Aspects for Trace Monitoring

Pavel Avgustinov[1], Eric Bodden[2], Elnar Hajiyev[1], Laurie Hendren[2],
Ondřej Lhoták[3], Oege de Moor[1], Neil Ongkingco[1], Damien Sereni[1],
Ganesh Sittampalam[1], Julian Tibble[1], Mathieu Verbaere[1]

[1] Programming Tools Group, Oxford University, United Kingdom
[2] Sable Research Group, McGill University, Montréal, Canada
[3] Programming Languages Group, University of Waterloo, Waterloo, Canada

Abstract. A *trace monitor* observes the sequence of events in a system, and takes appropriate action when a given pattern occurs in that sequence. Aspect-oriented programming provides a convenient framework for writing such trace monitors. We provide a brief introduction to aspect-oriented programming in AspectJ. AspectJ only provides support for triggering extra code with single events, and we present a new language feature (named *tracematches*) that allows one to directly express patterns that range over the whole current trace. Implementing this feature efficiently is challenging, and we report on our work towards that goal.
Another drawback of AspectJ is the highly syntactic nature of the event patterns, often requiring the programmer to list all methods that have a certain property, rather than specifying that property itself. We argue that *Datalog* provides an appropriate notation for describing such properties. Furthermore, all of the existing patterns in AspectJ can be reduced to Datalog via simple rewrite rules.
This research is carried out with *abc*, an extensible optimising compiler for AspectJ, which is freely available for download.

1 Introduction

When checking temporal properties at runtime, it is convenient to use a special tool for instrumentation. Ideally we would like to give a clean, declarative specification of the property to be checked, and then leave it to a tool to insert the appropriate instrumentation, possibly applying optimisations to reduce the overheads inherent in checking properties at runtime.

Aspect-oriented programming shares many of these goals, and in fact its stated ambitions are even grander, namely to improve software modularity in general. Briefly, an aspect observes all events (method calls, field sets/gets, exceptions, . . .) that occur in a system, and when certain events of interest happen, the aspect runs some extra code of its own. The events of interest are specified by the programmer via special patterns named *pointcuts*; the intercepted events are named *joinpoints*.

In this paper, we aim to assess the suitability of AspectJ (the most popular aspect-oriented programming language) for checking temporal properties. We do this via a familiar example, namely that of checking the safe use of enumerations (no updates to the underlying collection may happen while an enumeration is in progress).

In AspectJ one can specify only patterns that range over individual events, and we present a language extension where patterns can range over the whole computation history instead. It is quite hard to implement such a feature efficiently, and we report on the success we have had in approaching the efficiency of hand-coded solutions.

Another difficulty with AspectJ is that the patterns are very syntactic. It is common, for instance, that one needs to intercept calls to ‘any methods of a class *C* that may change the state of *C*’. In AspectJ the solution is to list all such methods by name. We propose to use *Datalog* instead to write queries that directly capture the property in question. Datalog is a little more verbose than the pattern language of AspectJ, but we show AspectJ patterns are merely syntactic sugar: they can all be translated into Datalog via a set of simple rewrite rules.

2 Aspect-oriented Programming

In this section, we present aspect-oriented programming using fail-safe *Enumeration*s as a motivating example. In subsequent sections, we will show how the aspect-oriented implementation of this example can be further improved using tracematches and Datalog pointcuts.

The *Enumeration* interface is an older version of the more well-known *Iterator* type: in particular it provides a *nextElement* method, and also *hasMoreElements*. An important difference is that implementations of *Iterator* are expected to be *fail-fast*: if the underlying collection is modified while iteration is in progress (through any method other than *Iterator.remove*()) an exception should be thrown. There is no such expectation for implementations of *Enumeration*.

To illustrate, suppose we have a vector *v* that is accessed by two concurrent threads. Thread 1 creates an enumeration (say *e*) over *v*, and does some enumeration steps. In the meantime, thread 2 modifies *v* by adding an element. When thread 1 does another enumeration step, its result is undefined. This situation is illustrated in Figure 1.

THREAD 1:	THREAD 2:
...	
Enumeration e = **new** MyEnum(v);	
...	
Elt a = (Elt) e.nextElement();	...
...	v.add(b)
a = (Elt) e.nextElement();	...

Fig. 1. Unsafe use of *Enumeration*.

Of course there is an easy way to make implementations of *Enumeration* safe. First, add a *stamp* field of type *long* to both the *Vector* class, and to any class implementing *Enumeration*. One can think of this stamp as a version number: we use it to check whether the current version of a vector is the same as when the enumeration was created. Furthermore, every *Enumeration* should have a *source* field, which records the data source (a *Vector*) being enumerated.

Whenever a new enumeration *e* over a vector *v* is created, we make the following assignments:

```
e.stamp = v.stamp;
e.source = v;
```

The version of a vector *v* changes upon each modification, so whenever a change is made to *v*, we execute

```
v.stamp++;
```

Finally, whenever we do an enumeration step, it is checked that the version numbers are still in synch:

```
if (e.source != null && e.stamp != e.source.stamp)
  throw new ConcurrentModificationException();
```

We must make the check that the source is not null in case the enumeration *e* is in fact not over a vector, but instead over some other collection type.

2.1 Aspects

Aspect-oriented programming provides us with the means to implement the check outlined above in a nice, modular fashion. Intuitively, an aspect can inject new members into existing classes (the new *stamp* and *source* fields above). An aspect can also intercept events like the creation of an enumeration, and execute some extra code.

In AspectJ, aspects are implemented via a *weaver* that takes the original system and the aspect, and it instruments the original system as described in the aspect. As a consequence, aspects achieve the goal set out at the beginning of this paper: the instrumentation code is neatly separated from the system being observed.

An outline of the aspect for the example of fail-fast enumeration is shown in Figure 2. Note how we introduce the *stamp* field on *Vector* by the declaration on Line 3. It is declared *private* — that means it is visible only from the aspect that introduced it.

