An Open Trace-Based Mechanism

Paul Leger **Éric Tanter**

[1]PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile, Santiago, Chile

http://pleiad.cl

Abstract. *Trace-based Mechanisms (TMs) support the definition of programs that observe and react to a program execution; they have numerous applications in domains like error detection, security, and modular definition of crosscutting concerns. Various TMs have been proposed, specifically tailored to address a particular domain; each of them have different features and semantics, which are not configurable. Unfortunately, applications typically need to handle errors, security and crosscutting concerns in concert. However, there is no single TM that allows programmers to take advantage of the specific features and semantics of existing TMs, as well as devise new variants. As a consequence developers must "code around" TMs in contort ways to satisfy their specific needs. In this paper, we present an open implementation of a trace-based mechanism, called OTM, integrated in JavaScript. In OTM, sequences of events are defined using plain objects, and crucial semantic elements like multiple matching and life cycle of sequences are open to customization.*

1. Introduction

Trace-based Mechanisms (TMs for short) support the definition of programs that observe and react to a program execution; they have numerous applications in domains like error detection, security, and modular definition of crosscutting concerns. Various TMs have been proposed, specifically tailored to address a particular domain [Allan et al. 2005, Goldsmith et al. 2005, Herzeel et al. 2006, Martin et al. 2005, Ostermann et al. 2005]. A TM observes the execution trace of a program at runtime and typically executes a piece of code when a given pattern occurs in that trace. In the rest of the paper, we use the term *execution trace* to refer to the actual trace of events generated by the execution of a program and the term *sequence* to refer to the pattern of the trace that should be matched. In line with aspect-oriented programming [Kiczales et al. 1997b], we name *advice* the piece of code triggered when a sequence matches.

Each of these mechanisms have different features and semantics, which are not configurable. Unfortunately, applications typically need to handle errors, security and crosscutting concerns in concert, and so it becomes interesting to combine some of these approaches. However, there is no single TM that allows programmers to take advantage of the specific features and semantics of existing TMs, as well as devise new variants. As a consequence developers must "code around" TMs in contort ways to satisfy their specific needs. Extending our recent work [Leger and Tanter 2010], which are the guidelines for this proposal, we present an Open Trace-based Mechanism (OTM). OTM adopts object-oriented programming and open implementation [Kiczales et al. 1997a] as

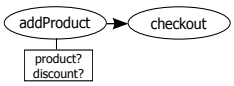

Figure 1. The sequence for the discount feature.

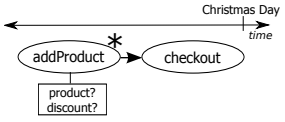

Figure 2. The sequence for the *restrictive* discount feature.

Figure 3. A possible execution trace of the Boutique Web application.

its core design principles. A program that supports open implementation allows developers to customize strategies of implementation of the program, and at the same time, still hiding details of its implementation. OTM permits to support the different features and semantics of current TMs as well as to devise new variants. OTM is implemented for AspectScript [Toledo et al. 2010], an aspect-oriented programming extension of JavaScript.

In OTM, sequences are defined using plain JavaScript objects, bringing the full benefits of the base language (in particular, the combination of higher-order functions and objects) to define sequences that match execution traces. In addition, OTM allows a developer to implement its own *multiple matching strategy*, which defines the multiplicity of simultaneous matching of a sequence. Besides, OTM permits to define the *life cycle strategy* of a sequence. This feature is useful to specify, for instance, that a sequence is aborted when some external condition is satisfied, *e.g.* lifetime of a sequence. These strategies configure important semantic features that are hard-coded in existing TMs.

1.1. Variations in Semantics

Let us consider a simple Web application, like offered by Amazon, to illustrate the semantic variety of existing TMs. The *Boutique* store sells clothing online: the catalogue can be viewed on a website, and clients can place their orders online. A client adds products from the catalogue to a virtual shopping cart in a dedicated page, which displays each product added with its price. Besides, this page contains a checkout form asking for a desired payment method. Now, consider that we need to add the *discount* feature, which potentially adds a different discount for each product when it is added to the shopping cart. The discounts are applied when the client does checkout. Each discount is only valid for a period of time; however, the discount feature must respect the discount that was active when the product was *added* to the shopping cart. Given this Web application, the discount feature is clearly a *crosscutting concern* that can be modularized by existing TMs.

Figure 1 shows the sequence necessary to implement the discount feature. The sequence matches calls to function addProduct followed by checkout. When this sequence matches, the associated advice is executed and applies the corresponding discount to the product. This feature can also be implemented by two simple aspects, however, the first aspect must have a *stateful advice* to store the references to product and discount when this aspect matches the call to addProduct; and the second aspect, which matches the call to

checkout, must be *aware* of the first to apply the corresponding discount.

Figure 3 shows a possible execution trace of the Web application. Depending on the TM semantics, different discounts could be applied due to the number of times the sequence matches: *zero* if the sequence definition means that no event can occur between the calls to addProduct and checkout, *one* if the TM only supports matching a single sequence at a time, and *three* if the TM supports multiple matching, as in tracematches [Allan et al. 2005]. Actually, even if the TM supports multiple matching, it may do so in a different way. For instance, with Halo, the above example results in *two* matches [Herzeel et al. 2006]. This is because Halo only matches simultaneous sequences when their associated contextual information differs. Here, the sequence 1 and 2 contain different contextual information; conversely the sequence 1 and 3 contain the same contextual information.

