

# Towards An Open Trace-Based Mechanism

position paper

Paul Leger    Éric Tanter  
PLEIAD Laboratory  
Computer Science Department (DCC)  
University of Chile – Santiago, Chile  
<http://www.pleiad.cl>

## ABSTRACT

Real-world applications have to deal with issues related to security, as well as errors and crosscutting concerns. Different trace-based mechanisms with distinctive features have been proposed to solve these particular issues. For example, PQL matches sequence of unordered events, and tracematches match traces expressed with regular expressions. Despite that applications present these issues at the same time, there is not a single trace-based mechanism that supports the distinctive features of current mechanisms. Besides, lack of an expressive trace-based mechanism does not permit to include new features, therefore, developers end up “code around” these mechanisms to satisfy particular needs. In this position paper, we compare and relate the specific characteristics of current trace-based mechanisms. Finally, we present a model for an open trace-based mechanism.

## 1. INTRODUCTION

Nowadays, real-world applications have to deal with issues related to security, as well as errors and crosscutting concerns. *Trace-based Mechanisms* (TMs for short) have shown their usefulness to solve some of these issues [1, 3, 5, 8, 10]. A TM observes the execution of the software at runtime and (possibly) executes a code fragment when the TM matches a specified trace of the execution. The researchers have proposed specific TMs to resolve these particular issues. Unsurprisingly, applications present these issues at the same time, therefore, they need to use different TMs at the same time, because there is not a single TM that supports the distinctive of current mechanisms. Besides, lack of an expressive TM does not allow developers to include new features, therefore, they end up “code around” these mechanism in contort ways. In this position paper, we first describe an abstract operational model of TMs (Section 2), which we then use to relate and compare the specific characteristics of existing TMs (Section 3). Section 4 describes the design of an open TM based on this abstract model. The open model is formulated in a class-based object-oriented setting and follows the design guidelines of open implementations [7]. We illustrate the range of openness of the model by describing several concrete extensions. For example:

- Express sequences using *operators*, which can enjoy all the power of the base language to be defined.
- Manage expressively the granularity of the matching, from a particular sequence to all sequences of a TM.
- Control expressively the multiplication of sequences of a TM.

## 2. AN OVERVIEW OF TMS

In this section, we describe an abstract operational model of TMs. We divide this abstract model in two parts: first, we describe

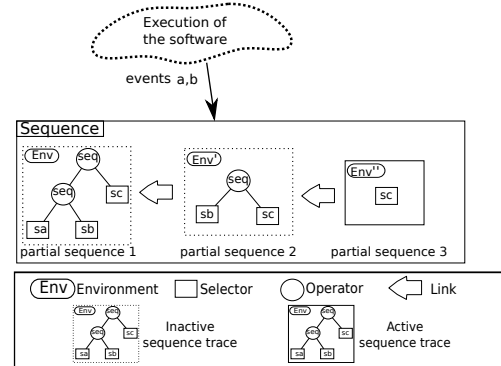


Figure 1: A sequence in action.

the abstract model necessary to match a specified trace of execution. Later, we extend this abstract model to manage and control the matching of several traces of executions at the same time.

### 2.1 Matching a Trace

Figure 1 shows that the execution of the software has generated the trace of events *a b* and that the *sequence* has matched these two events. According to the figure, the sequence needs to match the *c* event to finish the matching. When the sequence finishes the matching, our model executes an extra piece of code similar to an advice in AspectJ [6]. For space reasons, we only focus on the matching of sequences in this paper.

