

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 5. If a thread already has the same lock, and it could also safely release this lock, one solution is to let the `around` advice automatically remove the synchronization by using `rm_proceed()`, which means that users do not need to acquire a new lock if they already have a legal one. Otherwise, an acquired illegal lock exception is thrown.

More generally, if the synchronization cannot be eliminated at compile time, there are still ways to release the bottleneck presented by synchronization in multithreaded programs. For example, in the pseudo-code shown in Figure 6, if the locked object `ArrayList sublist` could be split during the `around` advice, the original thread could be split into two sub-threads, each of which contains and proceeds a sub-list of the original `ArrayList sublist`. Again, if the program is executing on a distributed system, during the `around` advice, the original thread could even ask remote nodes in a distributed dynamic aspect machine [13] to finish each sub-task, namely each `proceed()` in Figure 6.

3.4 Converting synchronized blocks into transactions

As discussed in [15, 20, 1], transactions provide strong atomicity semantics for all referenced objects, providing a natural replacement for critical sections defined by Java `synchronized`. Optimistic execution of transactions provides good parallel performance in the common case of non-conflicting object accesses with out the need for fine-grain locking mechanisms that further complicate correctness and introduce significant overhead. The best way to provide the benefit of transactional memory (TM) to the existing parallel program is to replace locks with a new language construct such as `atomic{B} [1]` that executes the statements in block B as a transaction. The implementation that provides atomicity and isolation depends on different software transactional memory (STM), but the general idea of the conversion could be implemented by the synchronized block join point.

Consider the simple string interning example in Figure 7. With transactional execution, there is no need to use anything other than the non-locking `HashMap` since the caller specifies its atomicity requirements, creating a single logical operation out of the `Map get()` and `put()` operations. Concurrent reads to the map happen in parallel due to speculation and the speculation is handled automatically by the system. The detailed implementation can be found in [15, 20, 1].

4. Shadow Matching

This section describes the way the synchronized blocks are recognised: i.e., how the shadows, static locations in the code where join points will be matched, of the synchronized block are identified. The theoretical definition of a join point shadow is as follows: “Every dynamic join point has a corresponding static shadow in the source code or bytecode of the program. The AspectJ compiler inserts code at these static shadows in order to modify the dynamic behavior of the program” [8].

Although the aim is to recognise synchronized blocks of the forms presented in Section 2, this section investigates in detail the requirements of various kinds of application in order to form a basis for the SBJP and associated SBBJP models:
String intern() {
    synchronized(map) {
        Object o = map.get(this);
        if (o != null)
            return (String)o;
        map.put(this, this);
        return this;
    }
}

aspect TestAspect {
    void around(Object map): synchronize()
    {
        $withincode(* * .intern(.))$
        $args(map)\$
        atomic{
            rm_proceed(map);
        }
    }
}

Figure 7. Converting synchronized blocks into transactions.

1. the ability to weave before advice, after advice and around advice for a synchronized block including the lock;
2. the ability to weave before advice, after advice and around advice for a synchronized block body excluding the lock; and
3. the ability to extract the context of execution at the join point.

4.1 Synchronized block in general case

In order to find a synchronized block, the initial approach is to pair the monitor instructions sequentially from the byte-code, which needs to construct the complete control flow graph of the code following the method described in [2]. This technique is based on finding dominators and successors. Given that a dominator contains an entermonitor, the synchronized blocks are defined as the region from the dominator to two of its branches, both of which ends with a successor node containing an exitmonitor. One of the branches denotes the normal exit, the other denotes the exception exit. Figure 8 represents the control flow graph for the simple synchronized block shown in Figure 1. In this example, the successors of dominator node 1 are: (a) nodes 2, 3 and 4, which is a normal exit branch; and (b) nodes 5, 6 and 7, which contain an exception handler as an exception exit branch. This typically, a doubly nested synchronized block is not recommended, but since it is not forbidden by the Java grammar, distinguishing the nested synchronized blocks is needed for detecting lock re-entrancy or deadlock problems. Therefore, instead of using two exit nodes for the join point model, the union of all