Similarly, we introduce the *stamp* and *source* fields on the *Enumeration* interface, along with appropriate accessor methods (Lines 6–12). This has the effect of introducing these new members on every *implementation* of *Enumeration* as well.

This mechanism of introducing new members onto existing classes is an admittedly rather crude form of *open classes*; we shall briefly mention some more disciplined alternatives below.

Now our task is to program the requisite updates to these newly introduced fields. In AspectJ, one does this through so-called *advice* declarations. A piece of advice consists of a pattern (the *pointcut*) describing the event we wish to intercept, some extra code to execute, and an instruction when to execute that code (before or after the event).

Figure 3 shows three pieces of advice. The first piece, on Lines 1-6, intercepts all constructor calls on implementations of the enumeration interface, where the constructor call has the data source *ds* of type *Vector* as its actual argument. We are assuming, therefore, that all enumerations over vectors are created via such constructor calls. As indicated earlier, here we have to set the version number (*stamp*) of the enumeration, as well as its *source* field.

The next piece of advice in Figure 3, on Lines 8-12, intercepts updates to the *Vector* class, and whenever they occur, the version number is incremented. Here we have

```
public aspect SafeEnum {

  private long Vector.stamp = 0;

// introduce new members on every implementation of Enumeration
  private long Enumeration.stamp;
  private void Enumeration.setStamp(long n) { stamp = n; }
  private long Enumeration.getStamp() { return stamp; }

  private Vector Enumeration.source;
  private void Enumeration.setSource(Vector v) { vector = v;}
  private Vector Enumeration.getSource() {return vector;}

// ... intercept creation, update and nextElement ...
}
```

Fig. 2. Making *Enumeration* safe.

employed a named pointcut *vector_update* to describe all calls to methods that may change the state of *Vector*, and we shall look at its definition shortly.

The final piece of advice in Figure 3 occurs on Lines 14-19. This intercepts calls to *nextElement*, and it checks whether the version number on the enumeration agrees with that on the vector. If they do not coincide, an exception is thrown.

```
1  synchronized after(Vector ds) returning (Enumeration e) :
2      call(Enumeration+.new(..)) && args(ds)
3  {
4   e.setStamp(ds.stamp);
5   e.setSource(ds);
6  }
7
8  synchronized after(Vector ds) :
9      vector_update() && target(ds)
10 {
11   ds.stamp++;
12 }
13
14 synchronized before(Enumeration e) :
15      call(Object Enumeration.nextElement()) && target(e)
16 {
17   if (e.getSource() != null && e.getStamp() != e.getSource().stamp)
18    throw new ConcurrentModificationException();
19 }
```

Fig. 3. Advice for safe enumeration.

The final piece of code we must write to complete this aspect is the pointcut for intercepting calls that may change the state of the *Vector* class. The received way of doing that is to carefully examine each method in *Vector*, and list it in the pointcut. The result is shown in Figure 4. Note that to reduce the number of disjucts, we have used wildcards in the name patterns.

```
pointcut vector_update() :
    call (* Vector.add* (..)) ||
    call (* Vector.clear ()) ||
    call (* Vector.insertElementAt (..)) ||
    call (* Vector.remove* (..)) ||
    call (* Vector.retainAll (..)) ||
    call (* Vector.set* (..));
```

Fig. 4. Pointcut for updates on *Vector*.

To use the aspect we have just written, one just includes it on the command line of the compiler, and the result is an instrumented version of the original program, now with the ability to catch unsafe uses of enumerations over vectors, whenever they occur.

2.2 Pros and cons of aspects

The advantages of using aspects are apparent. It allows easy, flexible instrumentation, while retaining the advantages (in particular good compiler error messages) of a high-level programming language. Experiments show that for the above example, the overheads introduced by aspects (as compared to making the changes by hand in the original program) are negligible. Finally, AspectJ is a fairly mature programming language, with good tool support, and numerous textbooks for newcomers to get started.

Not all is rosy, however. Our purpose is to check a property of traces – that no updates occur during enumeration – and while that property is *encoded* in the above aspect, it would be much preferable to state the property directly, in an appropriate specification formalism. The compiler should then generate the checking code from the specification. Also the pointcut in Figure 4 leaves much to be desired: for a library class like *Vector* it might be acceptable, but what about a class that might change over time? Whenever a new method is introduced, we have to remember that the pointcut may need to be altered as well. Both of these problems (direct specification of trace properties and semantic pointcuts) will be addressed below.

There are some further disadvantages of aspects that we shall not discuss further, but it is still worthwhile to mention them here. For now, the semantics of aspects remain an area of active research. In particular, a crisp definition of the AspectJ language itself is still lacking. More generally, aspects introduce many problems with modular reasoning about programs, because they can interfere with existing code in unpredictable ways.

Finally, above we have made light of the problem of modifying library classes like *Vector* and *Enumeration*. Without support in the JVM, this is hard to achieve, and if we wish to use a compile-time weaver some trickery is needed to replace every *Vector* in an application by our own subclass *MyVector*. These changes, while somewhat akward, can be concisely expressed in AspectJ as well; a complete version of the above aspect, with these changes incorporated, is available on-line as part of a more general benchmark suite [2].

2.3 Further reading

The AspectJ language was introduced by Kiczales *et al.* in [55]. It is now widely used in practice, and there is a wealth of textbooks available, for instance [19, 27, 41, 56, 57, 63]. We found especially Laddad's book [57] very helpful, because it discusses a wide variety of applications. It also identifies some common design patterns in aspect-oriented programming.

Method interception as found in aspect-oriented programming has its origins in previous work on meta-programming with objects, in particular [14, 54]. Of course there have been earlier systems that provided similar forms of method interception, for instance the POP-2 language [23] or even Cobol [59]. It was only with the advent of aspects, however, that this language feature was recognised as a structuring mechanism in its own right: before that, it was mostly used for debugging purposes.

