Static Analysis of Aspects

Damien Sereni and Oege de Moor
Programming Tools Research Group, Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK

Abstract

Aspects are a novel programming language feature, to express concerns in program design that crosscut traditional abstraction boundaries. Aspects are specified as *pointcut designators* (patterns in the call stack), coupled with *advice* (code whose execution is triggered by the given pattern). We propose a more primitive syntax for pointcut designators, based on regular expressions. This primitive syntax facilitates a new static analysis that in turn enables a more efficient implementation of aspects.

1 Introduction

Some aspects of program design, such as tracing and logging, crosscut the traditional abstraction boundaries of procedures and modules. When such crosscutting occurs, it is desirable that the aspect can be added later, as a separate program unit, that is then woven into the original base program. This is the motivation for the paradigm of *aspect-oriented programming* [19]. The most popular implementation of this paradigm is an extension of Java, called AspectJ [18].

One way to describe the weaving process is akin to the observer pattern [13]: the aspect code monitors the execution of the base program, and when certain sequences of events occur, additional code that belongs to the aspect is run. An aspect can be applied recursively, monitoring its own execution, and splicing in code at appropriate times. In AspectJ, the sequences that are observed are an abstraction of the call stack, and the patterns in the aspect that trigger the execution of new code refer to patterns in the call stack. A definitional interpreter (in the style of [11]) for this dynamic view of aspect weaving was described in a pioneering paper by Wand *et al.* [26]. An alternative formal model can be found in [9].

Naturally dynamic weaving is costly, and it is important to find more efficient compilation strategies, and indeed AspectJ does implement one such strategy. The salient features of the AspectJ optimizer are explained by Masuhara *et al.*, through partial evaluation of Wand's interpreter [21]. In brief, the patterns are compiled to matching automata, and the compiled program maintains a stack of states for each such automaton. By inspecting the top of this stack, one can tell in constant time whether new aspect code needs to be executed. The runtime overhead of maintaining these stacks is however significant.

This paper makes two contributions to the compilation of aspects:

- We propose a more primitive language for describing patterns in the call stack, based on regular expressions. The AspectJ pattern combinators can be expressed in our language. The more primitive nature of our proposal makes static analysis easier, and it may be a little more familiar to readers outside the AspectJ community.
- We present a new meet-over-all-paths analysis, which enables a compiler to determine statically whether the any call stack at a given point in the program could match one of the patterns in the aspect. This eliminates most of the runtime overheads in the strategy of Masuhara *et al.*

We illustrate these ideas in the context of an experimental implementation of aspects as an extension to a Pascal-like language, which was carried out by the first author.

The structure of the paper is as follows. Section 2 reviews the terminology of aspect-oriented programming and it introduces our new language of patterns. We then illustrate these with an example, namely counting swap operations in an implementation of quicksort. With reference to that example, we briefly review previous work on interpreting and compiling aspects. Section 3 proposes our new analysis, and the way it is implemented. Section 4 discusses the results of this analysis on some sample programs. Next, in Section 5, we outline the optimisations that the new analysis enables. The paper concludes with a discussion of related work, and possible directions for future investigation. In particular, we indicate how the results of this paper could be used in a refactoring tool, which gives automatic assistance in factoring out aspects in legacy code.

2 Aspect-oriented Programming

Aspect-oriented programming builds upon a long tradition of meta-programming systems (especially [17]) and there exist several variants of the same ideas [1, 15, 20]. The most widely adopted, however, is that in AspectJ [18], and therefore we follow the terminology of that exposition. Space does not allow an in-depth review of the applications of aspects, and the reader is referred to [5] for a comprehensive overview. AJD is a much smaller aspect-oriented language than AspectJ, designed as part of the Aspect SandBox project, which aims to build a set of tools to experiment with aspects [21, 26].

To illustrate the ideas of this paper, we use a small procedural language, which is a variant of Pascal. It could be regarded as a subset of AJD that lacks features for object orientation. The restriction is not fundamental, and the techniques of this paper could be easily extended to richer languages. We shall return to this point in Section 6.

2.1 Preliminaries

Base program, aspects and advice. An aspect-oriented program consists of a *base program* and a number of *aspects*. The aspects can be viewed as observers of the base program, taking action when certain events occur in its execution. A piece of code that describes an intervening action is called *advice*. Aspects are thus understood relative to an operational semantics of the base language.

Join points. Machine configurations where advice might intervene are called *join points*. Variations of aspect-oriented programming can be explained as variations in the level of abstraction of these join points. We follow AJD by defining join points as an abstraction of the control stack, ignoring all other notions of machine configuration. To wit, a join point is a sequence of procedure calls, procedure executions, and advice executions. A procedure call corresponds to the invocation of a procedure at the call site, whereas an execution refers to entry into the procedure's body. In the syntax of the ML programming language [22], we might declare the type of join points thus:

$$\begin{array}{rcll} \mathit{jp} & = & \mathit{jp\_element\ list} & \\ \mathit{jp\_element} & = & & \mathit{call}\ \mathbf{of}\ (\ \mathit{name} * \mathit{name} * \\ & & & \qquad \mathit{actual\_param\ list}) \\ & & | & \mathit{exec}\ \mathbf{of}\ \mathit{name} \\ & & | & \mathit{aexec} \end{array}$$

Note that the join point for a procedure call includes not only the name of the called procedure, but also that of the calling context, as well as the values of actual parameters. Strictly speaking, the calling context is redundant, but it simplifies the description of useful sets of join points. In AspectJ, a much richer join point model is assumed, in particular reflecting the inheritance hierarchy. All these enrichments are however of a static nature, and the focus of this paper are dynamic join points. A semantics of such static aspects is sketched in [2].

Pointcut designators. To specify where advice should intervene, each piece of advice is coupled with a predicate that singles out the set of join points where that advice should be executed. Such a set of join points is called a *pointcut*, and the predicate that describes it is called a *pointcut designator*, or PCD for short. A PCD may include variables, to match against the actual parameters of a procedure call.

We deviate from AJD and AspectJ in the syntax for expressing pointcut designators. A pointcut designator is a regular expression whose alphabet consists of *element designators* (EDs). An element designator is a predicate over join points elements. The abstract syntax of element designators is given by:

$$\begin{array}{rcll} \mathit{ed} & = & & \mathit{pcall}\ \mathbf{of}\ \mathit{name} \\ & & | & \mathit{pwithin}\ \mathbf{of}\ \mathit{name} \\ & & | & \mathit{args}\ \mathbf{of}\ \mathit{var\ list} \\ & & | & \mathit{and}\ \mathbf{of}\ (\mathit{ed} * \mathit{ed}) \\ & & | & \mathit{or}\ \mathbf{of}\ (\mathit{ed} * \mathit{ed}) \\ & & | & \mathit{not}\ \mathbf{of}\ \mathit{ed} \\ & & | & \mathit{true} \end{array}$$

The *pcall* n designator matches join point elements of the form *call* $(n, \_, \_)$. The next designator is *pwithin* n; it matches calls that are made from the context n, that is *call* $(\_, n, \_)$. Finally, one can match for actual parameters using *args* $(x_1, x_2, \ldots, x_n)$. This element designator matches *call* $(\_, \_, [a_1, a_2, \ldots, a_n])$, binding each variable x_i to a_i. The logical operators are interpreted as expected; in particular, negation can only be applied to element designators that do not contain free variables.