4.2 Synchronized block with extra exit nodes

The above two branches of synchronization can become confusing when there are return or break statements in the body of a synchronized block, because there would then be multiple normal exit nodes from the synchronized block. As shown in Figures 10 and 11, what appears to be a simple synchronized block actually contains three exit nodes sharing the same dominator (see control flow graph in Figure 13 for the multi-exit synchronized block shown in Figure 10). In such a case, defining the points immediately after the synchronized block would be ambiguous. Therefore, instead of using two exit nodes for the join point model, the union of all

Figure 8. Control flow graph of simple synchronized block.

- the inner synchronized block consists of nodes 2, 3, 4, 5 and 6; its exception handler is 4, 5 and 6; and
- the outer synchronized block consists of nodes from 1 to 10; its exception handler is 7, 8 and 9.

Once the name of the locked objects, r1 and r2 in Figure 9, are registered, these two synchronized blocks can be recognised.

Figure 9. Double synchronized blocks nest.
branches which contain an exit node sharing the same dom-
inor is considered as a single synchronized block. How-

```
@synchronized (Object obj) {
  if(obj.getStatus())
    return ;
}
```

Figure 10. Example synchronized block contains return statement.

```
while(true) {
  synchronized (Object obj) {
    if(obj.getStatus()) break ;
  }
}
```

Figure 11. Example synchronized block contains break statement.

ever, some of the “extra” exit nodes might be illegal, even
though they are consistent with Java grammar. In the case
shown in Figure 11, the code can be compiled and run with-
out error as a single threaded program. But it could cause a
severe unreleased lock problem in a multi-threaded program
because of the misuse of the break statement, which exits a
synchronized block without safely releasing the lock. If the
program exits via the break, there will be no exitmonitor
paired with the dominating entermonitor, and the object
will be locked forever. The implementation of the SBBJP
could find this kind of improper use of synchronized blocks
and report it in the form of a misuse warning.

4.3 Synchronized block body join point

According to the approach taken by [6], there is a require-
ment to distinguish between picking out the whole synchro-
nized block and just the body of the block. For example, for
inserting a piece of advice just after obtaining the lock, as
shown in Figure 3, the application catches the body of the
synchronized block as the shadow, so as to use proper before
advice to get the time of acquiring or releasing the lock. To
distinguish these two kinds of join points, a new SBBJP is
implemented and synchronized_body is used as the key-
word for the pointcut.

Catching the synchronized block body does not consider
the verbose exception handling associated with synchronization.
As shown in Figure 8, the shadow is just the code cov-
ering node 2. For a synchronized block with more than two
exits, the shadow of the SBBJP will be a combination of
several non-continuous areas. Moreover, any return state-
ment inside the around advice for a block body should be
followed by an exitmonitor statement to ensure that the
lock is released by the block body properly.

5. Context Exposure

Although synchronized blocks do not have arguments in the
same way as other join points, such as method calls,
they often depend on contextual information, particularly the
locked object, to which programmers may want access. In
addition, unlike synchronized methods, synchronized blocks
expose the Java monitor bytecode instructions to the pro-
grammer, which could be regarded as another kind of con-
text exposure to allow the SBJP to do flexible adaptation, at
least including:

- eliminating unnecessary synchronization; and
- re-inserting synchronization that has previously been re-
  moved.

Knowing that a synchronized block is represented as a
region of code between paired entermonitor and
exitmonitor bytecode instructions allows one to deter-
mine, at compile time, the execution behaviour of the syn-
chronized block in some detail. The monitor instruction pairs
can be considered as a first set of arguments for the syn-
chronized block (see Section 5.1). In order to make these
compile-time meaningful, only synchronized blocks includ-
ing the lock are considered for this context exposure. This
exempts SBBJP which is not able to have unnecessary syn-
chronization removed. Moreover, the object treated by the
synchronized block may be of interest for applications as
another contextual argument (see Section 5.2).