The static features of aspects, namely the ability to inject new class members into existing classes also has a long history. Harrison and Ossher coined the term *subject-oriented programming* [48], but arguably their composition mechanisms are much more powerful than those offered by AspectJ, as their open classes can be symmetrically composed. Recent years have seen a lot of research on giving open classes a more disciplined basis, for instance [26]. Nested inheritance [64] and virtual classes [37, 65] have similar goals, while satisfying stronger formal properties.

While AspectJ is presently the most popular aspect-oriented programming language, it is certainly not the only language available. CaesarJ adds dynamic deployment of aspects, creating new instances of aspect classes and attaching them to computations at runtime; it also has a notion of virtual classes instead of AspectJ's member injections [7]. A long list of current aspect-oriented programming systems can be found at [6].

Following closely on the growing popularity of aspect-oriented programming, researchers have started to address the problem of defining its semantics. An early attempt was a definitional interpreter by Wand *et al.* [76]; this offered little help, however, in reasoning about aspect code. More refined models have since been proposed by Walker *et al.* [74], Bruns *et al.* [21], and Aldrich [4]. Aldrich's model is especially attractive because it gives a basis for modular reasoning about aspects. We have ourselves adapted his language design to a full extension of the AspectJ language [66].

Our own interest in AspectJ started with a study of the runtime overheads [36]. At the time, it was believed that such overheads are negligible, but it turns out that certain features (in particular the *cflow* pointcut and *around* advice) can lead to substantial costs at runtime. We therefore decided to implement our own extensible, optimising compiler, named the *AspectBench Compiler*, or *abc* for short [8]. Using its analysis infrastructure, we were able to eliminate most of the overheads we identified earlier [9] (one of the optimisations had been proposed earlier in [70], for a small toy language). *abc* is however not only intended for optimisation; it is also designed as a workbench for experiments in language design. The two major case studies we have undertaken so far are tracematches [5] (discussed in the next section), and open modules [66] (mentioned above). A detailed overview of all the work on *abc* to date, as well as a comparison with the other AspectJ compiler *ajc*, can be found in [10]. *abc* itself can be downloaded from [1].

3 Tracematches

Tracematches are a new feature that we have introduced into AspectJ. As mentioned earlier, normal advice in AspectJ is triggered by single events. Instead, in tracematches one can specify a regular pattern that is matched against the whole computation history so far.

We need to be a bit more precise about the nature of events at this point. In AspectJ, pointcuts intercept composite events like method calls, which have a duration. Instead, when we talk about a trace, we mean the sequence of before/after actions associated with such composite events: these are atomic.

To illustrate, an example tracematch is shown in Figure 5. It is intended to introduce an autosave feature into an existing editor system. A tracematch consists of three parts: the declaration of the symbols (events) of interest (Lines 3 and 4), a regular pattern (Line 6) and a piece of code (Line 8). Here there are two symbols: the end of a *save* operation, and the end of the execution of a command. The pattern specifies five consecutive occurrences of the *action* symbol. Because we have declared an interest in saves as well, that means the pattern only matches if five actions occur, with no intervening saves. When that happens, the extra code is run, and here that is just the *autosave* method.

```
1  tracematch() {
2
3    sym save  after :    call ( * Application.save() ) || call ( * Application.autosave() );
4    sym action  after :  call ( * Command.execute() );
5
6    action [5]
7
8    { Application.autosave(); }
9  }
```

Fig. 5. An example tracematch.

This is an important point: the symbol declarations determine what trace we match against. The original trace is *filtered*, leaving out all events that do not correspond to a declared symbol. The pattern is then matched against all suffixes of the filtered trace, and when it matches, the code in the body of the tracematch is executed. Note that we never filter out the very last event that happened: if we did, one could run the code some time after an actual match occurred, with some irrelevant events in between. This process of filtering and matching is illustrated in Figure 6.

The above tracematch is atypical because it does not bind any variables. Local tracematch variables may be declared in the header, and are bound by the matching process. In Figure 7, we have displayed a tracematch that is equivalent to the aspect for safe enumeration discussed earlier. This tracematch does bind two variables, namely the vector *ds* and the enumeration *e* (Line 1). Here there are three symbols of interest (Lines 3-5): creating an enumeration, doing a next step, and updating the source. We wish to catch unsafe uses of enumerations, and this is expressed by the pattern (Line 7). First we see

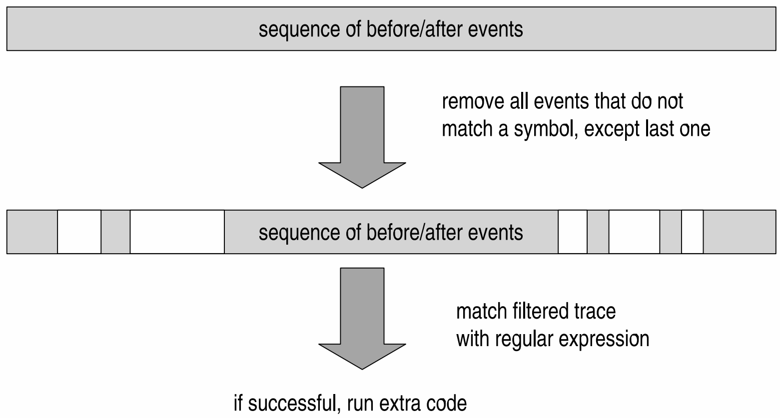

Fig. 6. Filtering of traces (no variables).

an enumeration being created, then zero or more 'next' steps, one or more updates and finally an erroneous attempt to continue the enumeration.

```
tracematch(Vector ds, Enumeration e) {

    sym create_enum after returning(e) : call (Enumeration+.new(..)) && args(ds);
    sym call_next before : call (Object Enumeration.nextElement()) && target(e);
    sym update_source after : vector_update() && target(ds);

    create_enum call_next∗ update_source+ call_next

    { throw new ConcurrentModificationException (); }

}
```

Fig. 7. Tracematch for safe enumeration.