1.2. Expressiveness

Let us now look at an extension of the previous example that highlights the need for new semantic variants of trace-based mechanisms. The Boutique store has an additional policy (updating the name of the feature to *restrictive* discount), according to which a limited number of discounts are really applied (say three): they are the discounts that generate the best benefits for the client. In addition, this policy only permits to apply the discounts if the client does checkout before Christmas Day. The sequence of Figure 1 does not work because only some matched sequences must eventually apply the discount; *i.e.* the three sequences whose contextual information contain the product and discount that generate the best benefits compared to the rest. The sequence shown in Figure 2 implements this new feature in a hypothetical TM that matches this sequence before Christmas Day and with a matching strategy that matches only one sequence at a time. When this sequence matches, it contains as contextual information a *list* of all products and discounts. The associated advice applies the discounts to the products of the list that generate the best benefits according to the store policy. Existing TMs cannot implement this feature using a single sequence because they cannot define sequences that gather lists of bindings as contextual information. The current solution in existing TMs is to implement this feature using stateful advices, which store each individual product and discount on purchase, and an additional aspect that eventually applies the chosen discounts on checkout if the date is before Christmas Day. With OTM, as will be illustrated in Section 5, the restrictive discount feature is straightforward to express.

1.3. Contributions

OTM is an open trace-based mechanism that allows developers to instantiate their own TMs to suit their specific needs. Our proposal is focused on opening the following points:

Sequence definition. Most TMs define sequences using a domain-specific language, which is typically not Turing complete. Instead, OTM allows developers to define sequences using the full power of the base language. In addition, OTM permits to reason about the matching of sequences because they are first-class values. For example, it possible to know if a sequence defined as $a^\star$ matched with the execution trace: *a* or *a a*.

Multiple matching strategy. As illustrated, almost all TMs have different multiple matching strategies that are not configurable, and can yield drastically different results. OTM allows developers to define and configure their own multiple matching strategies.

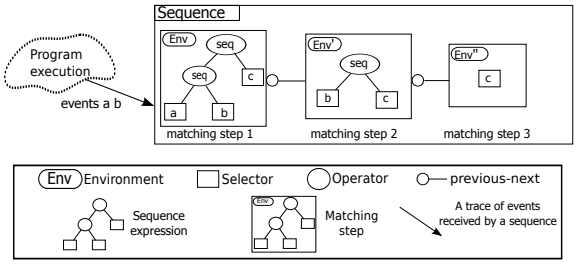

Figure 4. Matching a execution trace.

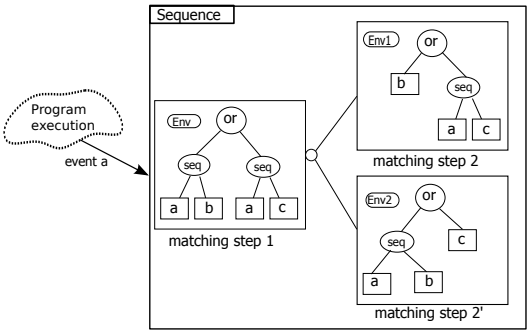

Figure 5. The non-deterministic effect of the Or operator.

Life cycle strategy. Unlike current TMs, OTM allows developers to control the life cycle of sequences. For example, OTM can remove or match all (or a group of) sequences if some external condition is satisfied, *e.g.* the lifetime of a sequence.

In addition, OTM is the first concrete TM for the JavaScript language. To introduce OTM, this paper puts a strong emphasis on understanding the different elements that compose a TM; these are analyzed in Section 2. Section 3 discusses existing TMs based on these elements. Section 4 presents the model and the implementation of OTM. Section 5 explains the implementation of restrictive discount feature using OTM. Section 6 concludes and discusses future work.

Availability. OTM, along with the examples presented in this paper, is available online at http://pleiad.cl/otm

2. An Overview of TMs

In this section, we describe an abstract operational model for TMs. We divide this abstract model into three parts. First, we describe the abstract model necessary to define a sequence. Second, we extend the model with the life cycle concept of a sequence. Finally, we introduce the multiple matching concept of a sequence into this model.

2.1. A Simple Sequence

Figure 4 shows the execution trace of a program that generates the events *a b*, and a *sequence* that matches the execution trace *a b c*. A sequence *accepts* events, as a consequence of which it *advances*, until it finally *matches* the execution trace. According to the figure, the sequence has accepted the events *a* and *b* and it still needs to accept the *c* event to match. When the sequence matches, the sequence triggers the execution of an advice, which uses the environment of bindings (contextual information) returned by this sequence. Next, we explain each element of Figure 4.

A sequence is composed of a set of linked *matching steps*, which represents the history of the matching of a sequence. The first matching step (matching step 1) represents the *root* of the sequence and the last matching step represents the last event that the sequence needs to accept to match.

