Back to the Future: Pointcuts as Predicates over Traces

Karl Klose
Darmstadt University of Technology
D-64283 Darmstadt, Germany
Karl.Klose@gmx.de

Klaus Ostermann
Darmstadt University of Technology
D-64283 Darmstadt, Germany
ostermann@informatik.tu-darmstadt.de

ABSTRACT

Pointcuts in aspect-oriented languages can be seen as predicates over events in the computation of a program. The ability to express temporal relations between such events is a key feature towards more expressive pointcut languages. In this paper, we describe the design and implementation of a pointcut language within which pointcuts are predicates over the complete execution trace of the program. In particular, pointcuts may refer to events that will happen in the future. In this model, advice application is an iterative process that stops once a fixed-point is reached. On the negative side, we do not have a "killer example" for these kinds of pointcuts, there are still some serious limitations, and our implementation strategy is not suitable for a practical language. However, we think that considering pointcuts as predicates over the whole computation and advice application as a fixed point problem is an interesting new perspective on pointcuts for the FOAL audience.

1. INTRODUCTION

In aspect-oriented programming, dynamic join points are points in the execution of a program, and pointcuts are predicates over join points. In the past, aspect-oriented programming has concentrated on pointcuts that are only predicates over the current join point and can thus efficiently be implemented by means of static weaving. However, these pointcuts are not powerful enough to express *relations* between different join points.

AspectJ has one special construct to relate different join points in the form of the **cflow** pointcut designator [7]. More recent works try to make more information about history of the computation available for describing pointcuts [3, 10].

In this paper we want to increase the expressiveness of pointcuts even further and consider pointcuts as predicates over the whole execution trace. We have designed and implemented a prototypical aspect-oriented languages within which the execution trace is reified as a deductive database in Prolog [9] and pointcuts are queries over this database.

Advice application is a fixed point problem in this language. Our implementation computes a solution to the fixed point problem with an iterative process, within which from all advices that are applicable for a given trace, the advice that starts at the smallest point `T` in time is executed by resetting the application to the state at time `T`, and executing the advice and the remainder of the application. This process is iterated until a fixed-point is reached.

This model allows pointcuts to refer to future events in the computation, and also allows sophisticated interactions between advices. However, the power that comes with this model is also not without disadvantages. For example, it is easy to create examples with paradox aspects where the aforementioned iterative process does not have a fixed-point.

The remainder of the paper is structured as follows. Sec. 2 presents our language Gamma. Sec. 3 presents some example programs. Sec. 4 discusses the problem of semantics and some preliminary results on the applicability of domain theory and static analysis techniques to guarantee that iterative advice-application is well-behaved. Our prototype implementation is presented in Sec. 5. Sec. 6 discusses related work. Sec. 7 concludes.

2. AN OVERVIEW OF THE GAMMA LANGUAGE

Our prototype language Gamma is an aspect-oriented language on top of a minimal object-oriented core language. This object-oriented core language is based on the teaching calculus L2 from Sophia Drossoupolou [4], which is similar to Featherweight Java [6], but also has an object store and hence supports assignments, aliasing etc. A formal syntax, operational semantics, and type system are described in [4]. Here we will use and describe this core only informally.

Gamma supports classes and single inheritance. The only primitive type is `bool`. Methods have a return type and always one argument whose name is always `x`. If no argument is required, we add a dummy argument of type `bool`.

Gamma is expression-oriented, so the body of a method is an expression. All names have to start with a lower-case letter. This makes interoperability with Prolog a little bit easier because then every name in this language can directly be used as a Prolog atom. Minimal I/O is available via a

`print` expression that can print strings or objects.

In addition to fields and methods the class `main`[1] can contain aspects, which consist of a keyword indicating the kind (`before` or `after`), a pointcut term (a pattern in the execution trace) and *advice*, code whose execution is triggered by the pointcut. A pointcut is basically a Prolog query (with some restrictions) in which at least one predicate should have the variable `Now` as its first argument. The value of this variable determines at which point in the computation the advice should be executed.
An advice is similar to a normal method but it can use unified variables of the query as expressions. If `this` is used inside an advice body it always refers to the first `main` instance that has been created.

The execution trace is represented as a collection of facts in a Prolog database, where each fact corresponds to one step of the interpreter. The number of the current step, or timestamp, is always stored as the first argument of a fact.