It might appear that there is no need to mention the intervening enumeration steps via *call_next* *. However, because of our definition of matching via filtering, that would be wrong. The pattern is matched against all suffixes of the filtered trace, and not to arbitrary subsequences.

The precise meaning of filtering in the presence of local tracematch variables is defined in the obvious manner: instantiate the free variables in all possible ways, and then match as we did before. This process is illustrated in Figure 8. As the figure suggests, if a match occurs with multiple variable bindings, the extra code is run for each of those bindings separately.

While it is nice to understand the semantics of tracematches in terms of all possible instantiations of its free variables, that does not provide a basis for implementation. We therefore also require an operational semantics. It is fairly obvious that this semantics will keep a finite state machine for the pattern. Each state of the machine is labelled with a *constraint* that describes the variable bindings made to arrive at that state. To wit, these constraints are equalities (variable = object), inequalities (variable $\neq$ object),

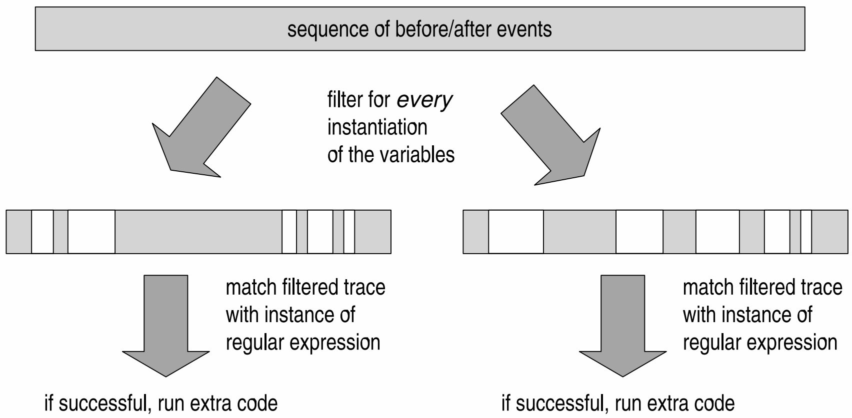

Fig. 8. Filtering of traces (with variables).

or combinations with conjunction and disjunction of these. A detailed definition of the operational semantics can be found in our original paper on tracematches [5].

Unfortunately a direct implementation of the operational semantics does not yield a practical system. The main problem is that of memory leaks, and there are two possible sources of these. First, we may hang on too long to existing objects, merely because they were bound to a tracematch variable. Second, partial matches may stay around forever, despite the fact that they can never be completed. In fact, we keep our constraints in disjunctive normal form, so 'partial matches' correspond to disjuncts in our representation of constraints.

To solve the problem of memory leaks, we have devised a static analysis of the tracematch, which classifies each variable v on each state s in one of three categories:

collectable when all paths in the automaton from s to all final states contain a transition that binds v. In that case we can use weak references for bindings of v. Furthermore, when the garbage collector nullifies that weak reference, we can discard all disjuncts that contain it.

weak not collectable, but the advice body does not mention v. We can still use weak references for bindings of v, but it would be incorrect to discard a disjunct upon nullification.

strong not collectable and not weak. A normal strong reference must be used to store bindings of v.

Note that this is a purely local analysis on the tracematch, involving no analysis of the instrumented system, so that it does not significantly affect compile times.

The technique appears to be highly effective in practice. As an example, we have applied this instrumentation to JHotDraw, the popular open source drawing program. It has a feature for animating a drawing; that in fact introduces an unsafe use of enumerations, because one can edit the drawing while the animation is in progress. The results of measuring memory usage over time are shown in Figure 9. We compared a number of different systems. First, we evaluated our tracematch implementation with leak detection and prevention disabled, using strong references for everything. This line

(TMNoLeak) stops after a few steps because execution becomes infeasibly slow. PQL is a runtime trace property checking system created by Monica Lam and her students at Stanford [61]. We tried several version of this benchmark with PQL (PQL and PQL-Neg), and both show linear memory growth over time. Next the figure shows a naive aspect (AjNaive), that instead of using new fields associates time stamps via an identity hash map. The figure also shows a smarter aspect (AjNormal), that uses a weak identity hash map for the same purpose, and finally our optimised implementation of tracematches. The aspect shown at the beginning of this paper also has constant space usage. More details of these experiments can be found in a technical report [11].

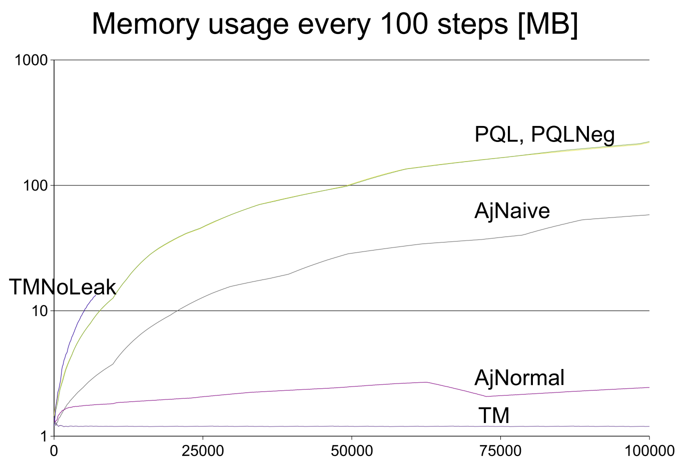

Fig. 9. Memory usage for SafeEnum (moving average to show trends).

