A join point for loops in AspectJ

Bruno Harbulot
bruno.harbulot@cs.man.ac.uk

John R. Gurd
jgurd@cs.man.ac.uk

Centre for Novel Computing, School of Computer Science,
University of Manchester, Oxford Road, Manchester M13 9PL, UK

ABSTRACT

The current AspectJ join points represent locations in the code that are at the interface of the Java objects. However, not all the "things that happen"[1] happen at the interfaces. In particular, loops are a key place that could be advised for parallelisation. Although parallelisation via aspects can be performed in certain cases by refactoring the Java code, it is not always possible or desirable. This article presents a model of loop join point, which allows AspectJ to intervene directly in loops.

The approach used for recognising loops is based on a control-flow analysis at the bytecode level; this avoids ambiguities due to alternative forms of source-code that would effectively produce identical loops. This model is also embellished with a mechanism for context exposure, which is pivotal for giving a meaning to the use of this join point. This context exposure is particularly useful for writing pointcuts that select specific loops only, and the problem of loop selection is also presented in the paper.

Finally, *LoopsAJ*, an extension for the `abc` compiler that provides AspectJ with a loop join point, is presented. It is shown how to use this extension for writing aspects that parallelise loops.

1. INTRODUCTION

When parallelising code in order to improve performance, loops are the natural places to make changes. There are sometimes several alternative ways of parallelising the same loop, depending on various parameters, such as the nature of the data being processed, or the architecture on which the application is going to be executed. In certain cases, it is possible to use aspects for parallelising loops, in particular for choosing a method of parallelisation [3]. However, since there is currently no join point for loops in AspectJ [5], the method proposed in [3] resorts to refactoring the base-code. In order to eliminate this inconvenience, this paper proposes a loop join point model for AspectJ which allows direct parallelisation of loops, without refactoring of the base-code.

[1](to use the AspectJ guide phrasing for introducing the concept of join point).

Foundations Of Aspect-oriented Languages workshop (*FOAL 2005*), held in conjunction with *AOSD 2005* in Chicago, USA.

Section 2 presents a formal definition of the loop join point model. This includes the definition of a loop and the way it can be identified. Although this approach is based on Java and AspectJ, the model can potentially be applied to other languages. Section 3 embellishes the loop join point model with a relation to the data handled by the loops. Section 4 explains the specific requirements for loop selection, and describes the associated difficulties, compared with other kind of join points. Section 5 introduces *LoopsAJ*, a prototype implementation of a weaver capable of handling the loop join point model, based on `abc` [2].Section 6 shows how to write aspects for parallelisation using the loop join point. Section 7 describes some of the problems related to base-code containing exceptions. Section 8 briefly introduces ideas about other potential fine-grained join points: a "loop-body" join point and an "if-then-else" join point. Finally, Section 9 concludes.

2. LOOP JOIN POINT MODEL

This section presents the definition of a loop join point model. It could be applied to various aspect-oriented systems, but the presentation focusses on the approach used in AspectJ. Section 2.1 describes the general approach used to recognise loops in the code. Section 2.2 gives a summary of compiler theory related to loop recognition. Finally, in Section 2.3, the definition of a loop is progressively restricted in order to build a model suitable for a join point.

The first step is to identify what the *shadow* of the loop join point is. The shadow of a join point is defined as follows: "*[A] join point is a point in the dynamic call graph of a running program [...]. Every [such] 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*" [4]. The main elements required for a join point shadow are:

- a weaving point for *before-advice*,
- a weaving point (or maybe several points) for *after-advice*, and
- the eventual ability to weave *around-advice.*

Then, for the dynamic part, the model should make it possible to extract information regarding the execution context at the join point.

2.1 From source or from bytecode

The first decision to be made is whether the join point is recognised at source code level or at bytecode level. The way loops are programmed in Java is not necessarily directly reflected in the generated bytecode. For example, instinctively, most Java programmers would consider the body of a for-loop to be the lines of code within the curly brackets following the `for(;;)` statement. However, a loop with the same effect can also be written in different ways, for example as a while-loop, or with some of the `for` statements displaced, as shown in Figure 1.

```
for (int i = 0 ; i < MAX ; i++) {
   /* A */
}

int j = 0 ;
int STRIDE = 1 ;
for ( ; j < MAX ; j += STRIDE ) {
   /* A */
}

int k = 0 ;
while (k < MAX) {
   /* A */
   k++ ;
}
```

Figure 1: Simple examples of equivalent loops.

In addition, the main conditional expression of a loop may encompass several instructions, in particular if it involves a call to a method or a complex expression, as shown in Figure 2. Although the condition may not seem to be part of the loop body, it could always be refactored so as to be part of it (for example through a temporary `boolean` variable). Moreover, the compiled code does not necessarily reflect the way a complex expression has been written in the source code.

```
int i = 0 ;
while (condition(i) || (i>10)) {
   /* A */
   i++ ;
}

int j = 0 ;
boolean ok = condition(j) || (j>10) ;
while (ok) {
   /* A */
   j++ ;
   ok = condition(j) || (j>10) ;
}
```

Figure 2: Loop with more complex conditions.

Since the main concern is to recognise the behaviour of the code, rather than the way it was written, the choice was made to base the representation of loops at the bytecode level rather than at the source code level. As a result, the representation is more robust to variations in programming style. However, this choice introduces limitations regarding (a) the potential specific handling of abrupt exit (see Section 2.4), and (b) the nature of the control-flow graphs. Indeed, as explained in more detail in Section 7, the model expects a reducible (or well-structured [1, 10]) graph. When exceptions are not used, Java source-code produces bytecode with reducible control-flow graphs, but this is not necessarily the case for bytecode produced by other means.

