Extending the Field Access Pointcuts of AspectJ to Arrays

Kung Chen[1] and Chin-Hung Chien[2]
[1]*Dept. of Computer Science, National Chengchi University, Taiwan*
chenk@cs.nccu.edu.tw
[2] *Centra R&D Division, Hon Hai Precision Industry Co., Ltd., Taiwan*
richard.chien@foxconn.com

ABSTRACT

Join points and advices are two fundamental constructs of aspect-oriented programming languages. AspectJ provides a large set of useful pointcuts that enables aspect-oriented programmers to pick out target join points for advice weaving in a highly flexible manner. However, the field access pointcuts of AspectJ do not support array objects in full. When an element of an array field is set or referenced, the corresponding index values and assigned value are not exposed to the advice. This paper presents an extension of AspectJ's field access pointcuts to arrays that exposes such useful context information. We have implemented this extension using the abc compiler for AspectJ. The core of our implementation is a finite-state machine based pointcut matcher that can handle arrays of multiple dimensions in a uniform way.

1: Introduction

Aspect-oriented programming (AOP) [1] aims at modularizing crosscutting concerns such as profiling and security that are generally spread throughout the components of a software system and tangled with core functionalities. In AOP, a program consists of many functional modules (base program) and some aspects that encapsulate the crosscutting concerns. An aspect module provides two kinds of specifications to realize crosscutting concerns of the base program. The first kind defines *pointcuts*, which select a set of well-defined points in the execution of a program, called *join points*, designating where to crosscut other modules. The second one defines *advice*, which is a piece of code associated with a pointcut that will be executed when any join point in the pointcut is reached. The complete program behavior is derived by some novel ways of composing functional modules and aspects according to the specifications given in the aspects. This is called weaving in AOP. Weaving results in the behavior of those functional modules impacted by aspects being modified accordingly.

AspectJ is a seamless aspect-oriented extension to the Java programming language [2]. It provides a powerful join point model and an expressive pointcut language to select join points for advising. The join points in AspectJ include method call or execution, constructor call or execution, field reference (read) and set (write) access, exception handlers, etc. Pointcuts are specified by combinations of primitive pointcut designators (PCDs). All join points have a corresponding PCD; other main PCDs include "*within*", which restricts the scope of selected join points, and "*args*", which exposes the argument values associated with the underlying join point.

This paper concerns the two primitive pointcut designators of AspectJ that select the join points of field reference and set operations, namely *get(FieldPattern)* and *set(FieldPattern)*. They pick out each field reference or set join point whose signature matches *FieldPattern*. Although they are very useful for installing advice to monitor selected class fields, there are still some limitations in using these field pointcut designators. Specifically, when the fields we are interested in are arrays, these pointcut designators do not catch the desired join points as we would expect, and neither do they expose the index values of the array element that is being set or retrieved. This will restrict the application scope of these field access pointcuts to a certain degree. Therefore, we propose to enhance them with better support for array fields and implement this enhancement using the Aspect Bench Compiler (*abc*) [3] for AspectJ.

The remainder of this paper is structured as follows. Section 2 describes the problem background of the proposed extension in more detail. A brief review of relevant features of AspectJ is also included therein. Section 3 presents the proposed array field pointcuts and states in detail our design considerations behind them. Section 4 describes the implementation we develop for the array field pointcuts. Finally, Section 5 concludes.

2: Problem Background

This section describes the issue of AspectJ's field access pointcut that motivates our work and states the specific extension we propose to enhance AspectJ. To make this work self-contained, this section begins with a brief review of the relevant features of AspectJ.

Advice in AspectJ is an anonymous method bound to a pointcut and tagged with one of the three keywords: *before*, *after*, or *around*. The before advice and the after advice are executed before and after the join points selected by their pointcuts, respectively. The case for the around advice is more subtle; it is executed in place of its join points. Inside the around advice, we may choose to resume the execution of its join points by calling the special built-in method *proceed()*, or simply bypass their execution.

