An Expressive Stateful Aspect Language

Paul Leger[1] Éric Tanter[2] Hiroaki Fukuda[3]
[1]Universidad Católica del Norte, Chile
[2]PLEIAD Lab, Computer Science Department, University of Chile
[3]Shibaura Institute of Technology, Japan

Abstract

Stateful aspects can react to a program execution; they support modular implementations of several crosscutting concerns like error detection, security, event handling, and debugging. However, most proposed stateful aspect languages have specifically been tailored to address a particular concern. Indeed, most of these languages differ in their pattern languages and semantics. As a consequence, developers need to tweak aspect definitions in contortive ways or create new specialized stateful aspect languages altogether if their specific needs are not supported. In this paper, we describe ESA, an expressive stateful aspect language, in which the pattern language is Turing-complete and patterns themselves are reusable, composable first-class values. In addition, the core semantic elements of every aspect in ESA is open to customization. We describe ESA in a typed functional language. We use this description to develop a concrete and practical implementation of ESA for JavaScript. With this implementation, we illustrate the expressiveness of ESA in action with examples of diverse scenarios and expressing semantics of existing stateful aspect languages.

Keywords: Aspect-Oriented Programming, Stateful Aspects, ESA, Typed Racket, JavaScript

1. Introduction

Separation of concerns [1] establishes that a program should be decomposed in a set of *modules*, and that each module should address a given concern of the software. Modules are crucial for raising the understandability, maintainability, reusability, and evolvability of software. However, concerns like logging and event handling cannot be implemented in one module; these are known as *crosscutting concerns*. Aspect-Oriented Programming (AOP) [2] allows developers to use aspects, as embodied in *e.g.* AspectJ [3], to modularize crosscutting concerns. In the pointcut-advice model of AOP [4], an aspect specifies program execution points of interest, named *join points*, through predicates called *pointcuts*. When an aspect matches a join point, it takes an action, called *advice*. Typically, an aspect matches a program execution point in isolation, or in the context of the current call stack. However, the modularization of some crosscutting concerns requires aspects to match a *trace* of join points (*e.g.* debugging [5], security [6], runtime verification [7], and event correlation [8]). Aspects that can react to a join point trace are called *stateful aspects* [9].

Several stateful aspect languages have been proposed [6, 8, 10, 11, 12, 13, 14], specifically tailored to address particular domains. Mainly because of this reason, these languages do not share the same semantics [14]. Some of them like Tracematches [10] support multiple matches, even simultaneously, of a join point trace pattern (just *pattern* for now). Each language provides its own sub-language to define patterns of interest. In addition, each language has its own *matching semantics* to define how a pattern is matched (*e.g.* multiple matches of Tracematches), and *advising semantics* to define how an advice is executed. To date, stateful aspect language design has been mostly focused on performance, leaving aside the exploration of more expressiveness:

Pattern language. The lack of expressiveness in pattern languages has at least two consequences. The first consequence is the lack of reuse and composition of patterns in most stateful aspect languages. The second one is the limited expressiveness to define patterns. We now illustrate the implication of the first consequence.

In the area of the event correlation [15], let us consider a *toggle airplane mode* feature in touch devices. If a user slides the sequence *up*, *down*, and *up* in the touch device, the airplane mode is enabled (or disabled, if it was enabled). Using Tracematches [10], a stateful aspect language for Java, we can define a tracematch, which is composed of a set of symbols, a regular expression pattern, and a piece of code. A symbol represents a set of join points, the pattern is defined as a regular expression over these symbols, and the piece of code represents the advice. To implement this feature, we can define the following stateful aspect in Tracematches:

```
tracematch() {
  //symbols
  sym s–up after: call (* Screen.up());
  sym s–down after: call (* Screen.down());

  //pattern
  s–up s–down s–up

  //piece of code (advice)
  {
    Device.toogleAirplaneMode();
} }
```

Consider now the addition of a new feature *toggle airplane mode with time*: this enables/disables the airplane mode during a period of time (*e.g.* 10 hours). This new feature is executed when a user realizes the sequence of the toggle airplane mode with a *left* slide as prefix. To implement this feature, an adequate solution would be to reuse the pattern of the previous feature, however, Tracematches like other stateful aspect languages do not allow developers to

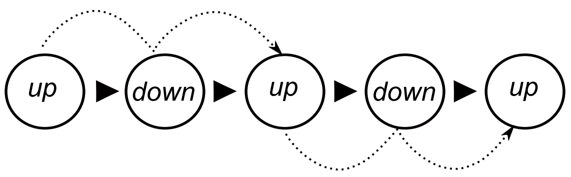


Figure 1: Matches of a naive implementation of *toggle airplane mode* with Tracematches.

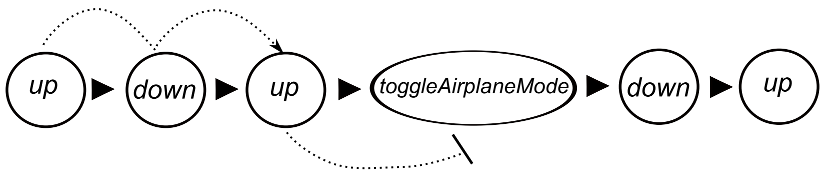


Figure 2: A correct implementation of *toggle airplane mode* with Tracematches needs an extra symbol.

reuse patterns. Therefore, it is necessary to write all pattern again. This means that the piece of code is not easily maintainable because if the pattern of *toggle airplane mode* changes, it is necessary to rewrite the pattern for this new feature.

Semantics. In stateful aspect languages, limited expressiveness also has two consequences in terms of semantics: common and fixed for all stateful aspects. Next, we exemplify the effect of the first consequence.

Although the aspect implementation of toggle airplane mode looks correct, this implementation does not work because Tracematches perform multiple matches of a pattern. As a result, once the pattern is matched, each subsequent *up* and *down* toggles the airplane mode. Figure 1 shows the previous point because we can observe that before that the first match of the pattern finishes (represented by the up dash line) with the call to the *up* function, a new potential match of the pattern starts (the down dash line). As the Tracematches semantics is not adequate in all cases, the programmer has to tweak the aspect definition to artificially introduce another symbol, toggled, which is then excluded from the regular expression because Tracematches require contiguous occurrences of the symbols in a pattern (see Figure 2):

```
tracematch() {
  sym s-up after: call (* Screen.up());
  sym s-down after: call (* Screen.down());

  //artificial symbol
  sym toggled after: call(* Device.toggleAirplaneMode());

  s-up s-down s-up {
    Device.toggleAirplaneMode();
} }
```

1.1. Contributions

The consequences of the lack of expressiveness of current stateful aspect languages motive this work, which proposes a precise description of an expressive stateful aspect language. Concretely, this proposal allows developers to:

- use first-class patterns, bringing the benefits of the reuse and composition of patterns (consequence $p1$). Apart from reuse and composition, these first-class patterns allow developers to cleanly use the factory design pattern [16] to build their own pattern libraries. In addition, developers can use a Turing complete language to define patterns (consequence $p2$). In this paper, we use first-class functions to describe our pattern language, but other first-class abstractions might be used as well (*e.g.* objects).
- customize the matching and advising semantics of every stateful aspect. With this, every stateful aspect can have different semantics (consequence $s1$) and developers can customize of any aspect (consequence $s2$). In order to achieve this goal, we follow *open implementation* design guidelines [17], so as to allow developers to customize strategies of a program implementation, and at the same time, still hiding details of its implementation.

To contrast our proposal, we develop a reference frame that compares and evaluates the existing proposals in terms of expressiveness. As a result, we clarify and discuss some differences between these proposals. This reference frame becomes an additional and novel contribution of this paper.[1]

Paper roadmap. Section 2 introduces and goes into detail about stateful aspect languages. Section 3 discusses and evaluates the state of the art of these languages. The limitations of existing proposals are shown through diverse examples in Section 4. Section 5 presents the description of an expressive stateful aspect language, named ESA; we describe ESA using a functional typed language, Typed Racket [18]. To illustrate how our proposal addresses the aforementioned limitations, we use a concrete and practical implementation of ESA for JavaScript in Section 6. Section 7 assesses the expressiveness of ESA through the emulation of several existing stateful aspect languages, and Section 8 concludes.

2. Stateful Aspects

Stateful aspects [9] support the modular definition of crosscutting concerns for which matching join point traces, as opposed to single join points, is necessary. As Figure 3 shows, a stateful aspect is composed of a (join point trace) pattern and an advice. The advice is executed *before*, *around*, or *after* of the last join point that must match with the pattern. In the three next sections, we

[1]This work extends and refines our previous work on open trace-based mechanisms, discussed in Section 3.

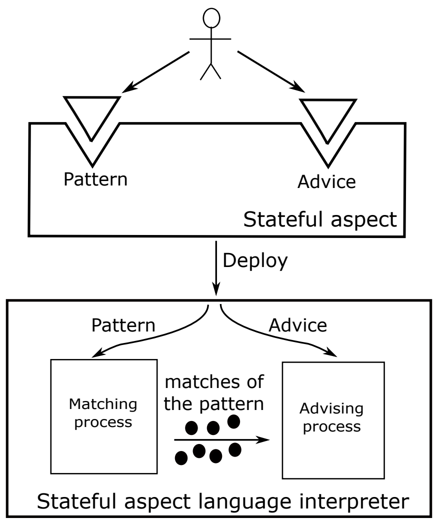


Figure 3: The anatomy of a stateful aspect and the main processes of its language.

describe the core elements (pattern language, matching process, and advising process) of a stateful aspect language.

2.1. Pattern Language

A stateful aspect language uses a pattern language to specify patterns. This section defines a pattern language and explains its main features.

Definition (Pattern Language)**.** *A language that allows developers to specify the (join point trace) pattern that should be matched by a stateful aspect.*

Some pattern languages allow developers to specify bindings that are gathered while a pattern is being matched. For example, patterns of Tracematches [10] are defined as regular expressions and can gather bindings. We illustrate pattern definitions using a runtime verification example implemented in Tracematches (presented in [19]):

```
tracematch (Vector v, Enumeration e) {
  sym createIter after returning(e): call(Enumeration Vector.elements()) && target(v);
  sym nextEl before: call(* Enumeration.nextElement()) && target(e);
  sym addVector after: call(* Vector.addElement(..)) && target(v);

  createIter nextEl* addVector+ nextEl
  {
    throw new ConcurrentModificationException();
} }
```

This tracematch provides the *fail-fast* feature for the Enumeration interface (this feature is only available for Iterator[2]). *Fail-fast* triggers an exception if

[2]http://docs.oracle.com/javase/6/docs/api/java/util/Vector.html

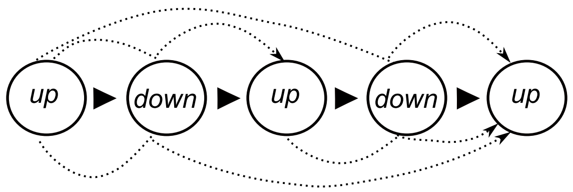


Figure 4: Potential matches of the pattern $up \triangleright down \triangleright up$.

the underlying collection is modified while an iteration is in progress. As the tracematch code shows, bindings gathered are specified in the header; in this piece of code, two bindings: a vector v and its enumeration e. In an aspect of Tracematches, when a symbol of the regular expression is matched, a binding can be gathered; for example, v is gathered when createIter is matched. As we need to catch unsafe uses of enumerations, the pattern first sees the creation of an enumeration then zero or more nextElement calls, one or more addElement calls, and finally an erroneous attempt to continue the enumeration. Note the subpattern nextEl* is necessary because the Tracematches semantics matches a pattern against all suffixes of a join point trace, which is represented by symbols declared in the tracematch. For example, the trace *elements* ▶ *nextElement* ▶ *addElement* ▶ *nextElement* does not match if the pattern does not contain nextEl*.

2.2. Matching Process

In most existing proposals, the matching process implementation is inside of the stateful aspect language. The definition of its process and some of its semantic variants are explained in this section.

Definition (Matching Process)**.** *A process that tries matching a given pattern against the current join point trace according to its semantics.*

Figure 3 shows that when a stateful aspect is deployed, the aspect language uses its matching process for the matching of the pattern of an aspect. Depending on the semantics of this process, a pattern may match multiples times. We use the implementations of the *toggle airplane mode* and *fail-fast* features to illustrate the impact of a matching process semantics.

Toggle airplane mode. Figure 1 shows that the Tracematches semantics matches twice the pattern of this feature in the trace *up* ▶ *down* ▶ *up* ▶ *down* ▶ *up*. This is so because the Tracematches semantics supports multiple matches, but it has the constraint of contiguous occurrences of the symbols in the pattern. Figure 4 shows that if the previous constraint is removed, the matching process would match up to four times; meaning that it gets more difficult to know whether the airplane mode is enabled or disabled.

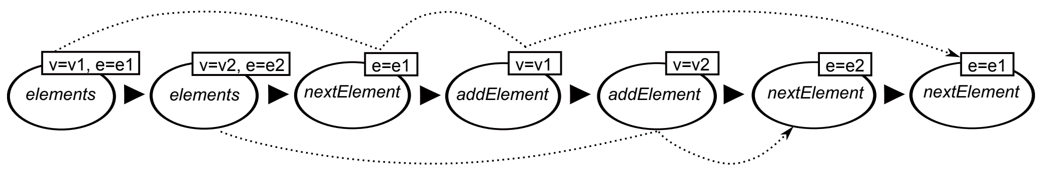


Figure 5: Two matches of the pattern used for *fail-fast* according to the Tracematches semantics. In the figure, the match of up side gathers the bindings `v1` and `e1`, and the match of down side gathers the bindings `v2` and `e2`.