2.2 Dominators, back edges and natural loops

The initial approach for finding loops in the control-flow graph follows the method described in [1, 10]. This method is based on finding *dominators* and *back edges*, defined as follows: "*Node* d *of a flow graph* dominates *node* n *[...] if every path from the initial node of the flow graph to* n *goes through* d. *[... The] edges in the flow graph whose heads dominate their tails [are called]* back edges. *(If* a → b *is an edge,* b *is the* head, a *is the* tail.*) [... Also,* a *is a* predecessor *of* b, *and* b *is a* successor *of* a *...] Given a* back edge n → d, *we define the* natural loop *of the edge to be* d *plus the set of nodes that can reach* n *without going through* d. *Node* d *is called the* header *of the loop*" [1].

Figures 3(a) and 3(b) represent, respectively, the (block-level [2]) control-flow graph and the associated dominator tree for the simple `for`-loop shown in Figure 1. In this example, the only back edge is 3 → 2, and its natural loop comprises blocks (nodes) 2 (which is the header) and 3.

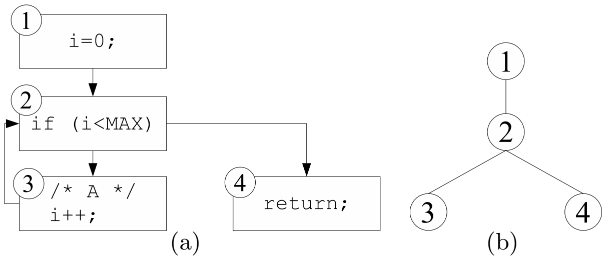

Figure 3: Control-flow graph (a) and dominator tree (b) for a simple `for`-loop.

Natural loops can be confusing because there could be several loops with the same header. As shown in Figure 4, what appears to be a single loop actually corresponds to two natural loops sharing the same header. In such a case, defining the points immediately *before* or *after* a natural loop would be ambiguous. Therefore, instead of using natural loops for the join point model, the union of all the natural loops sharing the same header is considered as a single *combined loop*. To avoid ambiguous cases, implementations should consider a node containing only an unconditional `goto` as the same node as its successor node. In the remainder of this article, the term "*loop*" will be used to mean a "combined loop", unless otherwise stated.

Following this style, an *inner loop* is a loop whose blocks are all contained within another loop, but do not share the latter's header. This also happens to match the natural definition of inner loops at the source level.

In the following sections, three categories of loops are presented, together with their characteristics pertinent to possible use as join points. The categories introduce increasing

[2] i.e., the nodes of the control-flow graph are *basic blocks* [1] of code statements.

```
int i = 0 ;
while (i<MAX) {
   if (cond(i++)) {
      /* A */
   } else {
      /* B */
   }
}
```

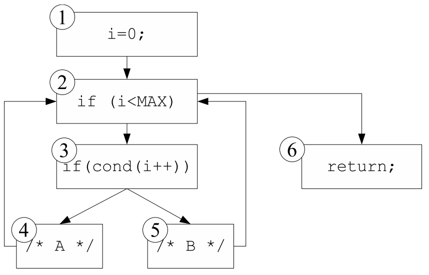

Figure 4: Two natural loops with the same header.

degrees of constraint which affect their ability to weave the three forms of advice: *before*, *after* and *around*.

2.3 Loops in the general case

A loop always has a unique entry point, namely its header. *Before-advice* can therefore be woven in a *pre-header*, that is, a node (block) inserted before the header to which the jumps from outside the considered loop are redirected, but the jumps from inside it are not (see Figure 5).

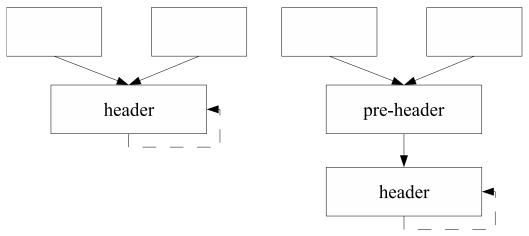

Figure 5: Insertion of a pre-header.

Without further constraint, it cannot be guaranteed that there is a unique point in the control flow that is executed immediately after execution of a loop. In order to introduce appropriate constraints, the following definitions are added. A node in a loop is an *exit node* if it can branch outside that loop. A node outside a loop which has predecessors inside that loop is termed a *successor node* of the loop.

Typically, a non-nested loop which contains a `break` statement has two exit nodes and one successor node, while a double loop nest with a `break` statement in the inner loop that branches outside the outer loop has two exit nodes and two successor nodes. For example, Figure 6 shows the source code and the corresponding (block-level) control-flow graph for a doubly nested loop:

- The inner loop consists of blocks 4, 5 and 6; its exit nodes are blocks 4 and 5; its successor nodes are blocks 7 and 8.
- The outer loop consists of blocks 2, 3, 4, 5, 6 and 7; its exit nodes are blocks 2 and 5; its (sole) successor node is block 8.

```
int i = 0 ;
outside:
while (i < maxI) {
   int j = 0 ;
   while (j < maxJ) {
      if (c(i,j))
         break outside ;
      j++ ;
   }
   i++ ;
}
/* A */
```

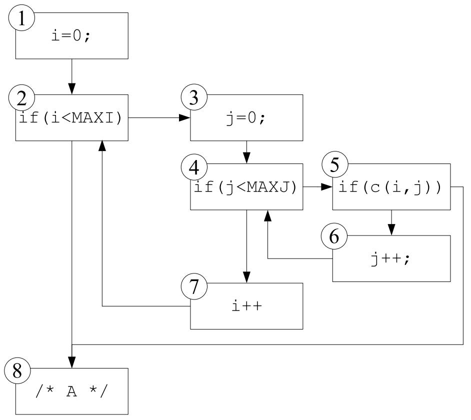

Figure 6: Two nested loops and break statement jumping outside outer-loop.