Within a PCD, a variable may only be repeated in the branches of a logical disjunction. Furthermore, variables cannot be used under the Kleene star. These restrictions simplify the matching of PCDs against join points at runtime. Indeed, PCDs cannot make tests that depend on dynamic values bound using the *args* primitive. We shall return to this point later.

Before, after, and around. As explained above, a pointcut designator describes where advice applies — but there is still the choice of executing the advice code before or after the selected join points. Each piece of advice is therefore also coupled with an indication of whether it happens before or after a join point. In AJD, there is furthermore the option of execution advice *around* a join point. In the corresponding advice execution, the statement *proceed* describes the original join point. This is a powerful feature, allowing the programmer to substitute completely different code for a procedure call. Our present implementation does not support *around* advice, and its possible inclusion will be further discussed in Section 6.

2.2 Example: Profiling quicksort

To illustrate the above definitions, consider a program that calls Hoare's quicksort routine [16], to sort the lines in an array of strings. The program has a global variable called *linesbuf* to hold the array of strings, and Figure 1 shows the procedures that are relevant to the discussion below. Each entry in the figure briefly describes what the procedure does — the open interval notation $x[a..b)$ denotes a consecutive segment of array x inclusive of $x[a]$, up to but not including $x[b]$.

$\mathit{compare}(i, j)$	:	returns true if $\mathit{linesbuf}[i] < \mathit{linesbuf}[j]$
$\mathit{swap}(i, j)$	:	swaps lines i and j
$\mathit{readln}(i)$	:	read a line into $\mathit{linesbuf}[i]$
$\mathit{writeln}(i)$	:	print $\mathit{linesbuf}[i]$
$\mathit{partition}(a, b)$	:	partitions $\mathit{linesbuf}[a..b)$, with pivot $\mathit{linesbuf}[a]$
$\mathit{quicksort}(a, b)$	:	sorts $\mathit{linesbuf}[a..b)$

Figure 1: Quicksort procedures.

```
aspect Counts
  var swaps, partitions : int;

  advice PCount
    before: $\{pcall(partition) \wedge pwithin(quicksort)\}; \{true\}^*$
    begin
      partitions := partitions + 1
    end

  advice SCount
    before: $\{pcall(swap)\}; \{true\}^*; \{pcall(quicksort)\}; \{true\}^*$
    begin
      swaps := swaps + 1
    end

  advice Init
    before: $\{pcall(quicksort)\}; \{\neg(pcall(quicksort))\}^*$
    begin
      partitions := 0;
      swaps := 0
    end

  advice Print
    after: $\{pcall(quicksort)\}; \{\neg(pcall(quicksort))\}^*$
    begin
      println "Partitions: " ++ partitions;
      println "Swaps: " ++ swaps
    end
end Counts
```

Figure 2: Profiling aspect for quicksort.

Now suppose that we wish to augment this program, to gather some statistics about the performance of quicksort, printing out the statistics after each run of the algorithm. We intend to count the number of calls to *partition*, as well as the number of *swap* operations. Note that there may well be other uses of *swap* apart from the obvious ones within the *partition* routine.

To achieve the desired effect, we use the aspect shown in Figure 2. The first piece of advice says that before each call to *partition* from the body of *quicksort*, the *partitions* counter should be increased. The next piece of advice says that before any call from *swap* within the context of a call to *quicksort*, the *swaps* counter should be increased. To achieve initialisation, we specify a pointcut that contains only non-recursive calls to *quicksort*, and of course the initialisation should be carried out before encountering any join point in that cut. By contrast, the results should be reported after each non-recursive call to *quicksort*.

2.3 Interpreting Aspects

Most programmers will agree that it is nice to localise the code that gathers the statistics in the *quicksort* example — provided the runtime cost is negligible. To assess those runtime overheads, let us first consider a straightforward interpreter for aspects.

Such an interpreter is described in detail by [26]. Sereni has ported those ideas to a variant of Pascal, implemented in OCAML [4], and the code can be downloaded from [23]. The basic idea is simple: an interpreter for the base language is augmented to keep track of the current join point. Whenever a new join point is created (at a procedure call, entry to a procedure body or advice execution), that new join point is matched against all the PCDs in the aspects that were applied to the program. If a match is found, the corresponding advice is executed, with an environment that contains values for the PCD variables that were bound in matching.

The maintenance and matching of join points is potentially costly. Consider, for example, the PCD we used to trigger initialisation in the *quicksort* example. To check that a join point satisfies this PCD, one must traverse the whole join point. It follows that upon each new join point creation, the interpreter may have to traverse the whole join point for each PCD that is in scope.

2.4 Compiling Aspects

Clearly such interpretative weaving of base program and aspects is too costly to be practical; some form of compilation must be used to transfer work from the dynamic matching to compile time. This is a clear case for partial evaluation [12]. Indeed, in [21], it is shown how the first Futamura projection yields a compiler from the interpreter of [26]. Unfortunately, however, without further improvements, that compiler still generates code that potentially traverses the whole join point for each PCD.

The same paper [21] explains how the AspectJ compiler solves this problem. A PCD *pcd* corresponds to a deterministic finite automaton, say M. This automaton is constructed as follows. First, we collect all potential join point elements from the program text, in a set called $\mathcal{A}$. Note that this is possible, because all relevant values are available at compile time, except for the argument values in a *call*: we just represent these by dummy place holders. It is likely that we overestimate the set of join point elements (because some of them do not occur in actual program runs), but that does not matter. Next, for the relevant *pcd*, we determine the set X of element designators that occur within *pcd*. Now, for each element designator, we can compute the set of join point elements that are true of it, giving a map

$$validates \quad : \quad X \rightarrow \mathcal{A}\, set$$

That is, *validates* takes an element designator e, and returns the set of join point elements in $\mathcal{A}$ that validate e. Take the finite automaton that corresponds to the regular expression that makes up *pcd*: this is an automaton where the transitions are labelled with element designators. Replace each transition labelled with e by a set of transitions, taking the labels from $validates(e)$. Finally determinise the result: thus we obtain the finite automaton M that corresponds to *pcd*.