A matching step is composed of a *sequence expression* and an *environment* of bindings. A sequence expression describes by construction an execution trace without

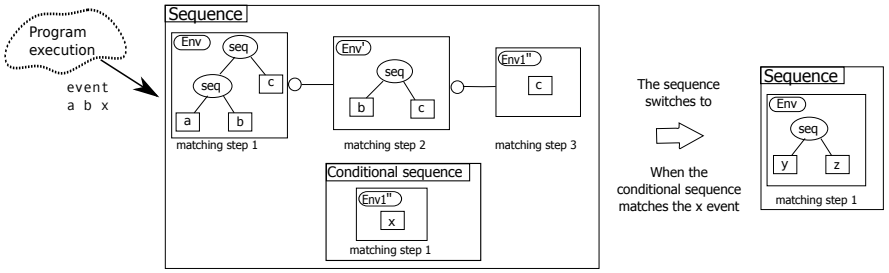

Figure 6. The switch of a sequence.

the contextual information (*i.e.* the static part of the trace), and an environment of bindings describes the contextual information of the trace (*i.e.* the dynamic part of the trace). In a sequence expression, *selectors* match single events of the trace and *operators* combine sequence expressions. The environment contains the set of bindings available for the matching step and for the execution of the advice. Every time a matching step is evaluated with an event and accepts this event, the matching step returns either the next matching step(s), in which case the sequence advances, or an environment if there is no next matching steps, in which case the sequence matches.

The nondeterminism of the matching. The evaluation of a matching step can create several matching steps if the sequence expression of this matching step contains a nondeterministic operator. For example, Figure 5 shows a single sequence, whose sequence expression of matching step 1 contains the or operator over two sequence expressions: *a b* and *a c*. When matching step 1 accepts the *a* event, two different matching steps are created: the first contains the sequence expression *a c* and the second contains the sequence expression *a b*.

2.2. The life cycle of a sequence

A *life cycle* strategy controls the evolution of the matching of a sequence passes until it matches. Existing TMs have specific and non-configurable life cycle strategies. For example, tracematches [Allan et al. 2005] allow developers to abort the matching of a sequence if it matches some specific event of the execution trace that is not in the sequence definition. Customizing the life cycle strategy can be useful in diverse areas like security. For instance, a sequence that represents a protocol of light security could switch to heavy security whenever a given conditional sequence matches.

Figure 6 shows the sequence that matches an execution trace *a b c* with a customized life cycle strategy. This life cycle strategy emulates the previous security example: it removes all matching steps and creates a new first matching step using a replacement sequence expression whenever a conditional sequence matches. For example, the figure shows that the execution trace that the sequence had to match switches to the execution trace *y z* when the conditional sequence matches the *x* event.

2.3. The multiple matching of a sequence

The *multiple matching* strategy controls the number of simultaneous matching of a sequence in an execution trace. Most TMs [Allan et al. 2005, Goldsmith et al. 2005, Herzeel et al. 2006, Martin et al. 2005, Ostermann et al. 2005] have specific multiple matching strategies, which are not configurable. For example, a sequence in

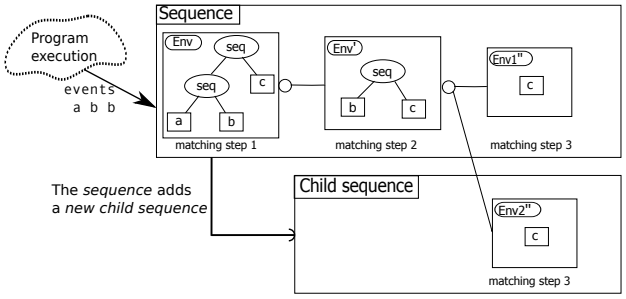

Figure 7. The creation of a child sequence.

Halo [Herzeel et al. 2006] simultaneously matches as patterns in Prolog, where these patterns in the left-hand side of rules may match a given term producing different bindings (or substitutions). Customizing the multiple matching strategy allows developers, for example, to declare a sequence that matches only once at a time; this strategy can be useful in cases like the Boutique Web application (Section 1.2).

A sequence evaluates all its matching steps for every event. If a matching step accepts two different events and creates two sets of matching steps, the sequence can add, according to its multiple matching strategy, a *new child* sequence that is composed of the second set of matching steps. Both the sequence and the new child sequence share a common prefix of matching steps. For example, Figure 7 shows a sequence that created a child sequence that shares the prefix of matching steps 1 and 2. In the figure, the sequence created a new child sequence because matching step 2 accepted two different *b* events and created two different matching step 3, and the sequence decided to create a new child sequence with the second matching step 3.

It is important to note that the nondeterminism discussed previously differs from the multiplicity of simultaneous matching of a sequence. The former generates different histories of the *same* sequence using the *same* event, whereas the latter generates different histories in *different* sequences using *different* events.

3. Related Work

In the previous section, we described the five elements that compose a TM: environments, selectors, operators, matching steps, and sequences. We now existing TMs in light of these elements.

Environment. An environment of bindings describes the contextual information of an execution trace. Expressive environments allow developers to match more precisely an execution trace and to provide more contextual information (values) to the execution of the advice when the sequence matches. In tracematches [Allan et al. 2005], the manipulation of environments is limited because it only permits to bind information directly related to events and to compare this information implicitily (using the equality operator). Environments in Alpha [Ostermann et al. 2005] can also only bind information related to events, however, the comparison is explicit; for example, developers can use a user-defined operator to compare information contextual. Halo [Herzeel et al. 2006] allows developers to bind information from any source (not only from events), *e.g.* time = getCurrentTime(). Like Alpha, the comparison of this information is explicit.