Fail-fast. Although the tracematch solution presented for this feature seems intuitive, it is not so intuitive when we analyze in detail. Tracematches support multiple matches of a pattern, if the *environment* of bindings gathered differs between the potential matches of this pattern. Figure 5 exemplifies this semantics because the pattern matches twice due to the fact that the environments of both matches differ ($\boxed{\text{v=v1 , e=e1}}$ and $\boxed{\text{v=v2 , e=e2}}$).

Tracematches semantics. The semantics of Tracematches is more complex than what this section describes. We go into detail about this semantics in Section 7.3, where we use ESA to express Tracematches.

2.3. Advising Process

The implementation of the advising process is also inside of a stateful aspect language in most existing proposals. We start defining its process and then explain its main features.

Definition (Advising Process)**.** *Given one or more simultaneous matches of a pattern, the advising process is a process that executes, according to its semantics, the advice of a stateful aspect for every one of these matches.*

The advising process is triggered when it receives one or more matches of the pattern (Figure 3). Each match abstraction contains the environment of bindings gathered during the matching stage. Then, the advising process executes the (same) advice for every match with its environment of bindings. We illustrate this process with the implementation of a *discount policy* feature on a Web application (presented in [12]).

Discount Policy. A Web application of an online store is used to order computers (*e.g.* DELL). A client chooses computers from the catalogue and adds them to a virtual shopping cart, which may contain more computers. The Web application contains a checkout form asking for a desired payment method. Consider that the store now wants to add a *discount policy*, where every computer has a potential discount that is applied when it is added to the cart. Each discount that is associated to a computer is only valid for a period of time; however, this discount must be applied (even if it is not valid anymore) when the client checks out. Implementing this discount policy is a crosscutting concern that can be modularized using a stateful aspect as follow:

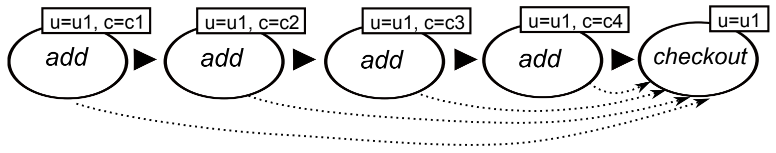


Figure 6: Four simultaneous matches of the pattern used to implement the *discount policy* feature.

```
tracematch (User u,Computer c) {
  sym add after: call(* Cart.add(User,Computer)) && args(u,c);
  sym checkout before: call(* Form.checkout(User)) && args(u);

  add checkout
  {
    Cart cart = u.getCart();
    cart.applyDiscount(c,DiscountPolicy.getDiscount(c));
} }
```

Figure 6 shows a join point trace that triggers four simultaneous matches for the tracematch above (notice these matches have different environments of bindings). The advice is executed four times, each one with an environment of bindings (*e.g.* $\boxed{\text{u=u1 , c=c1}}$), and each advice execution applies the discount to one computer. In this example, the advising process must compose the four executions of the (same) advice.

3. Existing Stateful Aspect Languages

Douence *et al.* [9, 20] initiated the body of work on stateful aspects. In addition, there is a large body of work on stateful aspects, which we review in this section. For each proposal, we focus on three components described in the previous section: pattern languages, matching process, and advising process.

Tracematches. The Tracematches proposal, implemented as an AspectJ [3] extension, is a widely-known and efficient stateful aspect language for Java. The proposal only allows developers to use a regular expression language to define patterns. Its patterns cannot be reused or composed. The matching process is implemented through a nondeterministic finite-state automaton, whose active states correspond to potential matches of a pattern. The advising process supports before, around, and after advice. For the around advice, Tracematches follow the AspectJ guidelines: when there are two or more matches of a pattern with the same join point, the advice executions are chained and nested.

Tracecuts. The Tracecuts stateful aspect [21] is a language that works for Java as an AspectJ extension. This mechanism is used to check the use of protocols (*e.g.* FTP [22], a communication protocol). In Tracecuts, if the join point trace does not follow a pattern, which represents a certain protocol, an action can be triggered. According to the authors of Tracecuts, the checking of some protocols needs to properly identify the nested entries and exits of the executions of different methods of a class. This feature is reducible to recognition of properly nested parenthesis, meaning that a finite state machine cannot correctly check the use of these protocols. Therefore, Tracecuts allow developers to express patterns using a context-free language. The matching process uses a pushdown automaton, and the advising process follow the same guidelines of Tracematches.

Alpha. Alpha [13] is an aspect-oriented extension of L2, a simple object-oriented language in the style of Java. Alpha uses Prolog queries to express patterns. The matching process is implemented through queries to a database that contains information about the static representation (*e.g.* abstract syntax tree) and the dynamic representation (*i.e.* execution trace) of a program. The matching process corresponds to the internal process of Prolog (*i.e.* a backward chaining algorithm [23]) to answer a query. Every solution to a Prolog query corresponds a match of a pattern. These solutions are passed to the advising process, which only supports before and after advice kind. Advices are executed in a consecutive manner for each solution, which contains a set of bindgins gathered.

Halo. Herzeel *et al.* [12] propose Halo, a Common Lisp extension. The Halo proposal allows developers to use *almost* all the base language to express patterns because loops and recursions are not allowed. Despite of these limitations, Halo patterns are first-class values. The matching process is implemented with the Rete algorithm [24], an efficient pattern matching algorithm used for expert systems [25]. In Rete, patterns are represented as rules that must be satisfied by a set of (matched) join points. The advising process only executes the advice with each set of bindings that satisfy the rules. The advice can be executed before or after the last join point matched.

EventJava. EventJava [8] allows developers to execute a piece of code (*i.e.* advice) when a set of distributed events (*i.e.* join points) has a correlation specified by developers. To specify the correlation, every distributed event contains a set of properties available to developers (*e.g.* the time at which the event is observed). The EventJava pattern language only supports an ad hoc for and if constructs to compare these events. No user-defined constructs to compare events are supported. For the matching and advising processes, EventJava follows the same guidelines of Halo, but the advice can only be executed after the last join point matched.

AWED. It is a language for Aspects With Explicit Distribution (AWED) [5, 26]. This stateful aspect language supports the monitoring of distributed computations in Java. In addition, this aspect language takes into consideration distributed causal relations in tasks of debugging and testing of middleware.

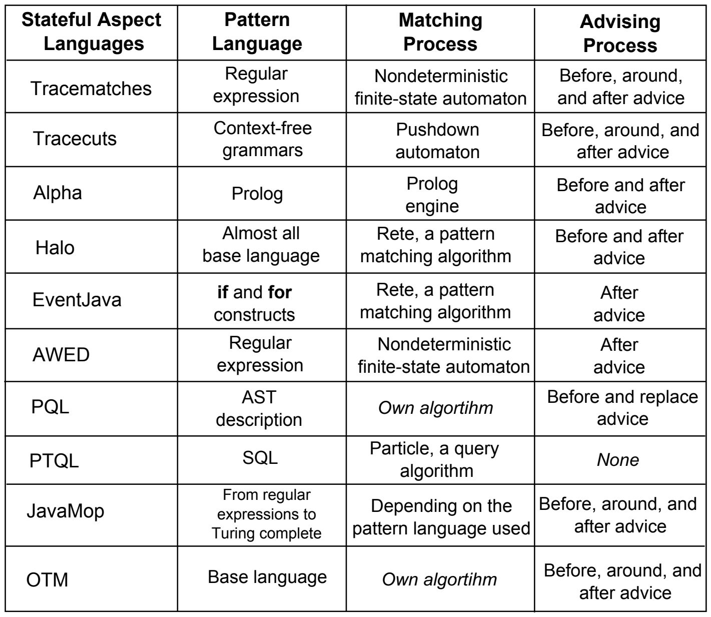
AWED patterns are expressed using a domain-specific language for regular expressions. Similarly to Tracematches, the matching process uses a finite state machine to carry out the match of patterns. For the advising process, AWED follows the same process of EventJava.

PQL. Program Query Language (PQL) [6] is a tool to detect errors and check-/force protocols of programming (*e.g.* file handling). This tool uses a static analyzer to reduce the possible matches and then use a *dynamic matcher* that really matches a given pattern. A developer expresses a pattern using an AST description (using a Java-like syntax). The matching process (*i.e.* dynamic matcher) uses a specialized state machine. The advice can only use the execute, which is used to execute a method before the last join point matched, or replace, used to replace the original computation of the last matched join point.

PTQL. Program Trace Query Language (PTQL) [11] is another tool to detect errors. Developers use the SQL language to express a pattern, which is actually a SQL query. Join points are stored in databases, which are used by its matching process, named PARTIQLE, to match a query. PTQL does not allow developers to take actions if a pattern is matched, *i.e.* there is no an advising process in PTQL.

JavaMop. JavaMop [14, 27, 28] is a generic and efficient runtime-verification framework for Java. Patterns in JavaMop can be expressed in different (previously defined) domain-specific languages: regular expressions, context-free grammars, linear temporal logic, string rewriting system [29], etc. This last pattern language is Turing complete. However, the JavaMop patterns are not first-class values, reusable, and composable. A fixed set of matching process semantics is available for the developers. As JavaMop compiles their code to AspectJ code, the JavaMop advising process follows the same guidelines of AspectJ for this process.

OTM. This paper is not our first try at implementing an expressive stateful aspect. In [30, 31], we implement a stateful aspect language for JavaScript, named OTM. The OTM pattern language is Turing complete and allows developers to reuse and compose patterns. Although the matching process of any OTM stateful aspect can be customized, developers have to update the definitions of their patterns to support a particular customization of the matching process. For example, the definition of a pattern for the single matching semantics differs from the definition of the same pattern for the multiple matching semantics. In other words, the matching process does not really customize the semantics, rather the power of the pattern language allows developers to "code around" patterns to achieve the required semantics. The advising process cannot be customized in OTM. In [32], OTM is extended to control causal relations among Ajax messages in JavaScript applications.



Stateful Aspect Languages	Pattern Language	Matching Process	Advising Process
Tracematches	Regular expression	Nondeterministic finite-state automaton	Before, around, and after advice
Tracecuts	Context-free grammars	Pushdown automaton	Before, around, and after advice
Alpha	Prolog	Prolog engine	Before and after advice
Halo	Almost all base language	Rete, a pattern matching algorithm	Before and after advice
EventJava	**if** and **for** constructs	Rete, a pattern matching algorithm	After advice
AWED	Regular expression	Nondeterministic finite-state automaton	After advice
PQL	AST description	*Own algortihm*	Before and replace advice
PTQL	SQL	Particle, a query algorithm	*None*
JavaMop	From regular expressions to Turing complete	Depending on the pattern language used	Before, around, and after advice
OTM	Base language	*Own algortihm*	Before, around, and after advice

Figure 7: Summary of stateful aspect languages discussed in this section.

Figure 7 sums up the description of the previously described stateful aspect languages. Most stateful aspect languages provide different pattern languages, where the language expressiveness varies. For instance, Alpha uses Prolog, PTQL uses SQL, and Tracecuts uses context-free grammars. Regarding the matching process, all these aspect languages support multiple matches of a pattern. Similar to pattern languages, semantics of matching processes of existing stateful aspect languages vary as well. For example, JavaMop [14] allows developers to choose one of three fixed specifications for the matching process for every stateful aspect. Finally, most advice processes of stateful aspect languages only support before and after advice.

3.1. Evaluation

Figure 8 evaluates the stateful aspect languages described in this section. The figure evaluates three components of these languages: pattern language, matching process, and advising process. Each evaluation measures which of the four consequences ($p1$, $p2$, $s1$, and $s2$) mentioned in Section 1 are addressed. The results of pattern language evaluations are heterogeneous because three proposals use a Turing complete pattern language with support of reusable and composable patterns (consequences $p1$ and $p2$), and two proposals only address one of the previous two consequences. Regarding matching process, only JavaMop and OTM support semantic variations for their aspects (consequence $s1$). Finally, we can observe there is no any support to customize the stateful

Stateful Aspect Languages	Pattern Language	Matching Process	Advising Process
Tracematches	○	○	○
Tracecuts	○	○	○
Alpha	●	○	○
Halo	◐	○	○
EventJava	○	○	○
AWED	○	○	○
PQL	●	○	○
PTQL	○	○	○
JavaMop	◐	◐	○
OTM	●	◐	○
ESA	●	●	●

Legend	**Pattern Language**	**Matching Process Advising Process**
○	Limited expressiveness without reusable and composable patterns	Fixed and common for all stateful aspects
◐	Turing complete, or reusable and composable patterns **(but not both)**	Not common for all stateful aspects, but with fixed semantics
●	Turing complete with reusable and composable patterns	Customizable semantics for stateful aspect

Figure 8: Evaluations of some stateful aspect proposals regarding their pattern language, matching and advising processes.

aspect language semantics. In the bottom of the list, we can appreciate that ESA, the proposal of this paper, completely addresses the four consequences.

It is important to mention that the main concern of existing stateful aspects is their performance. To achieve this concern, expressiveness and customizations have been sacrificed.

4. Limitations of Stateful Aspects

Using diverse variations of the discount policy of the Web store application (Section 2.3), this section shows the limitations of existing stateful aspect languages, in particular, the consequences $p2$ and $s2$ mentioned in the introduction of this paper. Each example is classified according to the components described in the previous section. In Section 6, we will revisit all these examples to present adequate solutions using an implementation of our proposal.