The field get and set pointcuts of AspectJ select any field references for advising. Combining them with other pointcut designators such as *args(aValue)*, we can easily monitor any attempt to access specific fields of certain classes. For example, the following aspect, taken from the programmer guide of AspectJ, guards a static integer field `x` of class `T` against any invalid change values.

```
aspect GuardedX {
  static final int MAX_CHANGE = 100;
  before(int newval): set(static int T.x)
                      && args(newval) {
  if (Math.abs(newval - T.x) > MAX_CHANGE)
      throw new RuntimeException();
  }
}
```

Here the *before advice* of aspect `GuardedX` will be triggered before any assignments made to the field `x` of class `T`, namely `T.x`. Moreover, the other PCD, *args()*, grabs the assigning value and passes it to the advice via the advice parameter, `newval`. Once executed, the advice will ensure that the change to the field is allowed.

While convenient for monitoring common fields, the two field access pointcuts do not handle array fields well. Consider the following simple class that uses an integer array `ar`.

```
01 public class FieldPointcuts {
02   static int ar[];
03   public static void main(
04                  String[] args) {
05     ar = new int[] {100}; //set
06     ar[0] = 200;          //get
07  }
08}
```

Suppose that we are interested in monitoring any changes made to the field `ar`. Using the pointcut designator "`set (* *.ar)`", we can only catch the assignment to `ar` as a whole in line 05, but not the array element assignment in line 06. By contrast, there is a field get join point in line 06 that can be singled out using the following field get pointcut and *after* advice:

```
after() returning(int[] a) : get(* *.ar)
```

However, this field get pointcut still cannot fulfill our requirements. First, we cannot get the new value assigned to the array element. Second, we cannot get the index value of the array element being assigned. Such shortcomings may severely constrain what one can do with these field pointcuts. For example, in another project [4], we have attempted to use the field set pointcut to construct an object reference graph for tracking the connectivity and heap memory usage of objects of a set of certain classes in a Java program. This simple scheme of reference recording appears to work. Indeed, recently we also learned of another work adopting an approach quite similar to ours [5]. Unfortunately, due to the lack of information on index values, we are not able to cover the case when the source end of an object reference is an array element. Hence we set out to look for an extension of the field set and reference pointcuts for our project.

Soon we realize that we are not the first to consider an extended pointcut for catching setting of array elements. Bruno Harbulot had implemented a version of *arrayset* pointcut using the Aspect Bench Compiler (*abc*) [3] of AspectJ. He proposed the following *arrayset* pointcut designator[1].

```
before(int i, Object s, Object[] a):
  arrayset() && args(i, s) && target(a)
{
  System.out.println (" Arrayset:
       ["+i+"/"+(a.length-1)+"] = "+s) ;
}
```

Here the pointcut designator `args(i, s)` exposes both the index value `i` and the object being assigned to an array element, and the pointcut designator `target(a)` exposes the array object being assigned. However, this extension bases its implementation on treating array element set as a call to a "`set(int index, Object newValue)`" method, and thus works only for one-dimensional arrays. Besides, there is no stand alone implementation of it that can be downloaded for further experiment. Therefore, we decided to implement a version of our own that can work for multi-dimensional arrays, and make it available to other interested users[2]. Our implementation is also based on *abc*, yet we take the standard field set pointcut as the basis and developed our extension on top of it.

3: Array Field Access Pointcuts

This section first states our design considerations behind the array field pointcuts and then specifies the proposed pointcuts in detail.

3.1: Design Considerations

In the beginning, it is tempting to design a general pointcut designator for capturing any array element access following Harbulot work, regardless of whether the array is a class field or not. However, the nature of Java arrays makes this approach too naïve to suit our purpose. Specifically, a multi-dimensional array in Java

[1] Post to the [abc-users]: Allowing ArrayRef as left-hand side operands for around-weaving, Nov. 18, 2004.

[2] The ArrayPT project at http://abc.comlab.ox.ac.uk/projects

is actually an array of arrays. On the byte-code level, an assignment to an element of a multi-dimensional array is actually realized by a sequence of assignments to elements of one-dimensional arrays. For example, consider the following assignment to the two-dimensional array ss.

```
ss[0][1] = “two join points”;
```

When translated to bytecode, there are two assignments to two one-dimensional arrays. If we use the pointcut shown before, *arrayset() && args(i, s) && target(a)*, there will be two join points matching it, one for each dimension of `ss`. Moreover, one of the join points is indeed an assignment to an intermediate array. One may think of generalizing the *args* designator to catch two-dimensional arrays and re-write the pointcut to the form: *arrayset() && args(i1,i2,s) && target(a)*. But this does not fully solve the problem. Consider the case where we also used another three-dimensional array in the same program as follows.

```
sss[0][1][2] = “two join points, too”;
```

Similarly, using the revised pointcuts, the assignment above will again leads to the matching of two join points. Such ambiguous join point matching is clearly unacceptable and we must avoid it.