In this case, "after" loop $\{4, 5, 6\}$ is both on the transitions between blocks 4 and 7, and between blocks 5 and 8.

In such cases, where there are several successor nodes, weaving an *after* piece of advice would require replication of the woven code at all edges between exit nodes and their successor nodes. Although it is, in principle, possible to achieve this, some aspect-oriented tools do not allow this kind of weaving.

2.4 Loops with a unique successor node

The problem of having multiple exit nodes only occurs when there are nested `break` or `continue` statements that branch outside the inner-most loop to which they belong. The default case (of a `break` statement with no label specified) corresponds to an exit node that branches outside the loop, but to the same successor node as the normal exit would go. In this case, weaving an *after* piece of advice could be done either at the end of each exit node (possibly at multiple points, as described previously) or at the beginning of the (unique) successor node (which thus guarantees a single weaving point). Weaving an *after-advice* (at a single weaving point) therefore consists of inserting a *pre-successor*, i.e., a new node inserted prior to the successor node, to which the jumps from the exit nodes to the (unique) successor node are redirected.

A loop with a unique successor node can also be reduced to a single node in the control-flow graph. This then makes it possible to weave an *around-advice* at the join point for the loop.

Just as there are two different constructs for writing after-advice depending on whether the execution returns normally or throws an exception[3], so might be abrupt exits be handled differently (due to `break` statements). However, there are cases where it is not possible to tell from the bytecode how such exits would differ from those due to the main condition of the loop evaluating to false. This is a limitation that might have been avoided if a source-code representation had been used, but it does make the model robust to changes in programming style, as illustrated by the code in Figure 7. The two loops in the figure might well produce the same bytecode and control-flow graph, in which case the use of `break` would not be distinguishable from the use of the "`&&`" operator. It would thus be impossible to treat an exit from the loop due to the `break` statement any differently than an exit from the loop due to `b` evaluating to `false`.

```
while (a && b) {
   /* Do something */
}

while (a) {
   if (!b)
      break ;
   /* Do something */
}
```

Figure 7: Considered special handling of `break` statements.

2.5 Loops with a unique exit node

The full potential of a loop join point can only be exploited if its model comprises information regarding the behaviour of the loop. In particular, it can be useful to predict as far as possible that the loop iterates over a specific range of integers or over an `Iterator` (see Section 3). However clever such a prediction may be, the programmer of an aspect dealing with loops might want to handle cases where there is no possibility of an abrupt exit (*i.e.*, there is no `break` statement in the loop). As shown in Figure 7, this case may also exclude loops with complex conditions (in particular expressions comprising *and* operations, which may create several exit points).

2.6 Summary

Three categories of loops have been identified, with increasing degrees of constraint. All three forms could be implemented by a different pointcut, each with different meaning and weaving capabilities. The more general form (several successor nodes possible) would only allow the weaving of *before*-advice, and possibly *after*-advice if the implementation of the weaver allows multiple weaving points. The intermediate form (unique successor node possible) and the restricted form (only one exit node and one successor node) would allow the weaving of *before*-, *after*- and *around*-advice. The latter also guarantees that there is a single condition for exit from the loop. This information is summarised in Table 2.6; it will be used for context exposure in Section 3.

3. CONTEXT EXPOSURE

Although loops do not have arguments in the same way as other join points (such as method calls), they often depend on contextual information to which programmers may want access. In particular, two forms of contextualised loops are frequently found:

- loops iterating regularly over a range of integers (presented in Section 3.1), and
- loops iterating over an `Iterator` (presented in Section 3.2).

Knowing that a loop is of one of these forms allows one to predict the execution behaviour of the loop in some detail. In order to make the resulting predictions meaningful, only loops with unique exit points and unique successors are considered in this section. This prevents loops which have any potential abrupt exits (e.g., using `break` statements) from consideration; a potential use of `break` would make the finding of a range of integers or of an `Iterator` less useful, since the loop might exit before the predicted end.

	Before	After	Around
several successor nodes	$\surd$	multiple weaving points	$\times$
several exit nodes, 1 successor node	$\surd$	$\surd$	$\surd$
1 exit node, 1 successor node	$\surd$	$\surd$	$\surd$

Table 1: Different loop types and their weaving capabilities.

3.1 Loop iterating over a range of integers

Loops iterating over a range of integers, following an arithmetic sequence, are one of the most frequent forms of loops. They consist of: initialising an integer local variable before the loop; incrementing this value by a constant (the stride) at the end of each iteration; and exiting the loop when the value reaches a given maximum value. This form of loop follows the pattern shown in Figure 1.

As explained in [3], exposing the iteration space is essential to make it possible to write aspects for parallelisation. The initial value, the stride and the final value will be available in the execution context of the loop join point model, when possible. Since these values are parameters ruling the execution of the loop, they could be considered, in aspect-oriented models such as AspectJ, as "arguments" of the loop.