4.1. Pattern Languages

Apart from the lack of the pattern reusability (consequence $p1$), there are other limitations in existing stateful aspect languages. For example, the limited expressiveness to define patterns that gathers a variable list bindings: a case of the consequence $p2$.

Let us at a variation of the current discount policy, named *limited discount policy*). This variation only applies the discounts to a limited number of the best discounts (*e.g.* three computers). This small restriction cannot be implemented as an update of the solution presented previously. This is so because the current solution does not use only one match that contains a list with all computers, which is necessary to choose the best discounts. A tracematch that matches a list of computers would be adequate, however, Tracematches do not allow developers to define a pattern that gathers a variable-size list of bindings. Therefore, we must code around the current tracematch advice to implement the update of this feature:

```
int computerCounter = 0;
ArrayList computers = new ArrayList();

tracematch (User u,Computer c) {
  //symbols and pattern as in Section 2.3

  {
    if (computerCounter++ < u.getCart().size())
      computers.add(c); //adding to the list of computers
    else {
      ArrayList computersWithBestDiscounts = getBestDiscounts(computers);
      //executing the original advice with every computer of the previous list
    }
} }
```

The original advice is only executed when the final match is triggered. In addition, the new advice is now *stateful* because of its mutable binding, computers. The behavior of stateful advices depends on bindings that are outside of it. Therefore, developers must keep in mind the state of outer bindings to know the real behavior of a stateful advice.

4.2. Matching Process

Not all stateful aspects need the same semantics (consequence $s1$). For instance, if we revisit the features of *fail-fast* and *toggle airplane mode* (Section 2.2), we can observe different semantics of the matching process is needed. Whereas the appropriate matching process for *fail-fast* must support multiple matches, the matching process semantics for *toggle airplane mode* must only support a single match at a time.

Apart from the single and multiple matching semantics, custom semantics per stateful aspect may be useful as well (consequence $s2$). Consider a different update to the policy, named *selective discount policy*, which only applies discounts to one computer by category (*e.g.* laptops, desktops, tablets) in the shopping cart. For example, if there are two laptops in the shopping cart, the discount policy only applies discounts to one laptop – the first added. To implement this new policy, it is necessary to custom matching semantics because it must permit a new potential match of the pattern only if a computer of a different category is added. As existing proposals do not allow developers to customize the matching process semantics, developers end up modifying advices to overcome the lack of this customization. For example, the implementation of this new policy in Tracematches requires the following modification around the advice:

```
ArrayList categories = new ArrayList();

tracematch(User u,Computer c) {
  //symbols and pattern as in Section 2.3

  {
    int category = getCategory(c);
    if (!categories.contains(category)) { //Is this category different?
      categories.add(category);
      //executing the original advice with the computer "c"
} } }
```

The piece of code above only applies the discounts when a category is different from the already used categories. Apart from the stateful feature, this advice overlaps its responsibility to apply discounts with the responsibility to correctly match the pattern (*i.e.* the pattern definition is tangled with the advice).

4.3. Advising Process

As mentioned in Section 2.3, the pattern of a stateful aspect can match simultaneously. If there are simultaneous matches of a pattern, the advice is executed several times: each one with an environment of bindings. Existing stateful aspect languages do not allow developers to customize the advising process (consequence $s2$), which is necessary in scenarios like adding the last variation to the discount policy of the online store.

Suppose the store Web application now allows clients to customize the pieces (*i.e.* hardware) of their computers. Thereby, the store established a new discount policy, named *personalized discount policy.* This new policy establishes that the discount policy only applies the discount to the computer with the greatest number of customized pieces. In this scenario, it is not possible to overburden the pattern definition to implement this restriction because the number total of pieces of every computer is only known when a client goes to checkout. The implementation of this new discount policy extension requires that the stateful aspect only executes the advice once, with the match whose bindings gathered contain the computer with more customized pieces:

```
int maxCustomizedPieceNumber = 0;
int computerCounter = 0;
int computerSelected;

tracematch(User u,Computer c) {
   //symbols and pattern as in Section 2.3
```

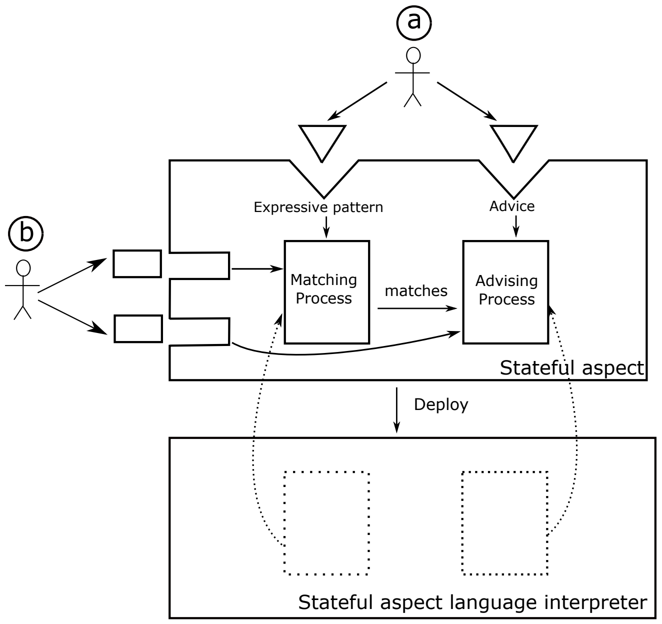


Figure 9: Requirements for Expressive stateful aspect language: *a)* an expressive pattern language and *b)* customizable semantics per stateful aspect.

```
  {
    Cart cart = u.getCart();

    //select the computer with the more customized pieces
    if (maxCustomizedPieceNumber < getCustomizedPiecesNumber(c)) {
      maxCustomizedPieceNumber = getCustomizedPiecesNumber(c);
      computerSelected = c;
    }

    if (++computerCounter == cart.size())
     //executing the original advice for "computerSelected"
} }
```

As we have seen before, coding around the advice is the most used option for the current spectrum of stateful aspect languages. In this piece of code, the original advice is only executed once, which uses computerSelected to apply the discounts.

4.4. Requirements for an Expressive Stateful Aspect Language

We have presented many examples that illustrate different kinds of limitations of existing proposals. In our opinion, a stateful aspect language that overcomes these limitations should consider an expressive pattern language and customizable semantics per aspect (see Figure 9):

Expressive Pattern language. A Turing complete language allows developers to express advanced patterns (consequence $p1$). In addition, first-class patterns are useful to cleanly reuse and compose patterns (consequence $p2$). Finally, the creation at runtime of patterns supports the definition of flexible patterns (*e.g.* adaptive, approximate, and even evolutive patterns).

Customizable Semantics. Unlike Figure 3, Figure 9 shows that *each* stateful aspect contains its matching and advising processes. Using this new approach, each aspect can have its own semantics, which should be according to its target concern (consequence $s1$). In addition, the figure shows that developers can customize the aspect semantics (consequence $s2$). The customization of semantics per aspect may be an extra complexity for developers because it requires a detailed knowledge of aspect language implementations. To address this complexity, *open implementations* [17] propose useful programming guidelines. Open implementation guidelines suggest intuitive default semantics for a program (*i.e.* the black-box case). For customizations (*i.e.* the white-box case), these guidelines suggest that abstractions used must not depend on a specific implementation of the program, meaning that customizations should be *self-contained*: they can be understood and implemented in an isolated manner.

5. ESA

This section presents the description of our expressive stateful aspect language, named ESA. We use a typed functional language, Typed Racket [18], to precisely describe ESA. For reading comprehension reasons, some implementation details of Typed Racket have been omitted (Appendix B shows these details).

ESA overview. In search of satisfying the requirements of Section 4.4, this description allows developers to implement a stateful aspect language, whose pattern language is Turing complete and semantics of each stateful aspect is customizable. In the following two sections, we first introduce the ESA pattern language, and then we explain how ESA allows developers to customize stateful aspect semantics. Although stateful aspect languages commonly uses a rich join point model (*e.g.* call join points, execution join points, field write join points, etc), we will only focus on function-call join points because they are enough to describe ESA.

5.1. Pattern language

In the standard formulation vision of the pointcut/advice model, a pointcut is a function that matches a single join point. The description behind of the ESA pattern language is a natural extension of the pointcut-advice model. We explain this affirmation in two parts. In the first part, patterns could not be used to gather bindings. In the second part, the ESA pattern language is extended to support definitions of patterns that gather bindings.

5.1.1. Without Bindings

Whereas a pointcut is a function Pc: JoinPoint → Boolean, a pattern is a function with the type Pattern: JoinPoint → Boolean ⋃ Pattern. A pointcut and a pattern take a join point and return a boolean value to determine whether there is a match or not with this join point. In addition, a pattern can return a (sub)pattern, which specifies next join points that should be matched by the pattern[3] (this approach is inspired by *continuation-passing style* [33]). For example, the implementation of a pattern to match a call to the up function of a touch device is:

```
(: s-up Pattern) ;;Pattern is the type name for "JoinPoint -> Boolean U Pattern"
(define (s-up jp)
  (eq? jp up))
```

The function above returns true if it matches the call to the up function; it returns false otherwise. The first line is used to define the type of a function in Typed Racket; in this code, the type of s-up is Pattern. The last line compares the references between up and jp, where jp is the function reference that is calling at that moment. In the piece of code above, the pattern can never return a pattern, meaning that the pattern behavior is equivalent to a pointcut.

```
(: call (Procedure -> Pattern))
(define (call fun)
  (λ (jp)
    (eq? jp fun)))

(define s-up (call up))
(define s-down (call down))
```

We can use higher-order functions to define patterns designators (*i.e.* functions that return patterns), which allow developers to reuse code and simplify the definitions of patterns. For example, the piece of code above shows the definition of the pattern designator call and its use to define the patterns s-up and s-down.

```
(: seq (Pattern Pattern -> Pattern))
(define (seq left right)
  (λ (jp)
    (let ([result (left jp)])
      (cond
        [(Pattern? result) (seq result right)]
        [(eq? result #t) right]
        [else #f]))))
```

In ESA pattern language, we can compose patterns. For example, the piece of code above is the implementation of the seq pattern designator, which is used to match a sequence of two patterns. The returned pattern by seq is used to match a sequence of a left pattern followed by a right pattern. The piece of code above shows that depending on the left evaluation, different values are returned. If left evaluation returns another pattern, a sequence pattern that is composed of

[3]Section 2.1 defines a pattern as a specification only (*e.g.* regular expression). In our proposal, a pattern is a function that can take actions (*e.g.* match).

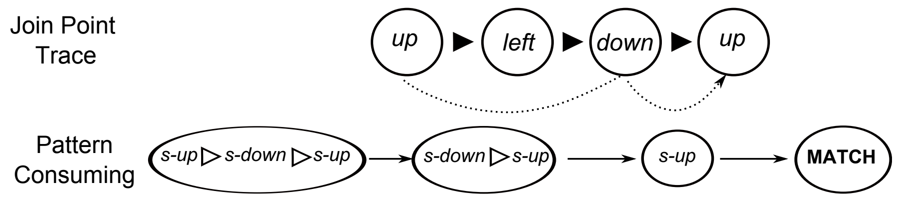


Figure 10: How a pattern is consumed during its matching.

the continuation of left and right is returned (Line 6). If left evaluation returns #t (*i.e.* true), right is only returned due to left matched completely (Line 7). Finally, if left does not match the current join point, false is returned (Line 8). Notice values returned on the lines 6 and 7 are patterns, which specify the next join points that must be matched.

```
(: toggle-airplane-mode Pattern)
(define toggle-airplane-mode (seq s-up (seq s-down s-up)))
```

We illustrate the use of seq to define the pattern of the *toggle airplane mode* feature. Figure 10 shows how the toggle-airplane-mode pattern (Section 1), defined by the piece of code above, varies throughout the matching of a program execution trace. This pattern changes every time it matches a join point, notice a pattern is drawn with non-filled triangle. In the beginning, the pattern begins with the pattern expressed by a programmer. For the first call to up, the pattern changes to the pattern *s-down* ▷ *s-up*. With the down call, the pattern changes to only *s-up*. Finally, once the user touches up on the screen, the whole pattern matches.

