The abc Group

Adding trace matching to AspectJ

abc Technical Report No. abc-2005-1

Chris Allan[1], Pavel Avgustinov[1], Aske Simon Christensen[2], Laurie Hendren[3], Sascha Kuzins[1], Ondřej Lhoták[3], Oege de Moor[1], Damien Sereni[1], Ganesh Sittampalam[1] and Julian Tibble[1]

[1] Programming Tools Group
University of Oxford
United Kingdom

[2] BRICS
University of Aarhus
Denmark

[3] Sable Research Group
McGill University
Montreal, Canada

March 23, 2005

a s p e c t b e n c h . o r g

Contents

1 INTRODUCTION **3**

2 TRACEMATCHES **4**

2.1 Examples 6

3 DESIGN CONSIDERATIONS **14**

3.1 Definition of Traces 14

3.2 Class of Language Used to Describe Traces 15

3.3 Matching a Pattern to a Trace 16

3.4 Binding Variables 16

3.5 Behaviour with multiple matches 17

3.6 Behaviour with multiple threads 18

4 SEMANTICS **18**

4.1 Roadmap 18

4.2 Events, Symbols and Tracematches 21

4.3 Semantics of Tracematches 21

4.4 Operational Semantics 23

4.5 Equivalence of the Semantics 24

4.6 From Semantics to Implementation 25

4.7 A Reference Implementation 26

5 IMPLEMENTATION **29**

5.1 Avoiding Space Leaks 29

5.2 Executing advice 31

6 Optimisations **31**

7 RELATED WORK **33**

8 CONCLUSIONS **35**

List of Figures

1 Grammar for a tracematch 5
2 An example tracematch 19
3 An example trace. 20
4 Matching a trace to a word. 24
5 The automaton for the *obs* tracematch. 28
6 Example use of safe iterator tracematch 32

Abstract

An aspect observes the execution of a base program; when certain actions occur, the aspect runs some extra code of its own. In the AspectJ language, the observations that an aspect can make are confined to the *current* action: it is not possible to directly observe the *history* of a computation.

Recently there have been several interesting proposals for new history-based language features, most notably by Douence *et al.* and also by Walker and Viggers. In this paper we present a new history-based language feature called *tracematches*, where the programmer can trigger the execution of extra code by specifying a regular pattern of events in a computation trace. We have fully designed and implemented tracematches as a seamless extension to AspectJ.

A key innovation in our tracematch approach is the introduction of free variables in the matching patterns. This enhancement enables a whole new class of applications where events can be matched not only by the event kind, but also by the values associated with the free variables. We provide several examples of applications enabled by this feature.

After introducing and motivating the idea of tracematches via examples, we present a detailed semantics of our language design, and we derive an implementation from that semantics. The implementation has been realised as an extension of the *abc* compiler for AspectJ.

1 INTRODUCTION

Aspect-oriented programming offers a new set of language features to increase modularity and separation of concerns. One could think of an aspect as a special kind of object that observes a base program: when certain patterns of actions happen in the base program, the aspect runs some extra code of its own. The actions that may be intercepted are called *joinpoints*, and the patterns are called *pointcuts*. The most popular implementation of these ideas is AspectJ, an extension of Java.

In AspectJ, pointcuts can only refer to the current program state, or more precisely, the current joinpoint, including an abstraction of the call stack. It is natural to explore a richer pointcut notation, which refers to the whole history of a computation, as a trace of the joinpoints encountered so far. There have been several such history-based approaches recently proposed. Walker and Viggers have introduced the idea of *tracecuts* as a history-based generalisation of pointcuts [23]. Other history-based proposals have been put forward by Douence *et al.* [7–10].[1]

Inspired by these pioneering efforts, the present paper takes an important step forward by extending the trace patterns with free variables. This innovation, which we term *tracematches*, enables a whole new class of applications, which we illustrate in Section 2. The key point is that matches can be made not just based on the kind of events, but also on the values bound to free variables. Thus, a tracematch can be used to pick out a trace of events relevant to individual objects.

Our motivating examples in themselves help to settle a number of important language design decisions, and we discuss those decisions in Section 3. A major goal of our design was to achieve a seamless integration of tracematches into the existing AspectJ language.

Although our examples provide a general feeling for our new tracematch language feature, we felt that it was important to give a rigorous definition and use this definition to lead to a correct and sound implementation. We first define a reasonably obvious declarative semantics, then we give a non-trivial operational semantics that could be used to guide a reference implementation, and finally we prove that the declarative and operational semantics are equivalent. These two semantics and the equivalence proof are given in Section 4.

We feel that proceeding in this principled fashion is an important contribution of the paper, since despite the fact that the meaning of tracematches is intuitive and crystal-clear, their implementation is quite subtle. The key problem to address is that tracematches must perform two interacting functions. First, tracematches *filter* the current trace so that they only match on symbols that are explicitly declared in a tracematch declaration. This is important because it means that the patterns don't need to be cluttered with irrelevant details and can focus on the events of interest. Second, tracematches must *consistently bind* variables across the whole match. This makes it easier to track the behaviour of individual objects in the pattern.

[1] A detailed comparison of our approach to these approaches is given in Section 7.

The declarative semantics makes these two notions of filtering and consistent binding precise, and serves to pin down exactly what behaviour we want. It is tricky to combine filtering and consistent binding in an implementation, however: intuitively, you only know what symbols to filter out once you have a binding for all the variables. In the implementation, you have to "guess" whether a symbol can be skipped and the operational semantics formalises that idea. In our experience it is very hard to get the implementation correct, and indeed, we got it wrong several times before we formally showed the equivalence of the declarative and operational semantics.

We have derived a concrete reference implementation from the operational semantics. Section 5 discusses some further implementation issues, in particular the choice of concrete representations for the main abstract data types. It is also here that we address the very important question of memory usage — a naive implementation of tracematches would suffer severe memory leaks. We also briefly discuss optimisations for tracematches in Section 6. The design has been fully implemented as an extension to the AspectBench Compiler *abc* [1] for the AspectJ language.

Finally, in Section 7 we discuss in more detail how our design differs from the works cited above, and we conclude in Section 8.

In summary, this paper presents the following original contributions:

- An important generalisation of earlier proposals for history-based approaches. Our approach, tracematches, introduces the notion of free variables in trace patterns.
- A new class of applications of history-based advice, enabled by this generalisation.
- A careful review of the design decisions for tracematches, in the light of these applications.
- A seamless integration of tracematches into the existing AspectJ language.
- A declarative semantics of tracematches, as well as an operational semantics, and a proof of their equivalence.
- A reference implementation that is derived from the operational semantics.
- A detailed discussion of implementation decisions, in particular regarding memory usage of compiled code.

In what follows, we shall assume the reader has a nodding acquaintance with AspectJ, and in fact most of our pure AspectJ code should be self-explanatory. There is a wealth of textbooks available on the subject, including [6, 13, 16, 17, 19].

2 TRACEMATCHES

Traditional aspects allow programmers to define advice – pieces of code that are run when the current program execution state (or 'joinpoint') meets some specified criteria. Tracematches extend this so that the program's entire execution history (or *trace*) can be examined to determine when the advice should run. The program's trace is modelled as a sequence of entries and exits from standard AOP joinpoints. To wit, the following aspect (in standard AspectJ) prints out a formatted version of a program trace. An *enter* event occurs before every joinpoint, and upon its completion we have an *exit* event.

```
1  aspect TraceGen {
2    before() : !within(TraceGen) {
3      System.err.println("enter: "+
4                         thisJoinPoint);
5    }
6    after() : !within(TraceGen) {
7      System.err.println("exit: "+
8                         thisJoinPoint);
9    }
10 }
```

```
⟨TRACEMATCH⟩ ::=
   [strictfp] tracematch (⟨VARIABLE DECLARATIONS⟩)
   {
     ⟨TOKEN DECLARATION⟩+
     ⟨REGEX⟩
     ⟨METHOD BODY⟩
   }

⟨TOKEN DECLARATION⟩ ::=
   sym ⟨NAME⟩ ⟨KIND⟩: ⟨POINTCUT⟩;

⟨KIND⟩ ::=
   before
 | after
 | after returning [( ⟨VARIABLE⟩ )]
 | after throwing [( ⟨VARIABLE⟩ )]
 | ⟨TYPE⟩ around [( ⟨VARIABLES⟩ )]

⟨REGEX⟩ ::=
   ⟨NAME⟩
 | ⟨REGEX⟩ ⟨REGEX⟩              AB — A followed by B
 | ⟨REGEX⟩ | ⟨REGEX⟩            A|B — A or B
 | ⟨REGEX⟩ *                    A* — 0 or more As
 | ⟨REGEX⟩ +                    A+ — 1 or more As
 | ⟨REGEX⟩ [ ⟨CONSTANT⟩ ]       A[n] — exactly n As
 | ( ⟨REGEX⟩ )                  (A) — grouping
```

Figure 1: Grammar for a tracematch

A tracematch defines a pattern and a code block to be run when the current trace matches that pattern. The grammar for a tracematch is shown in Figure 1. Each tracematch consists of three parts: the declaration of one or more symbols (events of interest), a pattern involving those symbols, and a piece of code to be executed. A *match* occurs when a suffix of the current program trace, when restricted to the symbols declared in the tracematch, is a word in the regular language specified by the pattern. Here is a very simple example of a tracematch:

```
1 tracematch () {
2   sym f before: call(* f(..));
3   sym g after: call(* g(..));
4
5   f g
6
7   { System.out.println("fg!"); }
8 }
```

Line 1 is the tracematch header, which defines any tracematch variables (none in this case). Next, on lines 2-3, we define two symbols. The symbol f matches *enter* events on joinpoints that match the pointcut $call(*\ f(..))$. Similarly, the symbol g matches *exit* events on joinpoints that match the pointcut $call(*\ g(..))$. The regular expression on line 5 specifies that advice is triggered on traces that end with a call to f and g. Finally, line 7 gives the advice body to be executed.

In matching the pattern to the trace, any events in the trace that are not declared as symbols in the tracematch are ignored, and only events declared as symbols can trigger the match. Hence, this tracematch

matches any exit from a call to g which was preceded by an enter of a call to f without any exits from calls to g in between.

2.1 Examples

We present a number of typical examples to demonstrate the practical uses of tracematches and to motivate our design. For each example, we describe a problem, give a straightforward solution in terms of tracematches, and show an equivalent solution in plain AspectJ. The various features of the tracematch extension will be explained alongside the examples as they are used.

Autosave Consider an editor of some sort. We wish to add an 'autosave' feature that ensures a copy of the file is saved to disk after every five actions. This is easy to do with a tracematch. We begin by declaring two symbols: the first one for saves — either those explicitly initiated by the user, or automatic ones (lines 2-4). The other symbol is a call to execute a command — this is what we mean by an 'action' (lines 5-6). Whenever we see five consecutive actions (as specified by the regular pattern on line 8) the autosave() method is called (line 10). Here, the syntax '[5]' means exactly 5 repetitions of the same symbol. Any constant-valued expression can go here, so the actual value could be put into a static, final field for clarity.

This example illustrates an important design decision: all events are ignored, except those that match one of the explicitly declared symbols. The save symbol is included in the alphabet, but not in the regular expression, in order to prevent the expression from matching if the actions are interrupted by a save action.

```
1  tracematch () {
2    sym save after:
3        call (* Application.save())
4      || call (* Application.autosave());
5    sym action after:
6        call (* Command.execute());
7
8    action [5]
9    {
10     Application.autosave();
11   }
12 }
```

Now consider how the same effect is achieved in pure AspectJ. We maintain a counter to keep track of the number of actions since the last save (line 2). Whenever a save event happens, the counter is reset to 0 (lines 4-7). Furthermore, upon the completion of each action, we increase the count by 1: if the total reaches 5, the autosave() method is called (lines 10-12).

```
1  aspect Autosave {
2    private int actions_since_last_save = 0;
3
4    after(): call (* Application.save())
5           || call (* Application.autosave()) {
6      actions_since_last_save = 0;
7    }
8
9    after(): call (* Command.execute()) {
10     actions_since_last_save ++;
11     if (actions_since_last_save == 5)
12       Application.autosave();
13   }
14 }
```

Note how the state of the matching process (the counter) is explicit in the AspectJ solution. In this example, that leads to only minor complications, but as we shall see below, often the burden of such state maintenance is much greater.