We now describe how M is used for efficient implementation of PCD matching. Write $M(x)$ for the state that is reached via a join point x. Instead of repeatedly running M on each new join point $x = [x_1, x_2, \ldots, x_n]$ from scratch, we could keep a stack $[M(x), \ldots, M([x_{n-1}, x_n]), M([x_n])]$. When a new PCD is constructed, it is either by pushing new elements on to the front of x, or by popping some, and thus the transitions can be computed in constant time. It follows that the overheads of matching are reduced to a constant, for each individual PCD. Similary we need to keep a stack of variable bindings for each PCD that contains $args(x)$ designators.

This is certainly an improvement, but keeping these stacks of states is still a significant overhead, roughly proportional to the number of PCDs. The paper [21] describes further methods of reducing the constant factor (one of which we shall elaborate in Section 5), but the overhead remains proportional to the number of PCDs.

Intuitively, in examples such as that shown above, it should be possible to completely eliminate the matching process, and determine for each point in the program exactly what PCDs will apply at runtime, and which cannot apply. It would then be possible for a compiler to generate the tangled program that one might have written prior to the invention of aspects. This is what we set out to do in the remainder of this paper.

It should be mentioned that the above compilation scheme, as we have described it, relies on the static nature of element designators: it can be determined at compile time whether a join point element validates an element designator or not. The element designators found in AspectJ do not have this static property, as they can include tests of runtime values, so in reality the AspectJ compiler is somewhat more complex.

3 Analysing Aspects

Our objective is to determine for each procedure call in the abstract syntax tree of the program the set of all possible call stacks (or equivalently, join points) at that call. Given this information, it would be possible to obtain a source-to-source program transformer that takes an aspect-oriented program, and returns the base program with some additional code inserted (corresponding to applicable aspects), but without any code for matching pointcut designators.

It is important to note, however, that we cannot hope to achieve this for all possible advice. Both with the regular expressions we use for denoting PCDs and with the language used in the Aspect SandBox project, it is possible to write a pattern which makes compile-time determination impossible. We will come back to this point in Section 5. In the meantime, we shall describe our approach to computing the relevant analysis information.

3.1 The Analysis

With each piece of advice is associated a pointcut designator *pcd*, which denotes a set of join points. As discussed in Section 2, it is possible in our language to compute (an over-approximation of) the set of join point elements in a program at compile time, which we denote by $\mathcal{A}$. We may hence define a function:

$$\textit{join\_points} : \textit{pcd} \rightarrow \mathcal{A}^* \textit{ set}$$

which associates to each pointcut designator the (usually infinite) set of join points that it represents. This simplifies our discussion of the analysis and is part of the reason for the restrictions we have placed on pointcut designators. In the remainder of this paper, for notational convenience, we identify a pointcut designator *pcd* with the set *join_points*(*pcd*).

Now during the execution of the base program, a piece of advice with pointcut designator *pcd* is executed if the current join point lies in *pcd*. It therefore suffices to compute, for each procedure call p in the program text, a set $L(p)$ containing all possible join points at evaluation of the call p. We shall define $L(p)$ to be a regular language, and in general it is an over-approximation; however it reduces the problem of determining advice applicability to tests on regular languages. Indeed,

- If $L(p) \subseteq \textit{pcd}$, then the advice always applies at p, and
- if $\textit{pcd} \cap L(p) = \emptyset$, then the advice can never apply at p.

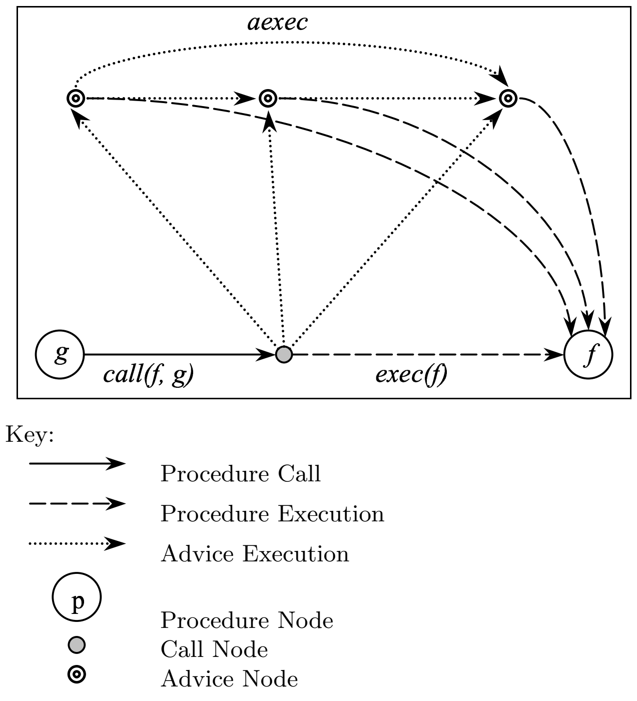

Figure 3: An advised procedure call

Observe that both tests can be reduced to language containment, as

$$\textit{pcd} \cap L(p) = \emptyset \Leftrightarrow L(p) \subseteq \neg\textit{pcd}$$

where $\neg L(p) = \mathcal{A}^* \setminus L(p)$.

3.2 The Call Graph

Aims We now proceed to describe how $L(p)$ may be computed. Our approach will be to construct a *call graph* for the program, in which each procedure call appears as a node (these are not the only nodes in the graph, however). Edges in this graph correspond to elementary operations affecting the control stack (such as procedure calls) and hence are labelled with join point elements — recall that a join point is a string of join point elements, and thus a path in the graph corresponds to a join point. The set $L(p)$ will in fact be the set of paths from the source vertex (the procedure *main*) to the vertex corresponding to p. We denote the source vertex by v.

The graph must thus be built such that every path from v to p is a valid join point at p (although it need not occur in actual program runs), and that every valid join point is represented by a path. There are therefore three kinds of edges, corresponding to the three *join point kinds* — procedure calls, procedure executions and advice executions.

The nodes in the graph correspondingly, represent procedure bodies (the *procedure node*), procedure calls (*call nodes*) and advice bodies (*advice nodes*). Note however that while there is a single procedure node in the graph for each procedure in the program, advice nodes are replicated at each procedure call.

Building the Graph We can now describe the relationship between these nodes by considering a single procedure call to f from g. For simplicity, we will first consider the case where there are no procedure calls within the bodies of advice, and return to the more general case later. See also Figure 3 for an illustration of this. The graph includes nodes for procedures f and g respectively. The call is represented by a node n, and edges are added: from g to n, labelled $\mathit{call}(f, g)$; and from n to f, labelled $\mathit{exec}(f)$. This represents the direct, or unadvised path to f from g. However, any advice that applies modifies this path by adding an advice execution event between the call to f and its execution. As we assume at present no knowledge of applicability of advice, any subset of the program advice can apply, chained however in the order in which they appear in the program text. To achieve this, a new node is created for each piece of advice in the program. Each of these has an *aexec* edge leading from the call node, and an *exec* edge leading to the execution node.