Using our pattern language, we can easily reuse patterns to create advanced ones. For example, the piece of code below shows the pattern toggle-airplane-mode-with-time, which reuses toggle-airplane-mode. In addition, the piece of code shows the seqn pattern designator, which reuses seq to create a pattern that matches a variable-size sequence of patterns. The foldl function, also known as reduce and accumulate, processes a list of patterns in the left order to return a new pattern.

```
(: toggle-airplane-mode-with-time Pattern)
(define toggle-airplane-mode-with-time (seq (call left) toggle-airplane-mode))

(: seqn ((Listof Pattern) -> Pattern))
(define (seqn patterns)
  (foldl (λ (pattern acc-pattern) (seq acc-pattern pattern))
         (first patterns) (rest patterns)))
```

5.1.2. Gathering Bindings in a Environment

The previous description of our pattern language is incomplete because a pattern cannot gather bindings while it is matching. Pointcuts and patterns should

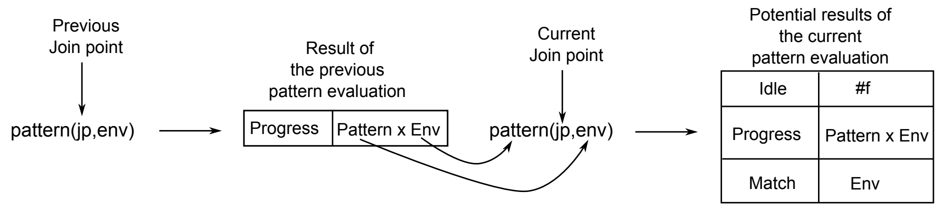


Figure 11: Evaluation of a pattern and its potential kinds of results.

be able to gather bindings. The standard and complete vision of the pointcut-advice model establishes a pointcut as a function Pc: JoinPoint $\rightarrow$ Env $\bigcup$ False. This definition means that a pointcut returns an environment of bindings (instead of true) if the pointcut matches the current join point. The ESA pattern language is an extension of the standard pointcut-advice model because a pattern is a function:

$$\text{Pattern: JoinPoint} \times \text{Env} \rightarrow \text{Env} \bigcup \text{False} \bigcup \text{Pattern} \times \text{Env}$$

A pattern now also takes an environment as a parameter. As Figure 11 shows, this environment contains the bindings previously gathered by a pattern. Like pointcuts, a pattern returns an environment when it matches. In addition, a pattern can return a pair (instead of a pattern only) composed of a pattern and an environment if the pattern *progresses* in its matching. To exemplify the extension of the pattern language, we redefine the pattern designators call and seq:

```
(: call (Procedure -> Pattern))
(define (call fun)
  (λ (jp env)
    (if (eq? jp env) env #f)))
```

```
1 (: seq (Pattern Pattern -> Pattern))
2 (define (seq left right)
3   (λ (jp env)
4     (let ([result (left jp env)])
5       (cond
6         [(Env? result) (cons right result)]
7         [(pair? result) (cons (seq (get-pat result) right) (get-env result))]
8         [else #f]))))
```

Patterns now take a join point and an environment, and returns an environment when the pattern matches. The environment is an object with functional behavior. Line 6 shows that the returned pattern by seq returns a pair that is composed of right and the environment gathered by the left evaluation. Line 7 returns the same previous pair with the difference that right is exchanged for the continuation of left with right. The functions with a name like get-xxx are getter functions that return xxx from an entity.

We illustrate the power of our pattern language through an enhancement of the feature *toggle airplane mode*. This feature is now triggered only if the trace *up* ▶ *down* ▶ *up* are performed within of a time interval of five seconds:

```
(define fast−toggle−airplane−mode
  (seqn (list (bind s−up 't0 get−time)
               s−down
              (time−diff (bind s−up 't1 get−time) t1 t0 5))))

;;where 'bind' is
(define (bind pattern id proc)
  (λ (jp env)
    (let ([result (pattern jp env)])
      (cond
       [(Env? result) (add−env result id (proc))]
       [else result]))))

;;where 'time−diff' is
(define (time−diff pattern t1 t0 time)
  (λ (jp env)
    (let ([result (pattern jp env)])
      (if (and (Env? result) (< (env−lookup result t1) (env−lookup result t0) time)))
        env result)))
```

To express the fast-toggle-airplane-mode pattern, we define two reusable and composable patterns: bind and time-diff. The first pattern binds a value when the passed pattern matches. The second one checks the time difference between two bindings stored in the environment when the pattern passed as argument matches.

An object-oriented design. The reader might wonder if the ESA pattern can be implemented on an object-oriented language (*e.g.* Java). It is not so difficult to design an object-oriented solution of this pattern language. Indeed, the ESA pattern language just defines a protocol that developers must follow. For instance, a possible object-oriented design defines a pattern as an interface with a match method, which is executed for every new join point; and pattern designators are (static) methods of a class (see Appendix A for more details).

5.2. Semantics

In most stateful aspect languages, aspects of a language share the same exact semantics [6, 8, 10, 11, 12, 13]. In ESA, every stateful aspect shares the same default semantics, which can be customized by developers. In order to follow the guidelines of open implementations, we use different and independent abstractions of any particular stateful aspect language implementation. Indeed, we use MatcherCells [34], a flexible algorithm to match program execution traces. In this section, we first explain the previous algorithm and then use to open the semantics of a stateful aspect.

5.2.1. MatcherCells

To flexibly match join point traces, MatcherCells use *self-replicating algorithms* [35], algorithms that emulate the reactions of a set of biological *cells* to a trace of *reagents*. The reaction of a cell to a reagent can be the creation of an identical copy of itself with a small variation in order to persist in the environment, nothing, death, or some of these combinations. An algorithm that follows self-replicating behavior is defined by a pair, where the first element is the set

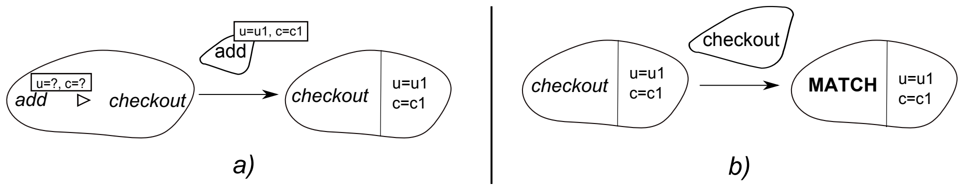


Figure 12: *a)* The left cell creates a cell that expects to match the next join point and gathers bindings. *b)* When a cell matches the last join point specified by a pattern, the cell creates a *match cell*.

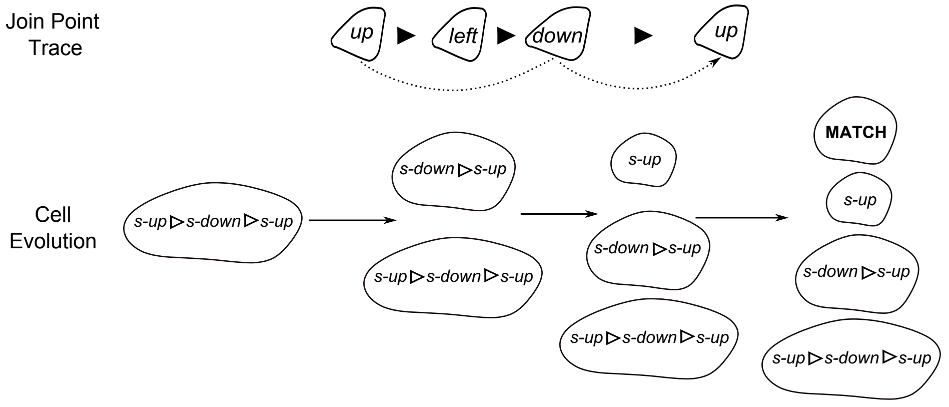


Figure 13: Evolution of a seed during the matching of a pattern.

of first cells (*a.k.a.* seeds) and the second one is the set of rules that governs the evolution of these cells.

In MatcherCells, a cell contains the pattern of a stateful aspect, bindings gathered during the matching, and a reference to its creator. Cells react to join points, which correspond to reagents. Using the example of the *discount* feature, Figure 12a shows that if a cell matches a join point, this cell creates a new cell that expects to match the next join point specified by the pattern. In addition, this new cell contains an environment of bindings gathered when the join point was matched. Figure 12b shows that when there is no next join point to match, a *match cell* is created to indicate a match of a pattern.

Using the example of the *toggle airplane mode* feature, Figure 13 shows the evolution from a seed, which creates a set of cells during a join point trace. The figure also shows that there is a match cell when the cell with the *s-up* pattern matches. As the cell with *s-up* is never killed (actually, any cell is never killed in the figure), each subsequent *up* join point will trigger a new match of the pattern.

MatcherCells allow developers to add rules to control the evolution of a set of cells. Figure 14 shows four different evolutions, where each one has a different

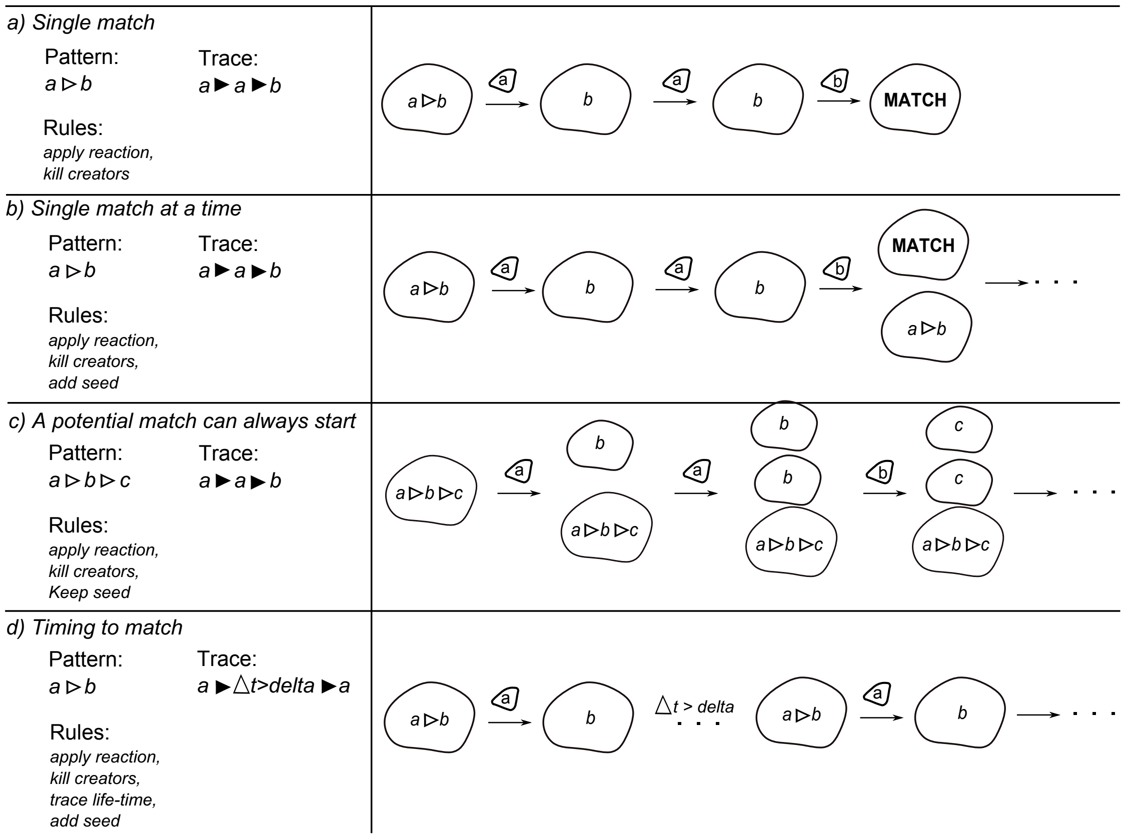


Figure 14: Different matching semantics to match a pattern (figure adapted from [34]).

set of rules that are used to customize the matching semantics of a pattern. Although evolutions have different rules, we can appreciate that all evolutions have the *apply reaction* rule, which only applies the reaction of each cell to a join point. Figure 14a shows that the *kill creators* rule kills the cells that create a new cell. Adding this rule, a pattern cannot match multiple times anymore. Figure 14b shows that the *add seed* rule adds a seed if there are no cells or only match cells. This rule allows a pattern to match again. Figure 14c shows that the *keep seed* rule always keeps a seed to permit to start a new potential match of a pattern at any moment. Finally, Figure 14d shows that the *life-time for a trace* rule kills all cells whose period of time of the join point trace time has exceeded a determined period. This rule allows developers to only match traces of join points that occur at a period of time.

Inspired by the *Decorator* design pattern [16], rules of MatcherCells are functions, which can be composed in order to customize the matching semantics. For instance, the *single match* semantics (Figure 14a) is achieved by the composition of *kill creators* with *apply reaction*. In the two sections, we explain how these rules are defined and used to support the customization of the processes of matching and advising of each stateful aspect in ESA.

AOP terminology. To avoid confusion to the reader, we will keep the use of AOP terminology in the rest of the paper. Apart from replacing reagents with join points, we replace cell with *smatch* (a stage in the match of a pattern) and match cells with the term match. We keep the terms seed and rules because we think these terms can be used without misunderstandings.

5.2.2. Matching Process

Unlike existing proposals, every ESA stateful aspect has own matching process that can be customized by developers (see Figure 9). We use MatcherCells to allow developers to define a matching process.