A sequence matches a trace of execution, which in this case is the trace composed of events: *a b c*. In addition, a sequence contains a set of linked *partial sequences*. A partial sequence represents the trace of execution that the sequence need to match. The first partial sequence represents the *whole* sequence and the last represents the last event that the sequence needs to match. Only active partial sequences are only evaluated and so can match events. The set of partial sequences represents the history of the matching of a sequence, which we then use to support the multiple matching of sequences. A partial sequence is composed of an Abstract Sequence Tree (AST), like AST for source code of a programming language, and an *environment* of bindings. In this AST, *selectors* are the terms and *operators* are combinators. Selectors match single events of the trace of execution and operators combine partial sequences. The environment contains a set of bindings available for a partial sequence and for the execution of the extra piece of code when the sequence matches. Every time that a partial sequence matches an event, the root operator of the AST creates a new par-

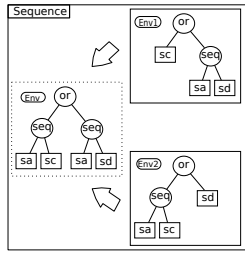


Figure 2: The non-deterministic effect of the Or operator.

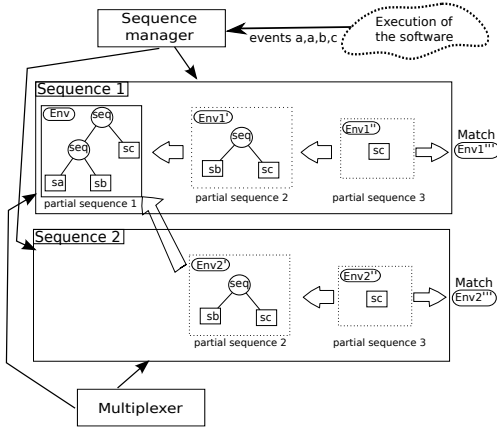


Figure 3: A TM in action.

tial sequence that is based and linked with the previous one. This new partial sequence contains a modified environment and represents the remaining sub-sequence of the sequence. A sequence can activate and deactivate its partial sequences. For example, when the partial sequence 2 matched the  $b$  event, the partial sequence 3 was created; and the sequence activated the partial sequence 3 and deactivated the partial sequence 2. Although the *whole* sequence did not match with the  $b$  event, this sequence advanced because now this sequence only has to match the  $c$  event to match entirely.

Operators can create one or more partial sequence when an event is matched. For example, Figure 2 shows a single sequence that has a partial sequence with the or operator over two sequences:  $a$   $c$  and  $a$   $d$ . When the  $a$  event occurs, two different histories can happen: the first or second sequence advance in the matching. It is so because the or operator is non-deterministic and so produces two different histories of the matching of the same sequence.

## 2.2 Matching Several Traces

Now, we extend this abstract model to manage and control the matching of several sequences inside a TM. Figure 3 shows a new scenario and two new components of our model: *sequence manager* and *multiplexer*. In the figure, the trace of execution has generated the events  $a$   $a$   $b$   $c$  and the sequence manager has received and sent these events to the two sequences, which has matched.

The sequence manager manages the matching of all sequences inside a TM. For instance, this components permits to abort or reset the matching of all or a specific group sequences. In this case, the sequence manager only sends the events to sequences.

The multiplexer controls the multiplication of sequences. When a sequence generate an *equivalent* partial sequence the multiplexer

decides if this equivalent partial sequence becomes in a new sequence. For example, Figure 3 shows that a sequence is multiplexed into two, which end up matching. This double matching is due to four reasons: *i*) the trace of execution generates twice the  $a$  event, *ii*) the sequence 1 always has the partial sequence 1 activated, *iii*) the multiplexer decides to create a new sequence when the sequence 1 matches the second  $a$  event and creates an equivalent partial sequence 2, *iv*) and when the trace of execution generates finally the events  $b$  and  $c$ , sequences 1 and 2 advance up to the matching. In other words, when the execution of the software generates the second  $a$  event, the partial sequence 1, which is activate, matches the event, therefore, creates a new partial sequence 2. As there are two equivalent partial sequences, the multiplexer decides to create another sequence with the second partial sequence 2.

It is important to note that the nondeterminism differs from the multiplication of sequences because the first generates different histories of the same sequence using the same event; instead the second generates different histories in different sequences using different events.