Previous program state can be exposed to tracematches by capturing variables in the symbol pointcuts. These variables are defined in the tracematch header (similarly to the definition of pointcut variables in ordinary pointcuts) and bound by the normal pointcut variable binding constructs in the symbol pointcuts. Unlike ordinary pointcuts, symbol pointcuts in tracematches do not define variables of their own. A tracematch variable is visible in the advice body and in the symbol pointcut by which it is bound. All tracematch variables that are used by the advice body must be bound by at least one symbol in any symbol string matching the pattern. This ensures that these variables have always been given a value whenever the body is executed.

Whenever the same tracematch variable is bound more than once in a trace (by the same symbol or by different symbols) the variable is not rebound to the new value. Rather, it is checked that the old and the new value are equal (in the sense of ==). If this is not the case, the new symbol is ignored for this particular trace. In other words, a program trace is defined to match the regular expression when there exists some set of values that can be consistently substituted for the pointcut variables in such a way that the program trace matches the defined expression. When more than one set of variable bindings exists that cause the expression to match, the code block is executed multiple times, once for every possible match. This allows the tracematch to match patterns in the behaviour of individual objects, for example to enforce conditions on the order in which the object's methods are called (by reporting runtime errors when the conditions are violated).

Contextual logging Our next example is intended to illustrate the use of variable binding in tracematches. The application is to log the actions of the users of a database: whenever a user has logged in, we want to report the queries of that user. For simplicity, we consider a system where only one user is logged in at any time.

Variables that are to be bound in the pattern of a tracematch are declared in its header (line 1). Here there are two such variables, namely the user u and a query q. The first symbol we declare is the one that binds u, via a call to the login(..) method (lines 2-4). We also track logout actions, so that we stop logging when the user has finished (lines 5-6). Finally, we declare a symbol for query events (lines 7-9), and intercept the value of the query in variable q. The pattern is then very simple: we just look for queries that follow a login event (line 11). Whenever this matches a suffix of the current trace, we print an appropriate logging message that reports both the user u and the query q (lines 13-14).

```
1  tracematch (User u, Query q) {
2    sym login after returning:
3        call (* LoginManager.login(User,..))
4     && args(u,..);
5    sym logout after:
6        call(* LoginManager.logout());
7    sym query before:
8        call(* Database.query(Query))
9     && args(q);
10
11   login query+
12   {
13     System.out.println(u +
14         " made query " + q);
15   }
16 }
```

Note that it does not make sense to replace + in the above pattern (line 11) by $*$, for that would imply that the q parameter might not have been bound. It is a requirement (checked statically by the compiler) that any variable that is used inside the advice body must be bound by some symbol in all possible traces matched by the regular expression.

Now consider how the same functionality is encoded in pure AspectJ. We need a boolean variable to keep track of whether a user has been logged in, and another variable to record the user (lines 2-3). (At a pinch, the two might be combined, as the user field is *null* precisely when the boolean field is $false$ — but we

find that less transparent.) Whenever the login() call succeeds, we set the boolean to true, and update the user field as well (lines 5-11). Corresponding updates are made upon a logout (lines 13-17). When a query happens, and a user is logged in, the logging message is output (lines 19-26).

```
1  aspect Logging {
2    private boolean loggedIn = false;
3    private User user;
4
5    after(User u) returning:
6        call (* LoginManager.
7              login(User, Password))
8     && args(u,..) {
9      loggedIn = true;
10     user = u;
11   }
12
13   after():
14       call (* LoginManager.logout()) {
15     loggedIn = false;
16     user = null;
17   }
18
19   before(Query q):
20       call (* Database.query(Query))
21    && args(q) {
22     if (loggedIn) {
23       System.out.println
24           (user + " made query " + q);
25     }
26   }
27 }
```

Apart from being more verbose, this AspectJ solution is also less flexible than the one based on a tracematch. Suppose we wanted to extend this logging aspect to log the actions of multiple users who can be logged in at the same time. For the tracematch version, all we would have to do would be to extend the login, logout and query symbols to capture some unique information such as a session id (assuming this is made explicit in the base program). This would then tie the login and query events together by their shared session id, allowing the user from the login symbol to be available to the advice body. To add the same functionality to the AspectJ version, the variables would have to be replaced by mappings and the code changed accordingly.

Observer This example demonstrates a way of implementing the well-known *Observer* design pattern. Here we have a set of *Subject* objects, which represent the state of some entity being modelled, and a set of *Observer* objects which are attached to a particular Subject and need to be notified in any changes to their Subject's state. The solutions below both provide a way of doing this without the Subject needing to be aware that it is being observed.

The tracematch based solution declares two events of interest, in lines 2-7: the creation of an observer o (where the subject s is passed as an argument), and updates to that same subject s. We then specify the sequence of events that will cause the view of the observer to be updated, on line 9. In words, we perform an observer update upon creation of the observer, and subsequently upon each update of the subject that follows the creation of the observer. Note how this example illustrates our use of variable bindings in patterns: it is the observer creation that binds variables s and o. The update symbol then only matches execution events with those same variable bindings.

```
1 tracematch (Subject s, Observer o) {
2   sym create_observer after returning(o):
3       call(Observer.new(..))
4    && args(s);
```

```
5     sym update_subject after:
6         call(* Subject.update(..))
7      && target(s);
8
9     create_observer update_subject*
10    {
11      o.update_view();
12    }
13  }
```

A similar AspectJ solution is shown below. It needs to maintain a vector of observers for each subject. The association of the vector to each subject is achieved via a so-called *intertype declaration* on lines 2-3. This inserts a new field called *observers* into the Subject class (or, if it is an interface, into each implementor of that interface). When a new observer is constructed, we add it to the observers of subject s (line 8), and we also update its view (line 9). Then, whenever the subject is updated, we update each of its observers (lines 15-20). A comparison of the AspectJ solution with our formulation in terms of tracematches highlights an important point, namely that in a tracematch, the advice is executed for *all* matching bindings. The iteration that is explicit in the AspectJ solution is implicit when using tracematches.

```
1  aspect Observer {
2    private Vector Subject.observers
3      = new Vector();
4
5    after(Subject s) returning(Observer o):
6        call(Observer.new(..))
7     && args(s) {
8      s.observers.add(o);
9      o.update_view();
10   }
11
12   after(Subject s):
13       call(* Subject.update(..))
14    && target(s) {
15     Iterator obsit = s.observers.iterator();
16     while(obsit.hasNext()) {
17       Observer o
18           = (Observer) obsit.next();
19       o.update_view();
20     }
21   }
22 }
```

For brevity, we have chosen a minimal implementation of the observer pattern, but the use of tracematches also simplifies the more advanced formulation in [15]. In that seminal paper, Hannemann and Kiczales demonstrate convincingly that many design patterns are more easily expressed in AspectJ than in Java. Here we present a further significant improvement over that work.

In addition to **before** and **after**, symbols can also be declared as **around**. An **around** symbol matches *enter* events of the corresponding joinpoints, just as **before** does. However, the advice body gets executed *instead* of the original joinpoint, rather than just before it.

There are certain restrictions on how **around** symbols can be used. We require that either *all* the events that could be the last in the matched sequence are of **around** type, or none are. This restriction is necessary as advice bodies for **around** advice are incompatible with the ones for **before** and **after**, for two reasons: first, around advice must return a value of the return type declared for the advice (declared just before the **around** keyword); second, similarly to ordinary around advice, around advice in tracematches can call the special method **proceed**() to invoke the original join point. We will say more about **proceed**() in Section 3.5. Additionally, we require that **around** symbols can only appear at the end of matched sequences, as there is

no sensible meaning for **around** if there is no advice to be executed at the matched joinpoint.

Flyweight We now consider the *Flyweight* design pattern. The purpose of this pattern is to avoid a huge number of small objects being created. To achieve that, a pool of instances is maintained; where possible, each constructor call is intercepted and instead an object from the pool is returned. For simplicity we stipulate that objects of the *FlyWeight* type that were created with the same argument to the constructor are considered equivalent. An implementation of the flyweight pattern thus requires that we cache the result of constructor calls, only creating one object for each different argument value.

The flyweight pattern has a natural description in terms of a tracematch. We look for the first object creation with a given argument; and after that, any constructor call with the same argument is intercepted. We therefore declare two symbols (lines 2-7). Note that the latter symbol is an instance of **around**. The advice body does not call **proceed**(), which means that the original constructor call is not performed. Instead, it just returns the value returned the last time the constructor was called with the same argument. (line 11).

```
1  tracematch (Flyweight fw, Object arg) {
2    sym return_an_obj after returning(fw):
3        call(FlyWeight+.new(Object))
4     && args(arg);
5    sym create_another_obj FlyWeight around:
6        call(FlyWeight+.new(Object))
7     && args(arg);
8
9    return_an_obj create_another_obj
10   {
11     return fw;
12   }
13 }
```

We now consider an encoding of the flyweight pattern in pure AspectJ, as displayed below. Again this is a minor simplification of the code of Hannemann and Kiczales [15]. It explicitly maintains a table of those objects that have previously been used as arguments to a flyweight constructor, and the associated object that was returned (lines 2-3). An IdentityHashMap is used for this to mimic the object identity behaviour of the tracematch version. Then, upon each constructor call (lines 6-7), we check whether a table entry exists for the given argument (line 9). If so, the corresponding object is returned (lines 10-11). Otherwise, a new object is created, and stored in the table before returning (lines 14-16).

```
1  aspect FlyWeightAspect {
2    private Map constructedObjects
3       = new IdentityHashMap();
4
5    FlyWeight around(Object arg):
6        call(FlyWeight+.new(Object))
7     && args(arg)
8    {
9      if (constructedObjects.containsKey(arg))
10       return (FlyWeight)
11           constructedObjects.get(arg);
12     else
13     {
14       FlyWeight fw = proceed(arg);
15       constructedObjects.put(arg,fw);
16       return fw;
17     }
18   }
19 }
```

The code using a tracematch is marginally shorter, but in our view that is not its main advantage. The true merit of the tracematch is that it directly states the programmer's intention, crisp and clear, without a need to encode the essential idea.

Safe iterators Our next example concerns the safe use of iterators. It is usually the case that the data source that underlies an iterator may not be changed during the iteration process. It is fairly common to explicitly encode that behaviour in the implementation of iterators, by throwing an exception if an iterator is used after the collection has changed, but it would be nicer to specify it as a separate concern, once and for all.