Predicting what the range of integer values is going to be at the time of execution is not always possible. In order to be exposed to the join point model, these values have to be determinable before the join point is encountered. The availability of these values will depend on the capabilities of the implemented static analysis in the shadow matcher. Determination of these values ought to be implemented in a conservative way, discarding the cases where it cannot be certain that these values will not change during the execution of the loop.

3.2 Loop iterating over an Iterator

Another frequent form of loop (found in particular in Java programs) is that conducted by an `Iterator`. In a manner similar to that presented in Section 3.1, the instance of `Iterator` controlling the loop can be seen as an "argument" to be included in the join point context.

[3]"`after() returning(...):`" executes the advice after a normal execution, "`after() throwing(...):`" executes the advice if an exception has been thrown, and "`after():`" executes the advice in both cases.

3.3 Parallel with Java 5 for-construct

Java 5 offers a new way to write for-loops iterating over all the elements of an array or `Collection`, similar to "for-each" constructs in certain other languages; this is shown in Figure 8.

```
/* Before Java 5 */
Collection c ;
for (Iterator it = c.iterator() ; it.hasNext(); ) {
   Object obj = it.next() ;
   /* Do something with obj */
}

/* Since Java 5 */
Collection<Object> c ;
for (Object obj: c) {
   /* Do something with obj */
}
```

Figure 8: Example of new Java 5 for-loops.

For iterating over the elements of an array or of a `Collection`, the for-loop construct before Java 5 relies on the abstraction provided by an array index or, respectively, by an `Iterator`. Java 5 gives a new abstraction, more meaningful in terms of data representation. The data guiding the loop execution is directly and explicitly included in the way the for-loop is written in the source code.[4] This is a useful piece of information regarding this kind of loop, and the loop join point model should also be able to expose it, wherever possible. It can also be useful for loop selection, as described in Section 4, and for certain forms of parallelisation, as described in Section 6.

4. LOOP SELECTION

This section analyses and proposes solutions to the problem of writing pointcuts for loops. In particular, the aim is to determine which characteristics can be used for making a selection. In aspect-oriented systems such as AspectJ, the means of selection for a join point is, in most cases, ultimately based on the naming of some source element characterising the join point, possibly by using a regular expression. For example, to advise a method call or a group of methods, the pointcut has to contain an explicit reference to some names characterising the method signatures, whether it be a pattern matching the name of the methods, or a pattern matching the parameter types. Since loops cannot be named, it is impossible to use a name-based pattern to write a pointcut that would select a particular loop.

Neither loop labels, nor Java 5 (or C#) metadata, can be used to identify a particular loop in a method. Firstly, the loop labels will not be kept in the bytecode (and, in any case, they are rarely used, unless motivated by a `break` statement branching outside an inner loop). Secondly, Java 5 metadata cannot be applied to statements (apart from variable declarations).

If it is known for certain that all the loops within a method are to be advised, it would be possible, in AspectJ, to use pointcut constructs such as `withincode` or `cflow` to restrict the pointcut to all the loops contained in the methods traditionally picked up by those constructs. However, selecting only one of several loops within the same method turns out to be impossible without any further mechanism.

[4]This is solely a source-code enhancement; the bytecode still contains `Iterators` (for `Collections`) or temporary variables (for arrays).

In order to solve this problem, it is proposed that selection of loops is made to rely on the data being processed, as well as the method in which the join point's shadow is located. In this case, the context —or what was called the "arguments" of the loop in Section 3— can be used to refine the selection. For example, the programmer might want to write a pointcut that would only select loops iterating over a specified range of integers, over a particular array, or over a particular `Collection`. Such an example is shown in Figure 11 (Section 6): the parallelising advice only applies to arrays of `byte`s.

More speculatively, there might be a potential application for metadata, which could be introduced in the declarations of the local variables that refer to the arrays, `Collection`s or `Iterator`s utilised as "arguments" to certain loops.

5. IMPLEMENTATION IN abc

Although the loop join-point model could potentially be implemented in various aspect-oriented tools, based for example on Java or C#, the focus has been put on a model integrable into AspectJ. The implementation uses `abc`,[5] an alternative AspectJ compiler, for two main reasons:

- extensibility was at the core of the `abc` design [2]; and
- `abc` relies heavily on the Soot framework [12], which provides most of the infrastructure for performing the analyses, in particular those described in Section 2.2.

This section describes an extension for `abc`, known as *LoopsAJ*, which implements a loop join point for AspectJ and subsequently provides the `loop()` pointcut. The latter picks out loops with unique exit points (as described in Section 2.5) and provides contextual information where possible. Other pointcuts for the other forms of loops could also be provided (by lowering the degree of constraint imposed in the shadow matcher).

5.1 Shadow matching

The Soot framework, and subsequently `abc`, use Jimple, which is a three-address representation of bytecode. This makes it possible to look for loops at bytecode level (as described in Section 2.1). The shadow matcher and all pre- or post-transformations operate on this representation.

LoopsAJ extends the method that finds the shadows in each method, so that it looks for loops as well. For each method processed, the control-flow graph and its corresponding dominator tree are built using the Soot framework toolkits. Then combined loops are identified, as described in Section 2.2.