In this section, we appreciate six components of TMs: environments, selectors, operators, sequences, sequence managers, and multiplexers. In next section, we will compare and relate the expressiveness of these components in existing TMs.

## 3. THE EXPRESSIVENESS OF TMS

TMs [1, 3, 5, 8, 10] vary in their expressiveness in terms of environments, selectors, operators, sequences, sequence managers, and multiplexers. In this section, we relate and compare the expressiveness of these components.

**Environment.** An environment of bindings represents the contextual information associated to the sequence. The expressiveness of environments allows developers to match more precisely a trace of execution and to provide more contextual information (values) to the execution of the code fragment when the sequence matches. In tracematches [1], the manipulation of environments is limited because it only permits to bind information related to the event and compare implicitly this information using the equality operator. Environments of Alpha [10] can only contain information related to events. Halo [5] allows developers to contain contextual information from any source (not only from the event), and then comparing this information explicitly.

**Selector.** A selector matches single events. The precision to match events depends on the granularity of the event model of the base language and the expressiveness to define selectors. Although this expressiveness varies, most of current TMs [1, 3, 5, 8, 10] cannot use the power of the base language to define selectors. For instance, in tracematches and Halo, selectors are pointcuts defined in a dedicated declarative language<sup>1</sup>. Selectors of Alpha are Prolog queries, which differs from the base language (a Java subset). In PTQL [3], selectors are fields of a register of a data base.

**Operator.** An operator relates selectors and/or operators, therefore, it permits to define partial sequences. The expressiveness to define operators varies according to TMs. For example, the expressiveness of selectors of Alpha is enough to express operators because it can express sequences using Prolog queries<sup>2</sup>. In Halo, operators

<sup>1</sup>In tracematches, a selector is really a symbol that is composed of a pointcut and a modifier of the event.

<sup>2</sup>To be more precise, the selectors are facts and the operators are rules.

are Lisp functions defined by the `def-rule` construct. The operators of PTQL are SQL operators like `or` and `and`. Tracematches match traces of execution using regular expression operators. For example, if the alphabet is  $\{sa, sb\}$  and the regular expression of the sequence is  $sa\ sa\ sa$  and the regular expression of the trace of execution is  $sa\ sa\ sb\ sa$ , so tracematches do not match this trace because the  $sb$  symbol of the execution is not in the regular expression of the sequence. In a nutshell, the regular expression of the trace of execution must happen exactly as the sequence is defined.

**Sequence.** Although an environment and an AST of the sequence define a partial sequence, they do not define entirely the process of matching of a sequence. A sequence handles the set of partial sequences to carry out the matching and the history of this matching. Sadly, to the best of our knowledge, there is no TM that allows developers to reason or reflect about the matching of a particular sequence. Reasoning about a sequence permits, for example, to abort or reset the matching of a particular sequence if some condition is satisfied. Concretely, consider the familiar example for AOP community: *autosave*. A document is automatically saved if it is edited a number of times (e.g. say three) without being saved. The wanted sequence should match when the trace of execution generates three events of edition, but this sequence should abort if it matches the sequence composed of the *save* event because the document has already saved.

**Sequence manager.** The sequence manager manages the matching of all sequences inside a TM. To the best of our knowledge, there is no TM that permits to manage this component. Reason about the matching of sequences permits, for example, to abort or reset the matching of all sequences if some condition is satisfied. Concretely, controlling the sequence manager can be useful in areas like security. For instance, a sequence that represents a protocol of *light security* could change to *heavy security* whenever another sequence matches. This example shows that controlling the sequence manager permits to obtain behavior similar to *morphing aspects* [4] in TMs.

**Multiplexer.** The multiplexer controls the multiple matching of sequences. In most of these mechanisms [3, 5, 8, 10], the sequence are always multiplexed. Tracematches are a particular case because a sequence is only duplicated when two equivalent partial sequences have environments of bindings with different values. The multiple matching is used, for example, in tracematches resolve to the problem of the observer pattern [2] because this TM matches multiple sequences that binds different values in the *subject* and *observer*.