When an ESA stateful aspect is deployed, its matching process creates a seed, a smatch that contains the pattern of the aspect. The matching process evaluates the seed with every new join point. If the seed matches a join point, this seed can create other smatches. If any smatch is a match, it means that the stateful aspect matches its pattern. Depending on the used composition of rules, a stateful aspect might match multiple times, even at the same time.

$$\text{react: SMatch} \times \text{JoinPoint} \times [\text{Env} \times \text{Pattern} \times \text{SMatch} \rightarrow \text{Env}] \rightarrow \text{SMatch}$$

A smatch uses the react function to evaluate a join point. If the smatch matches the join point, the function returns a new smatch, otherwise the function returns the same smatch. If there is no next join point to match, the new smatch is really a match. The last and optional parameter[4] of react is a function that allows developers to add information to a smatch in its creation (more on this at the end of this section).

$$\text{rule: List<SMatch>} \times \text{JoinPoint} \rightarrow \text{List<SMatch>}$$

A rule is a function that takes as parameters a list of smatches and a join point, and returns the list of smatches that are evaluated with the next join point. For example, the *apply reaction* rule implementation is:

```
1 (: apply−reaction Rule)
2 (define (apply−reaction smatches jp)
3   (remove−duplicates (append smatches
4                               (map (λ (smatch) (react smatch jp)) smatches))))
```

The apply-reaction function returns the smatches reactions. A smatch, whose reaction is itself, is in the list of smatches and their reactions (Line 4). This means that this smatch is duplicated when both lists are joined. To prevent this duplication, the remove-duplicates function is used. Using rule designators (*i.e.* functions that return rules), developers are able to create rules that can be composed:

[4]The notation [...] denotes an optional parameter.

```
(: kill-creators (Rule -> Rule))
(define (kill-creators rule)
  (λ (smatches jp)
    (let ([next-smatches (rule smatches jp)])
     (diff next-smatches (get-creators (get-sons next-smatches smatches))))))

(: add-seed (Pattern -> (Rule -> Rule)))
(define (add-seed pattern)
  (λ (rule)
    (λ (smatches jp)
      (let ([next-smatches (rule smatches jp)])
        (if (empty? (filter no-match? next-smatches))
           (cons (make-seed pattern) next-smatches)
           next-smatches)))))

(: keep-seed (Pattern -> (Rule -> Rule)))
(define (keep-seed pattern)
  (λ (rule)
    (λ (smatches jp)
      (let ([next-smatches (rule smatches jp)])
       (if (= (count-seeds next-smatches) 0)
        (cons (make-seed pattern) next-smatches)
        next-smatches)))))
```

These rule designators are parametrized by a rule, which corresponds to the rule that should be applied, in most cases, before the current one. The rule returned by kill-creators first applies a previous rule (*e.g.* apply-reaction) to obtain a list of smatches, where the smatches that created new ones are removed for the evaluation with the next join point. The add-seed[5] rule designator returns a rule that adds a seed if there are no smatches. Finally, keep-seed always keeps a seed. The composition of rules allows developers to define the full semantics of a matching process. For example, the compositions of rules to obtain the different semantics of Figure 14 are:

```
(define single-match  (kill-creators apply-reaction))
(define single-match-at-a-time ((add-seed pattern) single-match))
(define a-potential-match-can-always-start ((keep-seed pattern) single-match))
(define timing-to-match ((add-seed pattern) ((trace-life-time delta) single-match)))
```

Adding context information to smatches. Some rules may need that all smatches contain specific context information. For example, trace-life-time needs that all smatches contain the time in their environments when a potential match starts:

```
(: trace-life-time (Number -> (Rule -> Rule)))
(define (trace-life-time delta)
  (λ (rule)
    (λ (smatches jp)
      (let ([next-smatches (rule smatches jp)])
       (filter (λ (smatch)
                 (< (- (get-time) (env-lookup (get-env smatch) 'time))
```

[5]Notice that add-seed is in fact a higher-order rule designator, parameterized by the original pattern or a new one.

```
                    delta))
                smatches)))))
```

To add context information to smatches, the third and optional parameter of the react function is used. This parameter is a function that receives the values with which a smatch will be created, and returns the initial environment of bindings of a smatch. For example, to annotate a smatch with the trace time, one needs to provide the following function:

```
(: creation-with-time (Env Pattern SMatch -> Env))
(define (creation-with-time env pattern creator)
  (env-add env 'time (if (Seed? creator)
                         (get-time)
                         (env-lookup (get-env creator) 'time))))
```

In some scenarios like trace-life-time, intrinsic attributes (*e.g.* time, physical space, etc) should be present in each smatch (from seed to match) in order to support operations between smatches. In other words, these attributes crosscut through the matching of a trace [36]. The feature of customized information of smatches allows developers to modularize the treatment of these intrinsic attributes.

5.2.3. Advising Process

Like the ESA matching process, the advising process is open to customization. When one or more smatches become matches, the advising is executed with these matches. This process executes the aspect advice with every match. ESA entirely reifies this process through a function with the following signature:

$$\text{AdvisingProcess: Advice} \times \text{List<Match>} \times \text{JoinPoint} \rightarrow \text{AdviceReturn}$$
$$\textit{where}$$
$$\text{Advice: JoinPoint} \times \text{Env} \rightarrow \text{AdviceReturn}$$

An AdvisingProcess function takes three parameters: the aspect advice, the list of matches of the pattern, and the current join point. An ESA advice also follows the standard vision of the pointcut-advice model [4], meaning that an advice is a function parametrized by the matched join point and an environment of bindings. The list contains the matches obtained when the matching process used the current join point to evaluate their smatches. The type of the value returned by the AdvisingProcess and Advice functions must be the same (*i.e.* AdviceReturn). We illustrate the open ESA advising process with two different matching processes:

```
(: single-advice-execution AdvisingProcess)
(define (single-advice-execution advice matches jp)
  (advice jp (get-env (first matches))))


(: simultaneous-advice-executions AdvisingProcess)
(define (simultaneous-advice-executions advice matches jp)
  (last (map (λ (match) (advice jp (get-env match))) matches)))
```

The single-advice-execution function corresponds to an advising process that only executes its advice once with the first match (discarding the rest of matches if there are). This advising semantics is useful, for example, for implementing the *toggle airplane mode* feature (Section 1) because simultaneous advice executions generate unexpected results in the airplane mode of a touch device. Instead, the multi-advice-executions function represents the consecutive advice executions, where the value of the last execution is returned. In Section 2.3, the tracematch implementation for the *discount policy* feature takes advantages the semantics of simultaneous advice executions to apply the discounts to every computer.

5.3. Stateful Aspects in ESA

Finally, we describe how to define and weave a stateful aspect in our proposal.

5.3.1. Defining a Stateful Aspect

We have described the core elements of ESA separately so far. This section now presents how these elements are integrated to make a stateful aspect.

make-aspect: Pattern $\times$ Advice $\times$ [Rule] $\times$ [AdvisingProcess] $\rightarrow$ StatefulAspect

The make-aspect function takes a pattern, an advice, and two more optional parameters to create a stateful aspect. The first optional parameter, which is a rule, is used to define matching process, and the second one allows developers to define the advising process. For example, the following stateful aspect implements the *toggle airplane mode* feature:

```
(make-aspect (seqn (list s-up s-down s-up)) (λ (jp env) (toggle-airplane-mode))
             single-match-at-a-time single-advice-execution)
```

As *toggle airplane mode* requires a touch device enables/disables the airplane mode only every trace of *up* ▶ *down* ▶ *up*, the single-match-at-a-time rule (Section 5.2.2) corresponds to the adequate matching semantics for this aspect. As mentioned in Section 5.2.3, simultaneous advice executions may unexpected results in the deployment of this feature; thereby, the advising process represented by the single-advice-execution function is needed.

Default semantics for an ESA stateful aspect. Multiple matches and simultaneous advice executions are distinguishing characteristics of most stateful aspect languages. However, the understanding of these features is complex for developers because it is necessary to reason about multiple and simultaneous potential triggers of a stateful aspect. Therefore, we establish that the default semantics for every ESA stateful aspect only permits a single match and advice execution. Taking into account this default semantics, we can simplify the definition of the previous stateful aspect as follow:

```
(make-aspect (seqn (list s-up s-down s-up)) (λ (jp env) (toggle-airplane-mode))
```

5.3.2. *Weaving a Stateful Aspect*

The weaving of a stateful aspect consists in evolving its list of smatches and executing the advice with each match found:

$$\mathsf{weave: StatefulAspect} \times \mathsf{JoinPoint} \rightarrow \mathsf{AdviceReturn}$$

```
(define (weave asp jp)
  (let
    ([temp-smatches ((get-rule asp) (get-smatches asp) jp)]
     [matches (filter is-match? temp-smatches)])
  (begin
    (update-smatches asp (filter is-not-match? temp-smatches))
    (if (> (length matches) 0)
      ;;execute advice with bindings of each match cell
      ;;else execute the join point proceed
))))
```

The evolution of the list of smatches is determined by a rule, which represents the matching process of a stateful aspect. If matches are found after the rule is applied, these matches are removed from the list and the advice is executed for every one of them.

5.4. *Summary*

We have described the three components of ESA. Developers can reuse and compose patterns (consequence $p1$) with a Turing complete language (consequence $p2$). The semantics of stateful aspects can be customized by each stateful aspects (consequences $s1$ and $s2$). The design of these components in ESA are modular (*e.g.* the pattern language is only a protocol that the matching process must follow).

6. ESA-JS: ESA for JavaScript

ESA-JS is a complete and practical implementation of ESA for JavaScript, dynamic prototype-based language with higher-order functions. With ESA-JS, we address the limitations of existing stateful aspect languages described in Section 4. Apart from ESA-JS, we have developed a version of ESA for ActionScript, named ESA-AS3. Although this last implementation is not described here because of space reasons, ESA-AS3 is available on the ESA Web site (http://pleiad.cl/esa).

A brief overview of AspectScript. ESA-JS is currently implemented as a seamless extension of AspectScript [37], an aspect language for JavaScript. In AspectScript, pointcuts and advices are functions; and aspects and join points are objects. In addition, AspectScript supports dynamic deployment of aspects with expressive strategies of scoping [38].

```
1 var toggleAirplaneMode = {
2   pattern: seqn([s-up,s-down,s-up]),
3   advice: function(jp,env) {
4      Device.toogleAirplaneMode();
5   },
6   kind: ESA.AFTER
7 };
8
9 ESA.deploy(toogleAirplaneMode);
```

The piece of code above shows the implementation of the *toggle airplane mode* feature in ESA-JS. An ESA-JS stateful aspect is a JavaScript object (Line 1). In this implementation of ESA, the patterns and advices are functions (lines 2-3) , *i.e.* first-class values in JavaScript. As in AspectScript, ESA-JS also supports dynamic deployment of stateful aspects (Line 9). Notice this piece of code represents a correct implementation of the feature because of default semantics of ESA (*i.e. single match at a time* and *single advice execution*).

6.1. Pattern Language

In Section 4.1, the solution presented for the *limited discount policy* has the problem that its advice is *stateful*. This is because existing stateful aspect languages are insufficiently expressive to allow developers to define patterns that gather a variable-size of list of bindings. Using ESA-JS, we can define patterns that gather a variable-sized list of bindings, implying a stateful advice is not need anymore:

```
var limitedDP = {
  pattern: starUntil(addComputer, call(checkout)),
  advice: function(jp,env) {
    var computerList = env.computers;
    var computersWithBestDiscounts = getBestDiscounts(computerList);
    //apply discounts to the computers with the best discounts
  },
  kind: ESA.BEFORE
};
```

The pattern of `limitedDiscountPolicy` matches and gathers all computers added to the cart until the user does check out. The advice of this aspect simply applies the three best discounts of the list of computers stored in the environment, `computerList`. The piece of code below shows the implementation of the pattern designators `starUntil` and `addComputer`:

```
var starUntil = function (star, until) {
  return function (jp,env) {
    var result = until(jp,env);
    if (isEnv(result)) {
      return result;
    }

    result = star(jp,env);
    if(isEnv(result)) {
      return [starUntil(star, until),result];
    }
    return false;
} };

var addComputer = bind(call(cart.add), function(jp,env) {
    var addedComputer = jp.args[0]; //1st argument passed to the function "cart.add"
    return env.bind("computers", addedComputer);
  });
```

The starUntil pattern designator returns a pattern that matches the star pattern zero or more times until the until pattern matches. The pattern returned by addComputer matches a call to the cart.add method. In addition, the returned pattern, using the env.bind method, binds the computers identifier to the added computer. In ESA-JS, if two values are bound to the same identifier (*e.g.* computers), they are aggregated as a list, where the most recent value is added at the end of the list.

6.2. Matching Process

Before our proposal, developers cannot customize the matching semantics of a stateful aspect. Therefore, developers must code around the advice to address this limitation. Section 4.2 exemplifies this limitation with the *selective discount policy* of the online store. This extension to the discount policy establishes that one computer by category must apply its discount. Neither a modification to the pattern nor to the advice of the stateful aspect is needed to implement this discount policy extension in ESA-JS. Only the customization of the matching process is required:

```
var selectiveDP = limitedDP;  //reusing the aspect limitedDP

selectiveDP.matchingProcess =
  keepSeed(selectiveDP.pattern)(filterByCategory(killCreators(applyReaction)));
```

The composition of rule above represents the *potential match can always start* matching semantics (Figure 14c) plus the filtering by categories. After applying the reaction to each smatch and killing the smatches that created new ones, the rule returned by the evaluation of filterByCategory removes the new smatches that contain computers whose categories are not different from current ones. The implementation of the filterByCategory rule designator is:

```
var filterByCategory = function(rule) {
  return function(smatches,jp) {
    var nextSmatches = rule(smatches,jp);
    var categories = getCategories(getComputers(nextSmatches));

    //newSmatches are the difference between next and current smatches
    var newSmatches = diff(nextSmatches,smatches);

    var smatchesWithRepeatedCategories = newSmatches.filter(function(newSmatch) {
      return categories.some(function(category) {
         return getCategory(newSmatch.env.computer) == category;
      });
    });

    //returns nextSmatches without the smatches (recently added) with repeated categories
    return diff(nextSmatches, smatchesWithRepeatedCategories);
} };
```

Categories of every computer are obtained from the environment of bindings that each smatch contains. The category of the each new smatch is compared to current categories to know whether this category is new or not. Finally, only the new smatches with new categories are returned and kept inside of the stateful aspect.

Multiple matches based on binding dependences. In the piece of code above, notice that the decision of filtering smatches depends on category, a binding that is not directly included in the pattern definition. Unlike ESA-JS, the matching process is closed for existing stateful aspect languages, therefore, it is not possible that their semantics of multiple matches depends on bindings that are not specified on the definition of a pattern (*e.g.* Tracematches [10]).

6.3. Advising Process

The advising semantics is also closed and fixed for stateful aspect developers in the existing proposals. Thereby, such as Section 4.3 showed, the implementation of the *personalized discount policy* needs to intrusively modify the advice of the aspect presented.