Finally, the interaction of different pieces of advice at a single procedure call must be considered, as the previous construction accurately describes the execution of a single piece of advice only. In this, we follow the semantics given in [26], which differ from that of AspectJ. We therefore consider that if more than one piece of advice apply at a procedure call, the resulting *aexec* join point elements are stacked, regardless of the type of advice (in AspectJ, this is only the case for *around* advice). Thus there is an additional set of edges, describing the possibility of chaining aspects. Assuming that aspects are ordered as $a_0, \ldots, a_n$, there is an edge from a_i to a_j for all $j > i$. The type of the advice (*before* or *after*) does not affect the graph. This construction is illustrated in Figure 3 for the case of three aspects.

This construction of the program graph satisfies the condition that every possible join point at a procedure call is represented by a path from the source vertex to the vertex representing that call. Unfortunately, the size of the graph is quadratic in the number of advices in the program. Indeed, for n aspects $n(n-1)/2$ edges are necessary to capture all the possible sequences of advice executions. However, it is possible to dramatically reduce the size of the graph prior to analysis by considering the topmost element of the call stack. In practice, many pointcut designators are of the form

$$\{\mathit{pcall}(f) \wedge \ldots\};\ \ldots$$

Indeed, PCDs which are not of this form can often cause infinite loops (more details are given in [26]). Now in this case it is easy to see that this can never apply to a call to a procedure $g \neq f$, and hence it is unnecessary to include the node for this advice for such calls. Generalising this, it is usually possible to eliminate a large proportion of advice nodes and edges by checking whether the first element of the PCD can apply to the call being considered. The top of the call stack will always be a *call* event, and this corresponds to the label of the edge leading to the procedure call node.

In the previous paragraphs, we have described the construction of the call graph in a restricted case (namely, that the only procedure calls occur in the base program). This is thoroughly unrealistic, and we now must complete the description in the general case. The same construction cannot be applied for procedure calls within advice bodies, as this would be a potential cause of infinite regress. Indeed, recall that a fresh copy of each of the advice nodes is created for each procedure call in the text (ignoring the quick pruning performed at that point, as we are considering the worst case). Therefore if a piece of advice a_i contains a procedure call pc, a fresh copy of a_i is created to advise that call, which in turn creates a new instance of pc, leading to nonterminating behaviour. To remedy this problem, procedure calls from advice bodies are not treated in the same way as calls from the base program. A procedure call pc from an advice node n_i does not give rise to new copies of the advice nodes, instead it shares the set S of advice nodes to which n_i belongs. This is illustrated in Figure 4. There is a final complication to the construction, as our quick pruning means that S will not in general contain the entire set of advice in the program. If the call pc requires an advice a_j not in S, a fresh copy of a_j is created and added to S, and is chained with the other nodes in S. To make this discussion precise, we have included the algorithm used for this construction in ML-like pseudocode in Figure 5. This defines a function *add_call_advice* which is invoked for all procedure calls in the base program, and adds the advice nodes and edges for the call, as well as those for calls from advice bodies. A mention is appropriate here of the effect that this sharing of advice nodes has on the analysis. In principle, it could lead to a certain loss in sharpness, by introducing new and usually spurious paths in the call graph. In practice, this does not affect the results of the analysis; furthermore an iterative technique for recovering sharpness will be presented in the next paragraph.

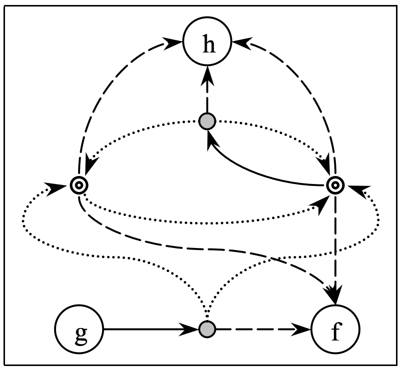

Figure 4: Procedure calls within advice bodies

Iterative Analysis We conclude our description of the call graph with a mention of the effect of procedure calls within advice bodies on the sharpness of the analysis. Not only do these complicate the construction of the graph, they also have an adverse effect on the analysis of that graph. Figure 6 depicts a simple case of this, where there are two procedures f and g and a single piece of advice. We assume that it has already been determined (by the "quick" analysis mentioned previously) that the advice can never apply at g, hence the presence of only one advice node in the graph. The body of the advice, however, contains a call to g. What should be observed here is that even though f does not call g, there is a path from f to g *via* the advice. This of course enlarges $L(g)$, the approximation to the set of possible join points at g. In particular, consider a (plausible) pointcut designator of the form:

$$\{\mathit{pcall}(g)\}\ ;\ \{\mathit{true}\}^*\ ;\ \{\mathit{pcall}(f)\}\ ;\ \{\mathit{true}\}^*$$

This describes any call to g within the dynamic scope of f. This dynamic scope, however, includes the advice shown

```
let $add\_call\_advice$ $procedure\_call$ =

let ($applicable$, $notapplicable$)
    = partition all advice in the program with the quick
    pruning method for $procedure\_call$
let $new\_advice\_nodes$
    = make a fresh node for each advice in $applicable$

let $to\_process$ = ref $new\_advice\_nodes$
and $added$       = ref $new\_advice\_nodes$

for each $anode$ in $to\_process$ do

    let $pcalls$
        = get all procedure calls from the body of the
         advice corresponding to $anode$

    for each $pcall$ in $pcalls$ do
        add the edges for $pcall$
        let ($app'$, $notapp'$)
            = quick pruning for $pcall$

        for each $anode'$ in $app'$ do
            if $name(anode')$ in $names(added)$ then
                add the advice edges for advising $pcall$ with
                the advice corresponding to $anode'$
            else
                let $newv$
                = create a fresh copy of $anode'$

                add the advice edges for $pcall$ and $newv$

                $added := \{newv\} \cup added$;
                $to\_process := \{newv\} \cup to\_process$
            end
        done
    done
done

chain the nodes in $added$ together
```

Figure 5: Constructing the Call Graph

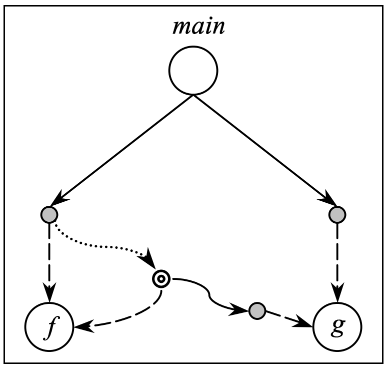

Figure 6: Effect of calls within advice bodies on sharpness

as an examination of Figure 6 will reveal. Consequently, at least one of the join points described by this pattern must lie in $L(g)$, thus we would not be able to determine that this can never apply. It may be the case, however, that the advice shown in the graph is found never to be applicable at f. In this case, deleting the node for this instance of the advice and re-processing the graph would allow a better estimate of $L(g)$ to be produced — in particular, one that is disjoint with the language described by the pattern above.