The following tables give an overview of the facts used in traces and their arguments:

Fact	Meaning
get(T, C, A, F, V)	reading field access
set(T, C, A, F, V)	writing field access
calls(T, C, A, M, V)	method call
endCall(T, B, R)	end of method call
newObject(T, C)	object creation

Arguments	Meaning
A	Address of target object
B	Timestamp of begin of call
C	Class of target/to create instance of
F	Name of field
M	Name of method
R	Return value
T	Timestamp
V	Current/new value of field

The pointcut language is closely connected with the Prolog syntax. In fact every pointcut is a Prolog query. Pointcut terms consist of predicates that can be combined by commas (a comma means `and` in Prolog). A predicate can contain variables (which have to start with a capital letter, like `Var`), anonymous variables (`_`), predicates or atoms (which start with a lower letter, like `main`. To negate a prolog term, the predicate `not`[2] can be used. The special variable `Now` identifies the timestamp where the pointcut matches, i. e. the point where advice should be inserted. Variables that have been used in the pointcut can be used as expressions in the advice. Fig. 1 shows a short program with an aspect which demonstrates how variables used in the query (here `Address`) can easily be accessed from within the advice.

To compare variables used for timestamps, the predicates `pred(T1,T2)` and its transitive closure `isbefore(T1,T2)`

[1]Due to space limitations and for the sake of clarity we simplified our language: only the class `main` can have aspects and `after` as well as `around` advices have been omitted
[2]In Prolog one should rather use `\+` than `not`, but this syntax is easier to parse.

```
class main extds Object {
 bool var;
 before set(Now,_,Address,_,_) {
  print(Address)
 }
 bool main(bool x){
   this.var := true
 }
}
```

Figure 1: Class with aspect

are available. The pointcut `not(set(T,_,Addr,Field,_)), get(Now,_,Addr,Field,_), isbefore(T, Now)` for example matches read access to any field of an object that has not been set before. In contrast, `set(Now,_,Addr,Field,_), set(T,_,Addr,Field,_), pred(Now,T)` matches any assignment to a field immediately followed by another assignment to the same field. If timestamp variables are not related, they can match *any* point in the trace.

More complex predicates, like the well-known `cflow`, can be easily formulated as rules:

```
% T2 is in the control flow of the call at T1
cflow(T1, T2) :-
  calls(T1,_,_,_,_),
  endcall(T3,T1,_),
  isbefore(T1,T2),
  isbefore(T2,T3).
```

Similarly the content of the store and the call stack at each time can be reconstructed from the `set` resp. `calls` facts using `isbefore`.

3. APPLICATION EXAMPLES

Consider an environment where a set of graphical objects are potentially manipulated by some `operation`. There is a display, which should only be updated if at least one element has been changed and the number of updates should be minimal. Fig. 2 shows a solution in GAMMA. The pointcut matches at the end of the execution of `main.operation` if a call to `point.setpos` lies in its control flow.

```
before calls(T1, main, _, operation, _),
       cflow(T1, T2),
       calls(T2, point, _, setpos),
       endCall(Now, T1, _) {
   this.display.update(true)
  }
```

Figure 2: A display example

This example shows how the pointcut languages allows us to simply refer to the history of the execution, making the aspect both short and easy to understand. In other pointcut languages, one would have to manually store parts of the history together with complicated imperative logic in order to achieve the same effect.

The pointcut in Fig. 2 only refers to past events. This is different in the next example. Fig. 3 illustrates a kind of

eager authentication, which performs authentication *before*, but only if, a call to a *protected* **database** function is made inside the control flow of **server.execute**. Such an aspect may appear in a scenario where a complex command has to be executed and it is necessary to be logged in if *any* of the subcommands will require authentication during its execution.

```
before calls(Now,server,_,execute,_),
       cflow(Now,T),
       calls(T,database,_,protected,_) {
  this.db.authenticate(true)
}
```

Figure 3: An authentication example

By the usage of logical programming and predicates as the base of our pointcut language we gain a great flexibility to express temporal related pointcuts without making the language too complex to use and understand.

4. PARADOX ASPECTS

In the preceding examples it was intuitively clear what the semantics of the programs should be, even in the case that a pointcut refers to the future. But we can easily construct examples where it is hard to say how the semantics of the program should be defined. We will discuss some of these examples and different strategies to solve the problems these examples impose.