We address this problem by fixing the array target(s) of interest. Since our main purpose concerns heap memory usage, we only need to focus on arrays which are fields of some class; arrays local to a method are stack-allocated and thus irrelevant. Hence we do not attempt to capture all array element assignments but only assignments to class fields which are arrays. To realize this, we include a specification of field signature in our array set/get pointcuts. As will become clearer later, this decision will save us from the ambiguous matching of join points described above.

Furthermore, assignments to a multi-dimensional array can take several forms. Our array field set pointcuts must be flexible enough to select any assignment form. For example, consider the following variety of assignments to the three array fields of class `Watch`.

```
01 class Watch {
02 String [][][] sss= new [2][2][2];
03 String [][] ss= {{“x”,”y”},{“w”,”z”}};
04 String []   s= {“abc”, “def”};
05 sss[0]       = ss;
06 sss[0][1]    = s;
07 sss[0][1][1] = “change”;
08 ss [0] = s;
09 ss[0][1] = “Me too”;
10 s[0] = ss[1][1];
… }
```

There are six array element assignments and thus six array field set join points above (Line 5, 6, 7, 8, 9 and 10). Our array field set pointcuts should be able to select all of them, either individually or as a group. Similar requirements also exist for the array field get pointcut.

Finally, we intend to make our new pointcuts a conservative extension of the standard field pointcuts. In particular, they should be as orthogonal as possible to the other AspectJ pointcuts that can work with the standard field pointcuts. This will make writing pointcut composition using our array field pointcuts as easy as using the original field pointcuts.

3.2: Specification of the New Pointcuts

Basically, our array field pointcuts look very like the standard field pointcuts. Specifically, the pointcut designators we propose for them take the following forms: *arrayset(aFieldSignature)* and *arrayget(aFieldSignature)*. The former catches assignments made to all array fields that match the *signature* specification, while the latter catches references to array fields.

The key change is that these new pointcut designators concern only class fields of array types. Hence they must take array index values into account. We shall explain array field set pointcut first. Essentially, all array filed set join points are treated as having a variable number of arguments: the sequence of index values and the value the field is being set to. At a join point, these values can be obtained using an *args()* pointcut designator and then passed to the advice for further processing. For example, the following aspect uses the *arrayset()* and *args()* pointcut designators to monitor the assignments to any array fields of class `Watch` declared above.

```
aspect Monitor {

  before(int ix1, int ix2, int newVal):
     arrayset(* Watch.*) &&
     args(ix1, ix2, newVal) { //advice
  if (newVal > B.bounds[ix1, ix2]) {
    ArraySetSignature sig =
    (ArraySetSignature)tjp.getSignature();
    String field = sig.getFieldType() +
                   sig.getName();
    throws new
     RuntimeException("Bad change"+ field)
  }
 }
```

Here we use the pointcut designator *args(ix1, ix2, newVal)*to get the array index values and the assigned value of the array field assignments. Moreover, like the standard pointcuts in AspectJ, the other join point context can be accessed inside the advice body through the object, *thisJoinPoint*, (abbreviated as *tjp* above). But note that, to handle the particular requirements of *arrayset*, we have extended the standard field set signature, *FieldSetSignature*, to a new signature called *ArraySetSignature*. Through the *ArraySetSignature* object, `sig`, we can obtain the detailed information regarding the field being set.

Astute readers may notice that class `Watch` has three array fields of different dimensions, and wonder assignments to which of them will be captured by the aspect `Monitor`, or whether we can capture any particular set(s) of assignments of the six assignments given there. Indeed, this is exactly one of the

requirements we stated in Section 3.1. Here the pointcut designator *args(ix1, ix2, newVal)*also serves this requirement of selective matching of array field set join points. Specifically, it makes the aspect capture only array assignments whose left-hand side has two index values, which in this case are the assignments in Line 6 and 9. Had we instead specified the pointcut designator *args(ix, newVal)*, aspect `Monitor` will capture the assignments whose left-hand side has only one index value, namely, assignments in Line 5, 8, and 10.