The openness of the matching process is insufficient to modularly implement the new discount policy. This is so because the computer with the greater number of customized pieces is only known when the user does check out (*i.e.* inside of the advising process). The *personalized discount policy* requires that the aspect only executes its advice, which applies the discounts, with an environment that contains the computer with more customized pieces. Using ESA-JS, it is not again necessary to modify the advice:

```
var personalizedDP = limitedDP;  //reusing the aspect limitedDP

personalizedDP.advising = function(advices,matches,jp) {
  var env = matches[0].env;
  var computerList = env.computers;

  //obtaining the computer with more customized pieces
  var computerWithMorePieces = computerList.max(function(computer1,computer2) {
    return getCustomizedPiecesNumber(computer2) - getCustomizedPiecesNumber(computer1);
  });

  //replacing the list of computer with only one computer
  env.computers = [computerWithMorePieces];
  advice(jp,env);
};
```

As we can appreciate above, the computer with more customized pieces, computerWithMorePieces is obtained from the environment of the smatch. Then, env.computers is replaced with a new array that only contains the binding computerWithMorePieces. Finally, the advice with this modified environment.

```
var counter = 0;
var aspect = {
  pointcut: call(foo),
  advice: function(jp,env) {
    if (++counter == PATTERN_LENGTH) {
      print("match");
      counter = 0; }
  },
  kind: AspectScript.BEFORE
}
```

```
var statefulAspect = {
  pattern: repeatPattern(foo,PATTERN_LENGTH),
  advice: function(jp,env) {
       print("match");
  },
  matching://depends on the experiment (single or multiple)
  kind: ESA.BEFORE
}
```

Figure 15: *(Left)* AspectScript aspect used for the experiment. *(Right)* ESA stateful aspect used for the experiment.

6.4. Performance

We ran performance tests of ESA-JS. The results of these tests were compared to Tracematches. For this experiment, an Intel Core 2 Duo, 2.66 GHz with 2 GB of RAM. Regarding software, we used Ubuntu 10.04 (kernel 2.6.32) with the Firefox JavaScript interpreter (version 1.8.0) for ESA-JS and the *abc*[6] compiler (version 1.3.0) for Tracematches.

```
//PATTERN_LENGTH varies from 5 to 30
start = getCurrentTime();
for (i = 0; i < PATTERN_LENGTH; ++i) {
  foo();
}
delta = getCurrentTime() - start;
```

The experiment measured the average time used to execute the piece of code above with an AspectScript aspect and an ESA-JS stateful aspect (Figure 15). The AspectScript aspect keeps a counter of matches and the ESA-JS stateful aspect with two different semantics, single match (the ESA default) and multiple match semantics. Finally, we execute the same experiment with an aspect of AspectJ and a stateful aspect of Tracematches. For the experiment, the base code above together with each aspect implementation is executed 500,000 times. The experiment was repeated for patterns of different lengths (from 5 to 30).

ESA-JS and Tracematches are aspect language extensions of AspectScript and AspectJ respectively. Figures 16 and 17 show the increment of the overhead of ESA-JS and Tracematches over their aspect languages. The overhead of ESA-JS is evidently less than Tracematches, and the Tracematches overhead quickly increases when the pattern is longer (which may be due to the index scheme used in long patterns [39]). In addition, Figure 16 shows the choice of the semantics of matching process strongly affects performance of a ESA-JS stateful aspect: performance of the default semantics is quite similar to an

[6]http://www.sable.mcgill.ca/abc/

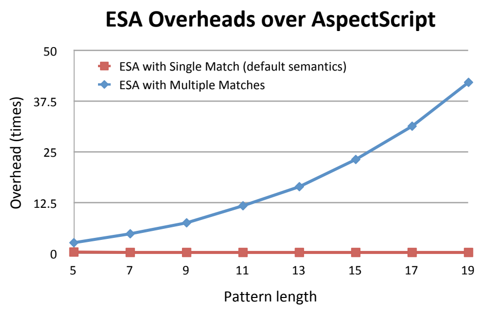


Figure 16: Overhead of ESA with two different matching semantics over AspectScript.

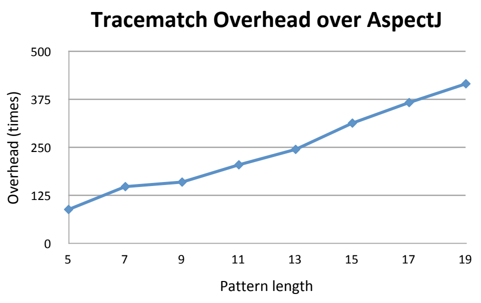


Figure 17: Overhead of Tracematches over AspectJ.

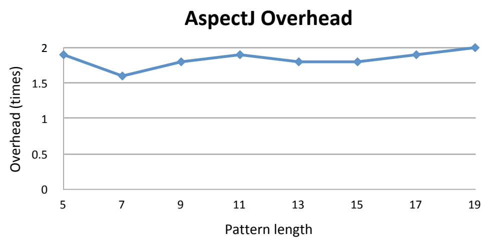


Figure 18: The overhead of AspectJ over the Java language.

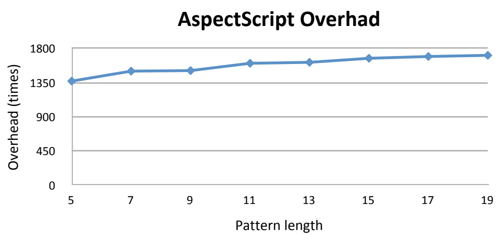


Figure 19: AspectScript overhead over JavaScript.

aspect of AspectScript; instead, the stateful aspect that uses the multiple match semantics differs from the aspect implementation in an exponential manner.

Although figures 16 and 17 show a less overhead in ESA-JS, these results do not mean that ESA-JS has less overhead for JavaScript than Tracematches for Java. Figures 18 and 19 shows the overhead of AspectScript is significantly greater than AspectJ. The AspectJ performance is very similar to Java (average of 1.3) instead of the AspectScript performance (average of 1,800).

Discussion. The current implementation of AspectScript is very slow compared to AspectJ. This so because AspectScript reifies every join point of the program execution to know whether an aspect matches or not at runtime; no partial evaluation or other optimization is performed. Instead, AspectJ optimizes aspect weaving aggressively, therefore the additional layer introduced by Tracematches is comparatively more costly. In addition, the choice of the matching process significantly impacts on the ESA-JS overhead. Finally, raw ESA-JS overhead in this experiment does not really reflect the fact that JavaScript is more widely used for interactive applications, that may even include remote communication. For instance, we have tested ESA-JS on a JavaScript Tetris game, without any noticeable difference[7].

7. Assessing the Expressiveness of ESA

We assess the expressiveness of ESA through the emulation of the semantics of some stateful aspect languages. For reading comprehension reasons, we use ESA-JS to express the semantics of Alpha [13], Halo [12], and Tracematches [10]. For each stateful aspect language, we show the necessary customizations of matching and advising processes.

7.1. Alpha

As Alpha uses Prolog, a pattern is really a query that is answered using a *backward chaining* algorithm [23]. Searching a data base, this algorithm finds a set of different answers for a query. Each answer is represented by an environment of bindings. In Alpha, the data base and the set of answers corresponds to a join point trace and the matches of a pattern respectively. The previous point means that if, in Alpha, a pattern matches twice or more times simultaneously, the advice is only executed using matches whose environments of bindings gathered at least differ in one binding (*e.g.* the environments $\boxed{\text{x=1 , y=1}}$ and $\boxed{\text{x=1 , y=2}}$ are different because of y).

Matching process. Alpha stateful aspects support the multiple matches of a pattern without any restriction, therefore, the matching process only uses the applyReaction rule (Section 5.2.2):

```
var alphaMatching = applyReaction;
```

Advising process. If there are two or more matches of a pattern simultaneously, the advice is only executed with matches with different bindings. To achieve this goal, we customize the function of the advice process (Section 6.3):

```
var alphaAdvising = function(advice,matches,jp){
  var envs = getEnvs(matches);
  //filtering environments that contain the same contextual information
  var filteredEnvs = envs.removesDuplicates(function(env1,env2) {
    return equal(env1,env2); //same bindings?
  });

  return last(consecutiveAdviceExecutions(jp,filteredEnvs));
};

//where
var consecutiveAdviceExecutions = function(jp,envs) {
  return envs.map(function(env) {
    return advice(jp,env);
  });
};
```

[7]The Tetris game can be tested online on the ESA Website: http://pleiad.cl/esa.

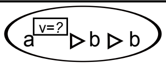
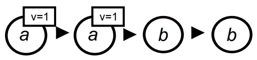
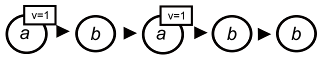
Pattern		
Join Point Trace	Alpha	Halo
	1	1
	2	1

Figure 20: For the same the join point trace, the subtle difference between the semantics of Halo and Alpha causes a different number of matches for the same pattern.

The alphaAdvising function first filters environments in order to only catch the environments with different bindings. Finally, the function executes the advice in a consecutive manner with each environment of filteredEnvs.

7.2. Halo

Halo uses the Rete algorithm [24] to match patterns. Unlike Alpha, this algorithm is based on *forward chaining* [23], meaning that Halo signals that if all potential matches of a pattern must at least differ in one binding. Although the Halo semantics is subtly different from Alpha, this difference causes different matches of a pattern. Figure 20 illustrates this difference, we evaluates Halo and Alpha with the same pattern and two join point traces. With the first join point trace, we can see that the number of matches for both proposals is one. This is because the pattern is simultaneously matched twice, but as both matches have the environment of bindings ($\boxed{\text{v=1}}$), a match is discarded. With the second join point trace, we can instead observe different number of matches. In Alpha, the are two matches of the pattern because its advising process actually filters matches with the same environment, and in this example, two different join point trigger the matches. With Halo, there is only one match because the filters starts inside of the matching process inside of the advising process.

Matching Process. As Halo filters the potential matches during the matching process, we need a rule (designator) to carry out this filter:

```
var differentBindings = function(rule) {
  return function(smatches,jp) {
    var nextSmatches = rule(smatches,jp);
    var newSmatches = difference(nextSmatches,smatches);

    //filtering new smatches with the same environment
    var filteredNewSmatches = newSmatches.filter(function(newSmatch) {
      var oldSisters = sisters(newSmatch,smatches); //get old sisters of newSmatch

      return oldSisters.some(function(oldSister) {
        return equalEnv(newSmatch,oldSister);
12        });
13      });
14
15      return difference(nextSmatches,filteredNewSmatches);
16  } };
```

The differentBindings rule designator allows developers to filters smatches with different environments of bindings. The rule returned by differentBindings only keeps a new smatch if its bindings are different from all its sisters or its creator is a seed (lines 7-15). Finally, the rule composition to express the matching process of Halo is:

```
haloMatching = keepSeed(pattern)(differentBindings(applyReaction));
```

Advising Process. In Halo, the advice process only executes the advice with every match in a consecutive manner:

```
var haloAdvising = function(advice,matches,jp){
  return last(consecutiveAdviceExecutions(jp,getEnvs(matches)));
}
```

7.3. Tracematches

Like Halo, the semantics of Tracematches intuitively signals that the advice is only executed using matches with different bindings. Unfortunately, this intuitive semantics is not completely correct because it does not take into account the effect of *symbols* on the matching of a program execution trace. In Tracematches, the use of symbols, which is used to define patterns, is one of the defining characteristic of the proposal. However, this characteristic complicates the understandability of its semantics at first glance. For example, we see that with two similar patterns $a \triangleright b^v \triangleright c$ and $a \triangleright b \triangleright c$ and the following join point trace $a \blacktriangleright b^1 \blacktriangleright b^2 \blacktriangleright c$, there are two matches of the first pattern but there is *no match* of the second one. To correctly understand the semantics of Tracematches, it is necessary to know the symbol semantics.

7.3.1. Symbol Semantics

Like JavaMop [14, 28], Tracematches use symbols to define the pattern of stateful aspect. A tracematch symbol is composed of a pointcut plus a set of bindings gathered by the pointcut. The symbols are used to define patterns as regular expressions that should match. In addition, these symbols are used *a)* to discard evaluations of smatches with unnecessary join points, *b)* to remove smatches that do not satisfy conditions imposed by these symbols, and *c)* to selectively create smatches. We now illustrate these three points using Figure 21, which shows three tracematches ($t1$, $t2$, and $t3$), and Figure 22 that shows the number of matches of these tracematches for the three different Java program executions:

```
tracematch() {
  sym a after: call(* *.a(..));
  sym b after: call(* *.b(int));

  sym c after: call(* *.c(..));

   a b  {print("match t1");}
}
```

t1

```
tracematch(int v) {
  sym a after: call(* *.a(..));
  sym b after: call(* *.b(int))
        && args(v);
  sym c after: call(* *.c(..));

   a b  {print("match t2");}
}
```

t2

```
tracematch(int v) {
  sym a after: call(* *.a(..));
  sym b after: call(* *.b(int))
        && args(v);
  sym c after: call(* *.c(..));

   a b c {print("match t3");}
}
```

t3

Figure 21: Pieces of code of three tracematches: $t1$, $t2$, and $t3$.

Java Program	Number of matches of t1	t2	t3
a(); b(1); b(2); z(); c();	1	2	2
a(); b(1); b(1); c();	1	1	0
a(); b(1); b(2); b(1); c();	1	2	1

Figure 22: Numbers of matches for each tracematch.