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 of the language use to define selectors. Although this expressiveness varies, most of current TMs [Allan et al. 2005, Goldsmith et al. 2005, Herzeel et al. 2006, Martin et al. 2005, Ostermann et al. 2005] 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[1]. Selectors of Alpha are Prolog queries, which differ from the base language (a Java subset). In PTQL [Goldsmith et al. 2005], selectors are fields of a register of a data base.

Operator. An operator combines sequence expressions to describe the execution trace without the contextual information that the sequence should match. Expressiveness of selectors of Alpha is enough to express operators because sequences can be expressed using Prolog queries[2]. The operators of PTQL are SQL operators like or and and. Halo can use the power of the base language to define operators, however, Halo cannot use the base language to define sequence expressions because selector definitions depend on a dedicated pointcut language. Tracematches sequences are expressed using regular expression operators. For example, if the alphabet is $\{sa,sb\}$ and the regular expression of the sequence is *sa sa* and the regular expression of the execution trace is *sa sb sa*, so tracematches do not match this trace because the *sb* symbol of the execution trace is not in the regular expression of the sequence. In a nutshell, the regular expression of the execution trace must happen exactly as the sequence is defined.

Matching step. The set of matching steps, where one is composed of an environment and a sequence expression, represents the history of the matching of a sequence. Sadly, to the best of our knowledge, there is no TM that allows developers to reason about the history of the matching of a sequence. Reasoning about the set of matching steps permits, for example, to know if a sequence defined as $a^\star$ matched with the execution trace: *a* or *a a* because it is possible to introspect what the execution trace is being matched.

Sequence. A sequence is the fundamental concept of TMs because it is used to match execution traces. Again, to the best of our knowledge, there is no TM that allows developers to reason about the matching of a sequence. Therefore, it is not possible to customize the strategies of life cycle or/and of multiple matching of a sequence.

4. Design & Implementation of OTM

This section presents the design and implementation of OTM, which is based on the abstract operational model presented in Section 2. OTM is developed for JavaScript as an AspectScript [Toledo et al. 2010] extension. The JavaScript language supports object-oriented programming based on prototypes, dynamic typing, and higher-order functions. As mentioned in Section 1, our proposal follows open implementation principles [Kiczales et al. 1997a] and object-oriented programming. OTM allows developers to define expressively the five elements that compose a TM: environments, selectors, operators, matching steps, and sequences. Similarly to Section 2, we divide the OTM explanation into four parts: the definition of a sequence, the creation of a sequence, the life cycle of a sequence, and the multiple matching of a sequence.

[1] In tracematches, a selector is really a symbol that is composed of a pointcut and a modifier of the event.

[2] To be more precise, the selectors are facts and the operators are rules in Prolog.

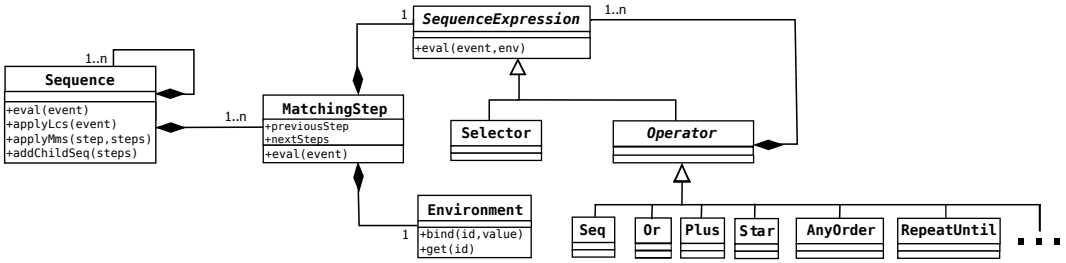

Figure 8. The OTM class diagram.

4.1. A Simple Sequence

As mentioned in Section 2.1, a sequence is composed of a set of matching steps, which represent the history of matching of the sequence. A matching step is composed of an environment of bindings, which describes the contextual information of an execution trace, and a sequence expression, which describes the execution trace without the contextual information. A sequence expression is composed of selectors and operators. In our model, the expressiveness of selectors depends on the event model of the base language, however, the expressiveness of operators depends on the power of the base language.

Figure 8 shows the OTM class diagram. A Sequence is composed of a set of MatchingSteps, an eval method, a life cycle strategy that is carried out by the applyLcs method, and a multiple matching strategy that is carried out by the applyMms method. In addition, a Sequence has the addChildSeq method, which adds a new child sequence to the sequence (more on this later).

A MatchingStep is composed of an Environment and a SequenceExpression. The Environment class represents environments, which bind and get values. The SequenceExpression class uses the composite pattern [Gamma et al. 1994] between the Selector and Operator classes to describe an execution trace. The Selector class represents selectors, which only match single events. The Operator abstract class represents operators and is used to combine sequence expressions. This abstract class is specialized through sub-classes that define specific operators. For example, Figure 8 shows six concrete operators: four for the typical regular expressions, one to express a sequence that matches several execution traces that occur in any order, and the RepeatUntil operator that expresses a sequence that matches repeated times an execution trace until this sequence matches another execution trace.