There are three symbols of interest here: the creation of an iterator on a particular data source (lines 2-4), the next() operation on that same iterator (lines 5-7), and update operations on the given datasource (lines 8-10). Then, whenever we see a creation, followed by some iteration steps, an update and then another iteration step, we know that an error has occurred. This is captured by the pattern on lines 12-13.

```
1  tracematch (Iterator i, DataSource ds) {
2    sym create_iter after returning(i):
3        call(Iterator DataSource.iterator())
4     && target(ds);
5    sym call_next before:
6        call(Object Iterator.next())
7     && target(i);
8    sym update_source after:
9        call(* DataSource.update(..))
10    && target(ds);
11
12   create_iter call_next*
13       update_source call_next
14   {
15      throw new
16          ConcurrentModificationException();
17   }
18 }
```

In the AspectJ version, we keep track of the state of the DataSource explicitly, in a map from the DataSource to some unique object (lines 3-4). We then reallocate this object each time the state changes (line 30). For each Iterator object, we remember its associated DataSource (lines 5-6 and 13) and the state the DataSource was in upon the creation of the iterator (lines 7-8 and 14). If the DataSource has changed state since the iterator was created, the next() operation fails (lines 21-24).

```
1  aspect SafeIterators
2  {
3    private Map ds_state
4        = new IdentityHashMap();
5    private Map it_ds
6        = new IdentityHashMap();
7    private Map it_ds_state
8        = new IdentityHashMap();
9
10   after(DataSource ds) returning(Iterator i):
11       call(Iterator DataSource.iterator())
12    && target(ds) {
13      it_ds.put(i, ds);
14      it_ds_state.put(i, ds_state.get(ds));
15   }
16
17   before(Iterator i):
18       call(Object Iterator.next())
```

```
19     && target(i)
20    {
21      if (ds_state.get(it_ds.get(i))
22          != it_ds_state.get(i))
23        throw new
24            ConcurrentModificationException();
25    }
26
27    after(DataSource ds):
28        call(* DataSource.update(..))
29     && target(ds) {
30      ds_state.put(ds, new Object());
31    }
32  }
```

Again, the intent is clearly visible in the tracematch solution, whereas the pure AspectJ solution is formulated in terms of how the constraint is actually implemented. Furthermore, this AspectJ implementation will cause severe memory leaks. Any DataSource and Iterator ever used will end up in the maps and not be garbage collected. For this example, this could be easily fixed by using *weak references* to ensure that mappings are removed from the maps when their keys are no longer in use.[2] A naïve tracematch implementation would of course suffer from the same problems, but in this case the compiler has the opportunity to analyze the specification and use weak maps whereever applicable. We will return to the issue of weak references in Section 5.

The AspectJ solution could be expressed a bit more simply (and without the memory leak problem) by using intertype declarations on the DataSource and Iterator classes. However, in order to inject intertype declarations, the AspectJ compiler must have access to modify these classes, either at compile time or by using a weaving class loader. Such access is typically not available for the Java standard library classes, so this proposed solution would not work with *e.g.* the standard collection classes. The tracematch implementation does not require weaving access to classes bound to tracematch variables, so in order to achieve as close to the same behaviour for the two solutions as possible, the hash map version was chosen.

Connection management In our final example, we use an aspect to control the opening and closing of some form of 'connection', for example to a database system. For the sake of the example, we assume that a Connection class has three methods, open(), query() and close(). The query() method should only be called on a Connection that is in the open state. We assume that the open() and close() methods take some time to execute, so should not be called unnecessarily, but also that open connections require some overhead, so connections should not be left open and unused for large periods of time.

The aspects below allow users of the Connection class to ignore the open() and close() methods, and just assume that they will be opened and closed when needed. To achieve this, a closed connection is opened immediately before a query is called on it, and an open connection that has not been used 'recently' is closed. We define a connection not having been used recently to mean "there have been 5 calls to some logging API since its last use", which in many systems would provide an acceptable heuristic.

The desired effect is achieved with two tracematches. The first is shown below. It declares symbols for opening, closing, querying, and creating a new connection (lines 2-12). The connection must be opened when we see the first query after a creation, or when we see that a query is performed immediately after a close. This is captured with the pattern on line 14, and it illustrates that all declared symbols must be matched: because *open_con* is one of the symbols, the pattern rules out a situation where the connection is open already.

```
1  tracematch (Connection c) {
2    sym open_con after:
3        call (* Connection.open())
```

[2] No WeakIdentityHashMap exists in the Java Standard Library, but such a class could of course be written specifically for this purpose.

```
4     && target(c);
5    sym close_con after:
6        call (* Connection.close())
7     && target(c);
8    sym query before:
9        call (* Connection.query(..))
10    && target(c);
11   sym create after returning(c):
12       call (Connection.new());
13
14   (create query)|(close_con query)
15   {
16     c.open();
17   }
18 }
```

The next step is to define a tracematch that closes a connection when it has been open too long. As said, our heuristic rule defining 'too long' is that there have been 5 logging calls since the last query. Declaring an explicit symbol for closing the connection (lines 2-4) guarantees that the connection has not been closed after the matching query event.

```
1  tracematch(Connection c) {
2    sym close_con after:
3        call (* Connection.close())
4     && target(c);
5    sym query before:
6        call (* Connection.query(..))
7     && target(c);
8    sym log before():
9        call (* Log.add(..));
10
11   query log [5]
12   {
13     c.close();
14   }
15 }
```

This example hints at the need for a language mechanism to name symbols outside a particular tracematch, to allow the same symbol to be used in multiple tracematches. However, often the amount of repetition can be minimised by naming the relevant pointcut, and therefore we have decided (at least for the moment) against such a mechanism.

Let us now consider a similar solution in plain AspectJ (a much fuller discussion of this type of application can be found in Laddad's textbook [17]). To track the number of logs since the last query on each open connection, we have a map from Connections to Integers (lines 2-3). An invariant of the code is that any connection that is a key in this map is open, and all other connections are closed.

When a connection is opened, we record it in the age map and set its age to 0 (line 8). When a connection is closed, we remove it from the map (line 14). When a query is intercepted, we must open the connection if it is currently closed, and its age is then reset to zero (lines 20-22). Finally, whenever a log call happens, we iterate over the set of all connections (lines 26-39). For each connection, we increase its age, and if this pushes the age of a connection to 5, the connection is closed and removed from the map.

```
1  aspect AJConnectionManagement {
2    private Map connection_age
3        = new IdentityHashMap();
4
5    after(Connection c):
6        call (* Connection.open())
7     && target(c) {
```

```
8       connection_age.put(c, new Integer(0));
9     }
10
11    after(Connection c):
12        call (* Connection.close())
13     && target(c) {
14      connection_age.remove(c);
15    }
16
17    before(Connection c):
18        call (* Connection.query(..))
19     && target(c) {
20      if (!connection_age.containsKey(c))
21        c.open();
22      connection_age.put(c, new Integer(0));
23    }
24
25    before(): call (* Log.add(..)) {
26      Iterator it
27        = connection_age.entrySet().iterator();
28      while(it.hasNext()) {
29        Map.Entry e = (Map.Entry)it.next();
30        Connection c = (Connection)e.getKey();
31        int age = ((Integer)e.getValue())
32            .intValue();
33        age++;
34        e.setValue(new Integer(age));
35        if(age == 5) {
36          c.close();
37          it.remove();
38        }
39      }
40    }
41  }
```

It is interesting to contrast this code with our earlier formulation in terms of tracematches. There, the statement of the intended behaviour is purely declarative, and we do not need to create an explicit iteration. Instead, the iteration happens automatically, for each binding that results from matching the regular pattern to suffixes of the current trace. This is similar to the use of iteration in our earlier discussion of the observer pattern.

3 DESIGN CONSIDERATIONS

We now review the crucial design choices for tracematches. In particular, we contrast our decisions with alternatives, focussing on those cases where others have made a different choice. In doing so, it is our aim to give a rational account of our design, deferring a detailed comparison with related work till Section 7. As we shall demonstrate, all decisions were informed both by the examples in the preceding section, as well as the desire to have a clean semantics that admits an efficient implementation.

3.1 Definition of Traces

A trace is a sequence of events in the execution of a program. Our events are defined as entries and exits from joinpoints. In the tracematch declaration itself, we attach the standard AspectJ advice kinds **before**, **after** and **around** to symbol declarations. Analogously to ordinary advice, **around** is treated similarly to **before** for the purposes of matching but then executes *in place of* the matched joinpoint rather than before

it.

An alternative way to define a trace might be to use joinpoints as events directly, which is similar to the way AspectJ defines the **cflow** pointcut. This means thinking of joinpoints as nodes in a program execution tree (a generalisation of the dynamic call graph with nodes for all joinpoints, not just calls), and define the trace as the sequence of joinpoints that have been visited so far. However, joinpoints are not atomic events – in particular they can be nested inside each other. It follows that the ordering of events in the resulting trace depends on the definition of 'visited' that is used. Whether a parent or child node is visited first depends on whether the trace is defined in terms of a preorder or postorder traversal.

Our definition is more flexible than this alternative approach. The above scenario gives the trace 'before parent; before child; after child; after parent'. By writing appropriate patterns, the programmer can achieve the same effect as either a preorder or postorder traversal would have done.

3.2 Class of Language Used to Describe Traces

Our trace patterns are described as regular expressions. The motivation for doing so is that regular expressions provide a concise, easily understood notation. Indeed, in typical use cases of tracematches, regular expressions offer just the right level of expressiveness. Furthermore, regular expressions lend themselves to static analyses: for richer formalisms, the question of language inclusion is typically undecidable, for example.

The only obvious alternative is to consider context-free language patterns instead. All examples in the literature that motivate such a generalisation involve dependencies on balanced method calls and returns. However, in these cases, the call stack dependencies can often be described using **cflow** pointcuts. These pointcuts allow the programmer to assert that program execution is below one of a given set of joinpoints in the execution stack. Together with **cflow** pointcuts regular trace patterns achieve a high degree of expressiveness.

There exist examples, however, where the use of **cflow** is not enough. In [23], Walker and Viggers describe a program with mutually recursive methods *safe* and *unsafe*, and present the challenge of identifying traces in which certain method calls happen when a call to *unsafe* encloses the call more closely than a call to *safe*. The use of **cflow** pointcuts within tracematch symbols is not sufficient to express this behaviour.

It is our view, however, that the desired behaviour is easily achieved via a simpler language feature, in conjunction with our design for tracematches. The hypothetical **cflowdepth**(pc, n) pointcut calculates the number of joinpoints in the current execution stack that match the pointcut *pc*. This would allow the above example to be expressed as follows:

```
1  pointcut safe(): execution(* *.safe());
2  pointcut unsafe(): execution(* *.unsafe());
3  tracematch(int i){
4  sym enterUnsafe before(int i):
5        unsafe()
6    && cflowdepth(safe||unsafe, i);
7  sym exitUnsafe after(int i):
8        unsafe
9    && cflowdepth(safe||unsafe, i);
10 sym callX before():
11       call(* *.x())
12   && cflowdepth(safe||unsafe, i);
13 enterUnsafe callX
14 { /* do something */ } }
```

Because **cflowdepth** is clearly useful in its own right, we do not present it as an inherent part of our tracematch design. In combination with the regular patterns of our tracematches, however, it obviates the need for context-free patterns.

3.3 Matching a Pattern to a Trace

An important design decision concerns the filtering of traces to the events of interest. We have decided to explicitly declare all "interesting" symbols, and restrict the trace to events that match one of these declared symbols. The pattern is then matched against this restricted trace. This decision avoids cluttering the pattern with spurious symbols for events that are irrelevant to the problem in hand. One subtle point is that we never discard the last event of a trace: this last event must match a declared symbol. This is to ensure that advice is only executed at the point a match occurs, and not at each ignored symbol thereafter.

One could consider defining the set of captured events implicitly as all events matched by symbols that occur in the regular expression (as opposed to all symbols defined in the tracematch). However, with this definition, it would not be possible to explicitly exclude certain events from the trace. If an event was included in the regular expression then by definition it could appear in some matched trace, and if it didn't appear then it would be completely ignored. As shown in the examples, being able to exclude events is highly useful.

3.4 Binding Variables

The most prominent feature of our tracematch design is the handling of variable bindings in the symbol pointcuts: multiple occurrences of the same variable in the pattern must be bound to a single value that is consistent across all the occurrences.

To see the rationale for this fundamental decision, note that variable bindings in tracematches serve two important purposes:

- To give code in the advice body access to context values at the joinpoints matched by the tracematch symbols. This is similar to variable binding in ordinary advice.
- To allow the tracematch to match traces in the behaviour of individual objects or groups of objects, rather than just control-flow traces. This mechanism is vaguely related to **per**-clauses for ordinary aspects (in the sense that these too associate pointcuts with individual objects) but serves a quite different purpose, as it is binding together traces of events rather than merely selecting an aspect instance.