## 4. A MODEL FOR AN OPEN TM

This section presents the design of an open TM, which is based on the abstract operational model presented in Section 2. The model follows the design guidelines of the open implementations [7] because this model allows developers to control its implementation strategy by the an expressive use of environments, selectors, operators, sequences, sequence managers, and multiplexers. In similar way to Section 2, this model is split into two: *partial sequence* and *sequence manager*. The first model permits define sequences by the use of environments, selectors, and operators. The second model permits to describe sequence managers, which are used to manage and control the multiple matching of the sequences.

### 4.1 Defining a Partial Sequence

As mentioned in Section 2.1, a partial sequence represents the

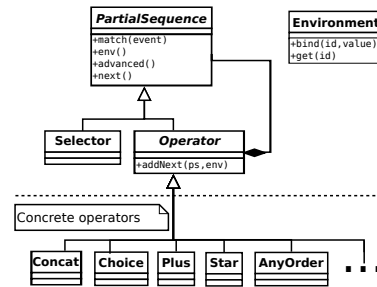


Figure 4: The class diagram of a partial sequence.

trace of execution that the sequence need to match. A partial sequence is a structure composed of an environment and an AST of the sequence. This AST settles the relationships between selectors and operators. The environment of bindings defines the set of values available in a partial sequence. In this model, the expressiveness of selectors depends on the event model of the base language, but the expressiveness of the operators depends on the base language, which is generally Turing Complete.

Figure 4 shows the class diagram of a partial sequence. This diagram uses the composite pattern [2] between `PartialSequence`, `Selector`, and `Operator`. The `Selector` class represents selectors that only match single events. The `Operator` class represents operators that match compositions of partial sequences. This class is an abstract class, which is used to implement sub-classes that provide specific and diverse kinds of operators. For example, Figure 4 shows five operators: four to match regular expressions and one to match traces in any order. The `Environment` class represents environments that permit to bind and get values.

The `match` method takes an event and returns true or false whether it matches or not a single event (in the selector case) or a partial sequence (in the operator case). The `env` method returns the environment that contains the bound values until this point of the sequence. If a partial sequence does not match but advance (like Figure 1), the `advanced` method returns true. The `next` method returns the next partial sequence. Finally, the `addNext` method of the `Operator` class adds the next partial sequence (`ps`) with the associated environment (`env`). This latter method determines which is the next partial sequence in the process of matching of a sequence.

The protocol of use is the following: when a `PartialSequence` object sets the next partial sequence with itself, we will say the matching of the sequence did not advance. Instead, when a `PartialSequence` object sets or adds the next partial sequences with different objects, we will say the matching of the sequence advanced. As examples, we present the implementation of the `Concat` and `AnyOrder` classes.

Figure 5 shows the `Concat` class, which represents the operator that matches the sequence of two traces, where both traces are matched by the left and right partial sequences. The `match` method adds right as the next partial sequence if left matches with the event. However, if left does not match, it could have advanced in its matching. In this case, `match` adds as next a new `Concat` object that contains the `left.next()` as left and right as right. A new `Concat` object is created to maintain the history of the matching of the sequence through the two different partial sequences (before and after the event).

It is important to note, the `match` method of `Concat` class always returns false because the responsibility of the matching is delegated to the right partial sequence.

The following code is useful if we want to match the sequence

```

class Concat extends Operator {
  PartialSequence left, right;
  Seq(PartialSequence left, PartialSequence right) {...}

  boolean match(Event event) {
    Env env = env();
    left.setEnv(env);
    if (left.match(event)) { //left matched, continue with right
      addNext(right, left.env());
    }
    else if (left.advanced()) { //left only advanced
      addNext(new Seq(left.next(), right), left.next().env());
    }
    else {
      addNext(this, env);
    }
    return false;
  }
};

```

Figure 5: The Concat class.

composed of  $a b c$  events, where events are calls to functions  $a$ ,  $b$ , and  $c$ :

concat = new Concat(new Concat(sa, sb), sc);  
 The sa, sb, and sc objects are selectors that match the calls to aforementioned functions. The concat object is a partial sequence that represents the matching of the wanted sequence.

