



Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico


An expressive stateful aspect language


 Paul Leger^{a,*}, Éric Tanter^b, Hiroaki Fukuda^c
^a Universidad Católica del Norte, Escuela de Ciencias Empresariales, Chile

^b PLEIAD Lab, Computer Science Department, University of Chile, Chile

^c Shibaura Institute of Technology, Japan

ARTICLE INFO

Article history:

Received 25 September 2013

Received in revised form 18 January 2015

Accepted 2 February 2015

Available online 7 February 2015

Keywords:

Aspect-oriented programming

Stateful aspects

ESA

Typed racket

JavaScript

ABSTRACT

Stateful aspects can react to the trace of a program execution; they can 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 are 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.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

Modularity favors system evolution and maintenance by allowing separate concerns to be localized [1]. *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 [6,8,10–15], specifically tailored to address particular domains. Because of these domains, these languages do not share the same semantics [15]. Some of them like Tracematches [11] support multiple matches, even simultaneously, of a join point trace pattern (just *pattern* from now). Each language provides its own domain-specific language to define patterns of interest. In addition, each language has its own *matching semantics*

* Corresponding author.

E-mail address: pleger@ucn.cl (P. Leger).

to define how a pattern is matched, and *advising semantics* to define how an advice is executed. To date, stateful aspect language design has focused mostly on performance, leaving aside the exploration of more expressiveness in the following:

Pattern language The lack of expressiveness in pattern languages limits developers from *a*) reusing and composing patterns and *b*) to accurately define program execution trace patterns that must be matched.

Semantics In stateful aspect languages, limited expressiveness generates two problems in terms of semantics: *fixed* and *common* semantics for all stateful aspects. By *fixed* we mean developers cannot customize the matching and advising semantics of the aspect language. Additionally, if developers are able to customize the language semantics, all aspects must share these customizations because semantics is *common* for all aspects.

1.1. Contributions

The problems related to the lack of expressiveness of current stateful aspect languages serve as motivation for this work, which proposes a precise description of an expressive stateful aspect language.¹ Concretely, the contributions of this paper are:

- **Four problems identified.** Through this paper we expose four problems associated with stateful aspect languages. Two problems belong to the lack of reuse, composition, and expressiveness of current pattern languages. The two remaining problems are related to the common and fixed semantics of these aspects.
- **An Expressive Stateful Aspect language (ESA) description.** We describe a stateful aspect language, named ESA, which addresses the previous problems. Using ESA, developers can:
 - use first-class patterns, meaning that a pattern is a value of the language (e.g., function, object) that can be composed to other language values to create more complex patterns. First-class patterns offer the benefits of the reuse and composition of patterns. 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.
 - customize the matching and advising semantics of every stateful aspect. With this, every stateful aspect can have different semantics and developers can customize any aspect. To achieve this goal, we follow *open implementation* design guidelines [17], allowing developers to customize strategies of a program implementation, while hiding details of its implementation.
- **A concrete and practical implementation of ESA.** We use the proposed description to implement a concrete and practical version of ESA for JavaScript, named ESA-JS. This version supports modern browsers such as Firefox and Chrome without the need of an add-on. In addition, we implement ESA-AS3, a proof of concept of ESA for ActionScript.
- **A reference frame of comparison.** To contrast our proposal with existing stateful languages, we develop a reference frame that compares the existing proposals in terms of expressiveness. As a result, we clarify and discuss some differences between these proposals.

Paper roadmap Section 2 introduces and goes into detail on stateful aspect languages. Section 3 discusses the state of the art of these languages. Evaluations and limitations of existing proposals are shown through various examples in Section 4. Section 5 presents the description of an expressive stateful aspect language; 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 some existing stateful aspect languages. Section 8 discusses design considerations of our proposal, and Section 9 concludes.

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.

2. Architecture of a stateful aspect language

Stateful aspects [9] support the modular definition of crosscutting concerns for which matching join point traces, as opposed to single join points, is necessary. In Aspect-Oriented Programming (AOP) [2], join points are the events that can be gathered by an aspect, and the variety of their types (e.g., method calls, object creations) depends on the join point model supported by the aspect language [4]. As Fig. 1 shows, a stateful aspect is composed of a (join point trace) pattern and an advice. The advice is executed *before*, *around*, or *after* the last join point that must match with the pattern. Depending on the deployment strategy used, a stateful aspect may react from a part to all history of a program execution.²

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

² Although the concrete implementation of our proposal uses different deployment strategies (Section 5) [19], the discussion of these strategies is beyond this paper.