5.1 Eliminating synchronization

In order to reduce overhead, synchronization actions may
be safely removed without compromising program seman-
tics, thus improving performance. This elimination could be
done manually or by some algorithm. Ruf [17] provides an
algorithm for removing unnecessary synchronization oper-
atics, thus improving performance. This elimination could be
done manually or by some algorithm. Ruf [17] provides an
algorithm for removing unnecessary synchronization oper-
atics, thus improving performance. This elimination could be
done manually or by some algorithm. Ruf [17] provides an
algorithm for removing unnecessary synchronization oper-
atics, thus improving performance. This elimination could be
done manually or by some algorithm. Ruf [17] provides an
algorithm for removing unnecessary synchronization oper-
atics, thus improving performance. This elimination could be
done manually or by some algorithm. Ruf [17] provides an
algorithm for removing unnecessary synchronization oper-
atics, thus improving performance. This elimination could be
done manually or by some algorithm. Ruf [17] provides an
algorithm for removing unnecessary synchronization oper-
atics, thus improving performance. This elimination could be
done manually or by some algorithm. Ruf [17] provides an
algorithm for removing unnecessary synchronization oper-
atics, thus improving performance. This elimination could be
done manually or by some algorithm. Ruf [17] provides an
algorithm for removing unnecessary synchronization oper-
ations from statically compiled Java programs. One of the
approaches is simply to prove that the program spawns no
threads, making contention impossible, upon which, all syn-
chronization maybe removed. Unnecessary synchronization
also occur when a JMM lock is replaced by some other
kind of lock (such as a distributed lock) when the program
becomes distributed, as shown in Section 3.

The removal of synchronization in Java corresponds to the
removal of paired entermonitor-exitmonitor in-
structions and their associated exception handler at bytecode
level. In order to handle possible exceptions caused by multi-
 threaded execution, the synchronized block actually contains
at least one try-catch block just after it acquires the lock,
and this try-catch block will end by releasing the lock. If
there is any return inside the synchronized block, an extra
exitmonitor is added before the return to ensure the lock
is released safely.

Figures 12(a) and 12(b) represent, respectively, a tra-
ditional synchronized block containing a return, and the
associated code which eliminates the synchronization. The
elimination of the synchronization removes the associated
try-catch block and the potential jump from synchronized
block body to the exception handler. Figures 13 and 14 de-
5.2 Re-inserting synchronization

Since synchronization could be removed at compile time by the advice for SBJP, it is also necessary to be able to undo this modification. In order to simplify re-insertion of the synchronization, as well as comply with the semantics of AspectJ, the advice for SBJP does not really remove the monitor instructions and associated exception handler at compile time; instead, it creates a new method, `rm_proceed()`, which contains the code inside the synchronized block without synchronization. The original synchronized block including synchronization is contained in method `proceed()`. Thus any eliminated synchronization can be recovered simply by replacing method `rm_proceed()` by method `proceed()`.

5.3 Synchronized object

Both forms of SBJP and SBBJP presented in Section 4 must hold a locked object. The characteristics of this object dominate the behaviour of the synchronized block. Through analysing the attributes of the object, execution details of a synchronized block could be re-arranged. Again, exposing this extra information could be useful, for example for certain synchronization schemes which require the program explicitly to check the legality of a thread in acquiring a lock (see Figure 5) or for thread rescheduling (see Figure 6).

6. Implementation in abc

Due to the focus in the SBJP model being put on a join point model integrable into AspectJ, the implementation uses abc, an alternative AspectJ compiler, not only for the extensibility which is the core design of abc, but also for the Soot framework, which provides most of the infrastructure for performing the analyses.

This section describes two extensions to abc, known as SynAJ and SynBodyAJ, which implement SBJP and SBBJP and provide `synchronized()` and `synchronized_body()` pointcuts, respectively. The former picks out the synchronized block including the lock, provides contextual information and offers a flexible way to remove unwanted synchronization. The latter picks out the synchronized block body excluding the lock.