As long as a variable is only bound once in a trace, it is simply bound to the corresponding value. When a variable is bound more than once in the same trace (whether by the same symbol or by different symbols), there are a number of options for what the behaviour could be:

- Re-bind the variable, so that the value seen by the advice is the one bound most recently in the trace. This is similar to what is done for **cflow** pointcuts, where the values bound by the most closely enclosing joinpoint are the ones seen by the advice.
- Check for equality with the previous binding. The pointcut is extended with an implicit condition that the values bound must be the same as was previously bound to the same variables. If the value is different, the pointcut does not match, so the trace is rejected. In this design, the first value that is bound to a variable in a given trace is the only possible value for that variable.
- Allow multiple sets of bindings for any given trace. For any given set of bindings, events that cause symbols to bind with different values are ignored in the same way as events that are not matched by any declared symbols.

The last option here is the only one that fulfils the second purpose above. By viewing the trace as a set of parallel, object-specific traces, the behaviour of individual objects can be easily captured by the tracematch. As witnessed by the examples in Section 2, this is highly useful.

This mechanism can in most cases simulate the other two options mentioned above. To only capture the last binding of a variable, rewrite the regular expression so that only the last binding is part of the match.

To check equality between bindings, bind the values to different variables and check their equality as an extra condition in the advice body.

In defining the equality of values above, we have used primitive or reference equality (for primitive and reference types respectively), corresponding to Java's == operator. One could consider whether it would be more appropriate to define equality for reference types by the *equals*() method instead. The distinction here is between tracking an object and tracking a value. However, "tracking a value" does not really make sense, since the fact that two objects are equal according the the *equals*() method does not in any way imply that these objects are related in the data flow of the program. As the examples clearly show, the tracking of particular object instances is very useful in capturing properties of the program data flow. Again, equality between variables based on *equals*() can be checked by extra code in the advice body.

3.5 Behaviour with multiple matches

The same pattern can match a single execution trace in multiple ways, producing different variable bindings for each match. Our decision is to execute the advice once for each of the variable bindings. A typical example where this feature proved crucial is that of the Observer pattern, where multiple Observers are notified upon a change in the Subject.

This decision was motivated by our desire to mimick the behaviour of multiple pieces of ordinary advice that apply to the same joinpoint. From this perspective, it is natural that the advice body is executed multiple times, once with each different set of bindings for the tracematch variables. It is possible that a trace can match the regular expression in two different ways, but result in the same values being bound. When this happens the advice is still executed just once for those particular values. It is the bound values that distinguish the traces.

It is not obvious how to define the order in which the advice for the different bindings are executed, as this involves the ordering of sets of values. One could consider using the structure of the way the trace was matched or the order in which the values were bound to define an ordering, but this would not give a unique ordering, since different values can be bound by identical traces, and the same values could be bound in several possible ways. This means that any given ordering is not particularly intuitive, since it would be based on some underlying mechanism which is not visible to the programmer. Ordering based solely on the actual values bound to the variables is not possible in general, since some values might not have a natural ordering. At least until more work is done on exploring implementations and applications, we have chosen not to define any particular ordering on the execution of tracematch advice.

Matters get more complicated when the final symbol is an **around** symbol. Similarly to ordinary **around** advice, **around** advice in tracematches can call the special method **proceed**() to invoke the original joinpoint. If more than one trace matched at the same joinpoint, **proceed**() invokes the next match, and only during execution of the final match does **proceed**() invoke the original joinpoint. It is permissible for the advice body not to call **proceed**() at all, in which case the original joinpoint is skipped and no more matches are executed. Similarly, if **proceed**() is called multiple times, the original joinpoint (or the following match) executes multiple times.

Normally, **proceed**() takes no arguments. A sometimes confusing feature of AspectJ is that the **proceed**() call can be given arguments which are used to replace the original value of variables bound by the pointcut. We provide an analogous feature in tracematches by the following mechanism. An **around** symbol can optionally declare a signature for **proceed**() by giving a list of tracematch variables after the **around** keyword (similarly to ordinary **around** advice but without the types). The actual values passed to **proceed**() then replace the values at the original joinpoint to which the corresponding variables were bound. In the case of multiple matches, the following match sees the passed values as new values for the corresponding tracematch variables.

If any final symbols declare such a signature, all final symbols must give the same list of variables, all these variables must be bound by all of the **around** symbol pointcuts, and they must be bound by the same binding constructs. This ensures that these variables have the same values for all possible matches at a given joinpoint. Thus, if the advice always passes the parameters directly to **proceed**(), the behaviour is the same as if no variables were specified.

3.6 Behaviour with multiple threads

So far, we have implicitly assumed that all programs to which tracematches are applied are single-threaded. We need to define how tracematches behave in the presence of multiple threads. There are at least two sensible behaviours that could be defined in this case: the first is to treat each thread like a separate program, and match the traces of each thread individually. The second is to create a single trace of the entire program, by interleaving the events of each thread. Our decision is to allow either behaviour, leaving the choice to the programmer.

The first possible behaviour, matching thread traces independently, is useful where a tracematch needs to detect patterns in control flow. Examples of this are to detect control flow patterns that are known to lead to error conditions, or to enforce rules such as "a thread should not use an object of type X until it first acquires a lock on it".

The second possible behaviour, interleaving thread traces, can be used when a tracematch needs to detect a pattern of events with reference to a particular object, for example to enforce typestate restrictions. Examples include "all Connection objects must be in an open state before being used" or "only one thread may have a lock on an object at a time".

Because both of these behaviours have important uses, we allow the programmer to select the desired behaviour. The default is to interleave events across threads, but we introduce a modifer, **perthread**, which can be added to a tracematch declaration to declare that the traces of each thread should be matched independently.

Both of these kinds of tracematches have their own sources of extra overhead compared to the single-threaded version. For the thread-local version, we have to keep track of the tracematch matching state per thread, using thread locals or hash maps mapping from the current thread, similar to what needs to be done in the implementation of **cflow** pointcuts. For global tracematches, we need to make matching and tracematch state tracking code **synchronized** in order to ensure that atomic events are properly interleaved.

4 SEMANTICS

We now pin down the meaning of tracematches so that it is possible to give a high-level description of their implementation. We first define the semantics in a declarative manner, and then refine this into a more operational semantics, geared towards defining a reference implementation.

4.1 Roadmap

Before diving into the formalities, we first give a brief roadmap, motivating our formal decisions later on.

Declarative semantics For a tracematch without variables, we match every suffix of the current trace against the pattern. In doing so, the trace is *filtered*, by ignoring all events that do not correspond to any of the declared symbols. The last event in the trace should, however, always correspond to a declared symbol: this is just the requirement that advice is executed immediately when a match occurs. These three ideas (suffixes, filtering, and last event declared) are the three key features of the declarative semantics.

To give a declarative meaning to a tracematch that has free variables, we read it as a template for all possible instantiations, where each of the variables has been replaced by a specific runtime value (there may be an infinite number of such instantiations). Each of these instantiations is a tracematch *without* variables, and we have already given a meaning to those.

To illustrate these points, consider the tracematch in Figure 2. and the sequence of calls

```
v.f(); v.h(); w.g(); w.f(); v.g();
```

To keep the example short and manageable, we assume that each of f, g and h has void return type and an empty body. A full trace of the above call sequence is shown in Figure 3.

```
tracematch (X x)
{
  sym f before :
        call(* f(..))
        && target(x) ;
  sym g after :
        call(* g(..))
        && target(x);

  f g

  { System.out.println("fg!"); }
}
```

Figure 2: An example tracematch

As described above, the above tracematch should be seen as a template for all possible instantiations assigning values to x. In this case it is clear that the only relevant values for x are $x = v$ and $x = w$. Consider first the instantiation $x = v$. Then the events relating to $v.h()$, $w.g()$ and $w.f()$ are all filtered out, and the resulting trace is just $[e1, e20]$ (labels given in Figure 3). Now $e1$ matches the symbol f with $x = v$, while $e20$ matches the symbol g. Hence this matches the pattern $f\ g$, and the tracematch applies with binding $x = v$.

Now consider the instantiation $x = w$. Then the only events in the trace that match one of the symbols f and g with $x = w$ are $e12$ and $e13$. Hence the filtered trace is $[e12, e13]$. This trace does not match the pattern $f\ g$, and so the tracematch does not match with binding $x = w$. In fact, this also fails to match for another reason: the *last* event $e20$ of the trace is filtered out, while a tracematch only matches if the last event of the trace matches a declared symbol.

Now suppose that we added one more symbol to the declarations in the above tracematch, namely

```
  sym g2  after :
        call(* g(..));
```

We leave the pattern unchanged, however. Consider the (only possible) binding $x = v$, as in the above example. The filtered trace from Figure 3 includes the same events as before ($e1$ and $e20$), but this time the *exit* event from $w.g()$ (event $e12$) also matches $g2$. The filtered trace is therefore $[e1, e12, e20]$. This no longer matches the pattern $f\ g$, and now the tracematch does not match the trace, solely because we introduced a new declared symbol and therefore reduced the amount of filtering. The reader may wish to check for him/herself that the new tracematch would match the event sequence generated by

v.f(); v.h(); w.f(); v.g();

The declarative semantics is formally defined in Section 4.2 (the definition of events, symbols and tracematches) and Section 4.3 (the definition of matching).

Operational semantics The definition of tracematches with free variables via all possible instantiations is attractive, because it is simple and it gives us an effective way of reasoning about tracematches. It does not give any guidance on their implementation, however.

Without variables, it is not difficult to see how an implementation might go. Alongside the base program, we run a finite automaton. This finite automaton recognises precisely the language of the regular expression, interspersed with events that do not match any of the declared symbols. Furthermore, construct the automaton to match t if some suffix of t matches the given pattern. Finally, we stipulate that only transitions labelled with a declared symbol can enter a final state. This way the automaton captures all three of the important elements of matching in the declarative definition (filtering, suffixes and last event declared).

e1	enter: call(void FG.f()) on v
e2	enter: execution(void FG.f()) on v
e3	exit: execution(void FG.f()) on v
e4	exit: call(void FG.f()) on v
e5	enter: call(void FG.h()) on v
e6	enter: execution(void FG.h()) on v
e7	exit: execution(void FG.h()) on v
e8	exit: call(void FG.h()) on v
e9	enter: call(void FG.g()) on v
e10	enter: execution(void FG.g()) on w
e11	exit: execution(void FG.g()) on w
e12	exit: call(void FG.g()) on w
e13	enter: call(void FG.f()) on w
e14	enter: execution(void FG.f()) on w
e15	exit: execution(void FG.f()) on w
e16	exit: call(void FG.f()) on w
e17	enter: call(void FG.g()) on v
e18	enter: execution(void FG.g()) on v
e19	exit: execution(void FG.g()) on v
e20	exit: call(void FG.g()) on v

Figure 3: An example trace.

While the base program is running, we keep a flag on each state of the automaton, to track whether the current trace moves the automaton into that state. Note that as the transitions are labelled by symbols, and an event can be matched by more than one symbol, the automaton can be in multiple states simultaneously (a new event causes the automaton to take *all* matching transitions).

To start with, the flags are set on the initial states of the automaton. Because every trace can be a prefix of an accepted trace, the flags on the initial states remain set to true at all times. Now when a new event e happens in the base program, we match it against each of the symbols, and make the corresponding changes to the flags: if there is a transition from s' to s labelled with symbol a, if the flag on s' is set to true, and if a match the new event e, the flag on s is set to true. If no such transition to s exists, the flag on s is set to false. When a final state becomes reachable, advice is executed.

Now how can this be modified to take free variables into account? We use the same automaton construction, but instead of boolean flags to indicate reachability, we use *constraints*. A constraint label on state s records any assumptions made in reaching s with the current trace. One may think of a constraint as a logical formula that combines assignments of values to variables ($x = 1$), as well as the negations of such expressions ($y \neq 1$). In the same way we updated the boolean flags on states, so one can also update the constraint labels. New equations of the form $x = value$ are generated by AspectJ's pointcut matching.

To capture filtering of declared symbols on account of wrong variable bindings (for instance, filtering out $e12$ in the first example above), however, it is not enough to match only on declared symbols. We introduce a new symbol **skip** to capture events that are ignored in the matching (either because they match no declared symbol, or because of wrong variable bindings). The **skip** symbol matches exactly under the conditions that cause all declared symbols to fail to match. In particular, if an event is not matched by any declared symbol, then it is matched by **skip**. Also, if there exists one declared symbol that matches with variable binding $x = value$, then **skip** matches with binding $x \neq value$. This is the way negative bindings are entered into constraints.