Timewise our implementation is still quite a lot behind the hand-coded aspect at the beginning of this paper. The time taken for 100,000 animation steps is shown in Figure 10. TM indicates our optimised implementation, whereas AjGold is the 'gold standard' aspect shown earlier. We believe that a static analysis of the instrumented program can bring one closer to the gold standard, but for now that remains future work. While this result may appear disappointing, we should mention the instrumented animation is still quite usable on a normal PC.

Figure 11 shows some further substantial applications of tracematches. It would take us too far afield to discuss each of these in detail, but a number of interesting trends can be identified. The first column shows the name of the tracematch being applied, the second the base program being instrumented, and the third column displays the size of that base program. Note that we have used some non-trivial applications. The column marked 'none' shows the execution time, in seconds, of the non-instrumented application. The 'AspectJ' column displays the execution time of a hand-coded version in AspectJ for each benchmark. The final three columns measure our own implementation. 'leak' refers to switching off the above analysis, whereas 'noidx' means that we do not use a special indexing data structure to quickly identify the relevant partial matches when a new variable binding occurs. The final column is our optimised im-

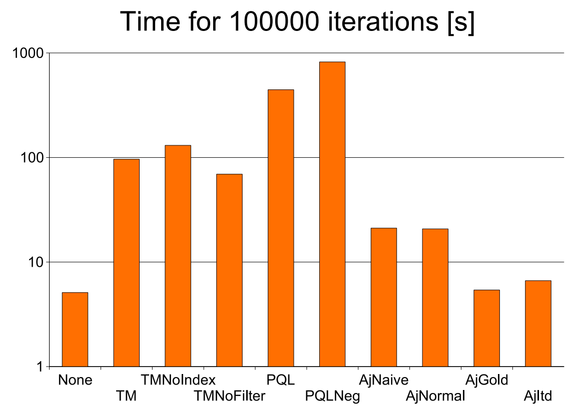

Fig. 10. Runtimes for SAFEENUM.

plementation with both leak prevention and indexing switched on. Clearly indexing is just as important as leak prevention, as indicated by the highlighted numbers in the top three rows. The interested reader is referred to [11] for full details of these and other experiments. The full experimental setup is available on-line for others to try with their own monitoring systems [2].

monitor	base	ksloc	none	aspectj	leak	noidx	TM
nulltrack	certrevsim	1.4	0.2	0.5	1.6	25.6	1.6
hashcode	aprove	438.7	345.0	458.9	>90m	>90m	845.1
observer	ajhotdraw	9.9	2.7	2.9	4.1	15.8	4.1
dbpooling	artificial	<0.1	70.0	4.5	5.0	4.8	4.9
luinmeth	jigsaw	100.9	13.6	18.0	21.9	20.9	22.4
lor	jigsaw	100.9	13.6	19.9	34.9	34.7	34.9
reweave	abc	51.2	4.5	5.4	9.1	9.0	8.7

Fig. 11. More tracematch benchmarks.

3.1 Summary of tracematches

The implementation of tracematches is surprisingly tricky to get right. Even ignoring the issue of space leaks, we found several bugs in our first prototype, which only came to light when we tried to prove the equivalence of the declarative and operational semantics. As shown by the above experiments, the implementation has now been thoroughly tested, and it is available in the standard distribution of the *AspectBench Compiler (abc)* for AspectJ.

The key to our efficient implementation consists of two parts: the prevention of memory leaks, and an efficient data structure for retrieving partial matches. Both of these only rely on local analysis of the tracematch itself, not of the instrumented program. We are currently investigating analyses of the instrumented program that may help to approach the efficiency of hand-coded solutions.

3.2 Further reading

The idea of generating trace monitors from specifications is an old one, and there exists a very large amount of previous research on this topic, *e.g.* [5, 12, 16, 17, 25, 28, 28, 31–35, 38, 40, 50, 61, 71, 72, 75]. These studies range from applications in medical image generation through business rules to theoretical investigations of the underlying calculus. The way the patterns are specified varies, and temporal logic, regular expressions and context-free languages all have been considered.

One theme shines through all of these previous works: trace monitors are an attractive, useful notion, worthy of integration into a mainstream programming language. This has not happened, however, because it turns out to be very difficult to generate efficient code when the trace monitor is phrased as a declarative specification.

Our own contributions have been to provide a solid semantic basis for trace monitors [5] (in particular a proof of equivalence between the declarative and operational semantics), and to devise optimisations that make trace monitors feasible in practice [11].

4 Datalog pointcuts

We now turn to the way individual events are intercepted in AspectJ. Recall the definition of the *vector_update* pointcut in Figure 4: it was just a list of the relevant methods. It would be much nicer to express the desired semantic property directly, and leave it to the weaver to identify individual methods that satisfy the property.

So in this example, what is the property exactly? We are interested in methods that may change the behaviour of the *nextElement* method on the *Enumeration* interface. Therefore, we seek to identify those methods of *Vector* that write to a memory location that may be read by an implementation of *nextElement*. How do we express that intuition in a formal notation?

The key idea is that the program could be regarded as a relational database. Pointcuts are then just queries over that database, which are used to identify *shadows*. A shadow is a piece of code which at run-time gives rise to an event (a joinpoint) that can be intercepted by AspectJ.

Examples of the relations that make up a program are shown in Figure 12. The first three refer to declarations; the *implements* relation is not transitive. The shadow relations identify calls, method bodies, and field gets. Of course this list is not complete: there are shadows for all kinds of events that can be intercepted in AspectJ. Finally, there is the lexical containment relation *contains*. Again this is not assumed to be transitive.

typeDecl (RefType T, String N, Boolean IsIntf, Package P)
  T has name N in package P, IsIntf indicates T interface or not
implements (Class C, Interface I)
  C implements interface I
methodDecl (Method M, String N, RefType C, Type T)
  M has name N, is declared in C, and has return type T
callShadow (Shadow S, Method M, RefType T)
  S is a call to M with static receiver type T
executionShadow (Shadow S, Method M)
  S is the body of M
getShadow (Shadow S, Field F, RefType T)
  S is get of F with static receiver type T
contains (Element P, Element C)
  C is lexically contained in P

Fig. 12. Program relations.