The anyOrder operator matches an unordered sequences of traces, where these traces are matched by a set partial sequences. This operator is non-deterministic like the *or* operator (Section 2) because the two or more partial sequences can match or advance with the same event, therefore, generating different histories of the matching. Figure 6 shows the implementation of the AnyOrder class that represents the AnyOrder operator. The match method, first, verifies if there is only a partial sequence and tries to match this last partial sequence. The match method returns true if the ps matches, but returns false otherwise. However, if ps only advanced, the match method adds ps.next() as next, which has the responsibility to finish the matching of the unordered sequences of traces.

When there are two or more partial sequences in the pss array, the match method tries to match every partial sequence (ps) of the pss array. The method adds a new AnyOrder object with the same pss array without ps if it matched; instead match adds a new AnyOrder object with the same pss array, but exchanges ps for ps.next() if ps only advanced.

The following code is useful if we want to match the unordered sequence composed of the events  $a$ ,  $b$ , and  $c$ :  
 AnyOrder anyOrder = new AnyOrder(new PartialSequence[] {sa, sb, sc});  
 The anyOrder object is a partial sequence that represents the matching of the disorderly aforementioned events.

## 4.2 Defining a Sequence Manager

The previous section showed how to build partial sequences. In this section, we describe our sequence manager model, which takes a partial sequence to manage and control the matching and the multiplication of sequences generated from this partial sequence.

Figure 7 shows the class diagram of our sequence manager. In this diagram, three classes are the core: Sequence, Multiplexer, and SequenceManager. The Sequence class represents a sequence, which contains a set of (active and inactive) partial sequences. The Multiplexer class controls the multiplication of sequences. Finally, the SequenceManager class manages the matching of sequences.

### 4.2.1 Sequence

A sequence can be declared using the following code:  
 Sequence s = new Sequence(ips, as);

```

class AnyOrder extends Operator {
  PartialSequence[] pss;
  AnyOrder(PartialSequence[] pss) {...}

  boolean match(Event event) {
    Env env = env();
    if (pss.length == 1) { //only one partial sequence remains
      PartialSequence ps = pss[0];
      ps.setEnv(env);
      if (ps.match(event)) { //last partial sequence matched
        setEnv(ps.env());
        return true;
      } else if (ps.advanced()) { //delegate the matching to ps.next()
        addNext(ps.next(), ps.next().env());
        return false;
      }
    }

    for(int i = 0; i < pss.length; ++i) {
      PartialSequence ps = pss[i];
      ps.setEnv(env);
      if (ps.match(event)) { //ps[i] matched
        addNext(new AnyOrder(pss.remove(i)), ps.env());
      } else if (ps.advanced()) { //ps[i] advanced
        addNext(new AnyOrder(pss.set(i, ps.next()), ps.next().env());
      }
    }

    if (!advanced()) { //no partial sequence matched nor advanced
      addNext(this, env);
    }

    return false;
  }
};

```

Figure 6: The AnyOrder class.

where ips is the initial partial sequence of the sequence and as is an ActivationStrategy object that represents the activation strategy of partial sequences. Every ActivationStrategy object has the activate method that is parameterized by a ps partial sequence. This method returns true if ps must active, otherwise the method returns false. We provide three default strategies:

Object	Strategy	It permits to ...
FIRSTANDLAST	First and last partial sequences are only active.	begin more than one sequence at the same time.
LAST	Last partial sequence is only active.	begin only one sequence at the same time.
ALL	all partial sequences are active.	multiplex sequences at any time.