Figure 9 shows the sequence diagram of the evaluation of a sequence when it receives an event. A Sequence object is evaluated with an event using its eval method. This method evaluates all matching steps of the sequence, and if one of them return an environment, in which case the sequence matches, the eval method returns that environment. If no matching step returns an environment, in which case the sequence does not match, the eval method returns false. The evaluation of a MatchingStep object is carried out by its eval method. This method takes an event (provided by the sequence) and returns the evaluation of its sequence expression. The evaluation of a SequenceExpression object is carried out by its eval method. This method takes the event and an environment (provided by the matching step) and can return three types of values:

matched	advanced	otherwise
Environment	Array of MatchingSteps	false

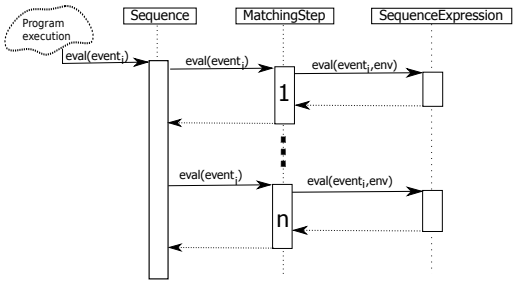

Figure 9. The sequence diagram of a sequence evaluation.

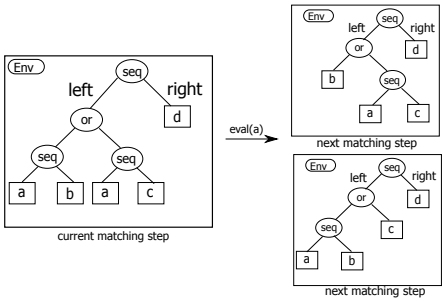

Figure 10. Two matching steps created by a matching step evaluation.

The eval method of a sequence expression returns an Environment[3] if this sequence expression *matched*. Instead, this method returns an array of matching steps if the sequence expression only *advanced*; the returned array represents the next matching steps of the sequence, which can be several because of nondeterminism. Finally, this method returns false if the sequence expression neither matched nor advanced.

Although the separation conceptual between selectors and operators to build a sequence expression is clear to explain and compare the components of a sequence, the implementation of selectors and operators in OTM follows a functional approach. The selectors and operators are plain functions that takes the same parameters of the eval method. While the evaluations of selectors can return false or an environment, the evaluations of operators additionally can return an array of MatchingSteps. This implementation is chosen due mainly to AspectScript pointcuts are JavaScript functions, and these pointcuts become selectors effortlessly.

A sequence expression that describes the execution trace of the *a* event followed by the *b* event can be declared using the seq function:

```
var sa = call(..),sb = propRead(..);
var seqExpAB = seq(sa,sb);
```

The seqExpAB object is a sequence expression composed of the selectors sa and sb, which match the *a* and *b* events respectively. Informally, we can say the seq function is a *sequence expression designator* because the seq evaluation returns a sequence expression.

seq. Figure 11 presents the implementation of the seq operator. The function returned by the seq evaluation first evaluates the left sequence expression and binds the result to res. If res does not represent a match or an advance, this function returns false. If res represents a match of left, this function returns the right sequence expression as an array of MatchingSteps. If res represents an advance, this function returns the next matching step(s) that are based on matching steps of the res array. For example, Figure 10 shows the evaluation of a matching step whose sequence expression is *(a b || a c) d* with an event *a*. This evaluation creates two matching steps with different sequence expressions, *(b || a c) d* and *(a b || c) d*, because the left sequence expression, *(a b || a c)*, only advances when it is evaluated with the *a* event.

anyOrder. The anyOrder operator expresses a sequence that matches several execution

[3]Returning true in the eval method of a sequence expression has the same meaning as returning the same environment originally passed to this method.

```
function seq(left,right) {
  return function(event,env) {
    var res = left(event,env);

    if (matched(res))
      return [new MatchingStep(right,res)];

    if (advanced(res))
      return res.map(function(step) {
                 return new MatchingStep(
                              new Seq(step.seqExp,next),step.env);
               });

    return false;
} }
```

Figure 11. The Seq operator.

```
function anyOrder(seqExps) {
  return function(event,env) {
    //only one sequence expression remains
    if(seqExps.length == 1)
      return seqExps[0](event,env);

    var nextSteps = [];
    for(var i = 0; i < seqExps.length; ++i) {
      var res = seqExps[i](event,env);

      if(matched(res))
        nextSteps.push(new MatchingStep(
                         new AnyOrder(remove(seqExps,i)),res));

      if(advanced(res)) {
        var resNextSteps = res.map(function (step) {
                             return new MatchingStep(
                               new AnyOrder(seqExps.concat([step.seqExp])),step.env)});
        nextSteps = nextSteps.concat(resNextSteps);
      }
    }

  return nextSteps.length > 0? nextSteps: false;
} }
```

Figure 12. The AnyOrder operator.

```
function repeatUntil(repeat,until){
  return function(event,accEnv){
    var finalEnv = until(event,accEnv);
    if (matched(finalEnv))
      return finalEnv;

    var newEnv = repeat(event,accEnv);
    if (matched(newEnv))
      return [new MatchingStep(repeatUntil(repeat,until),
          union(accEnv,newEnv))];

    return false;
} }
```