With this aim, the result of analysing the graph is not only $L(p)$ for each call p, but also the graph with all "impossible" edges removed. If a piece of advice a has been found to always apply at a call p, these edges are those that bypass the advice node corresponding to the instance of a applied to p (both advice executions and procedure executions from advice higher in the chain). Conversely, if it has been found that a can never apply at p, the impossible edges are those leading to the node corresponding to a. Care must be taken, however, as our treatment of procedure calls within advice bodies means that certain advice execution edges (chaining advice nodes together) are effectively shared between several procedure calls. We should therefore only remove an advice edge if it "impossible" for *all* procedure calls that it advises.

The analysis can be iterated until a fixed point is reached, which corresponds to the best estimate for $L(p)$ at each p. This iteration can also compensate for the minor loss of sharpness resulting from the sharing of advice nodes, as mentioned previously.

3.3 Meet-Over-all-Paths Analysis

We have constructed a graph from the source program such that the set of paths from the source vertex v to any vertex p is an superset of the set of possible call stacks at point p in the execution of the program. Using this, it is possible to obtain a regular expression $L(p)$ describing the set of join points at p, using an algorithm of Tarjan [25]. This regular expression can then be tested for inclusion with respect to each PCD in the program. This solves the problem but implies much duplicated computation. Indeed, the regular expressions $L(p_1)$ and $L(p_2)$ are closely related for vertices p_1 and p_2 that are "close" in the graph, and hence testing the same pointcut designator against both of them independently is wasteful. In the next section we will describe a method for performing these tests compositionally.

3.4 Compositional Analysis: Chips and Chops

We can accurately regard the number of advices as fixed and small in comparison to the number of procedure calls. It is feasible, therefore, to do some precomputation on the PCDs prior to the analysis proper.

We define two predicates on regular expressions r:

$$\begin{aligned} \mathit{Subset}\ \mathit{pcd}\ r &\equiv r \subseteq \mathit{pcd} \\ \mathit{Disjoint}\ \mathit{pcd}\ r &\equiv r \subseteq \neg \mathit{pcd} \\ &\equiv \mathit{Subset}\ (\neg \mathit{pcd})\ r \end{aligned}$$

The values that we are interested in are *Subset pcd* $L(p)$ and *Disjoint pcd* $L(p)$. By our previous observation, it is desirable to compute those compositionally in terms of p, that is by induction on the structure of $L(p)$.

In a companion paper [7] we have described a compositional algorithm for achieving just that (presented in the context of program analysis with side conditions specified as regular expressions). In this algorithm, a matrix $C(pcd)$ of regular expressions (the “chip-chop matrix” for *pcd*) is associated with each pointcut designator. We can think of $C(pcd)$ as a systematic arrangement of the parts of *pcd*. By generalising *Subset* and *Disjoint* to these matrices, so that:

$$\begin{aligned}(S\ pcd\ r)_{x,y} &\equiv r \subseteq C(pcd)_{x,y}\\(D\ pcd\ r)_{x,y} &\equiv r \subseteq C(\neg pcd)_{x,y}\end{aligned}$$

S and D may be computed compositionally on the structure of $L(p)$.

More specifically, we have:

$$\begin{aligned}S\ pcd\ (r_1\ ;\ r_2) &= (S\ pcd\ r_1) \cdot (S\ pcd\ r_2)\\S\ pcd\ (r_1 + r_2) &= (S\ pcd\ r_1) \wedge (S\ pcd\ r_2)\end{aligned}$$

where we define multiplication of Boolean matrices taking $\vee$ for addition and $\wedge$ for multiplication. These rules force $S\ pcd\ (r^*)$ to be the greatest fixed point of the equation:

$$X = (S\ pcd\ \varepsilon) \wedge (S\ pcd\ r) \cdot X$$

where ε denotes the empty string.

This defines a regular algebra which may be used directly in Tarjan’s algorithm. As *pcd* itself occurs as an entry of $C(pcd)$, the original problem is an instance of the generalisation. Note also that the performance advantage of this algorithm is increased if the analysis is iterated (to obtain better estimates of $L(p)$ in some cases, as previously mentioned), as the chip-chop matrices do not need to be recomputed.

4 Results

In the previous section we have described our method for analysing aspect-oriented programs with the aim of statically determining the points of application of advices, thus reducing the runtime overhead associated with aspect-oriented programming (we will give more details as to exactly how this might be reduced in Section 5). In this section, we will explore the effectiveness of this analysis on two small examples — as space does not allow us to explore the results of the analysis on larger programs, we use these to exemplify the main ideas. We will first come back to the example application of aspects given in Section 2, namely tracing the quicksort routine. We will then present an example for which it is actually necessary to iterate the analysis.

4.1 Tracing quicksort

Recall that we had augmented the behaviour of quicksort with an aspect *Count* made up of four advices: *SCount* and *PCount* for counting calls to *swap* and calls to *partition* respectively, *Init* for initialising counters, and *Print* for printing the tracing information (note that for clarity we use identifiers beginning with a capital letter for aspects and advices only). There were in total ten procedure calls in our program, which we have detailed in Figure 7, along with the results of the analysis. Our results are presented as follows (for a procedure call *pc* and an advice *a*):

- A ✓ means that *a* always applies at *pc*.
- An $\times$ means that *a* can never apply at *pc*.
- A blank entry means that the analysis is inconclusive.