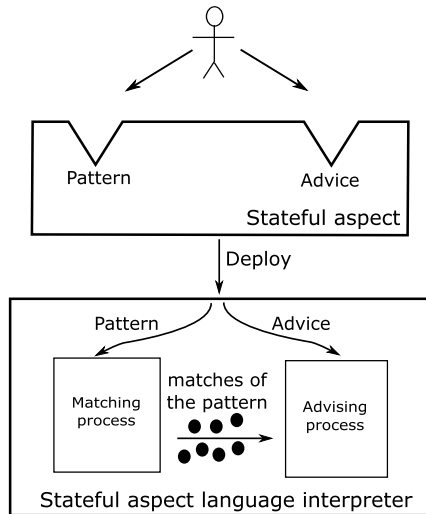


Fig. 1. The anatomy of a stateful aspect and main processes of its language.

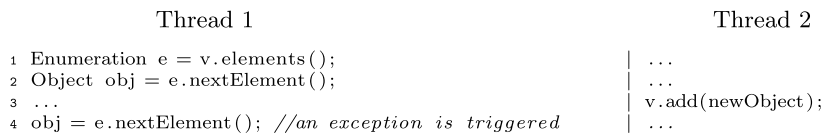


Fig. 2. A program execution that triggers an exception if Enumeration supports fail-fast.

Over the next three sections, we describe and illustrate the core elements of these languages: pattern language, matching process, and advising process. To illustrate, we use a runtime verification example implemented in JavaMop [15], a generic stateful aspect framework for Java. Patterns in JavaMop can be expressed in different domain-specific languages: regular expressions, context-free grammars, linear temporal logic, etc. An aspect in this framework is composed of a set of event declarations, a pattern, and a piece of code. An event declaration represents a set of join points. The pattern is defined over these events, and the piece of code represents the advice.

```

failFast(Vector v, Enumeration e) {
  //events
  event create after(Vector v) returning(Enumeration e):
    call(Enumeration.elements()) && target(v) {}
  event addVector after(Vector v): call(* Vector.add*(..)) && target(v) {}
  event nextEl before(Enumeration e): call(* Enumeration.nextElement()) && target(e) {}

  //pattern, 'ere' means that the pattern is defined by a regular expression
  ere: create nextEl* addVector+ nextEl

  //advice
  @match {
    throw new ConcurrentModificationException();
  }
}

```

The JavaMop stateful aspect above provides the *fail-fast* feature for the Enumeration interface, which triggers an exception if the underlying enumeration is modified while an iteration is in progress. In JavaMop, the aspect declaration starts with its name, e.g., failFast. Next, events are declared. To define the pattern, a developer first selects a pattern language with a keyboard (e.g., ere), then a pattern is defined. Fig. 2 illustrates this situation through the execution of two threads, where e is an enumeration over a vector v. In Line 3 of thread 2, v adds a new object, triggering an exception before the next call of nextElement in thread 1.

2.1. Pattern language

A stateful aspect language uses a domain-specific language to define patterns. Some pattern languages allow developers to specify bindings that are gathered while a pattern is being matched. For example, patterns of Tracematches [11] are

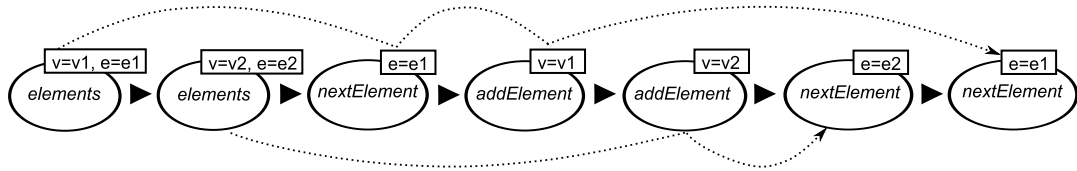


Fig. 3. Two matches of the pattern used for *fail-fast*. In the figure, the match of up side gathers the bindings *v1* and *e1*, and the match of down side gathers the bindings *v2* and *e2*.

defined as regular expressions and can gather bindings. Instead, PTQL [12] uses a SQL-like language to express patterns, where bindings are specified in the SELECT statement. Gathered bindings can be used to filter unwanted matches of the pattern and give context information to the execution of advice. 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.