We now have to decide on the query language for identifying shadows where the aspect weaver will insert some extra code. Many authors have suggested the use of logic programming for this purpose, in particular Prolog. There are numerous problems with that choice, however. First, it is notoriously hard to predict whether a Prolog query terminates. In the present setting, non-terminating queries yield uncompilable programs, which is undesirable. Second, to achieve acceptable efficiency, Prolog programs must be annotated with parameter modes, with the cut operation and with tabling instructions. Again, for this application that would not be acceptable. Yet, the arguments for using logic programming, in particular recursive queries, are quite compelling.

The appropriate choice is therefore 'safe, stratified *Datalog*' [39]. Datalog is similar to Prolog, but it does not allow the use of data structures; consequently its implementation is far simpler. The restriction to safe, stratified Datalog programs guarantees that all queries terminate. Yet, this restricted query language is powerful enough to express the properties we desire.

This is illustrated in Figure 13, which identifies the update methods *M* of the *Vector* class. It starts by finding the *Vector* class, and some implementation *I* of *Enumeration*; *I* contains a method *N* named *nextElement*. We check whether there exists a field *F* that may be read by *N*, while it may be written by *M*.

It remains to show how predicates like *mayRead*(*N*,*F*) can be defined. The key is that *N* does not need to directly read *F*; we must also cater for the situation where *N* calls another method, which in turn reads *F*. Similarly, the read shadow may not be directly lexically contained in the body of *N*, but perhaps in a method of a local class defined inside *N*. The relevant Datalog query is shown in Figure 14: *M* may transitively contain a shadow *G* which is a read of the field *F*.

```
vector_update_method(Method M) :-
  // M is a method in a class V named Vector
  typeDecl(V,'Vector',_,_),
  methodDecl(M,_,V,_,_),

  // N is a method named nextElement in an
  // implementation I of the Enumeration interface
  typeDecl(E,'Enumeration',_,_),
  implements(I,E),
  methodDecl(N,'nextElement',I,_),

  // N may read field F (possibly via a chain of calls)
  mayRead(N,F),

  // M may write field F
  mayWrite(M,F).
```

Fig. 13. Datalog for update methods.

```
mayRead(Method M,Field F) :-
  callContains(M,G),
  getShadow(F,G).

callContains(Method M, Shadow G) :-
  mayCall*(M,M'),         // static call chain from M to M'
  executionShadow(E,M'), // body of M' is E
  contains+(E,G),         // it contains a shadow G

mayCall*(Method X, Method Z) :- X=Z, method(X).
mayCall*(Method X, Method Z) :- mayCall(X,Y),mayCall*(Y,Z).

contains+(Element X, Element Z) :- contains(X,Z).
contains+(Element X, Element Z) :- contains(X,Y),contains+(Y,Z).
```

Fig. 14. Definition of *mayRead*.

4.1 An alternative surface syntax

Datalog is powerful, but for really simple pointcuts (like identifying calls to a method with a specific signature) it is verbose and awkward. By contrast, the AspectJ pointcut language shines in such examples, not least because any valid method signature is also a valid AspectJ method pattern. This is one of the reasons newcomers find AspectJ easy to pick up: if you know Java, you know how to express simple pointcuts. Can we give Datalog a similarly attractive syntax?

One might also be concerned about the formal expressive power of Datalog. When it comes to the finer points of AspectJ pointcuts, can they really be expressed as Datalog queries?

The answer to both of these questions is 'yes'. We have constructed a translation from AspectJ pointcuts to Datalog, which consists solely of simple rewrite rules. It is our intention to open up that implementation to advanced users, so they can define new query notations, along with rules for reducing them to Datalog.

Here is an example rule, used in the translation of *call* pointcuts.

> *aj2dl*(call(methconstrpat),C,S)
> $\rightarrow$
> $\exists$ X, R : (*methconstrpat2dl*(methconstrpat,C,R,X), callShadow(S,X,R))

The constructor *aj2dl*(*PC*,*C*,*S*) is used to drive the translation: it takes a pointcut *PC*, the current aspect class *C*, and a shadow *S*. This will be reduced to a Datalog query containing just *C* and *S* as free variables. Our implementation uses *Stratego*, which allows one to record the rewrite rules in concrete syntax, almost exactly as shown above [73].

4.2 Further reading

Various deficiencies of the AspectJ pointcut language are well-documented in the literature and on mailing lists [13,18,22,49]. Such dissatisfaction has led to several proposals for adopting Prolog as an alternative pointcut language [30,43,45]. Indeed, examples of systems that have built on those ideas are Alpha, Aspicere, Carma, Compose*, LogicAJ, Sally, and Soul [3, 15, 20, 42, 44, 47, 69]. However, the complex operational behaviour of Prolog renders it, in our view, too strong for the purpose of a pointcut language. We believe that this is the principal reason that none of these research prototypes has found widespread usage.

The program understanding community has long advocated a view of software systems as a relational database [24, 60]. They also considered new query languages, including Prolog, with [53] being an early example. It was soon realised that at a minimum, the query language must be able to express transitive closure [67]. Jarzabek proposed a language named PQL [52] that added such operators to a variant of SQL (not to be confused with PQL of Lam *et al.*, discussed below). The SQL syntax makes it rather awkward to express graph traversals concisely, however. A modern system that uses a tabled variant of Prolog is JQuery [51, 62].

The program analysis community has also frequently revisited the use of logic programming for specifying analyses. An early example making the connection is a paper by Reps [68]. More recently [29] provided an overview of the advantages of using logic programming to specify complex dataflow analyses. Very recently Martin *et al.* proposed another PQL (not to be confused with Jarzabek's language discussed above), to find bugs in compiled programs [58, 61]. Interestingly, the underlying machinery is that of Datalog, but with a completely different implementation, using BDDs to represent solution sets [77].