`abc` provides two kinds of classes representing a shadow-match: `BodyShadowMatch` and `StmtShadowMatch` (both extend `ShadowMatch`). The former is utilised when the shadow

[5]`http://www.aspectbench.org/`

is the whole method body; for example, when a method-execution pointcut is used. The latter is used for pin-pointing a specific statement (or group of statements) in the method; for example, when a method-call pointcut is used.

One of the requirements of `abc` is to insert `nop` operators in the shadow, at the points where *before* and *after* [6] pieces of advice might be woven. Given this, most of the `abc` infrastructure can already handle loop shadows if they are treated like `StmtShadowMatch` for *before* and *after* pieces of advice.

However, handling *around* pieces of advice requires a few modifications in the `abc` around-weaver [6]. One of the cases where a group of statements is used is the constructor-call shadow match. In this case, two consecutive statements are included in the shadow-match. However, loop shadows are not necessarily formed by consecutive statements. Indeed, at bytecode or Jimple level, the blocks forming a given loop may be spread across the method, with jumps from one block to another leaving blocks that do not belong to the loop interleaved between blocks that do. For this reason, `StmtShadowMatch` has been extended by `NonContiguousStmtGroupShadowMatch`, for which the around weaver has been modified in order to utilise its new type.

5.2 Transformations for context exposure

Exposing the context, as described in Section 3, depends on the cleverness of analysis and on the feasibility of certain transformations. For the context exposed to make sense, it has to be constant during execution of the join point.

In order to ensure this, as long as it is possible to predict that the transformation will not change the meaning of the loop, loop-invariant assignments are moved to the pre-header (before the shadow matching takes place), using a scheme inspired by [1, Ch 10.7].

5.2.1 Exposing the boundaries or the `Iterator`s

Further, in the case of loop iterating over a range of integers, if the context values are numerical constants, temporary variables are introduced and initialised in the pre-header, in order to make it possible to modify these values via calls to `proceed(...)` within an *around-advice*. An example transformation is shown in terms of source-code in Figure 9.

The part of the implementation that determines the feasibility of these transformations uses the dataflow analysis facilities provided by Soot; these have also been used to implement a code-motion method and a reaching-definition analysis [10, 1].

5.2.2 Exposing the originating array or `Collection`

It is not always possible to find an array to which the range of integers corresponds (i.e. when `minimum=0`, `stride=1`, and `maximum` is the length of the array). For example, if the boundaries and the array are passed as arguments to the containing method, finding the array that was the origin

```
/* Moving the invariants outside */
int i = 0 ;
while (i < 10) {
   /* ... */
   int stride = 3 ;
   i = i + stride ;
}
// -------------------------------------------------
/* First step: moving the invariants outside */
int i = 0 ;
int stride = 3 ;
while (i < 10) {
   /* ... */
   i = i + stride ;
}
// -------------------------------------------------
/* Second step: storing the boundaries in
   temporary variables */
int stride = 3 ;
int minimum = 0 ;
int maximum = 10 ;
int i = minimum ;
while (i < maximum) {
    /* ... */
    i = i + stride ;
}
```

Figure 9: Code-motion example.

of these values might require much more complex, cross-methods and points-to, analysis. The current implementation requires at least the statement initialising `maximum` to the length of the array to be within the same method as the loop.

Similarly, a `Collection` will only be exposed if the `Iterator` used for the loop comes from a call to `Collection.iterator()` and `Iterator.next()` is not called before the beginning of the loop.

5.2.3 Writing pointcuts

For loops iterating over a range of integers, the boundary values are passed via the `args` construct of AspectJ, to which `int` values are bound (for `minimum`, `maximum` and `stride`). Also, an extra argument will be bound to the originating array, if it has been found.

For loops iterating over an `Iterator`, the first argument of `args` will be bound to the corresponding instance of `Iterator`. Also, an extra argument will be bound to the originating `Collection`, if it has been found.

In cases where the originating array or `Collection` do not matter, it is recommended to use the double-dot wildcard notation ("`..`") [5], to make the argument optional. For example:

- `loop() && args(min, max, stride)` will match only loops iterating over an arithmetic sequence of integers for which the compiler was unable to find an array (although it may exist);
- `loop() && args(min, max, stride, ..)` will match all the loops iterating over a particular arithmetic sequence of integers; and
- `loop() && args(min, max, stride, array)` will match all the loops iterating over an array, a refer-

[6]It is not always possible to insert *after-advice*, as described in Section 2.

ence to which will be bound to pointcut parameter "`array`".

5.3 Limitations

The limitations in the current implementation of *LoopsAJ* divide into two categories: limits of analysis and predictability; and limits due to features not yet written in this preliminary version.

5.3.1 Analysis and predictability

One of the main limitations is the predictability on which the invariant code-motion is based. Although code-motion is currently done successfully in most useful cases, it will not be performed in cases where an invariant is not spotted by the data analysis. The implementation of such transformations ought to be conservative, that is to say, it should not be done unless it is certain that the resulting code will be equivalent.

Another limitation is the lack of points-to analysis in respect of `Iterator`s. Indeed, even though an `Iterator` instance may look like it is being iterated regularly in the loop (*i.e.* there is one and only one call to `next()` per iteration), nothing guarantees that no other thread is holding a reference to the same `Iterator` and is calling `next()` concurrently. In this respect, the exposure of `Iterator`s is probably not conservative enough. There may be a solution to this problem if a form of whole-program analysis were to be used. (This concurrency problem does not occur for loops iterating over a range of integers since `int` is a primitive type and the `int` values are local variables that cannot be modified by another thread.)