What if we wish to capture all the six assignments? This can be achieved by intentionally leaving out the *args()* pointcut designator and simply write the pointcut designator *arrayset(* Watch.*)*, which will capture all assignments to the array fields of `Watch`. A subsequent question is then how we obtain the index values and assigned value. The answer is simple: inside the associated advice, use the method *thisJoinPoint.getArgs(),* which will return an object array containing the index values and assigned value. Compared to using the *args* designator for picking out index values directly, calling this method inside the associated advice provides great flexibility in situations where the dimension of the array to be dealt with is unknown in advance, as well as when we intend to use one universal pointcut and a single piece of advice for catching assignments to a set of arrays of different dimensions. For example, the following advice catches assignments to all array fields of classes of package *data*.

```
void around() : arrayset( * data.*.* ) {
 try {
   ArraySetSignature sig =
    (ArraySetSignature)tjp.getSignature();
    Object[] args = tjp.getArgs();
    Integer rhsValue =
           (Integer)args[args.length-1];
   if (rhsValue.intValue() > MAX) {
     Log.add("Warning: " +
                        sig.getName());
   for (int i=0; i<args.length-1; i++) {
     Log.add("["+ args[i] +"]");
     Log.add(" exceeds MAX");
   }
   proceed();
 } catch(IndexOutOfBoundException e) {}
}
```

The case for *arrayget* is very similar. The only significant difference is when it is used with *args(),* the context values exposed by *args()* include only the index values, but not the value of the array element being read. To get that value, we can use the *after-returning* advice as follows.

```
after(int ix1, int ix2, Object val)
                 returning(Object val):
 arrayget(* Watch.*) && args(ix1, ix2) {
   // advice body
   …
 }
}
```

When specified in aspect `Monitor`, the advice above will capture the array element reference used in Line 10 of the `Watch` class example.

Furthermore, we can also use *arrayset/arrayget* with other pointcut designators such as *target(), within()* and *withincode()* following the same style with that of the standard field set/get pointuts.

4: Implementation

This section describes the main techniques for implementing our arrayset pointcut. We use the AspectBench Compiler (*abc*) for AspectJ [3] to implement the *arrayset* pointcut. Basically, we follow the standard steps outlined by the *abc* team to develop this extension of AspectJ [6], and our model implementation is the field set/get pointcuts of AspectJ. Among the steps, only the identification of join point shadows and the extension of the pointcut matcher of *abc* are non-trivial and worth mentioning. Furthermore, since the implementation of *arrayget* is very similar to that of *arrayset*, we shall focus on *arrayset* only in the following discussion.

In the AspectJ language model, every dynamic join point has a corresponding static shadow in the bytecode or source code of the program [7]. To insert a piece of advice into the underlying program, the AspectJ compiler must first find the designated join point shadow. Most shadows in AspectJ are easily identified from the bytecode: they are either a single bytecode statement or a bounded region of bytecode. For example, the shadow for a method call join point corresponds to a single bytecode instruction, *invokevirtual*, while the shadow for a method execution is the entire code segment of the method. The shadow for setting an array element also falls into the second category. However, unlike the case of method execution where the "*return*" instruction marks the clear end of a shadow, setting an element of a multi-dimensional array calls for a more elaborate test for the end of a shadow.

To illustrate the issues involved, we need to examine the bytecode pattern generated by a typical Java compiler for assignments to an element of a multi-dimensional array. Basically, all Java arrays are really one-dimensional arrays; multi-dimensional arrays are supported through arrays of arrays. Although setting an array element on the source code level, regardless of array dimension, is simply an atomic statement, it involves more intermediate steps on the bytecode level. Since the bytecode used by *abc* is *Jimple*, which is a simple, typed, and 3-address (stackless) intermediate representation of Java bytecode [8], we shall use Jimple code to show the details. Consider the following assignment to an element of the array field `a` of class `C` in Java.

```
class C {
 …
 Object a[][][] = new Object[2][2][2];
 …
 a[0][1][2] = new String("foo");
 …
```

```
}
```

The Jimple code to implement the above assignment is shown as follows.

```
#1: $r2 = r0.<C:java.lang.Object[][][]a>
#2: $r3 = $r2[0]
#3: $r4 = $r3[1]
#4: $r5 = new java.lang.String
#5: specialinvoke $r5.<java.lang.String:
    void <init>(java.lang.String)>("foo")
#6: $r4[2] = $r5
```

Instruction #1 references the array field `a`, thus marking the beginning of the array field join point shadow. Since `a` is a three-dimensional array, the *abc* compiler employs three intermediate one-dimensional arrays to set the element of `a`, namely, `r2`, `r3`, and `r4`. While the instruction #1 starts the task, the real assignment to `a[0][1][2]` is conducted only until instruction #6. Hence the join point shadow for this array element assignment must extend from instruction #1 to instruction #6. In general, the shadow for array element assignment is a block of Jimple instructions of indefinite length. Besides, there is no simple signpost, such as the "*return*" instruction, for marking the end of the shadow. Indeed, to determine whether a sequence of Jimple instructions implements an assignment to an array element on the Java source level, we have to keep a memory of the target array and all intermediate arrays ever encountered.