There are thus two important ideas in the operational semantics: the use of *constraints* and the anything-but-a-declared-symbol **skip**. Together they allow us to do the filtering of events incrementally, without knowing the variable bindings in advance.

In Section 4.4, we make the above intuition precise, and we give a formal definition of **skip**. Then, in

Section 4.5, a formal proof is presented that the declarative and operational semantics coincide. To avoid cluttering that proof, we shall already introduce constraints while discussing the declarative semantics in Sections 4.2 and 4.3. In Section 4.6, we spell out the incremental computation of the constraints that label the automaton states. Finally, in Section 4.7, all this is made concrete, by generating AspectJ code that directly implements the operational semantics.

4.2 Events, Symbols and Tracematches

Events and Traces An *event* occurs when a joinpoint is either entered or left. Accordingly, we define:

$$\mathit{event} = \{\mathbf{enter}, \mathbf{exit}\} \times \mathit{joinpoint}$$

A *trace* is then simply a finite sequence of events. An example trace is shown in Figure 3.

Constraints We shall model variable bindings as *constraints*, that is equations combined with the usual logical connectives. In particular a constraint may be an equation between a variable and a runtime value, $x = v$, or an inequation, $\neg(x = v)$. We write C for the set of all constraints. For a given tracematch, the relevant variables are those that are declared in its header.

Symbols The symbols defined in a tracematch are just AspectJ pointcuts. However, it will be convenient to abstract away from the precise details of matching AspectJ pointcuts to joinpoints. We will model symbols as functions from events to constraints:

$$\mathit{symbol} = \mathit{event} \rightarrow \mathit{constraint}$$

For a symbol a (a pointcut) and an event e, the constraint $a(e)$ defines the assignments of values to the variables of a obtained when matching a to e. If the pointcut does not match, then $a(e) = \mathit{false}$.

For example, in the example tracematch shown in Figure 2 with the trace shown in Figure 3, we have $f(e1) = (x = v)$ and $g(e12) = (x = w)$, while $g(e1) = \mathit{false}$ and $f(e20) = \mathit{false}$ (the symbols f and g are defined in Figure 2).

We will assume that for any event e, if a variable x appears in the constraint $a(e)$, then x is one of the variables declared in the tracematch. This is clearly satisfied by pointcut matching.

A symbol s is said to be a *ground* symbol if for any event e, $s(e)$ is either *true* or *false*. A ground symbol can match or fail to match, but does not bind variables.

Tracematches A tracematch is defined as a list of variables, a list of symbols, a pattern, and finally the body of the tracematch (code to execute when the pattern matches). The pattern is a regular expression over symbols. However, as we are only concerned with defining the semantics of *matching* here, we may ignore the body of the tracematch and define:

$$\mathit{tracematch} = \mathit{variable\ set} \times \mathit{symbol\ set} \times \mathit{symbol\ regexp}$$

We will fix a tracematch $tm = (F, A, P)$ in what follows. Hence F is the set of free variables of tm, A is the set of defined symbols, and P the pattern.

4.3 Semantics of Tracematches

Valuations A *valuation* is defined as a mapping from identifiers to runtime values, assigning values to each of the free variables of the tracematch:

$$\mathit{valuation} = F \rightarrow \mathit{value}$$

We define valuation on symbols as follows: the constraint resulting from matching $\sigma(a)$ to an event e is obtained by applying the valuation σ to $a(e)$:

$$\sigma(a) = \lambda e.\ \sigma(a(e))$$

In particular, as σ assigns a value to each variable occurring in $a(e)$, $\sigma(a(e))$ is a simple truth value.

For example, recall that $f(e1) = (x = v)$. If $\sigma = \{x \mapsto v\}$, then $\sigma(f)(e1) = (v = v) = \mathit{true}$, while if $\sigma' = \{x \mapsto w\}$, then $\sigma'(f)(e1) = (v = w) = \mathit{false}$ (provided v and w are distinct).

Valuations are lifted to patterns (regular expressions of symbols) by applying the valuation to each symbol in the pattern (in place).

Matching a trace to a word We define the *match* operator to take a sequence of symbols and a trace of events, and evaluate to the constraint that must be satisfied for the symbols to match the trace. If the number of symbols is the same as the number of events, the constraint is the conjunction of the constraints obtained by applying each symbol to the corresponding event. If the sequence of symbols and the trace of events are of different length, there can be no match, so the constraint is false. This can be written as:

$$\mathit{match}(\langle a_1, \ldots, a_n\rangle, \langle e_1, \ldots, e_m\rangle) = \begin{cases} (\wedge i : 1 \leq i \leq n : a_i(e_i)) & \text{if } n = m \\ \mathit{false} & \text{otherwise} \end{cases}$$

Note that we use the notation $(\oplus x : P(x) : v)$ in lieu of its equivalent $\bigoplus_{P(x)} v$ throughout.

The constraint that must be satisfied to match a trace to a sequence of symbols is just the conjunction of all the individual constraints obtained by matching each event to each symbol. If every symbol a_i is a ground symbol, the result is either *true* or *false*.

Filtering Recall that any events that do not match any defined symbol in a tracematch are simply ignored when matching. To formalise this, we define the *event set* of a tracematch, and the restriction of a trace to this set.

The event set of a tracematch tm under a given valuation σ is defined to be the set $\Omega(tm, \sigma)$ of events that are matched by some defined symbol in tm, with variable bindings compatible with the valuation σ. Formally, we define:

$$\Omega(tm, \sigma) := \{e \in \mathit{event} \mid (\exists a : a \in A : \sigma(a(e)) = \mathit{true})\}$$

Finally, we write the trace obtained from t by removing any events not in a set S as $t \downharpoonleft S$.

We can now define the match of a sequence of symbols to a sequence of events *relative to an alphabet* S. This is the match of symbols to events, ignoring any events not in S. A minor complication is that we must ensure that the *last* event in the sequence lies in S. This ensures that events that are ignored do not cause the tracematch to match repeatedly. We therefore define:

$$\mathit{match}_S(as, t) = (\mathit{last}\ t \in S) \wedge \mathit{match}(as, t \downharpoonleft S)$$

The match of a pattern (regular expression over symbols) to a sequence of events, still relative to an alphabet, is the disjunction of the matches of all strings denoted by the pattern to the given sequence of events:

$$\mathit{match}_S(p, t) = (\vee as : as \text{ in the language of } p : \mathit{match}_S(as, t))$$

The semantics of tracematches The semantics of tracematches can now be defined as follows. A tracematch tm is modelled by a function

$$[\![tm]\!] : \mathit{trace} \rightarrow \mathit{valuation\ set}$$

To wit, $[\![tm]\!](t)$ returns the set of valuations that cause tm to match some suffix of t. The body of tm will be executed exactly once for each such valuation.

Informally, the set of such valuations can be found as follows: replace the pointcut tm by the (possibly infinitely many) pointcuts obtained by applying every possible valuation σ to tm. Each of these involve no variables and can be matched against a trace straightforwardly. The result $[\![tm]\!](t)$ is the set of valuations that cause tm to match (some suffix of) t.

This can be formalised as follows. Write $u \prec v$ to mean that u is a suffix of v. Then

$$\begin{array}{l}[\![tm]\!](t) = \\ \{\sigma \in \mathit{valuation} \mid (\exists t' : t' \prec t : \mathit{match}_{\Omega(tm,\sigma)}(\sigma(P), t'))\}\end{array}$$

4.4 Operational Semantics

We have defined the semantics of tracematches in terms of applying all possible assignments of values to variables to a tracematch, and matching the resulting tracematches against a trace. We now wish to derive a more operational semantics that allows the resulting valuations to be effectively computed, leading to an implementation of tracematch matching.

Alphabet As before, A is the set of symbols that are explicitely declared in the tracematch. In addition, we introduce a symbol **skip** intended to capture both events that match no declared symbol (and so are ignored in matching), and events that could match some declared symbol but are ignored because of inconsistent variable bindings. This is defined by:

$$\begin{array}{l}\mathbf{skip}(e) := \\ \neg(\vee a : a \in A : a(e)) = \\ \wedge a : a \in A : \neg a(e)\end{array}$$

The constraint $\mathbf{skip}(e)$ defines the set of valuations that make e match no defined symbol a. We write $\Sigma = A \cup \{\mathbf{skip}\}$.

To illustrate, consider the event e_1 (see Figure 3) that occurs upon entering the call $v.f()$. Here we have $f(e_1) = (x = v)$ and $g(e_1) = \mathit{false}$, whence

$$\mathbf{skip}(e_1) = \neg(x = v \vee \mathit{false}) = (x \neq v)$$

Pattern We now aim to construct a finite automaton to implement matching of traces. To achieve this, it is necessary to transform the pattern P appearing in the tracematch to allow **skip** to occur.

For two sets of strings U and V, write $U \parallel V$ for the set of all possible interleavings of strings in U and V. It is easily checked that the class of regular languages is closed under interleaving.

The transformed pattern of the tracematch, named Pat, is the regular language

$$\mathit{Pat} = \Sigma^*(P \parallel \mathbf{skip}^*) \cap (\Sigma^* A)$$

A string s lies in Pat precisely when some suffix of s, possibly interleaved with some occurrences of **skip** representing ignored events, matches P. In addition, it is required that s end with a declared symbol (not **skip**).

To illustrate, for the tracematch shown earlier, $P = fg$, $A = \{f, g\}$, whence $\Sigma = \{f, g, \mathbf{skip}\}$. Then $\mathit{Pat} = \Sigma^* f \mathbf{skip}^* g$.

Executing advice We wish to execute advice whenever the current trace matches the pattern Pat. Unlike the declarative semantics described previously, there is no need to filter the trace (as **skip** symbols deal with events not in the alphabet) or to consider suffixes of the trace. We therefore execute the advice body for each solution of the constraint:

$$\mathit{match}(\mathit{Pat}, t)$$

Of course, if the constraint is $false$, there are no solutions and the advice body is not executed at all.

event		symbol	constraint
e1	enter: call(void FG.f())	f	$x = v$
e2	enter: execution(void FG.f())	**skip**	*true*
e3	exit: execution(void FG.f())	**skip**	*true*
e4	exit: call(void FG.f())	**skip**	*true*
e5	enter: call(void FG.h())	**skip**	*true*
e6	enter: execution(void FG.h())	**skip**	*true*
e7	exit: execution(void FG.h())	**skip**	*true*
e8	exit: call(void FG.h())	**skip**	*true*
e9	enter: call(void FG.g())	**skip**	*true*
e10	enter: execution(void FG.g())	**skip**	*true*
e11	exit: execution(void FG.g())	**skip**	*true*
e12	exit: call(void FG.g())	**skip**	$x \neq w$
e13	enter: call(void FG.f())	**skip**	$x \neq w$
e14	enter: execution(void FG.f())	**skip**	*true*
e15	exit: execution(void FG.f())	**skip**	*true*
e16	exit: call(void FG.f())	**skip**	*true*
e17	enter: call(void FG.g())	**skip**	*true*
e18	enter: execution(void FG.g())	**skip**	*true*
e19	exit: execution(void FG.g())	**skip**	*true*
e20	exit: call(void FG.g())	g	$x = v$

Figure 4: Matching a trace to a word.

As an example, Figure 4 shows a match between the trace given in Figure 3 and the string $f\mathbf{skip}^{18}g \in Pat$, together with resulting constraints. The complete constraint is $(x = v) \wedge (x \neq w) \equiv (x = v)$ (assuming v and w are distinct), whence the advice is run once, with valuation $x \mapsto v$.

4.5 Equivalence of the Semantics

We have defined two semantics for the match of a tracematch tm to a trace t, which we now reconcile. The two results of matching tm to t were defined as: the set of valuations

$$S = \{\sigma \mid \exists t' : t' \prec t : match_{\Omega(tm,\sigma)}(\sigma(P), t')\}$$

and the constraint

$$c = match(Pat, t)$$

As a notational convenience, we identify a constraint with the set of valuations that satisfy it. We therefore need to show that $\sigma \in S \iff \sigma \in c$.