First program. The tracematches $t2$ and $t3$ match twice, and $t1$ only matches once. Although $t1$ and $t2$ match the same program execution (a(); b(int);), the different values bound to v of the b symbol allows $t2$ to match twice. This is because a symbol is also identified by the set the of bindings gathered, meaning that calls b(1) and b(2) represent two different instances of the b symbol for $t2$ (see Figure 23a). Also, notice the call to the z function is not considered by either of the three tracematches because there is no symbol that matches this call. If there were a symbol that identifies the call to z in $t3$, there would not be any match in this program due to z does not appear between b and c of the $t3$ pattern ($a\ b\ c$) (see Figure 23b).

Second program. In $t2$, there is only matches once because in the two calls to b, the binding gathered is the same (v = 1). $t3$ matches zero times because its regular expression does not match with the trace of symbols $a\ b\ b\ c$ because there are two b symbols identified and not only one.

Third program. The binding gathered by the pointcut that matches the second call b(1) does not again allow $t3$ to match more times (see Figure 23c), meaning that $t3$ only matches the trace a(); b(2); c();.

We now express the Tracematches semantics in ESA-JS through the customizations of the matching and advising processes. To express the Tracematches matching process in ESA, we must adapt the three previous points of symbols semantics to our proposal: *a)* a smatch is only evaluated with join points that correspond to its expected next symbol, *b)* a smatch that matches a symbol that does not correspond to its next symbol is removed, and *c)* a smatch can only be created if its creator has not created another smatch

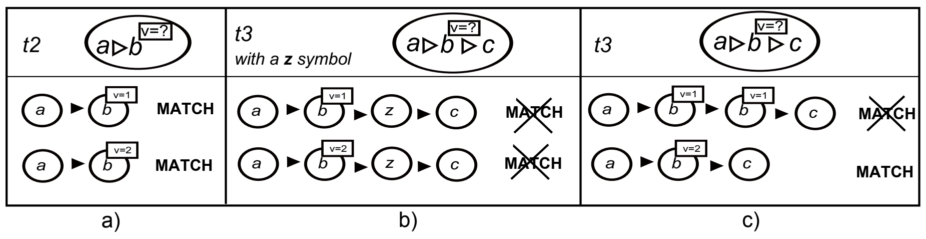


Figure 23: *a)* Two matches, each one with a different binding. *b)* No matches due to the z symbol. *c)* One match fails due to the symbol b with the same binding.

that corresponds to the same instance of a symbol (*e.g.* b(1) and b(2) are two different instances of the b symbol in $t2$ and $t3$). It is the c point which allows Tracematches to match multiple times a pattern. The advising process in Tracematches composes and executes advices like in AspectJ [3].

7.3.2. Matching Process

To achieve the points *a)* and *b)* of the previous section, it is necessary to define a new version of the applyReaction rule:

```
1  var tracematchApplyReaction = function(alphabet){
2    return function applyReaction(smatches,jp){
3      var nextSmatches = []; //empty array
4      //Visiting each smatch
5      smatches.foreach(function(smatch) {
6
7        if (DoesJPCorrespondTheExpectedSymbol(alphabet,jp,smatch)){
8          var nextSmatch = react(smatch,jp);
9          if(nextSmatch != smatch)
10           nextSmatches.push(smatch,nextSmatch);
11       }
12       else
13         nextSmatches.push(smatch);
14     });
15
16     return nextSmatches;
17 } };
```

The applyReaction rule is now a rule designator that takes as a parameter a set of definitions of symbols, named alphabet. For example, alphabet might contain the symbols a, b, and c of the three tracematches shown in Figure 21. To achieve the point *a)*, Line 7 shows that a smatch is only evaluated if the jp join point corresponds to the next expected symbol for this smatch. For example, this line allows the three tracematches to ignore the call to the function z in the first program of Figure 22. To achieve the point *b)*, the lines 10 and 13 show that smatch is not removed if smatch matches the next expected symbol (*i.e.* progress in its matching) or this smatch is not evaluated. In other words, a smatch is only removed if it matches an incorrect symbol. To satisfy the point *c)*, we use the differentBindings rule designator (Section 7.2) to filter duplicated instances of symbols (*i.e.* symbols with the same environment).

Finally, the rule composition to express the semantics of the Tracematches matching process is:

```
tracematchMatching = keepSeed(pattern)
                        (differentBindings(tracematchApplyReaction(alphabet)));
```

7.3.3. Advising Process

If the advice kind of a stateful aspect in Tracematches is before or after, the advising process executes the advice for every match in undetermined order[8]. Instead, if the advice kind is around, the advising process chains the advice executions, each one nesting the next one. Hence, we customize the advising process in the following way:

```
var tracematchesAdvising = function(advice,matches,jp) {
  var envs = getEnvs(matches);
  //randomize elements of envs
  envs = randomizeOrder(envs);

  //chain advice executions like AspectJ
  var chainedAdvices = chainAdvice(advice,envs,isAround());
  return chainedAdvices(jp);
}
```

Notice that it is not necessary to filter by environments that contain different bindings (like in Halo) because it is now filtered by the matching process.

7.4. Summary

Through the customization of matching and advice processes, we can instantiate different stateful aspect languages. In addition, any instantiation of ESA can enjoy of the expressive pattern language of our proposal. For example, any instantiation allows developers to define a pattern like $(a^v)^*$ that gathers one or more lists of bindings during its matching. The former pattern is not currently supported in most stateful aspect languages.

8. Conclusion

Because creating specialized stateful aspect languages or overburdening their aspects is a common task to address specify needs, we propose a precise description of an expressive stateful aspect language, named ESA. Our proposal is sufficiently expressive to encompass existing stateful aspect languages and new possible variants. ESA, which is accurately described in Typed Racket [18], concretely allows developers to *a)* use a Turing complete pattern language with full support for first-class patterns, bringing benefits of reusability and composition of patterns, and *b)* customize internal processes of each stateful aspect.

[8]In [10], the authors mention that have not defined any particular ordering on the advice executions. To the best of our knowledge, the authors do not discuss this subject again.

Using this description, we developed ESA-JS, a concrete and practical implementation of ESA for JavaScript. We illustrated and assessed the expressiveness of this proposal through several examples and implementing the semantics of some existing stateful aspect languages. To contrast ESA with existing proposals, we develop a deep reference frame that evaluates these proposals in terms of expressiveness.

Whereas the common concern for existing stateful aspect languages is performance, we explore a different and unusual concern such as expressiveness. Despite of our focus, we are aware that performance is important, thereby, the future of ESA is oriented towards to address this concern:

Elegible pattern language. In this proposal, developers can use a Turing complete language to define patterns. However, Turing expressiveness is not always necessary, *e.g. toggle airplane mode* (Section 1). Similar to JavaMop [14], we plan to allow developers to select the pattern language expressiveness. With this, ESA improves performance according to specific features of the selected pattern language.

Matching process. Although we showed that the selection of the semantics of the matching process can improve performance (Section 6.4), we think some semantics (*e.g.* multiple matches) needs to improve its performance. Bodden *et al.* in [40, 41, 42, 43] and Meredith's dissertation [29] studied several proposals in this line of research, *e.g. dependency advices* [41].

Availability. ESA-JS (and ESA-AS3) along with the examples presented in this paper, is available online at http://pleiad.cl/esa. ESA-JS currently supports the Firefox, Safari, Chrome, and Opera browsers without the need of an extension.

References

[1] D. Parnas, On the criteria for decomposing systems into modules, Communications of the ACM 15 (1972) 1053–1058.

[2] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Lopes, C. Maeda, A. Mendhekar, Aspect oriented programming, in: Special Issues in Object-Oriented Programming, Max Muehlhaeuser (general editor) et al., 1996.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An overview of AspectJ, in: J. L. Knudsen (Ed.), Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001), number 2072 in Lecture Notes in Computer Science, Springer-Verlag, Budapest, Hungary, 2001, pp. 327–353.

[4] H. Masuhara, G. Kiczales, C. Dutchyn, A compilation and optimization model for aspect-oriented programs, in: G. Hedin (Ed.), Proceedings of Compiler Construction (CC2003), volume 2622 of *Lecture Notes in Computer Science*, Springer-Verlag, 2003, pp. 46–60.

[5] L. D. Benavides Navarro, R. Douence, M. Südholt, Debugging and testing middleware with aspect-based control-flow and causal patterns, in: Proceedings of the 9th ACM/IFIP/USENIX International Middleware Conference, volume 5346 of *Lecture Notes in Computer Science*, Springer-Verlag, Leuven, Belgium, 2008, pp. 183–202.

[6] M. Martin, B. Livshits, M. S. Lam, Finding application errors and security flaws using PQL: a program query language, in: [44], pp. 365–383. ACM SIGPLAN Notices, 40(11).

[7] P. Avgustinov, J. Tibble, O. de Moor, Making trace monitors feasible, in: [45], pp. 589–608. ACM SIGPLAN Notices, 42(10).

[8] P. Eugster, K. Jayaram, EventJava: An extension of java for event correlation, in: S. Drossopoulou (Ed.), Proceedings of the 23rd European Conference on Object-oriented Programming (ECOOP 2009), number 5653 in Lecture Notes in Computer Science, Springer-Verlag, Genova, Italy, 2009, pp. 570–594.

[9] R. Douence, P. Fradet, M. Südholt, Trace-based aspects, in: R. E. Filman, T. Elrad, S. Clarke, M. Akşit (Eds.), Aspect-Oriented Software Development, Addison-Wesley, Boston, 2005, pp. 201–217.

[10] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, Adding trace matching with free variables to AspectJ, in: [44], pp. 345–364. ACM SIGPLAN Notices, 40(11).

[11] S. F. Goldsmith, R. O'Callahan, A. Aiken, Relational queries over program traces, in: [44], pp. 385–402. ACM SIGPLAN Notices, 40(11).

[12] C. Herzeel, K. Gybels, P. Costanza, T. D'Hondt, Modularizing crosscuts in an e-commerce application in lisp using halo, in: Proceedings of the 2007 International Lisp Conference, ILC '07, ACM, New York, NY, USA, 2009, pp. 11:1–11:14.

[13] K. Ostermann, M. Mezini, C. Bockisch, Expressive pointcuts for increased modularity, in: A. P. Black (Ed.), Proceedings of the European Conference on Object-Oriented Programming (ECOOP), volume 3586 of *LNCS*, Springer-Verlag, 2005, pp. 214–240.

[14] P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Roşu, An overview of the MOP runtime verification framework, International Journal on Software Techniques for Technology Transfer (2011) 249–289. Http://dx.doi.org/10.1007/s10009-011-0198-6.

[15] D. Luckham, The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Professional Computing Series, Addison-Wesley, 1994.

[17] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, G. Murphy, Open implementation design guidelines, in: Proceedings of the 19th International Conference on Software Engineering (ICSE 97), ACM Press, Boston, Massachusetts, USA, 1997, pp. 481–490.

[18] S. Tobin-Hochstadt, M. Felleisen, The design and implementation of Typed Scheme, in: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008), ACM Press, San Francisco, CA, USA, 2008, pp. 395–406.

[19] P. Avgustinov, E. Bodden, E. Hajiyev, L. Hendren, O. Lhoták, O. de Moor, N. Ongkingco, D. Sereni, G. Sittampalam, J. Tibble, Aspects for trace monitoring, in: K. Havelund, M. Nez, G. Rou, B. Wolff (Eds.), Formal Approaches to Software Testing and Runtime Verification, volume 4262 of *Lecture Notes in Computer Science*, pp. 20–39.

[20] R. Douence, O. Motelet, M. Südholt, A formal definition of crosscuts, in: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, REFLECTION '01, Springer-Verlag, London, UK, 2001, pp. 170–186.

[21] R. J. Walker, K. Viggers, Implementing protocols via declarative event patterns, SIGSOFT Softw. Eng. Notes 29 (2004) 159–169.

[22] J. Postel, J. Reynolds, File transfer protocol (ftp). request for comments 959, 1985.

[23] F. Bancilhon, D. Maier, Y. Sagiv, J. D. Ullman, Magic sets and other strange ways to implement logic programs (extended abstract), in: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems, PODS '86, ACM, New York, NY, USA, 1986, pp. 1–15.

[24] C. L. Forgy, Rete: A fast algorithm for the many pattern/ many object pattern match problem, Artificial Intelligence 19 (1982) 17–37.

[25] K. Darlington, The essence of expert systems, Essence of computing series, Prentice Hall, 2000.

[26] L. D. B. Navarro, M. Südholt, W. Vanderperren, B. De Fraine, D. Suvée, Explicitly distributed aop using awed, in: Proceedings of the 5th ACM International Conference on Aspect-Oriented Software Development (AOSD 2006), ACM Press, Bonn, Germany, 2006, pp. 51–62.

[27] F. Chen, G. Roşu, Towards monitoring-oriented programming: A paradigm combining specification and implementation, in: Workshop on Runtime Verification (RV'03), volume 89(2) of *ENTCS*, pp. 108 – 127.

[28] F. Chen, G. Roşu, Mop: An efficient and generic runtime verification framework, in: [45], pp. 569–588. ACM SIGPLAN Notices, 42(10).

[29] P. O. Meredith, Efficient, Expressive, and Effective Runtime Verification, Ph.D. thesis, University of Illinois at Urbana-Champaign, 2012.

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

[31] P. Leger, É. Tanter, An open trace-based mechanism, in: J. Aldrich, R. Massa (Eds.), Proceedings of the 14th Brazilian Symposium on Programming Languages (SBLP 2010), Salvador - Bahia, Brazil, pp. 123–138.