Procedure calls:

1:	*main*	$\longrightarrow$	$readln(N)$
2:	*main*	$\longrightarrow$	$quicksort(0, N)$
3:	*main*	$\longrightarrow$	$writeln(i)$
4:	*readln*	$\longrightarrow$	$swap(0, i)$
5:	*partition*	$\longrightarrow$	$compare(i, a)$
6:	*partition*	$\longrightarrow$	$swap(i, j)$
7:	*partition*	$\longrightarrow$	$swap(a, i-1)$
8:	*quicksort*	$\longrightarrow$	$partition(a, b)$
9:	*quicksort*	$\longrightarrow$	$quicksort(a, i)$
10:	*quicksort*	$\longrightarrow$	$quicksort(i+1, b)$

Results:

Call	*SCount*	*PCount*	*Init*	*Print*
1	$\times$	$\times$	$\times$	$\times$
2	$\times$	$\times$	✓	✓
3	$\times$	$\times$	$\times$	$\times$
4	$\times$	$\times$	$\times$	$\times$
5	$\times$	$\times$	$\times$	$\times$
6	$\times$	✓	$\times$	$\times$
7	$\times$	✓	$\times$	$\times$
8	✓	$\times$	$\times$	$\times$
9	$\times$	$\times$	$\times$	$\times$
10	$\times$	$\times$	$\times$	$\times$

Figure 7: Results of analysing the quicksort example

Calls 5 to 10 are the heart of the actual quicksort routine, while 1 to 4 make up the interface with the user. This program demonstrates the use of three different kinds of pointcut designators (recall that advices *Init* and *Print* have the same pointcut designator). The advice for *PCount* only matches on the topmost item of the stack and in this sense is static — it only depends on the textual location of the call. In contrast, the other two kinds of advice are dynamic and depend on the call stack. This is used to express the two properties that a call is within dynamic scope of quicksort and that a call to quicksort is not recursive, respectively.

In this case, the analysis has been successful in determining applicability for each advice at each procedure call. Thus the pointcut designators which we just described as dynamic are in fact static in the context of this base program. Given the table in Figure 7, it would be possible to transform the aspect-oriented program into its tangled, or *statically woven* counterpart by just inserting the body of the relevant advice at each point marked with a ✓. The programmer can thus write the clear, neatly separated version of the program without loss of efficiency.

4.2 Iterating the Analysis

The quicksort example can be resolved in a single pass of the algorithm, as there are no procedure calls within the bodies of the advices. In fact, even if such calls are present, iteration may not be necessary. However, we present a small example illustrating how iteration can actually add information.

Non-trivial examples where iteration is required would be difficult to present concisely, we therefore only give an artificial program fragment that has this property (in Figure 8). This consists of just two procedures f and g advised by two pieces of advice A and B. The complexity arises from the fact that A calls f, and that the pointcut designator

```
procedure f(x : int) : int;
begin
  if x = 0 then return 1
  else return x * f(x − 1)
  end
end;

procedure g(x : int) : int;
begin
  return x
end;

begin
  (* ... *)
  g(y);
  f(x)
  (* ... *)
end.

aspect X

  advice A
    before: {pcall(g)}; {true}*; {pcall(g)}; {true}*
    begin
      (* ... *)
      f(x)
      (* ... *)
    end

 advice B
    before: {pcall(f)}; {true}*; {pcall(g)}; {true}*
    begin
      (* ... *)
    end

end X
```

Figure 8: A program fragment with calls within advice

Procedure Calls:

1:	*main*	$\longrightarrow$	$g(x)$
2:	*main*	$\longrightarrow$	$f(y)$
3:	f	$\longrightarrow$	$f(x-1)$

Results:

Quick Pruning

Call	A	B
1		$\times$
2	$\times$	
3	$\times$	

First Pass

Call	A	B
1	$\times$	$\times$
2	$\times$	$\times$
3	$\times$	

Second Pass

Call	A	B
1	$\times$	$\times$
2	$\times$	$\times$
3	$\times$	$\times$

Figure 9: Results of iterating the analysis

for B matches exactly those calls to f within the dynamic scope of g. As A is invoked on a call to g, the call to f from the body of A occurs within the dynamic scope of g, thus extending the possible join points on execution of f. This means that B can actually apply in some cases (even though g is trivial and thus does not call f, directly or indirectly).

However, it is easy to see that in this case the possibility is purely fictitious: in fact g is not a recursive procedure, so that A can never apply! This is admittedly unrealistic, since a programmer is unlikely to write a piece of advice that never applies. We believe, however, than it can serve as an abstraction of what could occur in practice. It is desirable, therefore, that the analysis can deduce that since A can never apply, B can never apply either. This cannot be determined in its first pass, but is in fact detected on the second pass. We present the results in Figure 9.

Let us briefly comment on the efficiency of iterating the analysis. As mentioned before, the compositional algorithm used means that much information can be kept and does not need to be recomputed, and so naturally we expect any supplementary passes to be faster than the first pass. In practice, the gain in efficiency goes further than that — the time taken by subsequent passes is negligible compared to the time taken for the first pass. This is caused by the pruning of the graph after the first set of analysis results is obtained — essentially, in that process almost all infeasible edges disappear, leaving only a handful of troublesome cases that still need to be resolved.

5 Optimisations

The analysis that we have described in Section 3 determines, for each procedure call in the program, the "status" of each advice at that call (based on the advice's pointcut designator). This status is one of three possibilities: the advice can never apply at that call, *or* the advice always applies at this call, *or* applicability of the advice cannot be statically determined (that is, the analysis has been unsuccessful for this particular advice and call).

In the case that the analysis has been successful at a call p for a piece of advice with pointcut designator *pcd*, the advice body may be either discarded or inserted directly at the appropriate point in the code. The analysis thus eliminates the need for run-time matching of pointcut designators, which is certainly desirable and a significant performance saving if matching is implemented in the straightforward way. However, when the matching algorithm from the AspectJ compiler (as presented in [21]) is used, the savings will be less important — more time is spent keeping the current state of the automaton M_{pcd} for each pointcut designator *pcd*.

It is however easy to see how this may be eliminated: suppose that the analysis has been successful for a given pointcut designator *pcd* at *every* procedure call in the program. In this case, the current state of the corresponding automaton M_{pcd} will never be used, as *pcd* will never need to be matched during the execution of the program. It therefore becomes unnecessary to keep and update the state of M_{pcd}, and the functionality of the advice is woven with no run-time overhead.

Thus far, we have made the restriction that no free variables appear in *pcd*. We shall now extend the previous description to include the possibility of free variables in *pcd*. Recall that in our language, free variables are introduced by the `args` construct and bind to the values of parameters to procedures. Also, due to the restrictions that we have placed on the use of variables in our pointcut designators, they act as place holders only and have no influence on matching (and hence static analysis). It follows that whenever we can determine that a pointcut designator *cannot* apply at a program point p, we can eliminate the relevant code, just as in the absence of variables.

However, it is not possible to eliminate the matching code in the presence of variables, since we still must maintain a stack of variable bindings. Of course there are a number of common special cases where the stack is unnecessary, for instance when the variables are bound only in the leftmost element designator.