All these developments led us to the conclusion that Datalog is an appropriate query language for applications in program understanding, for pointcuts in aspect-oriented programming, and for program analysis. We have implemented the CodeQuest system as a first prototype, and are now working towards its integration into our AspectJ compiler *abc* [46].

5 Conclusion

We have shown how aspects can be used for checking temporal properties at runtime. The design and implementation of new features for such property checking is an interesting new field, requiring the joint efforts of experts in specification formalisms, in aspect-orientation, in program analysis and in compiler construction. We believe we have only scratched the surface, and hopefully this paper will encourage others to join us in our exploration. It would be especially interesting to consider other logics for expressing trace properties. One of us (Bodden) has already implemented a similar system [17, 71] based on LTL in lieu of regular patterns, also on top of *abc*.

It is important that a system for runtime instrumentation allows the programmer to make a judicious choice between static properties and dynamic ones. Our use of Datalog to describe compile-time analysis, for the purpose of identifying instrumentation points, is a step in that direction.

It is unlikely that a perfect specification notation can be found to express all desirable properties. It seems that an extensible syntax, where programmers can introduce new notations that are reduced to existing notions, provides a good compromise.

Acknowledgments We would like to thank Chris Allan, Aske Simon Christensen, Sascha Kuzins and Jennifer Lhoták for the collaboration that led up to this work. This research has been supported, in part, by EPSRC in the United Kingdom and by NSERC in Canada.

References

1. abc. The AspectBench Compiler. Home page with downloads, FAQ, documentation, support mailing lists, and bug database. `http://aspectbench.org`.
2. abc team. Trace monitoring benchmarks. `http://abc.comlab.ox.ac.uk/packages/tmbenches.tar.gz`, 2006.
3. Bram Adams. Aspicere. `http://users.ugent.be/~badams/aspicere/`, 2004.
4. Jonathan Aldrich. Open Modules: modular reasoning about advice. In Andrew P. Black, editor, *Proceedings of ECOOP 2005*, 2005.
5. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications*, pages 345–364. ACM Press, 2005.
6. AOSD.NET. Tools for developers. `http://www.aosd.net/wiki/index.php?title=Tools_for_Developers`, 2006.
7. Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of AspectJ. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer, 2006.
8. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. *abc*: An extensible AspectJ compiler. In *Aspect-Oriented Software Development (AOSD)*, pages 87–98. ACM Press, 2005.
9. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising aspectj. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 117–128, New York, NY, USA, 2005. ACM Press.

10. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. *abc*: An extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 293–334. Springer, 2006.
11. Pavel Avgustinov, Julian Tibble, Eric Bodden, Laurie Hendren, Ondřej Lhoták, Oege de Moor, Neil Ongkingco, and Ganesh Sittampalam. Efficient Trace Monitoring. Technical Report abc-2006-1, AspectBench Compiler Project, 2006. `http://abc.comlab.ox.ac.uk/techreports#abc-2006-1`.
12. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2003.
13. Ohad Barzilay, Yishai A. Feldman, Shmuel Tyszberowicz, and Amiram Yehudai. Call and execution semantics in AspectJ. In *Foundations Of Aspect Languages (FOAL)*, pages 19–24, 2004. Technical report TR #04-04, Department of Computer Science, Iowa State University.
14. Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: merging common lisp and object-oriented programming. In Norman K. Meyrowitz, editor, *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, volume 791, pages 152–184. ACM Press, 1986.
15. Christoph Bockisch. Alpha. `http://www.st.informatik.tu-darmstadt.de/static/pages/projects/alpha/index.html`, 2005.
16. Christoph Bockisch, Mira Mezini, and Klaus Ostermann. Quantifying over dynamic properties of program execution. In *2nd Dynamic Aspects Workshop (DAW05)*, Technical Report 05.01, pages 71–75. Research Institute for Advanced Computer Science, 2005.
17. Eric Bodden. J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, 2005.
18. Ron Bodkin. Pointcuts need a long form. `http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg05971.html`, 2006.
19. Oliver Böhm. *Aspectorientierte Programmierung mit AspectJ 5*. Dpunkt.verlag, 2006.
20. Johan Brichau, Kim Mens, and Kris de Volder. SOUL/aop. `http://prog.vub.ac.be/research/aop/soulaop.html`, 2002.
21. Glenn Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μABC: a minimal aspect calculus. In *Proceedings of CONCUR 2004*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224, 2004.
22. Bill Burke. *has* and *hasfield* pointcut expressions. `http://aosd.net/pipermail/discuss_aosd.net/2004-May/000958.html`, 2004.
23. Rod M. Burstall and Robin J. Popplestone. POP-2 reference manual. In Bernard Meltzer and Donald Michie, editors, *Machine Intellingence*, volume 5, pages 207–246. Edinburgh University Press, 1968.
24. Yih Chen, Michael Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, 1990.
25. María Augustina Cibrán and Bart Verheecke. Dynamic business rules for web service composition. In *2nd Dynamic Aspects Workshop (DAW05)*, pages 13–18, 2005.
26. Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM Press.
27. Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ development tools*. Addison-Wesley, 2004.

28. Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of java programs. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7. ACM Press, 2005.
29. Stephen Dawson, C. R. Ramakrishnan, and David Scott Warren. Practical program analysis using general purpose logic programming systems. In *ACM Symposium on Programming Language Design and Implementation*, pages 117–126. ACM Press, 1996.
30. Kris de Volder. Aspect-oriented logic meta-programming. In Pierre Cointe, editor, *2nd International Conference on Meta-level Architectures and Reflection*, volume 1616 of *Springer Lecture Notes in Computer Science*, pages 250–272, 1999.
31. Rémi Douence, Pascal Fradet, and Mario 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)*, pages 173–188, 2002.
32. Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Karl Lieberherr, editor, *3rd International Conference on Aspect-oriented Software Development*, pages 141–150, 2004.
33. Rémi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In *Aspect-oriented Software Development*, pages 141–150. Addison-Wesley, 2004.
34. Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura, and Mario Südholt. An expressive aspect language for system applications with arachne. In *Aspect-Oriented Software Development*, pages 27–38, 2005.
35. Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186. Springer, 2001.
36. Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the dynamic behaviour of aspectj programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 150–169, New York, NY, USA, 2004. ACM Press.
37. Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–282, New York, NY, USA, 2006. ACM Press.
38. Thomas Fritz, Marc Ségura, Mario Südholt, Egon Wuchner, and Jean-Marc Menaud. An application of dynamic AOP to medical image generation. In *2nd Dynamic Aspects Workshop (DAW05)*, Technical Report 05.01, pages 5–12. Research Institute for Advanced Computer Science, 2005.
39. Hervé Gallaire and Jack Minker. *Logic and Databases*. Plenum Press, New York, 1978.
40. Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 385–402, 2005.
41. Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. Wiley, 2003.
42. Trese group. Compose*. `http://janus.cs.utwente.nl:8000/twiki/bin/view/Composer/`, 2005.
43. Stefan Hanenberg Günter Kniesel, Tobias Rho. Evolvable pattern implementations need generic aspects. In *Proc. of ECOOP 2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 116–126. June 2004.
44. Kris Gybels. Carma. `http://prog.vub.ac.be/~kgybels/Research/AOP.html`, 2004.
45. Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *2nd International Conference on Aspect-oriented Software Development*, pages 60–69. ACM Press, 2003.

46. Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: scalable source code queries with Datalog. In Dave Thomas, editor, *Proceedings of ECOOP 2006*, Lecture Notes in Computer Science. Springer, 2006.
47. Stefan Hanenberg and Rainer Unland. Sally. `http://dawis.icb.uni-due.de/?id=200`, 2003.
48. William Harrison and Harold 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.
49. Jim Hugunin. Support for modifiers in typepatterns. `http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg01578.html`, 2003.
50. Peter Hui and James Riely. Temporal aspects as security automata. In *Foundations of Aspect-Oriented Languages (FOAL 2006), Workshop at AOSD 2006*, Technical Report #06-01, pages 19–28. Iowa State University, 2006.
51. Doug Janzen and Kris de Volder. Navigating and querying code without getting lost. In *2nd International Conference on Aspect-Oriented Software Development*, pages 178–187, 2003.
52. Stan Jarzabek. Design of flexible static program analyzers with PQL. *IEEE Transactions on Software Engineering*, 24(3):197–215, 1998.
53. Shahram Javey, Kin'ichi Mitsui, Hiroaki Nakamura, Tsuyoshi Ohira, Kazu Yasuda, Kazushi Kuse, Tsutomu Kamimura, and Richard Helm. Architecture of the XL C++ browser. In *CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*, pages 369–379. IBM Press, 1992.
54. Gregor Kiczales and Jim des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, 1991.
55. Gregor Kiczales, John Lamping, Anurag Menhdekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *European Conference on Object-oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.
56. Ivan Kiselev. *Aspect-oriented programming with AspectJ*. SAMS, 2002.
57. Ramnivas Laddad. *AspectJ in Action*. Manning, 2003.
58. Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2005. ACM Press.
59. Ralf Lämmel and Kris De Schutter. What does aspect-oriented programming mean to Cobol? In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 99–110, New York, NY, USA, 2005. ACM Press.
60. Mark A. Linton. Implementing relational views of programs. In Peter B. Henderson, editor, *Software Development Environments (SDE)*, pages 132–140, 1984.
61. Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 365–383, 2005.
62. Edward McCormick and Kris De Volder. JQuery: finding your way through tangled code. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 9–10, New York, NY, USA, 2004. ACM Press.
63. Russell Miles. *AspectJ cookbook*. O'Reilly, 2004.
64. Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 99–115, New York, NY, USA, 2004. ACM Press.

65. Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 41–57, New York, NY, USA, 2005. ACM Press.
66. Neil Ongkingco, Pavel Avgustinov, Julian Tibble, Laurie Hendren, Oege de Moor, and Ganesh Sittampalam. Adding open modules to AspectJ. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 39–50, New York, NY, USA, 2006. ACM Press.
67. Santanu Paul and Atul Prakash. Querying source code using an algebraic query language. *IEEE Transactions on Software Engineering*, 22(3):202–217, 1996.
68. Thomas W. Reps. Demand interprocedural program analysis using logic databases. In *Workshop on Programming with Logic Databases, ILPS*, pages 163–196, 1993.
69. Tobias Rho, Günter Kniesel, Malte Appeltauer, and Andreas Linder. LogicAJ. `http://roots.iai.uni-bonn.de/research/logicaj/people`, 2006.
70. Damien Sereni and Oege de Moor. Static analysis of aspects. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 30–39, 2003.
71. Volker Stolz and Eric Bodden. Temporal Assertions using AspectJ. In *RV'05 - Fifth Workshop on Runtime Verification*, volume 144(4) of *Electronical Notes in Theoretical Computer Science*, pages 109–124. Elsevier Science Publishers, 2005.
72. Wim Vanderperren, Davy Suvé, María Augustina Cibrán, and Bruno De Fraine. Stateful aspects in JAsCo. In *Software Composition: 4th International Workshop*, volume 3628 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2005.
73. Eelco Visser. Meta-programming with concrete object syntax. In *Generative programming and component engineering (GPCE)*, pages 299–315, 2002.
74. David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 127–139, New York, NY, USA, 2003. ACM Press.
75. Robert Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *ACM Sigsoft International Symposium on Foundations of Software Engineering (FSE-12)*, pages 159–169, 2004.
76. Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.
77. John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog and binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, volume 3780 of *LNCS*, pages 97–118. Springer-Verlag, November 2005.