[32] P. Leger, É. Tanter, R. Douence, Modular and flexible causality control on the web, Science of Computer Programming (2012). Available online.

[33] G. J. Sussman, G. L. S. Jr., Scheme: An interpreter for extended lambda calculus, in: MEMO 349, MIT AI LAB, Massachusetts Institute Of Technology Artificial Intelligence Laboratory, 1976, pp. 1–43.

[34] P. Leger, É. Tanter, A self-replication algorithm to flexibly match execution traces, in: Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012), ACM Press, Potsdam, Germany, 2012, pp. 27–32.

[35] J. V. Neumann, Theory of Self-Reproducing Automata, University of Illinois Press, Champaign, IL, USA, 1966.

[36] A. Holzer, L. Ziarek, K. Jayaram, P. Eugster, Putting events in context: aspects for event-based distributed programming, in: Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011), ACM Press, Porto de Galinhas, Brazil, 2011, pp. 241–252.

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

[38] É. Tanter, Expressive scoping of dynamically-deployed aspects, in: Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008), ACM Press, Brussels, Belgium, 2008, pp. 168–179.

[39] E. Bodden, Personal Communication, 2012. July 10, 2012.

[40] E. Bodden, P. Lam, L. Hendren, Clara: a framework for statically evaluating finite-state runtime monitors, in: 1st International Conference on Runtime Verification (RV), volume 6418 of *LNCS*, Springer, 2010, pp. 74–88.

[41] E. Bodden, F. Chen, G. Rosu, Dependent advice: a general approach to optimizing history-based aspects, in: Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development (AOSD 2009), ACM Press, Charlottesville, Virginia, USA, 2009, pp. 3–14.

[42] E. Bodden, Specifying and exploiting advice-execution ordering using dependency state machines, in: International Workshop on the Foundations of Aspect-Oriented Languages (FOAL), pp. 31 – 42.

[43] M. Parzonka, A Library-Based Approach to Efficient Parametric Runtime Monitoring of Java Programs, Master's thesis, Technische Universität Darmstadt, Darmstadt, 2013.

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

[45] OOPSLA 2007, Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007), ACM Press, Montreal, Canada, 2007. ACM SIGPLAN Notices, 42(10).

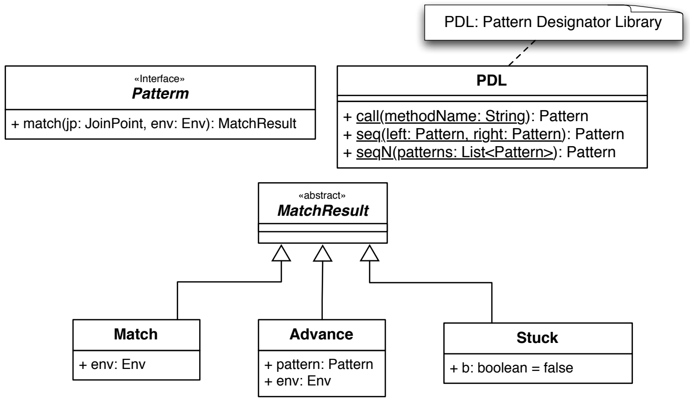

PDL: Pattern Designator Library
«Interface»
Patterm
+ match(jp: JoinPoint, env: Env): MatchResult
PDL
+ call(methodName: String): Pattern
+ seq(left: Pattern, right: Pattern): Pattern
+ seqN(patterns: List<Pattern>): Pattern
«abstract»
MatchResult
Match
+ env: Env
Advance
+ pattern: Pattern
+ env: Env
Stuck
+ b: boolean = false

Figure A.24: An object-oriented design for the ESA pattern language.

Appendix A. An Object-Oriented Design for the ESA Pattern Language

In this paper, we affirm the ESA pattern language can be implemented in other paradigms like object oriented. Figure A.24 shows a class diagram for our pattern language. A pattern is only an Interface, whose method match is executed for every new join point. This method can return every possible result of a pattern: an environment, a pair, or false. Finally, the PDL class contains a set of pattern designators. The following piece of code implements the pattern of the *toggle airplane mode* feature (Section 1) in Java with the proposed design:

```
Pattern s–up = PDL.call("up");
Pattern s–down = PDL.call("down");

ArrayList<Pattern> patterns = new ArrayList<Pattern>();
patterns.add(s–up);
patterns.add(s–down);
patterns.add(s–up);

Pattern toggleAirplaneMode = PDL.seqn(patterns);
```

Appendix B. Complete Description of ESA in Typed Racket

Using Typed Racket [18], this section presents the complete description of ESA. At the ESA website (http://pleiad.cl/esa), this description and a testsuite are available to download. For space reasons, we do not include the implementations of some helper functions in the following description.

Appendix B.1. Pattern Language

The ESA pattern language only requires functions that follow the signature of a pattern. In Typed Racket, the define-type construct allows developers to define types, and define-predicate is used to make a predicate for a (customized) type. The piece of code below defines the Pattern type and two predicates: one for a pair of Pattern and Env, and another for a Env. These predicates are useful to know what a pattern evaluation returns. Notice the definition of a pattern uses the Rec construct, which is necessary to define recursive types in Typed Racket.

```
;;Pattern type
(define-type Pattern (Rec Pat (JoinPoint Env -> (U Env False (Pair Pat Env)))))

;;Predicate for Pattern X Env
;;Note: Typed Racket does not support the definition of predicate for a particular
;;signature of a function, therefore, the 'Pattern' type must be replaced with 'Procedure'
(define-predicate PatternEnvPair? (Pair Procedure Env))
;;Predicate for Env
(define-predicate Env? Env)
```

Some pattern designators. The following piece of code shows the complete implementation of the call, seq, seqn, bind, where, and time-diff pattern designators. Only seqn and timediff subtly vary their implementations from Section 5.1:

```
;;To match the call to a function
(: call (Procedure -> Pattern))
(define (call fun)
  (λ (jp env)
    (if (eq? jp fun) env #f)))

;;To match a sequence of two patterns
(: seq (Pattern Pattern -> Pattern))
(define (seq left right)
              (λ (jp env)
                (let ([result (left jp env)])
                  (cond
                    [(PatternEnvPair? result) (cons (seq (get-pat result) right) (get-env result))]
                    [(Env? result) (cons right result)]
                    [else #f])))))

;;To match a sequence of 'n' patterns
(: seqn ((Listof Pattern) -> Pattern))
(define (seqn patterns)
  (foldl (λ: ([pattern : Pattern] [accPattern : Pattern]) (seq accPattern pattern))
         (first patterns) (rest patterns)))

;;To bind a value when a pattern matches
(: bind (Pattern Symbol (Env -> Env) -> Pattern))
(define (bind pattern id gather)
  (λ (jp env)
    (let ([result (pattern jp env)])
      (cond
        [(Env? result) (env-bind result id (gather env))]
        [else env]))))

;;To verify a condition (using bindings of the environment) when a pattern matches
(: where (Pattern (Env -> Boolean) -> Pattern))
(define (where pattern condition)
  (λ (jp env)
    (let ([result (pattern jp env)])
      (and (Env? result) (condition result)))))
```

```
;;To verify a period of time when a pattern matches
(: time-diff (Pattern Symbol Symbol Real -> Pattern))
(define (time-diff pattern t1 t0 time)
  (λ (jp env)
    (let ([result (pattern jp env)])
      (if (and (Env? result) (< (cast (env-lookup result t1) Real) (cast (env-lookup result t0) Real) time))
          env result))))
```

Appendix B.2. Adaptation of MatcherCells for ESA

We adapt the MatcherCells algorithm [34] to integrate into ESA. Seeds and matches are structures in this description, and a smatch is only type that is the union of Seed, Match, and a list that represents an intermediate stage between a seed and a match. The reaction of a smatch is carry out by the react function, which takes three parameters. The last parameter, ctx-inf, is optional, where keep-previous-bindings is its default value.

```
;;Seed structure
(define-struct: Seed
  ([pat : Pattern]
   [env : Env]))

;;Match structure
(define-struct: Match
  ([env : Env]
   [creator : SMatch]))

;;SMatch type
(define-type SMatch (Rec SM (U Seed Match (List Pattern Env SM))))

;;Reaction of a smatch
(: react (SMatch JoinPoint [#:ctx-inf (Env Pattern SMatch -> Env)] -> SMatch))
(define (react smatch jp #:ctx-inf [ctx-inf keep-previous-bindings])
  (let*
      ([pattern (get-pat smatch)]
       [env (get-env smatch)]
       [result (pattern jp env)]) ;;evaluating the pattern of the smatch
      (cond
        ;;According to 'result',this function returns a new smatch,seed,or the same smatch
        [(PatternEnvPair? result) (make-smatch (get-pat result)
                                               (ctx-inf (get-env result) (get-pat result) smatch)
                                               smatch)]
        [(Env? result)  (make-match result smatch]
        [else smatch]))) ;; the same smatch

;;Default context information for a smatch
(: keep-previous-bindings (Env Pattern SMatch -> Env))
(define (keep-previous-bindings env pat creator)
  env)
```

Appendix B.3. Matching Process

The matching process is defined by a composition of rules, where a rule is a function with the following signature in Typed Racket:

```
(define-type Rule ((Listof SMatch) JoinPoint -> (Listof SMatch)))
```

Some rules. Only the trace-life-time rule varies its definition from Section 5.2.2. In Typed Racket, the < function requires two Real parameters, therefore, it is necessary to *cast* the value stored in the environment.

```
;;This rule just applies the reaction to each smatch
(: apply-reaction Rule)
(define (apply-reaction smatches jp)
  (remove-duplicates (append smatches
                             (map (λ : ([smatch : SMatch])
                                    (react smatch jp)) smatches))))

;;This rule kill to every smatch that created a new one
(: kill-creators (Rule -> Rule))
(define (kill-creators rule)
  (λ (smatches jp)
    (let ([next-smatches (rule smatches jp)])
      (diff next-smatches (get-creators (get-sons next-matches smatches)))
      )))

;;This rule adds a seed when there is no smatches or only matches
(: add-seed (Pattern -> (Rule -> Rule)))
(define (add-seed pattern)
  (λ (rule)
    (λ (smatches jp)
      (let ([next-smatches (rule smatches jp)])
        (if (empty? (filter no-match? next-smatches))
            (cons (make-seed pattern) next-smatches)
            next-smatches))))))

;;This rule always keeps at least a seed
(: keep-seed (Pattern -> (Rule -> Rule)))
(define (keep-seed pattern)
  (λ (rule)
    (λ (smatches jp)
      (let ([next-smatches (rule smatches jp)])
        (if (= (count-seeds next-smatches) 0)
            (cons (make-seed pattern) next-smatches)
            next-smatches))))))

;;This rules kills a smatch when this lives more than a period of time
(: trace-life-time (Real -> (Rule -> Rule)))
(define (trace-life-time delta)
  (λ (rule)
    (λ (smatches jp)
      (let ([next-smatches (rule smatches jp)])
        (filter (λ (smatch)
                  (< (- (get-time) (cast (env-lookup (get-env smatch) 'time) Real))
                     delta)) smatches)))))
```

Some examples of matching semantics. Using the previous rules, it is possible to define some matching semantics, for example:

```
(: single-match-at-a-time (Pattern -> Rule))
(define (single-match-at-a-time pattern)
  ((add-seed pattern) single-match))
```

```
(: a-potential-match-can-always-start (Pattern -> Rule))
(define (a-potential-match-can-always-start pattern)
  ((keep-seed pattern) single-match))

(: timing-to-match (Real Pattern -> Rule))
(define (timing-to-match delta pattern)
  ((add-seed pattern) ((trace-life-time delta) single-match) ))
```

Appendix B.4. Advising Process

The return type of the advice and the advising process of a stateful aspect must be the exact same. To satisfy this constraint, we use *parameterized types* of Typed Racket. For example, the signature of an ESA advice is defined as follow:

```
;;Advice type
(define-type (Advice A) (JoinPoint Env -> A))
```

The following piece of code illustrates the use of parameterized types to define an advice that only prints a message (and returns Void):

```
(: print-call-to-foo (Advice Void))
(define (print-call-to-foo jp env)
  (printf "Calling to foo"))
```

The signature of a function that represents an advising process uses *polymorphic types* of Typed Racket to enforce the same return type for this function and the advice passed as parameter:

```
;;Advising Process type
(define-type AdvisingProcess (All (A) ((Advice A) (Listof SMatch) JoinPoint -> A)))
```

Some examples of advising semantics. The implementations of advising processes shown in Section 5.2.3 do not vary.

Appendix B.5. Stateful Aspect

As the piece of code below shows, a stateful aspect is a structure with five fields: pattern, advice, matching process, advising process, and a list of smatches. When a stateful aspect is created (make-aspect), the smatch list only contains a seed. The make-aspect function, which makes a stateful aspect, takes two optional parameters: mp and ap. These optional parameters represent the matching process and advising process respectively.

```
;;StatefulAspect structure. This uses a parameterized type for its advice
(define-struct: (A) StatefulAspect
  ([pattern : Pattern]
   [advice : (Advice A)]
   [matching : Rule]
   [advising : AdvisingProcess]
   [smatches : (Listof SMatch)])
   )

;;This function creates a stateful aspect
(: make-aspect (All (A) (Pattern (Advice A) [#:mp Rule] [#:ap AdvisingProcess] -> (StatefulAspect A))))
(define (make-aspect pat adv #:mp [mp (single-match-at-a-time pat)] #:ap [ap single-advice-execution])
  (StatefulAspect pat adv mp ap (list (make-seed pat))))
```