Figure 13. The RepeatUntil operator.

```
function eval(event) {
  //some code omitted here for simplicity
  var env = false; //env changes its value if there is a match
  steps = applyLcs(steps,event);
  steps.map(function(step) {
    var res = step.eval(event);
    ...
    if (advanced(res) && hasNext(step) && applyMms(step,res)) {
      //add a new child sequence
    }
  }
  return env;
}
```

Figure 14. The Sequence eval method.

traces that occur in any order. This operator is similar to the PQL anyOrder operator [Martin et al. 2005], except that in PQL only atomic events (not whole sequences) can be related by this operator. The anyOrder operator is non-deterministic, like the *or* operator shown in Section 2.1, because two or more sequence expressions can advance or match with the same event. Figure 12 shows the implementation of the anyOrder operator, which takes an array of sequence expressions as argument and returns the function that represents the operator. If there is only one sequence expression (the base case), this function returns the evaluation of this last sequence expression because this sequence expression represents the last execution trace that the sequence should match. Instead, when there are two or more sequence expressions in the seqExps array, this function evaluates each sequence expression; and depending on the returned value (res), this function takes one of two decisions. First, the nextSteps array adds a new matching step using a new anyOrder operator and the res environment if seqExps[i] matches. Second, nextSteps is updated with the resNextSteps array, which contains the next(s) matching steps of seqExps[i], if seqExps[i] only advances. Both decisions create matching steps that contain an AnyOrder operator with the seqExps array without the sequence expression that recently matched or advanced, meaning that the number of sequence expressions in this array decreases up to the base case.

repeatUntil. The sequence shown in Section 1.2, which matches several times an event up to this sequence matches another event, can easily be declared using the RepeatUntil

operator (see Figure 13). This operator takes two selectors as parameters[4]: repeat and until. The function returned by this operator the *accumulated* environment by all events that repeat has matched so far if until matches the current event. The accEnv environment is accumulated when until does not match the current event but repeat does. To accumulate values in the accEnv environment, this function returns a new matching step with the same operator and the *union* of the already-accumulated environment and the new environment (newEnv). The union function returns an aggregated environment, where values of common identifiers are gathered in lists. Like previous operators, the function returns false if both repeat and until do not match the event.

4.2. Creating A Sequence

The previous section shows how to declare operators, which are used to define sequence expressions. As mentioned in Section 2.1, sequence expressions describe the execution traces that the sequences should match. Thereby, the declaration of a sequence needs a sequence expression as one of its arguments:

```
var sequence = new Sequence(seqExp,lcs,mms);
```

A sequence is created with a sequence expression (seqExp), a life cycle strategy (lcs), and a multiple matching strategy (mms). Both strategies are plain JavaScript functions that can be omitted[5]. If some of these strategies are omitted, the sequence uses the corresponding default strategies.

Figure 14 shows the eval method of a sequence, which is called for every event. As the previous section mentioned, this method evaluates all matching steps and returns an environment if some of matching steps return an environment. In addition, this method applies the life cycle strategy and the multiple matching strategy using the methods applyLcs and applyMms respectively. The eval method always calls applyLcs, whereas the eval method calls applyMms when a matching step that already have next matching steps returns another set of matching steps.

4.3. The Life Cycle of a Sequence

As mentioned in Section 2.2, a life cycle strategy controls the evolution of the matching of a sequence passes until it matches. The applyLcs method applies the life cycle strategy. This method takes as arguments the set of matching steps of the sequence and an event, and this method returns a new version of this set. OTM allows developers to customize the life cycle strategy using the lcs function. OTM provides, among others, the except and switch functions as constructors as life cycle strategies.

The except function (Figure 15) creates life cycle strategies that abort sequences if some condition is satisfied. The condition is specified as a sequence; whenever it matches, the life cycle strategy removes all matching steps of the controlled sequence, thereby effectively resetting the sequence.

The switch function (Figure 16) creates life cycle strategies that switch the execution trace that sequences should match if some condition is satisfied. The condition is specified

[4]For space reasons, we only show the version that supports selectors as arguments. In Appendix A, we show the version that supports sequence expressions as parameters.

[5]All function parameters in JavaScript are optional.

```
function except(conditionalSeq) {
  return function(steps,event) {
    if(matched(conditionalSeq.eval(event))) {
      steps.removeAll();
    }
    return steps;
} }
```

Figure 15. The life cycle strategy to support the *except* feature.

```
function switch(conditionalSeq,replacementSeqExp) {
  return function(steps,event) {
    if(matched(condtionalSeq.eval(event))) {
      steps.removeAll();
      steps.push(new MatchingStep(replecementSeqExp,emptyEnv));
    }
    return steps;
} }
```

Figure 16. The life cycle strategy to support the *switch* feature.

as a sequence; whenever it matches, the life cycle strategy removes all matching steps of the controlled sequence and adds a first new matching step with a replacement sequence expression, thereby effectively switching the execution trace that the sequence should match.

4.4. The Multiplicity of a Sequence

As mentioned in Section 2.3, sequences can have *children* sequences. This means that apart from evaluating its matching steps, the eval method of a Sequence object should be updated to evaluate the children sequences. Thereby, this method could now return an *array* of environments (not only one), one for the evaluation of the sequence itself and one for every child sequence that matches.