More generally, there could be further dependency analysis to provide safeguards in case of concurrent execution of a join point. Again a whole-program analysis may be required to be sure that a loop can be executed in parallel. Such analyses can be much more complicated, and are beyond the scope of this article.

Whatever the implementation of a weaver capable of handling loop join points is, it should be stated clearly by the implementors how conservative their implementation is, in particular, how certain it is that a specified `Collection` is at the origin of an `Iterator`.

5.3.2 Future work

The *LoopsAJ* implementation is still being worked on. In its current state, only cases where the loop termination condition is of the form $i < max$ and where the increment is of the form $i = i + ...$ are handled. Eventually, other conditions, using $\leq$, $>$ or $\geq$, will be handled as well.

Also, the *around* weaving has only been implemented in cases that do not generate closures (see [6] for further details). One of the difficulties being currently addressed is to keep the block graphs up-to-date after weaving an *around*-advice. At the time of writing, the implementation works only partially on loop nests.

Moreover, the handling of traps is not always updated, which can lead to the generation of bytecode with incorrect exception tables.

6. ASPECTS FOR PARALLELISATION

This section shows an application of the `loop()` pointcut, namely parallelisation of loops.

The example advice shown in Figure 10 executes in parallel (using cyclic loop scheduling) all the loops contained in class `LoopsAJTest` which are recognised as iterating over a range of integers. As shown, the `loop()` pointcut combines ideally with the "*worker object creation pattern*" [7], which creates new `Runnable`s to execute join points on separate threads.

```
void around(int min, int max, int step):
  within(LoopsAJTest)
  && loop() && args (min, max, step, ..) {
     int numThreads = 4 ;
     Thread[] threads = new Thread[numThreads] ;
     for (int i = 0 ; i<numThreads ; i++) {
        final int t_min = min+i ;
        final int t_max = max ;
        final int t_step = numThreads*step ;
        Runnable r = new Runnable () {
           public void run() {
             proceed(t_min, t_max, t_step) ;
           }
        } ;
        threads[i] = new Thread(r) ;
     }
     for (int i = 1 ; i<numThreads ; i++) {
        threads[i].start() ;
     }
     threads[0].run() ;
     try {
       for (int i = 1 ; i<numThreads ; i++) {
         threads[i].join() ;
       }
     } catch (InterruptedException e) {  }
  }
```

Figure 10: Loop parallelisation using Java Threads.

The aspect shown in Figure 11 is slightly more complex. It executes in parallel, using MPI for Java,[7] the loops working on a array of `byte`s that are in method `LoopsAJTest.test`. The original array, `a`, is exposed to the pointcut. It is then sliced into an array `p` per MPI task. Then `proceed` uses array `p` instead of `a`, so the loop in each MPI task only iterates over its local portion of `a`.

When using these kinds of aspects, the programmer needs to make sure that the loops that are going to be executed in parallel can actually be parallelised. As explained in Section 5.3, no inter-dependency analysis is currently performed.

7. ISSUES RELATED TO EXCEPTIONS

This model and the way the loops are recognised do not work properly if exceptions are used in the methods advised. Firstly, exceptions handlers are activated according to position between two bytecode instructions. Weaving may insert code within the range of an exception handler when this may not be intended. Secondly, combined loops correspond approximately to loops written in the source-code, as long as the graph is reducible (or well-structured). This is the case for bytecode produced by Java source-code when the graph does not contain edges due to the potential handling of exceptions. However, taking the exceptions into account adds

[7]`http://www.hpjava.org/mpiJava.html`

```
import mpi.* ;

aspect MPIParallel {
  int rank ;
  int nthreads ;

  void around(String[] arg):
      execution(void LoopsAJTest.main(..))
      && args(arg) {
    try {
      MPI.Init(arg);
      rank = MPI.COMM_WORLD.Rank();
      nthreads = MPI.COMM_WORLD.Size();

      proceed(arg) ;

      MPI.Finalize();
    } catch (MPIException e) {
      e.printStackTrace() ;
    }
  }

  void around(int min, int max, int stride, byte[] a):
        loop() && args(min, max, stride, a, ..) &&
        withincode(* LoopsAJTest.test(..)) {
    try {
      MPI.COMM_WORLD.Barrier();
      int slice_length = a.length / nthreads ;
      byte[] p = new byte[slice_length] ;
      if (rank == 0) {
        for (int i = 0 ; i < slice_length ; i++) {
           p[i] = a[i] ;
        }
        for (int k = 1; k < nthreads; k++) {
           MPI.COMM_WORLD.Ssend(a, k*slice_length,
                        slice_length, MPI.BYTE, k, k) ;
        }
      } else {
        MPI.COMM_WORLD.Recv(p, 0, slice_length,
                      MPI.BYTE, 0, rank) ;
      }
      proceed(0, slice_length, 1, p) ;
      MPI.COMM_WORLD.Barrier();
    } catch (MPIException e) {
       e.printStackTrace() ;
    }
  }
}
```

Figure 11: Loop parallelisation using mpiJava.

```
public int foo (int i, int j) {
   while (true) {
      try {
         while (i < j)
           i = j++/i ;
      } catch (RuntimeException re) {
         i = 10 ;
         continue ;
      }
      break ;
   }
   return j ;
}
```

Figure 12: Example of loops involving exceptions.