As the *failFast* aspect code shows, in JavaMop, bindings gathered are specified in the header. In this piece of code, two bindings: a vector *v* and its enumeration *e*. When an event of the pattern is matched, a binding can be gathered; for example, *v* is gathered when *create* 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.

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 of the impact of its chosen semantics 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.

Fig. 1 shows that when a stateful aspect is deployed, the aspect language uses its matching process to match the aspect pattern. Depending on the semantics of this process, a pattern may match multiple times. We use the implementation of the *fail-fast* feature to illustrate this process.

The matching process of JavaMop, like most existing stateful aspect languages, supports multiple matches. Particularly, JavaMop can support multiple matches of a pattern, if the *environment* of bindings gathered by potential matches of the pattern differs between them. For example, Fig. 3 exemplifies this semantics because the pattern matches twice (represented by dotted continuation lines) due to the fact that the environments of both matches, v=v1 , e=e1 and v=v2 , e=e2, differ.

2.3. Advising process

The implementation of the advising process is also inside the 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.

As Fig. 1 shows, the advising process is triggered when it receives one or more matches of a pattern. 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 using a restrict variant of *fail-fast*: if a vector removes an element between *hasMoreElements* and *nextElement* invocations, an exception is thrown. In this variant, we have added two stateful aspect instance variables, *vec* and *enum*, that are bound to the vector *v* and enumeration *e* respectively when the *create* event is matched. In JavaMop, instance variables allow developers to link bindings from the matching to the advising process.

```
restrictedFailFast(Vector v, Enumeration e) {
    Vector vec;
    Enumeration enum;
```

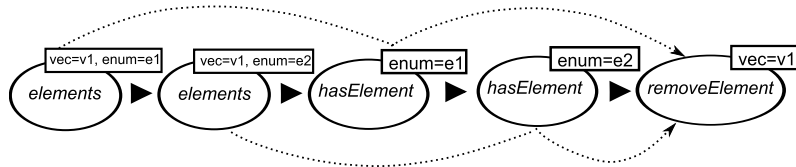


Fig. 4. Two simultaneous matches of the pattern of the restricted *fail-fast*. In the figure, the match of up side gathers the bindings *v1* and *e1*, and the match of down side gathers the bindings *v1* and *e2*.

```

event create after(Vector v) returning(Enumeration e):
  call(Enumeration Vector.elements()) && target(v) {vec = v; enum = e;}
event nextEl before(Enumeration e) : call(* Enumeration.nextElement()) && target(e) {}
event hasMore after(Enumeration e) : call(* Enumeration.hasMoreElements()) && target(e) {}
event removeVector before(Vector v) : call(* Vector.removeElement(..)) && target(v) {}

```

ere: create hasMore removeElement

```

@match {
  throw new ConcurrentModificationException(this.enum+" "+this.vec+ " has removed an element");
} }

```

Fig. 4 shows that the `removeElement` invocation triggers two simultaneous matches of the pattern of `restrictedFailFast`. Due to these matches, the advice is executed twice, each execution uses a different environment of bindings, `vec=v1, enum=e1`, `vec=v1, enum=e2`. In this example, the advising process must compose the two executions of the same piece of code given as 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.

Tracematches The Tracematches proposal, implemented as an AspectJ [3] extension, is an 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 parentheses, 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 follows the same guidelines of Tracematches.

Alpha Alpha [14] 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 to a match of a pattern. These solutions are passed to the advising process, which only supports before and after advice kinds. Advices are executed in a consecutive manner for each solution, which contains a set of gathered bindings.

Halo Herzeel et al. [13] 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 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 matched join point.

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 is 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. Similar to Tracematches, the matching process uses a finite state machine to carry out the matching 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 uses 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) [12] 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 advising process in PTQL.