A multiple matching strategy controls the number of simultaneous matching of a sequence in a trace of execution. The applyMms method applies the multiple matching strategy. This method permit to add, according to its multiple matching strategy, a *new child* sequence with a set of matching steps created by a matching step that already has next matching steps. If this method decides to add a new child sequence, this child sequence and the sequence share a common prefix of matching steps. OTM allows developers to customize the multiple matching strategy. The following function implements a multiple matching strategy that always adds a new child sequence.

```
function multiple(step,steps) {return true;}
```

step is the matching step, which already has next matching steps, that created the steps array. The *single* matching strategy (used in Section 1.2) never adds a child sequence. This strategy is used to match only one sequence at a time.

```
function single(step,steps) {return false;}
```

The OTM distribution provides, among others, the tracematches and halo functions as multiple matching strategies. Figure 17 presents the function that implements the Halo multiple matching strategy [Herzeel et al. 2006], which adds a new child sequence if its contextual information differs from the contextual information of the sequence and other children sequences. The halo function adds a new child sequence when at least one of matching steps of the steps array differs in the values of its environment from all environments of next matching steps (nextSteps).

Figure 18 presents the function that implements the tracematch multiple matching strategy [Allan et al. 2005]. Like Halo, the tracematch strategy adds a new child sequence if its contextual information differs from the contextual information of the sequence. Apart from that, this strategy also adds a new child sequence if this possible child sequence represents a new beginning of the matching of the execution trace. This

```
function halo(step,steps) {
  var nextSteps = step.getNextMatchingSteps();
  for(var i = 0; i < steps.length; ++i) {
    var repeated = false;
    for(var k = 0; k < nextSteps.length; ++k) {
      var nextStepsBySeq = nextSteps[k];
      for(var j = 0; j < nextStepsBySeq.length; ++j) {
        if(equals(steps[i].env,nextSteps[j].env))
          repeated = true;
      }
    }
    if (!repeated)
      return true;
  }
  return false;
}
```

Figure 17. The *Halo* multiple matching strategy.

```
function tracematches(step,steps) {
  if (!hasPreviousStep(step))
    return true;

  var nextSteps = step.getNextMatchingSteps();
  for(var i = 0; i < steps.length; ++i) {
    var repeated = false;
    for(var k = 0; k < nextSteps.length; ++k) {
      var nextStepsBySeq = nextSteps[k];
      for(var j = 0; j < nextStepsBySeq.length; ++j) {
        if(equals(steps[i].env,nextSteps[j].env))
          repeated = true;
      }
    }
    if (!repeated)
      return true;
  }
  return false;
}
```

Figure 18. The *tracematches* multiple matching strategy.

seemingly small difference implies that Halo and tracematches match a different number of times. For example, if the trace of execution is *a a a* and the sequence is *a a*, Halo matches once and tracematches matches twice. The tracematches function implementation is the same of the halo function implementation, the only difference is the initial condition (**if** (!hasPreviousSteps(step)) to permit to begin a new matching of the execution trace.

5. Revisiting The Restrictive Discount Feature

In Section 1.2, we needed to add the *restrictive discount* feature to the Boutique Web application. This feature applies the three discounts of all products added to the shopping cart that generate the best benefits for the client if the client does checkout before Christmas Day. As also mentioned in that section, this feature can be implemented through the sequence shown in Figure 2 in a hypothetical TM that uses the *except* life cycle strategy and the *single* matching strategy (see Section 4.3 and 4.4). The first strategy is used to avoid that the sequence matches after Christmas Day and the second strategy is used to match one sequence at a time.

Figure 19 shows the declaration and deployment the sequence shown in Figure 2 using OTM. Figure 19a shows the definition of selectors: cd matches any events that occurs after "25th of December, 2010", ap binds the product and the associated discount when it matches the call to addProduct, and co matches the call to checkout. Figure 19b shows the definition of the advice, which gets the lists of products and discounts from the environment passed as parameter. It then gets, shows, and applies the three discounts that generate the best benefits for the client. Finally, Figure 19c shows the definition and deployment of the rd sequence, which repetitively matches the event described by ap until this sequence matches the event described by co. In addition, the life strategy is customized to avoid that the sequence matches after the given date, and the multiple matching strategy is customized for that the sequence matches only once at a time. The deployment triggers the execution of the advice *around* the checkout execution.

6. Conclusion

As trace-based mechanisms are gaining interest and being applied in a large number of contexts, there is a need for understanding and consolidating their specific differences, and allowing programmers to fine-tune these systems as required. This paper has presented an

```
//matches any event that occurs after Christmas Day
var cd = function(event,env) {
      return (new Date(2010,12,25)).getTime() <
             (new Date()).getTime();
};

//matches calls to function addProduct
//  & binds the product and discount object
var ap = function(event,env) {
    if(event.isCall() && event.fun == addProduct) {
      var p = event.args[0]; //1st to addproduct
      var d = getDiscount(p.getId());
      return env.bind("product", p).bind("discount", d);
    }
    return false;
};

//matches all calls to function checkout
var co = call(checkout);
```

a) Selectors

```
var advice = function(event,env) {
  //getting the list of products and the list of discounts
  var products = env.product;
  var discounts = env.discount;
  //getting and showing the three best discounts
  var productsWithDiscounts = getDiscountMax3(products,discounts);
  showDiscounts(productsWithDiscounts,discounts);
  //applying the three discounts
  return event.proceed(applyDiscount(products,productsWithDiscounts,discounts));
};
```

b) The advice

```
var rd = new Sequence(repeatUntil(ap,co), //sequence expression
                      except(new Sequence(cd)), //life cycle strategy
                      single //single matching strategy
                     );

OTM.deploy(OTM.AROUND,rd,advice);
```

c) The sequence

Figure 19. Using OTM to implement the restrictive discount feature.