Therefore, we developed a finite-state machine (FSM) based pointcut matcher for identifying the join point shadow of array field assignments. The key steps of our shadow matcher are embodied in the FSM; hence we shall describe the operations of the FSM in detail. Basically, the FSM takes one Jimple assignment statement as the input at a time. It will examine the contents of the current assignment using some auxiliary functions and take proper state transitions until the ending state is reached. In particular, the FSM uses the following functions during its operation.

1. **getNext**(): returns the assignment statement following the current one.
2. **hasNext**(): checks the existence of a next assignment; that is, checks if the end of a method body has been reached.
3. **instanceOf**(*variable*, *type*): checks if *variable* is an instance of type *type*.
4. **getRhs**(*assignment*): returns the variable on the RHS of the *assignment*.
5. **getLhs**(*assignment*): returns the variable on the LHS of the *assignment*.
6. **equalBase**(*var1*, *var2*): checks if *var1* and *var2* refer to the same variable.

Figure 1 illustrates the valid state transitions of the FSM. There are four states in the FSM, namely *Start*, *SeenIMArray*, *HaveMatch*, and *NoMatch*, where *Start* is the initial state and *HaveMatch* and *NoMatch* are the ending states. The state transitions among them are determined by the following conditions.

- Cond. #1: *instanceOf*(*getRhs*(ca), ArrayFieldRef)
- Cond. #2: *instanceOf*(*getRhs*(ca), ArrayRef)
  && *equalBase*(*getRhs*(ca), ima)
  && *instanceOf* (*getLhs*(ca), Local)
- Cond. #3: *instanceOf*(*getLhs*(ca), ArrayRef)
  && *equalBase*(*getLhs*(ca), ima)
- Cond. #4: !*hasNext*()

Here "ca" stands for the current assignment statement under examination and "ima" for the current intermediate array involved in the assignment (assignee on the left-hand side). We have also used three Jimple-defined types, "ArrayFieldRef", "ArrayRef" and "Local", as the second argument to the *instanceOf* function in the conditions. These function calls check whether the first argument is a class-level variable of the array type, a method-level variable of the array type, or a method-level variable of any type, respectively.

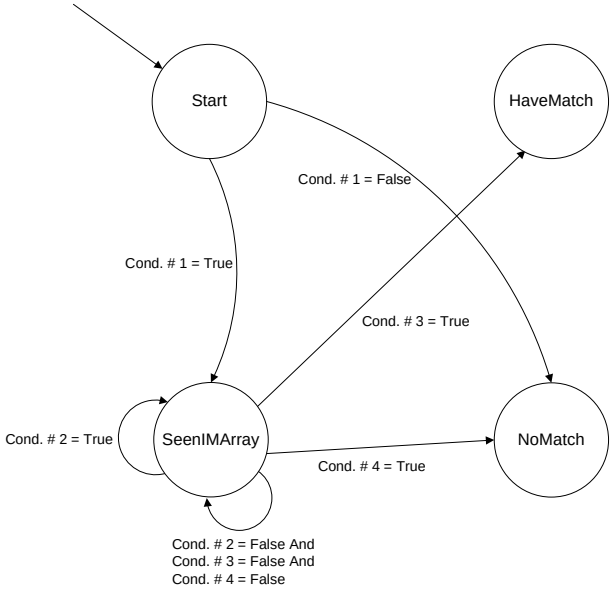

Figure 1: The FSM of the Shadow Matcher

The entry point of our pointcut matcher is the *matchesAt(…)* method, which is registered as a callback routine of the abc compiler. When the matchesAt(…) method is invoked for checking the existence of the arrayset join point shadow in the underlying method, . The first task of the matchesAt(…) method is to check that if the current assignment statement is a potential starting instruction of a join point for an arrayset pointcut, namely one with an array field of interest is referenced. If so, Cond. #1 is met and thus the FSM will set *ima* to the array variable assigned and move to the *SeenIMArray* state; otherwise the FSM simply moves to the *NoMatch* state and terminate the matching process. For example, the instruction #1 above indeed starts the

join point shadow. Thus the FSM will set *ima* to `r2` and move to *SeenIMArray* state.