As mentioned before, the execution of advice can enable pointcuts at *every* other position in the computation. This can easily produce a phenomenon that is similar to the "grand-mother paradoxon" in time travel: an aspect whose pointcut is enabled by the base program uses its advice to change the control flow in such a way, that the pointcut is not being enabled. Fig. 4 gives an example of such an aspect. The problem is that the trace of this program is not consistent in any of both cases: if the advice is not executed, its aspects pointcut is enabled but if it is executed, its pointcut is not enabled.

```
class main extds Object{
  bool create;
  before calls(Now,_,_,foo,_),
         newObject(T,a),
         isbefore(Now,T) {
    this.create := false
  }
  bool foo(bool x){
    if this.create
      then (new a; true)
      else false
  }
  bool main(bool x){
    this.create := true;
    this.foo()
  }
}
```

Figure 4: A paradox aspect

4.1 Properties of advice application

We can view advice application in our language as a non-deterministic transition system on traces, whereby $t \to t'$ means that the trace t' is the result of applying an advice to the trace t.

An activation point of a trace is a position in the trace, where advice has to be inserted due to an enabled pointcut or inserted advice has to be removed because the corresponding pointcut is not enabled any longer. For convenience we write $AP(t)$ for the *activation points of a trace* t. In general, several pointcuts may be applicable to a trace (i.e., there is more than one activation point), hence the transition system is non-deterministic. Unfortunately, this transistion system does not enjoy the confluence property, meaning that the final result depends on the non-deterministic choice of the next advice to apply. It also does not have a standardization property, informally meaning that there is no "best" choice for the next advice.

4.2 Traces as domains

Domain theory provides a general setting within which recursive equations have a proper solution. When we look at the process of advice application as a function on the set of all traces of a program P, $\mathcal{F} : \mathcal{T}_P \to \mathcal{T}_P$, one way to reason about termination is to apply the fixpoint theorem, a result of domain theory. As mentioned before, there may be several strategies to define such a function, so we will discuss those functions in general. A specific selection strategy is presented in Sec. 5.

If there is a partial order $\sqsubseteq$ that makes the set $(\mathcal{T}_P, \sqsubseteq)$ a **cpo**[3] and if the advice application operator $\mathcal{F}$ is monotonic and Scott-continuous w. r. t. $\sqsubseteq$, then the fixpoint theorem garantees that $\mu(\mathcal{F}) = \bigsqcup_{n\in\mathbb{N}} \mathcal{F}^n(\bot)$ exists with $\mathcal{F}(\mu(\mathcal{F})) = \mu(\mathcal{F})$. This fixed point thus can be constructed by repeatedly applying advice. The bottom element $\bot$ is in our case the trace of the base program without any advice. The challenging part is to define $\mathcal{F}$ and $\sqsubseteq$.

One example to construct such an order is as follows. Let $lap(t)$ (least activation point of t) be the first point where advice has to be inserted or must be removed. Furthermore, let $s = (s_0, \ldots, s_{n-1})$, $t = (t_0, \ldots, t_{m-1})$, $a = lap(s)$ and $b = lap(t)$ then consider the transitive and reflexive closure $\sqsubseteq_P$ of the following relation:

$$s \sqsubset_P t \;\Leftrightarrow\; t = (s_0, \ldots, s_{a-1}, \overbrace{u_0, \ldots, u_{k-1}}^{\text{trace of advice}}, v_0, \ldots, v_{l-1}) \quad (1)$$
$$\wedge\; b > a + k + 1 \quad (2)$$
$$\wedge\; n < a + l \quad (3)$$

In words, a trace t is greater than another trace s, if it can be obtained (possibly in more than one step) from s by simply inserting advice at the earliest possible point (1). Conditions (2)+(3) ensures that no advice will be removed, that each new advice is always executed *after* the last one and that the part of the trace after the advice invocation does not get longer.
It is easy to see that this order makes $\mathcal{T}_P$ a **cpo** because every chain starting with $s = (s_0, \ldots, s_{n-1})$ can have at most n distinct traces. However, this condition is very restrictive and hard to check statically. For example, pointcuts cannot

[3]A **cpo** is a partial ordered set where the supremum of each ω-chain is also contained in the set.

change the control flow after advice application in such a way that the trace gets longer (condition (3)).