JavaMop JavaMop [15,27,28] 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.

EventReactor Event composition model [30] is a computational model that is based on the definitions of software concerns as *event modules*. An event module groups a set of events, *i.e.*, join points, that are related to a concern and can react to this set of events. The software construction is achieved through the composition of these event modules, where an event module can be composed using input (events received) and output (events published) interfaces. EventReactor [10, 31,32] provides a set of language constructs to implement the event composition model. The proposal can be implemented for multi languages like Java and C. Similarly to JavaMop, patterns in EventReactor are expressed in a user-defined and potentially Turing complete language, but these patterns are not first-class values, reusable, and composable. The matching process supports multiple matches, which can be specialized per thread. In EventReactor, as an event can be “before calling to a function” and the advising process can react to any event described by an event module, the advice kind of EventReactor supports *before* and *after*.

OTM This paper is not our first attempt at implementing an expressive stateful aspect. In [33,34], 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 [35], OTM is extended to control causal relations among Ajax messages in JavaScript applications.

Fig. 5 sums up the reviews of stateful aspect languages analyzed in this paper. Most stateful aspect languages provide different pattern languages, where the language expressiveness varies. For instance, Alpha uses Prolog, PTQL uses SQL, and Tracematch 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 [15] allows developers to choose one of three fixed specifications for the matching process for every stateful aspect. Finally, most advising processes of stateful aspect languages only support before and after advice.

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	<i>Own algorithm</i>	Before and replace advice
PTQL	SQL	Particle, a query algorithm	<i>None</i>
JavaMop	From regular expressions to Turing complete	Depending on the pattern language used + a fixed set of strategies	Before, around, and after advice
EventReactor	User-defined and potentially Turing complete language	Depending on the pattern language used + a fixed set of strategies	Before and after advice
OTM	Base language	<i>Own algorithm</i>	Before, around, and after advice

Fig. 5. Summary of stateful aspect languages discussed in this section.

4. Shortcomings of existing stateful aspects

This section illustrates current shortcomings of existing stateful aspect languages. For this illustration we focus on the three components described in this paper: pattern languages, matching process, and advising process. Each component is evaluated with regard to four problems mentioned in the introduction of this paper. To clearly identify every problem, we assign a identifier:

- Pattern Languages:
 - *p1* is the *inability to reuse and compose* patterns.
 - *p2* is the *limited expressiveness* to define patterns.
- Matching and Advising Processes:
 - *s1* is the *fixed* semantics of stateful aspects.
 - *s2* is the *common* semantics for all stateful aspects. An aspect language with customizable semantics (*i.e.*, *s1* solved) does not address the problem *s2* if these customizations are at language level because all aspects would share these semantic variations.

Considering the same components and problems mentioned above, this section also evaluates and compares stateful aspect languages described in Section 3. This section concludes describing requirements to achieve an expressive stateful aspect language.

4.1. Illustrating shortcomings of stateful aspect languages

To illustrate the four problems mentioned above, we use variants of two different applications of a stateful aspect approach:

Toggle airplane mode In touch devices, a *toggle airplane mode* feature allows users to enable (or disable, if it was enabled) the airplane mode. Suppose a user can use this feature with the sequence *up*, *down*, and *up* in the touch device. The following aspect implements this feature:

```
toggleAirplaneMode() {
  event s-up after(): call (* Screen.up()) {}
  event s-down after(): call (* Screen.down()) {}

  ere: s-up s-down s-up

  @match {
    Device.toggleAirplaneMode();
  } }

```

Discount policy A Web application of an online store is used to order computers. 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 with 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 follows:

```
discountPolicy(Computer c, User u) {
  Computer comp;
  User user;

  event add after(User u, Computer c): call(* Cart.add(User, Computer))
    && args(u, c) { user = u; comp = c; }
  event checkout before(User u): call(* Form.checkout(User)) && args(u) {}

  ere: add checkout

  @match {
    Cart cart = this.user.getCart();
    cart.applyDiscount(this.comp, DiscountPolicy.getDiscount(this.comp));
  } }

```

4.1.1. Pattern languages

This section first illustrates the problem *p1* through of an extension of the *toggle airplane mode* feature. This section then uses an additional restriction to the discount policy of the online store example to illustrate the problem *p2*.