Unfortunately, it is in general impossible to statically determine applicability for arbitrary pointcut designators. As an example, consider the case of a simple procedure f with a single recursive call to f from within its body. Now the pointcut designator:

$$\begin{array}{l} \{pcall(f)\};\ \{true\}^*; \\ \{pcall(f)\};\ \{true\}^*; \\ \{pcall(f)\};\ \{true\}^* \end{array}$$

matches exactly those calls to f that occur within the scope of two previous calls to f (that is, the depth of recursion is at least three). Certainly this may apply to the recursive call to f from within its body, but this is not always the case — in the first, non-recursive execution of f it will not. Therefore no static analysis can in general solve the problem for this particular situation. It is worth noting that this phenomenon is not introduced by our new language for pointcut designators, as the previous expression has an equivalent in the AJD syntax [26], namely

$$\begin{array}{lll} & pcall(f) & \\ \wedge & cflowbelow(& pcall(f) \\ & & \wedge \quad cflowbelow(pcall(f))) \end{array}$$

(We discuss the primitives in this expression, and its relation to our notation, further in Section 6.) Because of this restriction on the power of static analysis of aspects, it would be desirable to reduce run-time overhead when we have determined some, but not all, of the information about applicability of a piece of advice. We will examine two possible directions for achieving this.

5.1 Optimising Pointcut Automata

The motivation for this optimisation lies in the observation that not all join point elements actually update the state of

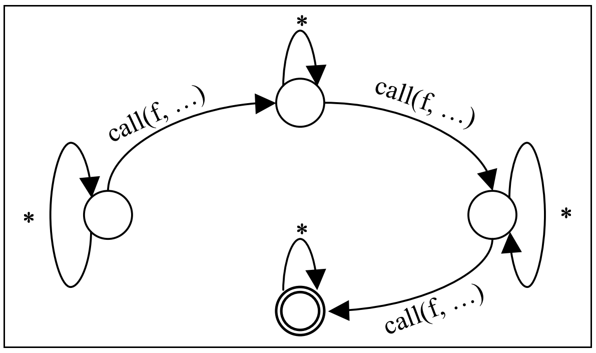

Figure 10: Deterministic automaton for a PCD

the automaton associated with a pointcut designator. As an example, consider the deterministic automaton M for the pointcut given above (Figure 5.1). We have used the wildcard "*" as a label for an edge leading from a node n to denote the set of join point elements not labelling any other edges leaving n. In this case, it is clear that we can know the state $M(x)$ for any join point x given the *restriction* x_Σ of x to the set Σ of join point elements corresponding to a call to f — intuitively it suffices to know the stack of calls to f. More formally, we can define:

$$\langle x_1 x_2 \ldots x_n \rangle_\Sigma = \langle x_i \mid x_i \in \Sigma \rangle$$

where we use the angle-bracket notation to denote sequences. For an arbitrary pointcut designator, it suffices to determine the set Σ of join point elements corresponding to useful edges – that is, edges whose start and end states are distinct. It is then clearly the case that $M(x) = M\,(x_\Sigma)$ for any join point x. Bookkeeping code need thus only be inserted exactly at those points (always procedure calls in our model) for which the join point element lies in Σ. This optimisation does not depend on the analysis described above, and it was already described in [21]. The AspectJ compiler implements this optimisation.

5.2 Isolating Aspect-Free Components of the Call Graph

Another optimisation is based on the observation that often the static undecidability of aspects will be limited to a few procedure calls in the program, and hence there will be large portions of the call graph in which this may be computed statically. We say that such portions of the call graph are "aspect-free", in reference to the fact that they can be compiled without the code responsible for matching pointcut designators (we are not saying that no advice can apply in those components of the graph, however). For concision, we say a procedure call node is "good" if all advice can be statically determined for that node. Given a node v, we also define *reachables*(v) to be the set of nodes reachable from v in the graph. The crucial point is that due to the stack nature of join points, the events occurring in *reachables*(v) do not influence the join point at any predecessor of v (as the graph may not be acyclic, we only say that a node is a predecessor of v if it is not also a successor of v). Therefore if all nodes in *reachables*(v) are good, we may safely eliminate all bookkeeping code (*i.e.* code that updates the current state of automata for pointcut designators) from all calls within $reachables(v)$, as this will not affect the states of the automata at other points in the program.

If this property is true of the source node (that corresponding to the body of the procedure `main`), then this just boils down to eliminating all PCD matching code from the program. However, it is much more general than that, as there may be several aspect-free components in the graph even if it is not globally aspect-free.

6 Related Work

As we indicated earlier, the research reported here is closely related to and much inspired by the Aspect SandBox project at UBC, and in particular its implementations of the experimental language AJD. In this section, we discuss some of the differences.

6.1 Regular expressions *vs. cflow*

The main difference is our language of pointcut designators, which is that of regular expressions over element designators. By contrast, in AJD the abstract syntax of pointcut designators is given by

$$\begin{array}{lcl} ajd\_pcd & = & top\ \mathbf{of}\ ed \\ & | & cflow\ \mathbf{of}\ ajd\_pcd \\ & | & and\ \mathbf{of}\ (ajd\_pcd * ajd\_pcd) \end{array}$$

as well as some further logical combinators that are not relevant to the present discussion. The first form matches a join point whose head matches the given element designator. The form $cflow\ p$ is matched against a join point x as follows. First, p is matched against x. If that succeeds, so does the matching of $cflow\ p$. Otherwise, the process is repeated with the tail of x, until a match is found, or no more elements remain. A PCD of the form $and(p, q)$ matches x if both p and q match x.

From the above description, it is easy to deduce a translation from the PCDs in AJD to our notation:

$$trans \ : \ ajd\_pcd \rightarrow pcd$$

To wit, we define (using abstract syntax on the left, and for brevity, concrete syntax on the right):

$$\begin{array}{rcl} trans\,(top\ e) & = & \{e\}\,;\{true\}^* \\ trans\,(cflow\ p) & = & \{true\}^*\,;trans\ p \\ trans\,(and\ p\ q) & = & trans\ p\ \cap\ trans\ q \end{array}$$

This translation is faithful in the absence of variables, in the sense that matching an ajd_pcd (say p) against a join point x yields the same result as matching $trans\ p$ against x. In the presence of variables however (introduced using the *args* element descriptor), the situation is more complex. AJD defines the matching process of *cflow* so that it takes a “minimal munch” from the left of the join point. This characteristic is not reflected in the above translation, although our implementation of regular expression matching does produce the same behaviour.

We have only very limited experience with writing pointcut designators, but it would seem that the regular expression syntax is just a more primitive counterpart to the notation in AJD. Admittedly, however, regular expressions tend to be somewhat more verbose. For example, consider

$$\{pcall(swap)\}\,;\{true\}^*\,;\{pcall(quicksort)\}\,;\{true\}^*$$

In AJD, one would write the much shorter

$$pcall(swap) \wedge cflow(quicksort)$$

(in the concrete syntax of AJD, the *top* constructor is invisible).