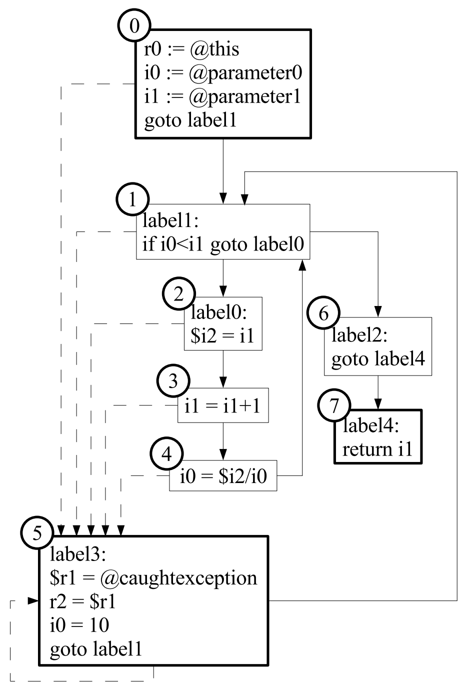

Figure 13: Complete block-level control flow graph.

extra edges to the graph, which may make the graph non-reducible. The main characteristics of non-reducible graphs are that: (a) loops may have several headers; and (b) there are still cycles in the graph after all the back edges have been removed.

To illustrate this problem, Figure 12 shows an example of code that involves loops and exceptions (taken from [9]). Figure 13 shows the corresponding complete block-level control-flow graph (including exceptions, shown as dashed lines) using the Jimple intermediate representation for this example, as produced by the control-flow graph viewer included in the Soot framework. The edges due to traps are dashed only in the illustration; in the system they are treated as regular edges. Without entering into the details of the syntax of Jimple, in this example, `i0` and `i1` represent `i` and `j`, respectively, in the Java source-code.

The back edges found using the method described in Section 2.2 are $4 \rightarrow 1$ and $5 \rightarrow 5$. The graph is not reducible because, after these back edges have been removed, a cycle made of nodes 1 and 5 exists. This gives a loop comprising nodes 1, 2, 3 and 4 —which corresponds to "`while (i<j) i=j++/i;`" in the source-code— and another loop comprising node 5 (which handles the exception in the source-code) only. Although the first loop is meaningful, and corresponds to what would be naturally expected by looking at the source-code, the second would cause *before-advice* to be inserted just before the exception is caught, and *after-advice* just before "`continue`" (without even dealing with the correctness of trap handling). This effect would not necessarily be meaningful or useful for advising this loop.

Moreover, such code is not robust to changes of compilation strategy. For example, a different compiler might insert an extra, "useless" `goto` statement between nodes 0 and 1 in this graph, yielding the control-flow graph shown in Figure 14. In this case there is a third back edge ($5 \rightarrow 8$), which gives a natural loop that could be assimilated into the outer "`while(true) { ... }`" loop in the source-code. The method used so far is not suitable for such cases involving exceptions, since the loop model should depend as little as possible on the compilation strategy utilised.

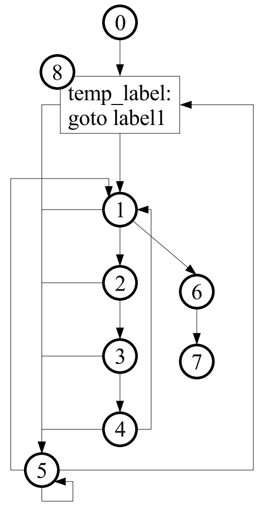

Figure 14: Another possible control-flow graph.

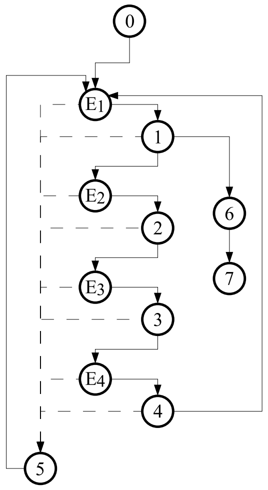

Figure 15: Control-flow graph with special nodes for exceptions.

The problem with exceptions lies in the edges they add to the graph. In particular, the edges between the predecessors of the first node that could throw an exception and the node catching the exceptions distort the dominator tree when trying to find the back edges. Because these edges do not actually come from the predecessors, but from a point just before the nodes that could throw an exception, a possible solution would be to change this representation and to introduce separate nodes for throwing exceptions. For each node A that could potentially throw an exception represented as an edge from A to B, a new node E_A would be inserted before A, so that all the edges pointing to A would be redirected to E_A, and an extra edge $E_A \rightarrow B$ would be added. The resulting control-flow graph for the example in Figure 12 is sketched in Figure 15. This is similar to the graph in Figure 13, but contains extra nodes E_1, E_2, E_3 and E_4, which preceed nodes 1, 2, 3 and 4, respectively, and represent the cases where an exception would be thrown in one of these nodes, thus preventing the operations in that node from being performed. This representation now gives two back edges ($4 \rightarrow E_1$ and $5 \rightarrow E_1$) corresponding to a single combined loop. To avoid ambiguity, chains of unconditional `goto`s should be considered as a single node if they can all throw exceptions to the same catching blocks. This approach has not yet been implemented in our prototype.

8. RELATED TOPICS

This section explores two related potential fine-grained join points (i.e. join points that recognise complex behaviour within a method and not only at the interface of the object), namely a loop-body join point (Section 8.1), and an "if-then-else" join point (Section 8.2).