abstract operational model of a trace-based mechanism, and built on it to propose an open implementation, called OTM. Currently, OTM is implemented in JavaScript, opening the door to a myriad of applications of TMs to Web programming. We have shown how existing TMs can be expressed in OTMs, and have exposed the definition of a number of variants, such as new operators and life cycle strategies.

While this work has focused on the expressiveness of sequences, we are currently exploring how to open another important aspect of trace-based mechanisms, namely the advice invocation protocol. A major challenge for OTM is how to bridge the gap between the expressiveness and flexibility that it offers and the need for optimizations. At present, opportunities for static analysis are greatly limited, due in part to the dynamicity of JavaScript, and in part to the flexibility of the OTM framework. Allowing programmers to annotate and fix certain elements of a sequence may help in regaining some performance as well as potential for compile-time checking.

Acknowledgments. We are grateful to Rodolfo Toledo for his help in clarifying our initial intuition.

References

Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2005). Adding trace matching with free variables to AspectJ. In [OOPSLA 2005 2005], pages 345–364. ACM SIGPLAN Notices, 40(11).

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley.

Goldsmith, S. F., O'Callahan, R., and Aiken, A. (2005). Relational queries over program traces. In [OOPSLA 2005 2005], pages 385–402. ACM SIGPLAN Notices, 40(11).

Herzeel, C., Gybels, K., and Costanza, P. (2006). A temporal logic language for context awareness in pointcuts. In Thomas, D., editor, *Workshop on Revival of Dynamic Languages*, number 4067 in Lecture Notes in Computer Science, Nantes, France. Springer-Verlag.

Kiczales, G., Lamping, J., Lopes, C. V., Maeda, C., Mendhekar, A., and Murphy, G. (1997a). Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering (ICSE 97)*, pages 481–490, Boston, Massachusetts, USA. ACM Press.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997b). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland. Springer-Verlag.

Leger, P. and Tanter, É. (2010). Towards an open trace-baced mechanism. In Leavens, G. T., Katz, S., and Mezini, M., editors, *Proceedings of the Ninth Workshop on Foundations of Aspect-Oriented Languages (FOAL 2010)*, pages 25–30, Rennes and Saint Malo, France.

Martin, M., Livshits, B., and Lam, M. S. (2005). Finding application errors and security flaws using PQL: a program query language. In [OOPSLA 2005 2005], pages 365–383. ACM SIGPLAN Notices, 40(11).

OOPSLA 2005 (2005). *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2005)*, San Diego, California, USA. ACM Press. ACM SIGPLAN Notices, 40(11).

Ostermann, K., Mezini, M., and Bockisch, C. (2005). Expressive pointcuts for increased modularity. In Black, A. P., editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 214–240. Springer-Verlag.

Toledo, R., Leger, P., and Tanter, É. (2010). AspectScript: Expressive aspects for the Web. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, pages 13–24, Rennes and Saint Malo, France. ACM Press.

A. The RepeatUntil Operator

```
function repeatUntil(repeat,until,repeatInitial) { //The third parameter is used to restart the repeat sequence expression
  repeatInitial = (repeatInitial == undefined)? repeat: repeatInitial;
  return function(event,env) {
    var resUntil = until(event,env);
    if(matched(resUntil))
      return resUntil; // match the operator

    var resRepeat = repeat(event,env);

    //if repeat matches
    if(matched(resRepeat)) {
      var accEnv = union(env,resRepeat);
      if (advanced(resUntil)) {
        return resUntil.map(function(step){
                 return new MatchingStep(repeatUntil(repeatInitial,step.seqExp),union(accEnv,step.env));
               });
      }
      return [new MatchingStep(repeatUntil(repeatInitial,until),accEnv)];
    }

    //if repeat only advances
    if(advanced(resRepeat)) {
      if (advanced(resUntil)) {
        var res = crossProduct(resRepeat,resUntil); //makes pairs of combinations of the resRepeat and resUntil matching steps
        return res.map(function(pair){
                   return new MatchingStep(repeatUntil(pair[0].seqExp,pair[1].seqExp,repeatInitial),union(env,union2(pair[0].env,pair[1].env)));
                 });
      }
      return resRepeat.map(function(step){
                         return new MatchingStep(repeatUntil(step.seqExp,until,repeatInitial),union(env,step.env));
                       });
    }

    //if repeat neither matches nor advances
    if (advanced(resUntil)) {
      return resUntil.map(function(step){
               return new MatchingStep(repeatUntil(repeat,step.seqExp,repeatInitial),union(env,step.env));
             });
    }
    return false;
  };
}
```

Figure 20. The RepeatUntil operator version that supports sequence expressions as parameters.