Once the FSM has entered the *SeenIMArray* state, the *matchesAt*(…) method will iterate through the rest of statements in the method and examine every statement encountered to take proper state transitions according to condition #2, #3 or #4. During the iteration, there are two kinds of situations which will make the FSM stay in the *SeenIMArray* state. The first kind is specified by Cond. #2, which will be true when an element of the current intermediate array (*ima*) is assigned to a local variable, such as instruction #2 and #3 above. This amounts to checking if the current assignment statement is an intermediate step of the assignment to an element of a multi-dimensional array. When such cases happen, we shall update the intermediate array (*ima*) to the local variable on the left-hand side of the current assignment. The other kind corresponds to the scenarios where some irrelevant statements are encountered and none of the Cond. #2, #3 and #4 is true, such as instruction #4 and #5 above.

Besides, there are two specific situations which will move the FSM out of the *SeenIMArray* state. The first one happens when we find the matching array element assignment that marks the end of an *arrayset* join point. Cond. #3 specifies how we know the matching assignment is encountered. Essentially, if the intermediate array (*ima*) appears as an element reference on the left-hand side of the current assignment, we know that the end of the *arrayset* join point shadow is found and we can move to the *HaveMatch* state, such as the instruction #6 above. Another way to leave the *SeenIMArray* state is when the end of the method body is reached, which makes Cond. #4 true and moves the FSM to the *NoMatch* state.

After successfully identifying the join point shadow for an *arrayset* pointcut, the rest of work is pretty standard. We shall prepare the context information and set up the *thisJoinPoint* object accordingly.

5: Conclusions

In this paper, we have stated our motivation to extend the field pointcuts of AspectJ to array objects. Our extension is designed to be a conservative one that follows the syntax and semantics of the standard field pointcuts as much as possible. We have also described our implementation of the array field reference pointcuts based on the Aspect Bench Compiler for AspectJ. Due to the nature of Java arrays, the identification of join points for accessing array elements is not directly supported by the pointcut matching mechanism of *abc*. Therefore, we have developed a finite-state machine based pointcut matcher to realize our requirements. In the future, we shall apply the new pointcuts to our project of Java object memory consumption tracking for further evaluating them.

6: References

[1] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., and Irwin, J., "Aspect-Oriented Programming", in *Proceedings of the 11th European Conference on OOP*, LNCS 1241, pp. 220-242, 1997.

[2] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W.G., Getting Started with AspectJ, *Communications of ACM 44*, No. 10, pp. 59-65. AspectJ website: http://www.eclipse.org/aspectj/

[3] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhot'ak, O. Lhot'ak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. "abc: An extensible AspectJ compiler". *Proceedings of the 4th International conference on Aspect-Oriented Software Development (AOSD 2005)*, Chicago, USA, pp. 87-98, 2005.

[4] Chen, K. and Chen, J.B., Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs, Third Asian Workshop on Aspect-Oriented Software Development (AOAsia 3), in Proc. IEEE COMPSAC 2007, Vol. 2, pp. 23-28.

[5] Morgan Deters, Nicholas A. Leidenfrost, Matthew P. Hampton, James C. Brodman, Ron K. Cytron, "Automated Reference-Counted Object Recycling for Real-Time Java", *Proceedings 10th IEEE Real-Time and Embedded Technology and Applications Symposium, May 2004.*

[6] Programming Tools Group, University of Oxford, Sable Research Group, McGill University and BRICS, Aarhus Universitet, "abc: An extensible AspectJ compiler", Slides from: http://abc.comlab.ox.ac.uk/documents/aosdnew.pdf

[7] Hilsdale, E. and Hugunin, J., "Advice Weaving in AspectJ", *Proceedings of the 3rd Inte'l Conference on Aspect-Oriented Software Development*, Lancaster, UK, pp. 26-35, 2004.

[8] Raja Vallee-Rai and Laurie J. Hendren, "*Jimple: Simplifying Java Bytecode for Analyses and Transformations*", Technical Report, Sable Group, McGill University, July 1998.