8.1 "Loop-body" join point

The model of loop join point presented thus far takes an outside view of the loop; the points *before* and *after* the loop are not within the loop itself. As a consequence, however many iterations there may be for a given loop, *before* and *after*-advice will be executed only once. For some applications, for example for inserting a piece of advice before each iteration, it might be desirable to advise the loop body. However, the semantics would be difficult to define.

Even in the source-code, there is ambiguity about where to weave *before* and *after* advice in such a case. For example, is the termination condition in the loop-body or not? (see Figure 16). This question is even more relevant for complex conditions that may include calls to methods.

```
int i = 0;
while (i<2) {
   /* Is ``before'' the loop-body right here, or
      should it be before (i<2) is evaluated?     */
   System.out.println("i: "+i) ;
   i++ ;
   /* Is ``after'' the loop-body here? Would ``i++''
      be included in the loop-body of the equivalent
      for-loop?       */
}

i = 0 ;
do {
   /* Before the loop-body */
   System.out.println("i: "+i) ;
   i++ ;
   /* Is ``after'' here, or should it be after (i<2)
      has been evaluated?    */
} while (i<2) ;
```

Figure 16: Loop-body join point: where are "before" and "after"?

Again, a basic-block control-flow approach may solve the problem. It may be possible to define that "before" the loop-body is the point at the begining of the header, included in the loop, and that "after" the loop-body is a point inserted on the back edge of the natural loop. If there were several back edges in the corresponding combined loop, an equivalent of the "pre-header" could be inserted between the back edges and the header, in order to keep a single weaving point. In the case of a `while`-loop or a `for`-loop, "before" the loop-body would also be before the evaluation of the condition.

Without any enhancement, such a model would not comprise any contextual information (or "arguments" to the loop-body).

8.2 "If-then-else" join point

Why stop at loops? Similar techniques could be applied so as to provide aspect-oriented languages such as AspectJ with a model for an "if-then-else" join point.

At source-code level, there is again the question of whether the evaluation of the condition should or should not be included in the "if-then-else" join point.

A basic-block control-flow approach may help to define a model. A possible way to find the shadows of "if-then-else" constructs might be in the combined use of dominators and *postdominators*. "*[We] say that node* p postdominates *node* i *[...] if every possible execution path from* i *to [the exit] includes* p" [10]. Given a node a that branches conditionally to other nodes (unconditional branching presents no interest), the smallest subgraph G of the control-flow graph that contains another node b such that a dominates all the nodes in G and b postdominates a, would represent an area of conditional execution, starting from a and joining back at b. Since a would dominate all the nodes in G, it would be the unique entry node to G. Since b would postdominate a, b would be the unique exit node from G. Just before the conditional jump in a would be the *before* weaving point, and just before b (for edges coming from inside G) would be the *after* weaving point.

Again, it is unclear what kind of contextual information could be included in such a model. Without it, such an "if-then-else" join point would represent areas of code that will only be partially executed (for a given (dynamic) join point).

However, going a step further by making it possible to advise `goto` statements directly in the bytecode, may break modularity and consistency, even within a method, which would counteract the benefits of using aspects.

Apart from the usual debugging and tracing applications of such join points, another successful approach for defining fine-grained join points (including conditional `if` blocks) has been applied to code-coverage analysis [11].

9. CONCLUSIONS

The paper demonstrates that it is possible to provide AspectJ (and perhaps other aspect-oriented systems) with a loop join point, which can be applied, in particular, to loop parallelisation.

The two main remaining difficulties are: (a) the cleverness of the analysis for context exposure; and (b) the mechanisms for loop selection. The context analysis is mostly implementation dependant. But the loop selection problem is more fundamental, especially because loops cannot be named or tagged. Loop selection based on contextual data can work, but is also limited with the current AspectJ join points. A possible way forward would be to use dataflow pointcuts, as presented in [8]. An extension of this pointcut that would predict the dataflow [8] would perhaps make it possible to determine at compile time which loops should be advised by a parallelising aspect, therefore reducing the overhead of run-time `cflow` (or `dflow`) checks.

[8]A `pdflow` pointcut could be imagined in a similar way as the `pcflow` pointcut mentioned by G. Kiczales in his keynote talk at AOSD'2003.

10. ACKNOWLEDGMENTS

B. Harbulot acknowledges the support of the Engineering and Physical Sciences Research Council (EPSRC) during his doctoral studies.

11. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1985.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *Proceedings of the 4th international conference on Aspect-Oriented Software Development (to appear).* ACM Press, 2005.

[3] B. Harbulot and J. R. Gurd. Using AspectJ to separate concerns in parallel scientific Java code. In *Proceedings of the 3rd international conference on Aspect-Oriented Software Development*, pages 122–131. ACM Press, 2004.

[4] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35. ACM Press, 2004.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.

[6] S. Kuzins. Efficient implementation of around-advice for the AspectBench Compiler. Master's thesis, Oxford University, UK, September 2004.

[7] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning, 2003.

[8] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. *Lecture Notes in Computer Science 2895, Proceedings of The First Asian Symposium on Programming Languages and Systems (APLAS'03).*, pages 105–121, 2003.

[9] J. Miecznikowski and L. Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In *Proceedings of CC'02*, 2002.

[10] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[11] H. Rajan and K. Sullivan. Aspect language features for concern coverage profiling. In *Proceedings of the 4th international conference on Aspect-oriented software development (to appear).* ACM Press, 2005.

[12] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical report, Sable Group, McGill University, Montreal, Canada, July 1998.