6.1 Shadow matching

In the Soot framework, as well as in abc, a three-address representation of bytecode, Jimple [19] is used, by which finding a synchronized block at bytecode level is made possible. Both SynAJ and SynBodyAJ extend the class that finds the shadows in each method so that it looks for both synchronized blocks and synchronized block bodies. For each method processed, the control flow graph and its corresponding dominator tree are built using the Soot framework toolkit. Then each synchronized block and associated body is identified.

abc provides two kinds of classes representing a shadow, namely `StmtShadowMatch` and `BodyShadowMatch`, both of which extend `ShadowMatch`. The former is used for pinpointing a specific statement or group of statements in the method, for example, when a method call pointcut is used. The latter is used when the shadow is the whole method body, for example, when a method execution pointcut is used. `LoopsAJ`, an extension of abc that implements a loop join point model [11], provides `GroupShadowMatch` when the shadow is a group of statements; this is also used in SynAJ and SynBodyAJ.
In dealing with before and after advice for both SynAJ and SynBodyAJ, one of the requirements of abc is to insert nop operators in the shadow, at the points where before and after advice might be woven. Given this, most of the abc infrastructure can already handle synchronized shadows for before and after advice.

Handling around advice requires modification of the around weaver, as in LoopsAJ [11]. But the requirements for synchronized blocks concerning the exception handling have to be satisfied in a more complicated way than those for loops. Also, due to the requirement to distinguish the synchronized block from its body, further modification inside the around weaver is required. Handling of the new type of shadows and, more importantly, keeping the control flow graph and the synchronized join point structures consistent with the method content are performed using SynchronizedAdviceApplication, a run-time controller class in abc. The controller can signal the weaver to do the specific matching work, before or after the original matching, or even replace the way of matching.

Both SBJP and SBBJP structures are created during shadow matching, and different weaving operations modify the set of instructions in the methods. Weaving an around advice is implemented by placing the statements that form the shadow into a separate method and replacing them by an invocation of that method. Because the SBJP structure corresponds to the instructions that form the synchronized block, it has to be updated to take this operation into account.

As the synchronized block has to deal with possible exceptions during execution of around advice, Soot uses the class Trap to describe the sequence of an exception handler in a method. Its methods setHead(), setTail() and setHandler() are utilised for all transformations of exception handlers. It is thus possible to implement a consistent exception handler behaviour for proceed() inside an around advice and other aspects.

### 6.2 Context exposure

Both SynAJ and SynBodyAJ expose the locked object as contextual information, as described in Section 5. In order to ensure that transformation caused by exposed context will not change the meaning of the synchronization, related field declarations and assignments are made to properly acquire and release the lock.

There are several potential ways to use the exposed context in SynAJ, as shown in Figure 15.

In case 1, an attribute of the object is changed before proceed().

In case 2, a field of the object is removed then a new one added before proceed(), which may cause compatibility problems for execution inside the synchronized block. For example, this occurs if the original synchronized block cast the object in ArrayList list to Thread type, but, during the around advice in case 2, the object in ArrayList list is String type. The programmer of the aspect should avoid this incompatible misuse.

In case 3, the object passed to proceed() is changed to another new object. In order to make the around advice make sense, not only should compatibility be dealt with, as in case 2, but also the locking associated with the new object should be done properly. Due to weaving limitations inside abc, the original around advice just passes the object as an argument which is like the rewritten Java code shown in Figure 16. Although sublist can be accessed and used inside the synchronized block as an argument, it is not guaranteed that sublist is locked safely, even though list is. Thus further modifications are made in order to unify the name of the lock and its associated object. SynBodyAJ has almost the same way of using the exposed context but, since SynBodyAJ does not take the lock into account, the modification for thread safety need not be made.

```java
ArrayList list = new ArrayList();
public void run() {
     synchronized(list) {
        Thread t = (Thread)list.get(0);
        // do something with list...
     }
}

// ...

/* 1. change an attribute of the object */
void around(ArrayList list): synchronized()
    & & within/*.run() & & args[list] {
    list.ensureCapacity(10);
    proceed(list);
}

/* 2. add or remove a field of the object */
void around(ArrayList list): synchronized()
    & & within/*.run() & & args[list] {
    list.clear();
    list.add("hello");
    proceed(list);
}

/* 3. change the object with which to proceed */
void around(ArrayList list): synchronized()
    & & within/*.run() & & args[list] {
    new Thread(new Runnable()) {
        public void run() {
            ArrayList sublist = list.clone().subList();
            proceed(sublist);
        }
    }.start();
}
```