### 4.2.2 Multiplexer

The multiplexer controls the multiplication of the sequences. The class diagram of Figure 7 shows the Multiplexer abstract class. This class is used to implement sub-classes that provide specific strategies of multiplication of sequences. In this paper, we provide two sub-classes: Multiple and Tracematch. The objects of Multiple class always multiplexes a sequence when it finds an equivalent partial sequence. A case more refined is the Tracematch class because it emulates the behavior found in tracematches [1].

The multiplex method takes a s sequence and a pss array of equivalent partial sequences found in s. The goal of this method is to decide which partial sequences of pss become sequences, which are returned in an ArrayList object. As examples, we present the implementation of the Multiple and Tracematch classes.

Figure 8 shows the Multiple class. The multiplex method creates new sequences of all equivalent partial sequences of pss with ALL object as activation strategy.

The semantic of multiplication of tracematches always permits to

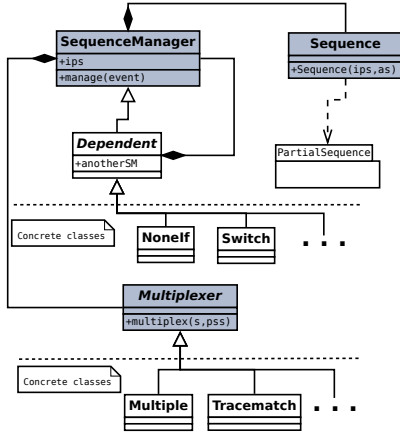


Figure 7: The class diagram of a sequence manager.

```
class Multiple extends Multiplexer {
    ArrayList multiplex(Sequence s, PartialSequence[] pss) {
        ArrayList ss = new ArrayList();
        for (int i = 0; i < pss.length; ++i)
            ss.add(new Sequence(pss[i], ActivationStrategies.ALL));
        return ss;
    }
}
```

Figure 8: The Multiple class.

begin a new sequence, and this semantics multiplexes a sequence if two equivalent partial sequences differ in the values of their environments. Figure 9 shows the TraceMatch class. The compareEnvs method verifies if two environments have the same set of values. The getEquivalentents method gets all equivalent partial sequences of ps in s. The previousIsFirstOfSeq method verifies if pss[i] is linked to the first partial sequence of the sequence. The multiplex method represents the same semantic of tracematches. The method gets all equivalent partial sequences of every element of pss, and for every equivalent pair, multiplex creates a new sequence if both equivalent partial sequences have different set of values in their environments. A new sequence is also created if pss[i] represents the initial partial sequence of the sequence.

#### 4.2.3 Sequence Manager

The SequenceManager class defines how to manage the matching of all sequences inside a TM. In Figure 7, we can see that SequenceManager has the instance variable ips, which represents the initial partial sequence of the sequence. In addition, this variable is used as a seed to create sequences that begin from the outset. The manage method manages the matching of all sequences. This method takes an event and returns an ArrayList object with the environments of the matched sequences. This class has the Dependent abstract class, which refines the behavior of the manage method for that depending on a specified trace of execution. Besides, this abstract class is used to implement sub-classes that provide specific strategies to manage matching of sequences. For example, the Nonelf class removes all sequences if it matches another trace of execution, or the Switch class that removes all sequences and uses a different initial partial sequence as a seed if it matches another trace of execution. As example, we explain the Nonelf class.