Now we need to define an advice application operator $\mathcal{F}$ that is monotonous and continuous. Our general strategy is to select always the earliest activation point in the trace whose advice has not yet been executed. However, we need additional restrictions in order to ensure that $t \sqsubseteq_P \mathcal{F}(t)$.

This can be ensured by a (conservative) static analysis of advice bodies. We present the basic ideas for an algorithm to identify programs that meet the desired restriction.

For each pointcut there is a set of shadows, locations in the source code, where it is *possible* that it could match. We say that an aspect A *precedes* another aspect B if the earliest shadow of A is below or equal to a shadow of B in at least one control path. Furthermore an A *affects* B, if the execution of A can affect the pointcut matching of B, i. e. if a shadow of B lies in that part of the program that can be possibly affected by the execution of A. One condition for the class of acceptable programs could be formulated as: **if** A **precedes** B**, then** B **must not affect** A (and therefore A **must not affect itself**) and the *affect* relation should not have any cycles.

The crucial point is clearly to identify the part of a program that can be affected by an aspect. We developed some basic techniques, but they are rather inaccurate and inefficient, so further work on this topic is required.

5. PROTOTYPE IMPLEMENTATION

As a result of the powerful pointcut language and execution model, some new problems arise in determining the semantics of a program. First, as pointcuts can refer to events in the future, we can not make judgments about advice invocations before the complete trace of the execution is available. Therefore we must run the program at least once to see, where advice has to be executed. Another problem is that the execution of advice itself can effect the execution of other aspects at *any point* in the execution trace, after and *even before* the point at which this advice has been inserted. Our solution to these problems is presented in this section.

Our approach is to iterate advice application, beginning with a trace that does not invoke any advice at all, to (hopefully) get better and better approximations of the final execution trace. If more than one pointcut matches, the first one is chosen to be executed. We model this procedure as an "advice-application-operator" $\mathcal{F}$ which takes a trace t and returns another trace t'.

The interpreter we use to run the program creates and stores a copy of its internal state at every step. The interpreter can thus be reset to any point in the execution of the program. This feature is necessary to restart the execution from the first point where changes in the behavior are expected (due to advice insertion or removal).

To find the points in a trace where advice has to be inserted, the pointcuts of all aspects must be evaluated. The queries are passed to the Prolog engine which returns a set of variable bindings for each positive answer. So we can determine the timestamp of the match by looking at the value of the variable `Now`. These timestamps along with the associated advices and the variable bindings describe the activation points in the trace. The set of APs that has been identified in the current trace is called `foundAPs`. By `oldAPs` we denote the set of APs that has been found in the last trace (this set is empty in the first iteration).

Now there are two things that can happen: a pointcut could match a position it did not match before or a pointcut did match a position in the old trace but does not in the actual one. In the first case, the AP is in the set `foundAPs`$\backslash$`oldAPs` otherwise it is in `oldAPs`$\backslash$`foundAPs`. The earliest such event (`currentAP`) is that $a \in$ `oldAPs`$\triangle$`foundAPs`[4] with the minimal timestamp (the textual order of the aspects is considered, if two APs have the same timestamp).

After determining `currentAP`, the interpreter (and with it the fact database) is reset to the timestamp of that point and the program is executed from this point in the next iteration. The interpreter does not need to considers other APs than `currentAP` because the execution trace of those that were before stays in the database and APs that lie in the future may become invalid due to the execution the advice. When the advice has been inserted, the timestamp of corresponding AP in `oldAPs` must be updated, because the matching point of the pointcut trace has moved due to the trace produced by the advice.

When the advice has been executed, the `currentAP` must be updated, because the timestamp of the matching position in the trace has moved. A fixed-point of the iteration is reached, if both sets, `oldAPs` and `foundAPs`, are the same.

There are two properties of advice application that follow from the procedure described above:

1. Advice application will only terminate if the base program (without advice) terminates.
2. If advice must be removed at any point in the trace, the iteration ends up in a cycle since the resulting trace has been processed before.

The program in Fig. 5 illustrates how a pointcut can refer to future events: it only matches those assignments to `varx` which are (not necessarily immediately) followed by an assignment to `vary`. The iteration process for this example is shown in Fig. 6. It also shows how the AP is updated after inserting the advice.