The proof is founded on the following crucial observation about our definitions. If we fix a valuation σ, then for each event e,

$$e \in \Omega(tm, \sigma) \iff \exists a : a \in A : \sigma \in a(e) \iff \sigma \notin \mathbf{skip}(e) \tag{1}$$

The first equivalence is just the definition of Ω, and the second equivalence follows directly from the definition of **skip**. We denote the concatenation of sequences r and s by $r +\!\!+ s$.

$(\sigma \in S \Rightarrow \sigma \in c)$ Let $\sigma \in S$. Then we can split t into p and q such that $t = p +\!\!+ q$ and $match_{\Omega(tm,\sigma)}(\sigma(P), q) = true$. By the definition of *match*, there exists a sequence of symbols $as = a_1 \ldots a_n$ in the lagnuage P such that

$$(last(q) \in \Omega(tm, \sigma)) \wedge match(\sigma(as), q \downharpoonleft \Omega(tm, \sigma))$$

Let $q' = q \downharpoonleft \Omega(tm, \sigma)$. First note that as $match(\sigma(as), q') = true$, it is the case that $\sigma \in match(as, q')$.

Now, consider an event q_i of q. Then there are two cases: either $q_i \in \Omega(tm, \sigma)$, or q_i is not in this set. In the first case, q_i is part of q', say it appears at position j. Then by observation (1), $\sigma \in a_j(q_i)$. In the second case, again by observation (1), $\sigma \in \mathbf{skip}(q_i)$. Therefore, it is clear that $\sigma \in match(as \parallel \mathbf{skip}^*, q)$.

Also, as a consequence of observation (1), for any event e and valuation σ, there exists some $a \in \Sigma$ such that $\sigma \in a(e)$. Hence $\sigma \in match(\Sigma^*, p)$. Finally, $last(q) \in \Omega(tm, \sigma)$, so we can conclude that $\sigma \in match(\Sigma^*(P \parallel \mathbf{skip}^*) \cap \Sigma^* A, p +\!\!+ q)$, as required.

$(\sigma \in c \Rightarrow \sigma \in S)$ Since $\sigma \in match(\Sigma^*(P \parallel \mathbf{skip}^*) \cap \Sigma^* A, t)$, we know that $last(t) \in \Omega(tm, \sigma)$, and we can split t into p and q such that $t = p +\!\!+ q$ and $\sigma \in match(P \parallel \mathbf{skip}^*, q)$. Then q is an interleaving of two strings of events r and s such that $\sigma \in match(P, r)$ and $\sigma \in match(\mathbf{skip}^*, s)$. Since P is over the alphabet A, for each event r_i of r, $\exists a : a \in A : \sigma \in a(r_i)$, so $r_i \in \Omega(tm, \sigma)$. For each event s_i of s, $\sigma \in \mathbf{skip}(s_i)$, so $s_i \notin \Omega(tm, \sigma)$ (by observation 1). Therefore, $r = q \downharpoonleft \Omega(tm, \sigma)$. Hence, $match_{\Omega(tm,\sigma)}(\sigma(P), q) = true$, as required.

4.6 From Semantics to Implementation

It is relatively straightforward to derive the implementation from the operational semantics defined above. The main difficulty is to compute $match(Pat, t)$ for the current trace t efficiently at runtime.

Let M be an automaton for Pat. For each state s of M, define $L(s)$ to be the language obtained by making s the only final state.

During execution, each state s of M is labelled by the constraint

$$lab(s, t) = match(L(s), t)$$

, where t is the current trace. It is shown below how to update these constraints when a new event is appended to the current trace. After we have computed the new decorated version of M, the advice body is executed for all distinct solutions of

$$\vee s : s \text{ is a final state of } M : lab(s, t)$$

Computing Labelled States We now turn to the question of how to compute $lab(s, t)$ efficiently, making an update when the trace t is extended by another symbol.

We define $lab(s, t)$ by recursion on t. The base case is

$$lab(s, \varepsilon) := match(L(s), \epsilon) = \begin{cases} true & \text{if } s \text{ is an initial state} \\ false & \text{otherwise} \end{cases}$$

Now assume that we have computed $lab(s, t)$ for a trace t, and we want to know its new value $lab(s, te)$ for an extended trace te. Write $s' \rightarrow^a s$ to indicate that there is a transition labelled a from s' to s in M. Then for all states s, it is straightforward to derive that

$$lab(s, te) \quad = \quad \vee a, s' : a \in \Sigma \wedge s' \rightarrow^a s : lab(s', t) \wedge a(e)$$

This is almost ready to translate into executable code, but note that the formula treats declared symbols and the newly introduced symbol **skip** on the same footing. That is not quite possible in the implementation, for we have an explicit pointcut that corresponds to each $a \in A$, but not for **skip**. We therefore split off **skip** as a special case. Since $\Sigma = \{\mathbf{skip}\} \cup A$, the above formula may be rewritten as

$$\begin{aligned} lab(s, te) \quad = \quad & (\vee s' : s' \rightarrow^{\mathbf{skip}} s : lab(s', t) \wedge \mathbf{skip}(e)) \\ & \vee (\vee a, s' : a \in A \wedge s' \rightarrow^a s : lab(s', t) \wedge a(e)) \end{aligned} \tag{2}$$

Our strategy upon occurrence of a new event e, then, is to first compute $\mathbf{skip}(e)$, and subsequently to apply the above formula.

4.7 A Reference Implementation

This abstract reference implementation may, at first sight, appear expensive. Note, however, that in AspectJ most of the pointcut matching the computation of $e(a)$) can be carried out statically [18], and consequently the above transition from $lab(s,t)$ to $lab(s,te)$ is also mostly static: it can be pre-computed at compile-time, except for variable bindings.

Let us assume that there is some suitable implementation of constraints, through a class called Constraint. It is worthwhile to generate a specialised implementation for each tracematch, but for simplicity we assume it is generic. The Constraint type has the obvious operations for the logical operations. A new equality constraint is generated by the static factory method $eq(varname, value)$.

The key step is the computation of $lab(s,te)$ from $lab(s,t)$ for all states s, whenever the trace t is extended by one event. The implementation maintains variables *labs* and *labs′* for each state s — the value of *labs* is $lab(s,t)$ for the current trace, and *labs′* is an intermediate result in the computation of $lab(s,te)$. Furthermore, a variable *skip* is used to store the constraint $\mathbf{skip}(e)$ for the current event e.

This computation is done in two stages: the first stage computes *labs′* as the disjunction of all $labsj \wedge \mathbf{skip}(e)$, where $sj \rightarrow^{\mathbf{skip}} s$. That is the first part of Equation (2). The second stage adds disjuncts for each matching symbol (thereby computing $lab(s,te)$ completing the formula given in Equation (2)).

It is straightforward to define an action to be taken when a symbol a matches: it suffices to define a piece of advice with pointcut a. Furthermore, variable bindings are given by AspectJ's advice mechanism.

Suppose that the defined symbols of the tracematch are named pointcuts $a_1(vs_1)$, ..., $a_n(vs_n)$ (where for each i, vs_i is the list of variable names bound in a_i). Define a pointcut

$$some : a_1(*) \vee a_2(*) \vee \cdots \vee a_n(*)$$

that matches when some of the a_i do, ignoring variable bindings. Also, for a list of variable names vs and a list of runtime values os, let $eqs(vs, os)$ denote the constraint $(\wedge : 1 \leq i \leq |xs| : eq(vs_i, os_i))$.

Then the pseudocode for the implementation of a tracematch is the following aspect:

```
aspect Tracematch
{
  // For each initial state sj (1<=j<=N)
    private Constraint labsj = true ();
    private Constraint labsjp = false ();
  // For each non-initial state sj (1<=j<=N)
    private Constraint labsj = false ();
    private Constraint labsjp = false ();

  private Constraint skip  = true ();

  // Pass I
  // for each symbol ai (1<=i<=n)
  ai(y1, ..., yk) :
    { skip = and(skip,
                 not(eqs(vsi,
                       [y1,...,yk]))); }

  some :
    { for each state sj
        labsjp = false ();
        for each state sl with sl ->skip sj
          labsjp = or(labsjp,
                      and(labsl, skip));
      skip = true(); }

  // Pass II
  // For each symbol ai (1<=i<=n)
```

```
  ai(y1, ..., yk) :
    { for each state sj
        for each state sl with sl ->ai sj
          labsjp = or(labsjp,
                      and(labsj,
                          eqs(vsi,
                            [y1,...,yk]))); }

  some :
    { for each state sj
          labsj = labsjp;
        for each final state sj
          for each solution s of labsj
            run the advice body
            with bindings s }

}
```

This pseudocode cannot be directly expanded into AspectJ. For, we have omitted to consider the **before**, **after** or **around** qualifiers for each piece of advice. These have an impact on advice precedence, and we must ensure that the events described above happen in exactly this order at each event.

There are two ways to remedy this. The first is a more careful source-to-source translation that takes these into account. In this case, the *some* advice may be duplicated into **before** and **after** versions. Note that any event is unambiguously matched by one of **before** and **after**, but never both, so that we can freely perform this duplication. The case of **around** advice does not require duplication of this advice, by our requirement that if any final symbol is of type **around**, then all possible final symbols are.

The other strategy for a concrete implementation is to implement tracematches as an extension to advice weaving, and to insert the above code at the appropriate points directly.

Finally, the main aim of the above translation is clarity, but it should be obvious that opportunities for further specialisation of the code abound. We shall explore these and related issues in Section 5.

An Example To conclude this section, we illustrate the translation of tracematches into AspectJ with an example. Recall the Observer example from Section 2. The code is repeated below for ease of reference:

```
aspect ObserveAspect
{
  tracematch(Subject s, Observer o){
    sym create_observer
          after returning(o):
          call(Observer.new(..)) &&
          args(s)
    sym update_subject after :
          call(* Subject.update(..)) &&
          target(s);
    create_observer update_subject*
    {
          o.update_view();
    }
  }
}
```

For brevity, call the creation event c and the update event u. The set of declared symbols is $A = \{c, u\}$. The finite automaton implementing the pattern *Pat* derived from this tracematch is shown in Figure 5. State 1 is the only initial state, and state 2 is the final state.

The concrete implementation of the pseudocode for this tracematch is given below (taking the source-to-source transform approach):

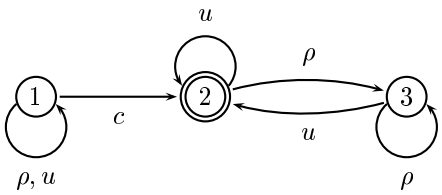

Figure 5: The automaton for the *obs* tracematch.

```
aspect ObserveAspect
{
  private Constraint lab1  = true ();
  private Constraint lab1p = false ();
  private Constraint lab2  = false ();
  private Constraint lab2p = false ();
  private Constraint lab3  = false ();
  private Constraint lab3p = false ();

  private Constraint skip  = true ();

  pointcut c(Subject s) :
    call(Observer.new(..)) && args(s);
  pointcut u(Subject s) :
    call(Subject.update(..)) && target(s);

  pointcut some() : c(*) || u(*);

  after(Subject s) returning(Observer o):
    c(s) {
    Constraint constr =
      and(eq("s", s), eq("o", o));
    skip = and(skip, not(constr));
  }

  after(Subject s): u(s) {
    Constraint constr =
      eq("s", s);
    skip = and(skip, not(constr));
  }

  after(): some() {
    lab1p = false ();
    lab2p = false ();
    lab3p = and(lab2, skip);

    skip   = true ();
  }

  after(Subject s) returning(Observer o):
    c(s) {
    Constraint constr =
      and(eq("s", s), eq("o", o));
    lab2p = or(lab2p, and(lab1, constr));
  }

  after(Subject s): u(s) {
```

```
48      Constraint constr =
49        eq("s", s);
50      lab2p = or(lab2p,
51                 or(and(lab2, constr),
52                    and(lab3, constr)));
53    }
54
55    after(): some() {
56      lab1 = lab1p; lab2 = lab2p; lab3 = lab3p;
57      for (s : lab2.sols()) {
58        adviceBody((Subject)s.valOf("s"),
59                   (Observer)s.valOf("o"));
60      }
61    }
62
63    void adviceBody(Subject s, Observer o) {
64      o.update_view ();
65    }
66
67  }
```

This highlights a clear optimisation — avoid recomputation of the constraints for each symbol by storing the value between the execution of the first and second pieces of advice for that symbol. This has been left out for clarity.