**Figure 15.** Context exposure examples.

**Figure 16.** Unlocked object by improper transformation.
6.3 Eliminating synchronization

The exposed monitor can make use of the synchronized block more flexibly under the control of SynAJ, as described in Section 5.1.

In its around weaver, abc provides the static class ProceedMethod which represents each proceed() method. SynAJ extends the method that uses the around weaver to weave the join point so that the proceed() method can be used inside the around advice.

In order to implement the rm_proceed() method, to remove the synchronization of a synchronized block inside the around advice, ProceedMethod has been extended by RemoveProceedMethod. As the removal operation can be undone, the original ProceedMethod is kept. When weaving the around advice, the extension is used by calling both of the following methods: proceedMethod.doWeave() and removeProceedMethod.doWeave(). Thus there are two different methods for the around weaver to call during around advice.

7. Discussion

This section discusses some characteristics of the SBJP based on the implemented model. First, the re-entrant checking method is studied. Secondly, a comparison is made between the performances obtained from SBJP and another implementation [7] as related work. Finally, ideas for a generalised “block” join point related to SBJP are briefly introduced.

7.1 Re-entrant checking

Re-entrant behaviour can be caused by misuse of double synchronized nested blocks, as described in Section 3. 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 17, the pointcuts to select the inner synchronized block are written in three distinct forms using a mixture of thisJoinPoint, args or cflowbelow [10] constructs in AspectJ.

These three approaches pick out the same inner synchronized block, but the resulting performance differs. The first two are actually the same, both of which rely on the processed data and make selection based on an instanceof test. They generally perform better than the third approach, which relies on counter based selection. Figure 18 shows results obtained with these three pointcuts. But if the processed data are not available, the cflowbelow pointcut can be used to pick out all of the inner blocks without knowing the processed data for nested synchronized blocks, which is helpful for anonymous selection.

7.2 Related work

Pairs of lock and unlock pointcuts have been recently proposed and used to pick out synchronized blocks [7]. These pointcuts match the entermonitor and exitmonitor instructions, respectively. Because pairs of lock and unlock pointcuts are used to catch synchronized blocks, the weaving capability is limited to before and after kinds of advice for the synchronized block. The paper [7] does not state whether the lock and unlock pointcut have the capability of removing the synchronization. The system obviously cannot weave around advice or any proceed() for converting the synchronized block into transactions. Moreover, it seems dangerous to use just half of the monitor as the pointcut, in case the other half of the pointcut is misused.

The above situation, together with the context exposure capabilities, is summarised in Table 1, where LOCK & UNLOCK refers to the work presented in [7].

8. Conclusion

This paper demonstrates the synchronized block join point and its associated synchronized block body join point 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, the paper shows 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 limitations of this model are due to its reliance on bytecode for recognising the synchronized block. This design decision has been made for the same reasons as AspectJ, which aims to make the aspect applicable to wider code bases. The information prepared for the shadow and weaving can only be collected at bytecode level. One of the limitation is the ability 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. Another limitation is related to the size of the synchronized block. A synchronized block might encompass more code than what is actually required in the critical section for the safety of the synchronization. This might cause extra overhead, however, the current SBJP model does not have the capability to resize the region of the synchronized block. A third limitation relates to the converting synchronized blocks into transactions. Transactional memory can be implemented in various systems in either software or hardware for different purposes. Therefore, code written to work on one Java virtual machine may not work on another system, since the atomic constructors only consider their own language and library specifications to be an implementation detail.

References