The AJD pointcut language is however complicated by some subtle variants of *cflow*, like *cflowbelow*, which is the same as *cflow*, but it operates on the tail of a join point. This extra operator is necessary to express pointcuts such as

$$\{pcall(quicksort)\}\,;\{\neg(pcall(quicksort))\}^*$$

Which would be written

$$pcall(quicksort) \wedge \neg(cflowbelow(quicksort))$$

We saw another example of the use of *cflowbelow* in Section 5.1.

It is our belief that regular expressions provide a nice set of primitives to define higher-level constructs, including those of the AJD pointcut language. We are not advocating that the *cflow* notation is replaced by regular expressions: we merely suggest that it may be worthwhile to offer the more primitive notation for situations where the current set of higher-level constructs proves awkward or inadequate.

6.2 *around* advice

We earlier described the notion of *around* advice. At a matching join point, the corresponding piece of advice is executed. When *proceed* statement is encountered, the procedure that originated this join point is executed, after which advice execution resumes. It is not necessary for the advice to contain a *proceed* statement, and it is thus possible to completely replace the existing procedure. Furthermore, the *proceed* statement takes parameters, which are used instead of those in the original procedure call.

This powerful feature does not present any conceptual difficulties for the analysis we have outlined. It certainly complicates the construction of the call graph, which so far did not need to take the type of advice (*before* or *after*) into account. It is thus for expository reasons that we decided not to consider *around* advice in this paper.

6.3 Objects

The construction of the call graph is also complicated by considering virtual methods. At each virtual method call, we need to determine what instances might be called from that point in the code. Fortunately, however, a great deal of research has been devoted to such virtual method call resolution (*e.g.* [14, 24]), and we foresee no problems in combining it with the analysis presented here.

7 Conclusion and Future Work

This paper has reported on a first exploration of static analysis of aspects. In particular, it has been shown how the runtime overheads of matching pointcut designators can be reduced, and sometimes completely eliminated. This is encouraging, and we feel it warrants a larger research effort, where these and similar techniques are applied to a realistic aspect language, so that meaningful performance experiments can be conducted.

The efficiency of our analysis is acceptable for small examples, taking seconds on programs of a few hundred lines. The main bottleneck is in the preprocessing phase, where the chip-chop matrix is constructed. This new application of chip-chop matrices, in addition to that in [7], suggests that it is worthwhile investigating efficient algorithms for their construction. Conway's monograph (where chip-chop matrices originated) [6] and Backhouse's thesis [3] contain a wealth of theory that can guide this research.

The first AspectJ compiler was a whole-program compiler, and currently it is being re-engineered to allow separate compilation, and use incremental recompilation where necessary. We are hopeful that our analysis can fit this setting: to re-use the work from a previous pass of the analysis, one can store the S and D matrices (Section 3.4) for each procedure. If a program change is known not to affect the call graph of a procedure, the matrices can be re-used.

Another interesting direction for future work is the application of static analysis to aid the refactoring of legacy code [10], extracting slices of the original program into aspects. Consider the tangled version of our *quicksort* example, where the relevant counters have been manually placed in the program. Standard program slicing [27] will extract the computations from the original program, thus giving us the relevant pieces of advice. If the original program already contains aspects, we can use the results of [28]. To then construct a new aspect, suitable pointcut designators have to be associated with each piece of advice: a static analysis could assist in finding suitable regular expressions. For this purpose, the output of Tarjan's algorithm on the call graph needs to be simplified, as the resulting regular expressions are seldom in the simplest possible form. It is our belief that tools to easily move from traditional code to aspect-oriented views are indispensable for the acceptance of this new paradigm. The present paper has investigated how to translate from aspects to traditional programs, but the reverse direction is equally important.

Finally, in large aspect-oriented programs, it is important for programmers to be warned of potential interactions between aspects. A static analysis for this problem was first proposed in [8]. We are hopeful that the results of this paper can be used similarly, namely to detect when two different pieces of advice may both be executed at the same program point.

Acknowledgements

The authors are grateful to Gregor Kiczales for helpful comments on an early draft, especially concerning AspectJ. They would like to thank Kris de Volder and Hidehiko Masuhara, for patiently explaining the finer points of aspects and their compilation during a very productive visit to Oxford in May 2002. Hidehiko pointed out some inaccuracies in a draft of this paper. Members of the tools group at Oxford also provided valuable discussion and feedback.

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *ECOOP '93 Workshop on Object-based Distributed Programming*, volume 791 of *Lecture Notes in Computer Science*, pages 152–184. Springer, 1994.

[2] U. Aßmann and A. Ludwig. Aspect Weaving by Graph Rewriting. In U. W. Eisenecker and K. Czarnecki, editors, *Generative Component-based Software Engineering (GCSE)*, number 1799 in Lecture Notes in Computer Science, Erfurt, 1999.

[3] R. C. Backhouse. *Closure algorithms and the star-height problem of regular languages*. Ph.d. thesis, Imperial College, London, 1975.

[4] The Caml language, 2002. `http://caml.inria.fr/`.

[5] Communications of the acm. Volume 44:10, October 2001. Special issue on aspect-oriented programming.

[6] J. H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, 1971.

[7] O. de Moor, S. Drape, D. Lacey, and G. Sittampalam. Incremental program analysis via language factors. submitted for publication, 2002.

[8] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, 2002.

[9] R. Douence, O. Motelet, and M. Sudholt. A formal definition of crosscuts. In A. Yonezawa and S. Matsuoka, editors, *Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186. Springer, 2001.

[10] M. Fowler. *Refactoring: improving the design of existing code*. Addison Wesley, 2000.

[11] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, second edition, 2001.

[12] Y. Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Higher-order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from *Systems, Computers, Controls* 2(5):45–50, 1971.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[14] D. Grove, G. Furrow, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM Press, 1997.

[15] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In A. Paepcke, editor, *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 411–428. ACM Press, 1993.

[16] C. A. R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[17] G. Kiczales and J. des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *European Conference on Object-oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[19] G. Kiczales, J. Lamping, A. Menhdekar, C. Maeda, C. Lopes, J. Loingties, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *European Conference on Object-oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[20] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[21] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In *Foundations of Aspect-Oriented Languages (FOAL), Workshop at AOSD 2002*, Technical Report TR #02-06, pages 17–26. Iowa State University, 2002.

[22] L. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[23] D. Sereni. A definitional interpreter for aspects. `http://www.comlab.ox.ac.uk/oucl/research/areas/progtools/aspects`, 2002.

[24] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.

[25] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the Association for Computing Machinery*, 28(3):594–614, 1981.

[26] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *Foundations of Aspect-Oriented Languages (FOAL), Workshop at AOSD 2002*, Technical Report TR #02-06, pages 1–8. Iowa State University, 2002.

[27] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, 1984.

[28] J. Zhao. Slicing aspect-oriented software. In *10th IEEE Workshop on Program Comprehension*, pages 251–260, 2002.