5 IMPLEMENTATION

We have implemented our design as an extension to *abc*, the extensible AspectJ compiler [1]. *abc* uses Polyglot, an extensible Java compiler [20], as its front-end. Polyglot works by defining a number of passes that transfrom the abstract syntax tree (AST). The source transformer of our extension is defined as such a polyglot transformation pass. This pass transforms the tracematch specific AST nodes into pure AspectJ, which is then compiled by *abc*.

The implementation given in the previous section is almost complete: it remains to decide on a concrete representation of the abstract type of constraints. We chose the simplest option, namely keeping the logical formula in disjunctive normal form. Below we shall refer to each component of that normal form as a *disjunct*: a constraint is a represented as a set of disjuncts. The type of disjuncts is specialised to the free variables of a tracematch. That is, a disjunct has a flag for each variable to say whether it is bound, and if not, a set of values it should *not* be equal to. If the flag is true, then the disjunct records the value bound.

5.1 Avoiding Space Leaks

A naive implementation of tracematches has the possibility of introducing memory leaks into a program. For example, consider the following tracematch which matches if a certain sequence of calls is made on an object.

```
1  tracematch {
2     sym first_call after(Object o):
3             call(Object+.method1()) &&
4             target(o)
5     sym second_call after(Object o):
6             call(Object+.method2()) &&
7             target(o)
8      first_call second_call
9      {
10       System.err.println("Error!");
```

```
11        }
12  }
```

The obvious tracematch compiler implementation would store a reference to every object that ever had method1() called on it to detect any subsequent calls to method2(). The problem is that if the base program calls method1() and then discards its references to the object, the second symbol will never match, but a reference to the object is still held by the tracematch implementation. This is not necessarily obvious to the programmer. Therefore, an implementation should use weak references as part of its strategy.

One can make the following observations. If an object is no longer referenced by the base program, symbols that bind the object cannot match. This means as far as the matching is concerned, the implementation only needs to keep weak references to the bound objects. If such a weak reference becomes invalid, a disjunct can be discarded if it is guaranteed that all paths through the DFA to a finishing state contain a transition that binds the same variable, because this transition can never be made. A disjunct can always use weak references for the values in the 'not' sets. Whenever one of these becomes invalid, it should be removed from the set as we can be certain that the corresponding variable will never be bound to that value.

However, if the variable is used in the tracematch body, it must be ensured that if the body is executed, the object is still alive. In this case, the implementation must keep a strong reference to the object if there is some path through the DFA to a finishing state that does not include a transition that binds the variable.

Given an implementation that follows the above strategy, it is useful to classify tracematches in terms of their memory behavior. Particularly, we want to detect if a tracecut could potentially lead to a memory leak. Tracecuts that cannot lead to memory leaks will be referred to as *safe*.

There are two classes of memory leaks to consider with tracematches.

- *Leaking bound objects.* The tracematch implementation potentially never releases bound objects.
- *Leaking disjunct objects.* The tracematch implementation creates an unbounded number of disjunct objects which are possibly never released.

Whenever the implementation creates a strong reference to a bound object, there is the potential for a memory leak of the first category. However, as long as at least one of the references held by the disjunct is weak and a future state is guaranteed to depend on this reference, this disjunct will be cleaned up when that reference becomes invalid, thus eliminating the strong reference.

If a simple value is bound by a tracematch variable, disjuncts cannot be discarded based on an invalid reference, so stricter safety rules are required. Furthermore, if a bound reference value is **null**, the same problem occurs. Generally, nullness cannot be determined at compile time.

Based on the above observations, we formulate the following safety conditions. First, to be safe in terms of **null** references, one of the following must hold.

- *Only one tracematch variable with reference type is used*

  or

- *Reference values bound by symbols that are not strictly final are not null at runtime.*

To establish the latter, the compiler should do a simple nullness analysis for bound reference values. If the compiler cannot establish non-nullness, the safety condition is only partially fulfilled. To avoid this, the programmer can satisfy the nullness analysis by explicitly testing for **null** in an **if** pointcut.

If one of the above conditions holds, a tracematch is safe if

- *Simple values are only bound by finishing symbols.*

  and

- *No tracematch variables with reference types are used in the body or at least one tracematch variable with a reference type is bound by all finishing symbols.* As soon as the object referenced by this variable is collected, the disjunct can be discarded.

This safety classification is a conservative one. It is possible for a programmer to construct a tracematch based on knowledge about the base program that is classified as unsafe but does not produce memory leaks. Therefore, an implementation should allow unsafe and partially safe tracematches, but issue a warning.

5.2 Executing advice

The implementation described in the previous section did not go into the details of how advice is executed once the matches have been found. The DNF representation of constraints means that each solution to a constraint can be obtained from the solution to a single disjunct (and similarly each disjunct's solution is a solution to the whole constraint). After the constraint for each state has been updated, the implementation needs to take the set of disjuncts for each final state and extract the substitutions needed to solve that disjunct. The advice is then executed for each of those substitions.

Conceptually, the repeated execution of the tracematch's advice is equivalent to the execution in AspectJ of multiple copies of the same advice body at the same joinpoint. However, as the number of matches is only determined at runtime, we have to simulate the effects of this ourselves.

If the final symbol in the trace is **before** or **after**, the implementation is straightforward; it simply needs to iterate through the list of substitutions, running the tracematch's advice once for each.

The only complication is that the set of disjuncts in each constraint must first be duplicated, in case the advice itself contains joinpoints that cause transitions in the DFA, thus modifying the constraint.

Executing **around** advice is somewhat more complicated, because it must be run in a nested fashion if there are multiple substitutions, in the same way as for multiple pieces of normal AspectJ **around** advice applying at the same joinpoint. A call to **proceed** should execute the advice again with the next substitution, and when the last substitution has been executed, the original joinpoint should be invoked. As with ordinary around advice, the execution should return to where it left off before **proceed** was called. Also, the advice may call **proceed** an arbitrary number of times, or wrap it within a closure object for later execution.

To achieve the described behavior, the implementation creates a specialized method that takes in a stack of substitutions and has the same return type as the around symbol. When the stack is non-empty, the top of the stack is removed and used to provide the substitution needed to execute the code block. Any calls to **proceed** in the advice are replaced with recursive calls to this generated method, passing in the remainder of the stack. Before the method returns, the current substitution is restored to the top of the stack to allow **proceed** to be called multiple times in the advice. When the stack is empty, a genuine call to **proceed** is made, resulting in the execution of the original joinpoint.

As described in Section 3.5, it is possible to define a signature for **around** which allows the modification of bound values in the call to **proceed**, before they reach the join point. If such a signature is defined, the implementation adds the corresponding parameters to the specialized method and to the advice method. For the first call to the method, the corresponding values from one of the substitutions are taken as arguments. Note that the choice of substitution is irrelevant since our restrictions on which variables can be part of the signature ensure that these variables have the same values for all substitutions. When the specialized method calls the advice method, it passes on its arguments, and when the advice method in turn calls the specialized method, it passes the values specified by the programmer in the **proceed** call.

6 Optimisations

A key issue in the design of a language feature is the balance between being sufficiently expressive to be practical, while being restrictive enough to make it possible to reason about the code (both by humans and by automated tools). As we have seen in Section 2, many useful tracematches can be expressed with our design. At the same time, a key goal of our design is to provide enough information to enable compiler tools to analyze tracematch behaviour and optimize their implementation. In this section, we propose analyses and optimizations that our design makes possible, and that we intend to implement in the future.

The state that a tracematch implementation must represent at run-time is a set of configurations of the

form (q, σ), where q is a state of the machine M matching Pat, and σ is the partial substitution of actual run-time values for tracematch variables that were implied by the events that caused M to transition to the state q. Whenever the executing program encounters a joinpoint matching a symbol of the tracematch (an event), and the variable bindings are consistent with σ, the state q is changed to the appropriate successor in M, and the substitution σ is updated with any new bindings implied by the match. In optimizing the implementation of tracematches, our goal is to reduce the size of set of configurations that must be maintained by removing configurations which can be proven to never lead to an accepting state.

Let us begin by considering the simple case of a tracematch with no tracematch variables. In this case, every substitution is empty, so a configuration is just a machine state q. By identifying joinpoint shadows in the program where each symbol may match, and by performing an interprocedural control-flow analysis of the program, an analysis can construct a finite state machine N modelling the possible executions of the program. Each state p in N is a joinpoint shadow matching a symbol $m(p)$ of the tracematch, and there is a transition from state p to p' if there is a potential control-flow path from the joinpoint shadow p to the joinpoint shadow p', passing through no other shadow matching any tracematch symbol. Then, when the program execution is at a joinpoint shadow p_0, and the tracematch is in state q_0 of M, it is possible to reach an accepting state of M only if there is a sequence of joinpoint shadows $p_0, p_1, \ldots, p_n$ such that it both is a path in N, and that following the transitions $m(p_0), m(p_1), \ldots, m(p_n)$ in M starting from q_0 leads to an accepting state. This can be determined at compile time by intersecting the automata M and N. If it is not possible to reach an accepting state of M, then at p_0, the compiler can generate code to omit the configuration q_0 from the set of configurations.

```
1  List l;
2  Iterator i;
3
4  while(condition) {
5      l = new ArrayList();
6      l.add("foo");
7      i = l.iterator();
8      while(i.hasNext()) {
9          System.out.println(i.next());
10     }
11 }
```

Figure 6: Example use of safe iterator tracematch

Dealing with tracematch variables requires a more sophisticated analysis, because whether a joinpoint shadow matches a tracematch symbol depends on the values of the tracematch variables. Optimizing even relatively simple tracematches requires analysis of the flow of these values and flow-sensitive alias information.

Consider, for example, the short program fragment in Figure 6, to which we apply the safe iterator tracematch from Section 2. The example creates a collection, adds an element to it, and iterates through it, all repeated within a loop. In an actual application, such code would likely be interspersed with other code, and most likely spread out in different methods, but the general pattern of operations is fairly typical.

In a naive tracematch implementation, each time an iterator is created, a new configuration would be created binding the tracematch variable ds to the current list object, and the tracematch variable i to the iterator object just created. If an analysis could statically prove that the list will not be updated between the creation of the iterator and a call to next() on the iterator, the tracematch would be known never to apply, and the configuration would not have to be created. An analysis cannot prove this without information about the values of l and i, however, because if l and i could arbitrarily point to any object, it would be possible to create an iterator in line 7 of the first iteration of the loop, and in a second iteration, add an element to the same list in line 6, and call next on the iterator in line 9.

To prove iterator safety for this simple example, an analysis would need to either know that the list to which an element is added in line 6 is distinct from every list on which an iterator has ever been created in line 7 in earlier loop iterations, or that the iterator on which next is called in line 9 is the iterator that

was created in line 7 of the same iteration of the loop, and that it is distinct from all other iterators created earlier in the program. Obtaining this information requires a flow-sensitive analysis to track the flow of objects through the program and determine whether objects are definitely equal (like a must alias analysis) or definitely distinct (like a may alias analysis). However, optimizing tracematches requires more information than an alias analysis can provide. While a traditional alias analysis determines the aliasing relationships between different variables at the same point in the program execution, optimizing tracematches requires knowing whether a variable points to the same or different object as some variable pointed to at a *different, earlier* point in the program execution. Such information could be obtained by starting with alias information about the earlier program point, and propagating it along all control flow paths to the later program point. We leave further development of such an analysis to future work.