The first property of our model, namely that advice application will only terminate if the base program does, is indeed very restrictive. The consequence is that certain program parts cannot be modelled as aspects, in particular aspects that force the program to terminate. e. g. the break condition of an algorithm. For the same reason our model cannot capture infinite computations.
We could overcome these limitations by changing the iteration process to execute advice (and maybe reset the inter-

[4] $A \triangle B$ is the symmetric difference of the sets A and B: $x \in A \triangle B \Rightarrow (x \in A \backslash B$ `or` $x \in B \backslash A)$

```
class main extds Object{
 bool varx;
 bool vary;
 before set(Now,_,_,varx,_),
        set(T,_,_,vary,_),
        isbefore(now,T){
  print("in advice")
 }
 bool main(bool x){
  this.varx := true;
  print("between assignments");
  this.vary := true;
  false
 }
}
```

Figure 5: Example for a "clairvoyant" aspect

```
Run #1 starting at 0
  => newObject(0, main)
  => calls(1, main, iota1, main, false)
  => set(3, main, iota1, varx, true)
between assignments
  => set(4, main, iota1, vary, true)
  => endCall(6, 1, false)
old act.points: ()
found act.points: (3)
new act.points:   (3)

Run #2 starting at 3
  => invokeAdvice(3)
in advice
  => endAdvice(4, 3)
  => set(5, main, iota1, varx, true)
between assignments
  => set(6, main, iota1, vary, true)
  => endCall(8, 1, false)
old act.points:   (5)
found act.points: (5)
new act.points:   (5)
Found fixed point after 2 runs.
Result is false
```

Figure 6: Example for an iteration

preter to a former state) directly at the step where its pointcut matched. This of course means to execute all queries at every step of computation. Since efficiency is not our primary consideration this may be tolerable, but it is not clear how and if this process can be described elegantly, for example in terms of domains as done above.

6. RELATED WORK

Walker and Viggers [8] discuss a kind of *temporal pointcuts*, called history patterns or *tracecuts*, to enrich the AspectJ [1] pointcut language with the abiblity to reason about former calls and their temporal relations. Moreover, data that has been passed as an argument can be accessed by the advice as it could be done via variable binding in our language. *Tracecuts* are patterns that are matched against a *history* of calls by a finite automaton. The implementation translates a program with tracecuts into AspectJ source code. However, the model of history patterns is not as rich as ours since it considers only method calls, whereas our approach may refer to almost any event in the computation.

Douence et al describe a pattern matching language based on Haskell [3] which allows pointcuts to relate different points in the execution history. The Java prototype uses an event monitoring system to accomplish pattern matching. The pointcut language used in this approach describes patterns as sequences of events. This is different to our language where the relation of joinpoints can be stated in a predicative way.

The work of Gybels and Brichau [5] is similar to our approach as they use logic programming and unification for pointcut matching. However, since the model behind their pointcut language does not cover the trace, it is not possible to encode pointcuts that relate different points in the execution. Furthermore the language only offers access to the current joinpoint, so it is not possible to access data from the store. Finally, the approach does not cover the usage of bound variables inside the advice.

Static analysis of aspect interaction is discussed by Douence et al [2], but their focus lies on detecting overlapping shadows of different aspects. They argue that aspects should be *orthogonal*, that means not covering the same join points, independent of the base program they are used with.

7. CONCLUSIONS

We have presented a powerful pointcut language, that makes it easy to write pointcuts that can reason about the execution trace and temporal relations between join points (facts) on a very abstract level. Our approach is so general that even referring to future events is possible. However, our results so far have some serious limitations. We need to find less restrictive ways to ensure termination of the advice application process. An efficient implementation and sophisticated tools for static analysis are also part of future work.

8. REFERENCES

[1] AspectJ homepage, 2003. http://aspectj.org.

[2] 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)*, volume 2487 of *LNCS*. Springer-Verlag, 2002.

[3] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proc. of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *LNCS*. Springer-Verlag, 2001.

[4] S. Drossoupolou. Lecture notes on the L2 calculus. http://www.doc.ic.ac.uk/~scd/Teaching/L1L2.pdf.

[5] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69. ACM Press, 2003.

[6] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 1999.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP '01*, 2001.

[8] K. V. Robert J. Walker. Communication history patterns: Direct implementations of protocol specifications. Technical report, 2004.

[9] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1994.

[10] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12)*, 2004.