Figure 10 shows the Nonelf class. The manage method, first,

```
class Tracematch extends Multiplexer {
    boolean compareEnvs(Env env1, Env env2) {...}
    PartialSequence[] getEquivalentents(Sequence s, PartialSequence pt) {...}
    boolean previousIsFirstOfSeq(PartialSequence ps) {...}

    ArrayList multiplex(Sequence s, PartialSequence[] pss) {
        ArrayList ss = new ArrayList();
        for (int i = 0; i < pss.length; ++i) {
            PartialSequence[] epss = getEquivalentents(t, pss[i]);
            for (int j = 0; j < epss.length; ++j)
                if (!compareEnvs(epss[j].env(), pss[i].env()) ||
                    previousIsFirstOfSeq(pss[i]))
                    ss.add(new Sequence(pss[i], ActivationStrategies.ALL));
        }
        return ss;
    }
}
```

Figure 9: The Tracematch class.

```
class Nonelf extends Dependent {
    ArrayList manage(Event event) {
        ArrayList envs = new ArrayList();
        ArrayList ss = getSequences();
        if (!anotherSM.match(event))
            return super.manage(event);

        ss.removeAll();
        ss.add(new Sequence(ips, ...));
        return new ArrayList();
    }
}
```

Figure 10: The Nonelf class.

verifies whether anotherSM matches or not with the event. In the case that anotherSM matches, the manage method removes all sequences and create a new sequence from ips. If anotherSM does not matches, this method calls the manage method of the super class.

## 5. CONCLUSION

In this position paper, we described an abstract operational model of TMs and use this abstract model to relate and compare the specific characteristics of existing TMs like PQL, PTQL, Halo, and Alpha. Later, we presented the design of an open TM based on this abstract model. The open model is formulated in a class-based object-oriented setting and follows the design guidelines of open implementations. This model is split into partial sequence and sequence manager model. The first model defines sequences and the second model, taking a partial sequence, defines how to manage and control the matching and multiplication sequences. Both models allows developers to specific strategies of implementation. We showed the openness of our model through concrete and expressive extensions.

Our model has different kinds of challenges. Some challenges are related to its practical adoption. For this reason, as future work, we plan to extend AspectScript [11], an AOP extension of JavaScript, to develop real-world applications that need our model to solve different the issues mentioned in the introduction. Other challenges are related to the understanding of relationships between sequences, multiplexers, and sequence managers. For example, there is a coupling between multiplexers and sequences because a multiplexer needs that sequences uses certain activation strategies to multiplex. Finally, some challenges are related to the static analysis of the sequences due to the highly dynamicity of their operators, *e.g.* consider the random operator that always returns a random partial sequence as next.

## 6. REFERENCES

- [1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In OOPSLA 2005 [9], pages 345–364. ACM SIGPLAN Notices, 40(11).
- [2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, October 1994.
- [3] Simon F. Goldsmith, Robert O’Callahan, and Alex Aiken. Relational queries over program traces. In OOPSLA 2005 [9], pages 385–402. ACM SIGPLAN Notices, 40(11).
- [4] Stefan Hanenberg, Robert Hirschfeld, and Rainer Unland. Morphing aspects: incompletely woven aspects and continuous weaving. In Karl Lieberherr, editor, *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 46–55, Lancaster, UK, March 2004. ACM Press.
- [5] Charlotte Herzeel, Kris Gybels, and Pascal Costanza. A temporal logic language for context awareness in pointcuts. In Dave Thomas, editor, *Workshop on Revival of Dynamic Languages*, number 4067 in Lecture Notes in Computer Science, Nantes, France, July 2006. Springer-Verlag.
- [6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In Jorgen L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.
- [7] Gregor Kiczales, John Lamping, Cristina V. Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering (ICSE 97)*, pages 481–490, Boston, Massachusetts, USA, 1997. ACM Press.
- [8] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In OOPSLA 2005 [9], pages 365–383. ACM SIGPLAN Notices, 40(11).
- [9] *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2005)*, San Diego, California, USA, October 2005. ACM Press. ACM SIGPLAN Notices, 40(11).
- [10] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In Andrew P. Black, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of LNCS, pages 214–240. Springer-Verlag, 2005.
- [11] Rodolfo Toledo, Paul Leger, and Éric Tanter. AspectScript: Expressive aspects for the Web. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, Rennes and Saint Malo, France, March 2010. ACM Press. To Appear.