7 RELATED WORK

It has long been recognised that history-based advice is a powerful and desirable feature in aspect-oriented programming. The contribution of this paper is to enhance the previous proposals through trace filtering and consistent variable bindings, as well as a seamless integration into AspectJ. Below we discuss these previous proposals, and we pin down how our own design differs from them. We also briefly review some related work in property checking — although the techniques are not called 'aspect-oriented', there are many overlapping ideas.

Douence et al. History-based advice first came to our attention through the work of Douence, Fradet, Motelet and Südholt [7–10]. In these works, they put forward a calculus of aspects, where advice can be triggered via a sequence of joinpoints. The syntax of their history advice is

$$\begin{array}{rcll} A & ::= & \mu a.A & \text{recursion} \\ & | & C \triangleright I; a & \text{base case of recursion} \\ & | & C \triangleright I; A & \text{sequencing} \\ & | & A \Box A & \text{choice} \end{array}$$

The first form is a recursive definition; the base case of such a recursion is the second form, where a stands for the recursive call. Both in the second and third form, C stands for a pointcut, and I for a piece of advice. Intuitively, if a joinpoint matches C, the advice I is executed, and control transfers to a (recursion) or A. Finally, $A_1 \Box A_2$ offers the environment a choice between two pieces of history advice A_1 and A_2: if A_1 succeeds, that is the preferred option, and A_2 executes only when A_1 fails.

As a concrete example, consider the history advice below, which is taken from [9]. It logs file accesses during a session (from a call to *login* to a call to *logout*):

$$\begin{array}{l} \mu\, a_1\ .\ \mathit{login} \triangleright \mathit{skip} \\ \qquad (\mu\, a_2\ .\ (\mathit{logout} \triangleright \mathit{skip}\ ;\ a_1) \\ \qquad\qquad\quad \Box \\ \qquad\qquad\quad (\mathit{read}(x) \triangleright \mathit{addLog}(x)\ ;\ a_2)) \end{array}$$

The reader is encouraged to contrast this formulation with the *contextual logging* example presented in Section 2.

As a formal calculus, the work of Douence *et al.* is more geared towards a formal understanding, and somewhat less towards a production programming language than ours. Nevertheless, there are clear similarities in the design: in particular, because only tail-recursive definitions are allowed, the patterns of execution are essentially regular languages.

An important difference is the association of a piece of advice with every pointcut. In our setting, this would mean that every symbol declaration has an associated piece of code. Clearly this is very powerful, but it also makes it very difficult to track what is happening in the matching process, especially when the advice has side effects. As we did not find any need for such expressiveness in our examples, we decided to eschew those complications.

A second important difference concerns the treatment of the choice operator. In the design of Douence *et al.*, ($\Box$) is asymmetric, favouring the left-hand component where possible. In our proposal $R|S$ and $S|R$ are equivalent patterns. Furthermore, if both R and S match, that may result in multiple variable bindings, and the advice is executed once for each binding. Several of the examples in Section 2 (in particular Observer and Connection Management) make essential use of such multiple bindings.

A very nice feature of the design of Douence *et al.* is that it enables interesting static analysis to determine possible interactions between aspects [9]. Based on the close similarities with our work, we are fairly confident that their results can be transferred to our setting, and implemented in *abc*.

There exists at least two implementations of the design of Douence *et al.*, namely in the JaSCo [22] (an integration with Java), and in the Arachne system [11]. Applications of the former are discussed in [5] and of the latter in [12]. Especially the examples of [12] provide strong indication of the importance of matching with variables as we have defined it: in that paper, the code is littered with explicit equality tests between variables. In tracematches, such equality tests are expressed by simply using the same variable multiple times.

Walker and Viggers The term 'tracecuts' was introduced by Walker and Viggers in [23]. Unlike the works discussed above, their design has also been integrated with an implementation of AspectJ. It is particularly interesting, therefore, to compare our design decisions to theirs.

An obvious difference is that their design uses an extension of context free grammars to define the set of traces to match, rather than the regular expression presented here. The set of languages used are not strictly context free, however. A 'semantic action block' can be associated with each token, to be executed whenever a current joinpoint matches a token. This block has access to information about the trace matched so far and can reject a match using the `fail` keyword, which results in the computation continuing as if the joinpoint had not matched the token. The presence of these blocks removes any restrictions on the set of languages that can be used to identify matches. We believe there is merit in restricting the set of languages that is recognised, not least from the point of view of program analysis: while it is trivial to decide whether one regular language is included in another, the problem is undecidable for context-free languages.

These semantic action blocks can also have side effects, which complicate the relationship between the tracecuts and the original program. Without side-effects, a tracecut simply observes the execution of the base program until the point where a match is discovered (so if a match is never found then the behaviour of the program is not altered.) When side-effects are allowed, a tracecut may interact with and modify the behaviour of the base program during the matching process, making them more complex than straightforward observers.

Another important difference is the way variable bindings are used. In Walker and Viggers' design, no context from a trace is stored automatically, but semantic action blocks can be used to explicitly store context. With this method, the automata do not need to take the values collected by a pointcut into account when deciding whether a token should match, instead they pass all the values into the semantic action blocks, which can use them to decide whether or not to accept a match, and optionally store them for later use.

This behaviour means that each tracecut only binds a single set of values at a time. As a result, tracecuts cannot be used to simultaneously track the behaviour of a set of individual objects, but only that of the entire program. This limitation means that most of the example tracematches given above that bind variables can not be expressed in their design.

The last important difference is that where the design above presents tracematches as a kind of higher-order operator over pointcuts, tracecuts are just another type of pointcut. This is made possible by treating a 'match' as a predicate rather than and event, as discussed previously. The most significant effect of this is that it permits one tracecut to be used in the definition of a token in another tracecut. That is, instead of specifying a trace directly as a sequence of events, it is specified in terms of a sequence of completions of other traces (which are themselves specified in this way). This idea is not possible in our design, where tracematches can result in multiple sets of variable bindings, because allowing tracematches to be used as pointcuts would break all constructs that assume a pointcut binds each of its variables to exactly one value.

Bockish et al. In [3], Bockish, Mezini and Ostermann put forward a very general notion of pointcuts that capture dynamic properties. Their proposal is implemented in the Alpha language [21]. Alpha provides Prolog queries over a rich representation of the program, including a complete representation of the execution history up to the current joinpoint. It thus provides a flexible testbed for experimenting with radical new pointcut idioms, albeit without regard for efficiency of the implementation. We believe it would be easy to implement our design for tracematches in Alpha, although such an implementation cannot rival the compilation techniques discussed here.

Property checking Recent years have seen a veritable explosion of work that aims to verify, either dynamically or statically, the correct usage of an API. The Safe Iterators example in Section 2 is a typical instance of the type of property involved. These works on property checking are almost entirely disjoint from the aspect-oriented programming community.

Typically one specifies erroneous traces in a separate specification language, and then the specification is statically checked against the code, or dynamic tests are woven in as appropriate. Examples of this line of work are [2, 14].

An important difference with the proposal discussed here is that tracematches are intended as a feature that is fully integrated in the programming language, here AspectJ. AspectJ has some very weak support for static checking of properties, namely the *declare warning* and *declare error* constructs. These take a pointcut and a message: when the pointcut is matched at compile-time, the error message is printed. An obvious generalisation is to provide trace versions of these constructs, and then the formalism is very close in expressive power to the works cited above. Such a feature would require a static analysis similar to the one discussed in Section 6.

Bodden In [4], Bodden introduces the notion of concern-specific languages (CSLs), which are specific to a cross-cutting concern like domain-specific languages are specific to a domain. He considers the concern of Runtime Verification, and shows how it can be implemented with an example language, namely linear-time temporal logic (LTL) over pointcuts. The language thus defined allows checking of certain run-time properties — among the examples Bodden gives are checking that a user is logged in when performing certain actions and proper use (i.e. timely acquiring and releasing) of locks during program execution. He exhibits an implementation which uses the *abc* framework to translate the LTL expressions into pure AspectJ.

The LTL predicates, as defined in this work, offer functionality that is quite similar to tracematches – properties of the program execution as a whole can be checked. Consequently, all the examples he presents have natural equivalents that can be expressed using tracematches. The converse is not necessarily true; in particular for tracematches that use variable bindings, LTL equivalents may be quite cumbersome or even impossible. Also, the focus of the work is verifying properties of the program execution rather than injecting code; tracematches offer more flexibility here. Bodden presents the idea that CSL implementations could benefit from building on top of each other, and this seems justified here: translating his language into tracematches seems easier than into pure AspectJ.

8 CONCLUSIONS

We have presented a novel design for integrating tracematches into the AspectJ language. The main innovation is our treatment of free variables in trace patterns. By defining the meaning of a tracematch through *all* consistent instantiations of these variables, many more examples are conveniently expressed. Inspired by these applications, we carefully reviewed the design space for tracematches and motivated our design decisions.

Of course the use of variables in trace patterns is non-trivial, and therefore we presented a precise declarative semantics, intended for reasoning about the behaviour of tracematches. We also presented an operational semantics as a step towards an implementation. The main insight in defining the operational semantics was the need for a new symbol in the alphabet, to capture the skipping of other symbols, due to

variable binding. The declarative and operational semantics were proved to be equivalent. This is quite a satisfactory result, because conceptually they are quite different.

Furthermore, the operational semantics directly led us to a reference implementation of tracematches. It would have been quite difficult to arrive at this implementation without the careful semantic analysis that preceded it.

There are also a number of pragmatic issues must be addressed in the implementation, in particular regarding the memory usage of tracematches. We also identified a number of further optimisation opportunities. Some of these required advanced program analyses, and we intend to report on careful performance experiments, involving those advanced optimisations, in a companion paper.

Finally, this language design exercise exemplifies our philosophy for aspect-oriented programming language research: a rigorous analysis of use cases, followed by a sound definition of the semantics, leading to a neat implementation. The implementation itself has been carried out using the *abc* compiler, a workbench for aspect-oriented language and compiler research.

References

[1] abc. The AspectBench Compiler. Home page with downloads, FAQ, documentation, support mailing lists, and bug database. `http://aspectbench.org`.

[2] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In Eerke Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods*, pages 1–20, 2004.

[3] Christoph Bockish, Mira Mezini, and Klaus Ostermann. Quantifying over dynamic properties of program execcution. In *2nd Dynamic Aspects Workshop (DAW05)*, pages 71–75, 2005.

[4] Eric Bodden. Concern specific languages and their implementation with abc. SPLAT workshop at AOSD. Download: http://www.bodden.de/publications, 2005.

[5] María Augustina Cibrán and Bart Verheecke. Dynamic business rules for web service composition. In *2nd Dynamic Aspects Workshop (DAW05)*, pages 13–18, 2005.

[6] 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.

[7] 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)*, pages 173–188, 2002.

[8] R. Douence, O. Motelet, and M. Sudholt. 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.

[9] 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.

[10] Remi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In *Aspect-oriented Software Development*, pages 141–150. Addison-Wesley, 2004.

[11] 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.

[12] 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)*, pages 5–12, 2005.

[13] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. Wiley, 2003.

[14] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 69–82, 2002.

[15] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA*, pages 161–173, 2002.

[16] I. Kiselev. *Aspect-oriented programming with AspectJ*. SAMS, 2002.

[17] Ramnivas Laddad. *AspectJ in Action*. Manning, 2003.

[18] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction*, volume 2622 of *Springer Lecture Notes in Computer Science*, pages 46–60, 2003.

[19] Russell Miles. *AspectJ cookbook*. O'Reilly, 2004.

[20] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152, 2003.

[21] Klaus Ostermann, Mira Mezini, and Christoph Bockish. Expressive pointcuts for increased modularity. In *ECOOP*, 2005.

[22] Wim Vanderperren, Davy Suvé, María Augustina Cibrán, and Bruno De Fraine. Stateful aspects in JAsCo. In *Workshop on Software Composition at ETAPS*, 2005.

[23] 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.