Problem p1 Consider an addition of the *toggle airplane mode* feature, which enables/disables the airplane mode during a period of time (e.g., 10 hours). This new feature is executed when a user moves the sequence of the toggle airplane mode with a *left* slide as prefix. To implement this feature, named *toggle airplane mode with time*, an adequate solution would be to reuse the pattern of the previous feature. However, most stateful aspect languages like JavaMop do not allow developers to reuse and compose patterns, i.e., problem *p1*. It is therefore necessary to write all of the pattern again:

```
toggleAirplaneModeWithTime () {
  //events as in toggleAirplaneMode

  event s-left after(): call (* Screen.left()) {}

  ere: s-left s-up s-down s-up //all pattern was defined again

  @match{
    Device.toggleAirplaneMode(10*60*60); //10 hours
  } }

```

The piece of code above is not easily maintainable, because if the pattern of `toggleAirplaneMode` changes, it is necessary to rewrite all the of pattern of `toggleAirplaneModeWithTime`.

Problem p2 Apart from the lack of reusing and composing a pattern, the limited expressiveness does not allow developers to directly define patterns that gather a variable list of bindings: problem *p2* is an example of this. Consider a variation of the current discount policy, named *limited discount policy*. This variation only applies the discounts to three computers that have the best associated discounts. This small variation cannot be implemented as an update of the solution presented

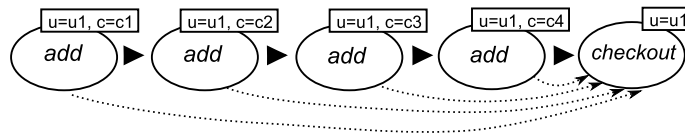


Fig. 6. Four simultaneous matches of the pattern used to implement the *discount policy* feature, where a user u_1 buys four products: c_1 , c_2 , c_3 , c_4 .

previously. This is so because the current solution does not use only one match that contains a list with all computers (see Fig. 6), which is necessary for choosing the best discounts. A stateful aspect that matches a list of computers would be adequate, however, most of these languages do not allow developers to define a pattern that gathers a variable-size list of bindings. A reader might wonder if the following piece of code can work:

```
limitedDiscountPolicy(User u, List l) {
  //discountPolicy code.
}
```

This piece of code above gathers a list instead of computers. This solution does not work because the list of computers is not available in the piece of code of the base language. Although this list would be available, this list would be useless because we cannot know which computers were added to the cart before the period of time of an associated discount finishes. Therefore, we need to code around the current advice to implement the update of this feature:

```
limitedDiscountPolicy(User u, Computer c) {
  int computerCounter = 0;
  ArrayList computers = new ArrayList();

  //instance variables, events, and pattern as in limitedDiscount

  @match {
    if (this.computerCounter++ < this.u.getCart().size())
      this.computers.add(this.comp); //adding to the list of computers
    else {
      ArrayList computersWithBestDiscounts = getBestDiscounts(this.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 bindings, `computers` and `computerCounter`. The behavior of stateful advices depends on bindings that are outside of it. Therefore, developers keep in mind the state of outer bindings to know the real behavior of a stateful advice. Although, in JavaMop, we use a Turing complete pattern language like *string rewriting system* [29], this advice has to be modified in contortive ways as well. This is because available pattern languages in JavaMop are sufficiently expressive to specify a join point trace, but not expressive enough to specify what and how bindings are gathered, e.g., a variable list of bindings.

4.1.2. Matching and advising processes

In this section, we first illustrate the problem s_1 through an example related to the advising process. We then use an example related to the matching process to exemplify s_2 .

Problem s_1 In an advising process, 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, i.e., problem s_1 . A customizable advising process semantics is necessary in scenarios like adding a new variation to the discount policy. 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 total number of pieces of every computer is only known when a client goes to checkout. The implementation of this new discount policy extension requires the stateful aspect to execute the advice only once, with the match whose bindings gathered contain the computer with more customized pieces:

```
personalizedDiscountPolicy(User u, Computer c) {
  int maxCustomizedPieceNumber = 0;
  int computerCounter = 0;
  int computerSelected;
```

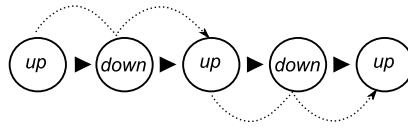


Fig. 7. Two matches of *airPlaneMode* due to semantics of multiple matches.

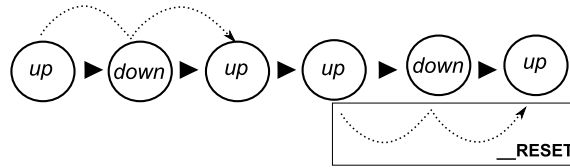


Fig. 8. A correct implementation of *toggle airplane mode* with JavaMop needs a special keyboard.

```
//events and pattern as in discountPolicy

@match{
  Cart cart = this.user.getCart();

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

  if (++this.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.

Problem s2 Although the pattern of *toggle airplane mode* implementation looks correct, this pattern does not work because most stateful aspect languages perform multiple matches of a pattern. As a result, once the pattern is matched, each subsequent *up* and *down* toggles the airplane mode. Fig. 7 shows the previous point where observe that before the first match of the pattern finishes with the call to the *up* function, new potential matches of the pattern start. Although semantics of multiple matches is useful in many cases (e.g., *fail-fast* and *discount policy*), this semantics is not adequate in all cases: problem *s2*. Particularly, JavaMop suppresses multiple matches of a pattern if this does not gather different bindings; therefore, the *toggleAirplaneMode* pattern only matches once in the whole program execution. To use a semantics like *single match at a time*, a programmer has to tweak the aspect definition. In the *toggleAirplaneMode* implementation in JavaMop, an ad-hoc keyboard, `__RESET`, has to be used inside of the advice. This keyboard reinitializes the stateful aspect pattern to match again (see Fig. 8):

```
rightToggleAirplaneMode() {
  //events and pattern as in toggleAirplaneMode

  @match {
    Device.toggleAirplaneMode();
    __RESET; //ad-hoc solution
  } }
```

In Section 6, we will revisit all these examples to present adequate solutions using an implementation of our proposal.

4.2. Evaluation

Fig. 9 evaluates the stateful aspect languages described in Section 3. The figure evaluates the pattern language, matching process, and advising process. Each evaluation measures which of the four problems 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 (problems *p1* and *p2*). Halo, JavaMop, and EventReactor address

Stateful Aspect Languages	Pattern Language	Matching Process	Advising Process
Tracematches	○	○	○
Tracecuts	○	○	○
Alpha	●	○	○
Halo	◐	○	○
EventJava	○	○	○
AWED	○	○	○
PQL	●	○	○
PTQL	○	○	○
JavaMop	◐	◐	○
EventReactor	◐	◐	○
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 per stateful aspect	

Fig. 9. Evaluations of some stateful aspect proposals regarding their pattern language, matching and advising processes.

one of the previous two problems. Halo uses composable and reusable patterns, but its pattern language is limited by a fixed and small set of patterns. JavaMop and EventReactor allow developers to create domain-specific languages, which may be Turing complete, however, their patterns are not first-class values and reusable.

Regarding the matching process, JavaMop, EventReactor, and OTM support semantic variations for their aspects (problem *s1*). Finally, we observe there is no support to customize the stateful aspect language semantics. At the bottom of the list, we see that ESA, the proposal of this paper, completely addresses the four problems.

4.3. Requirements for an expressive stateful aspect language

We have presented many examples that illustrate different kinds of limitations of existing proposals. In our opinion, to overcome the aforementioned limitations, it is crucial to consider an expressive pattern language and customizable semantics per aspect (see Fig. 10):

Expressive pattern language A Turing complete language allows developers to express advanced patterns, *e.g.*, patterns that gather a variable-sized list of bindings. In addition, first-class patterns are useful for cleanly reusing and composing patterns.

Customizable semantics Unlike Fig. 1, Fig. 10 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 (problem *s1*). In addition, the figure shows that developers can customize the aspect semantics (problem *s2*), and not choose a particular semantics from a pre-defined set of semantics available in a stateful aspect language.

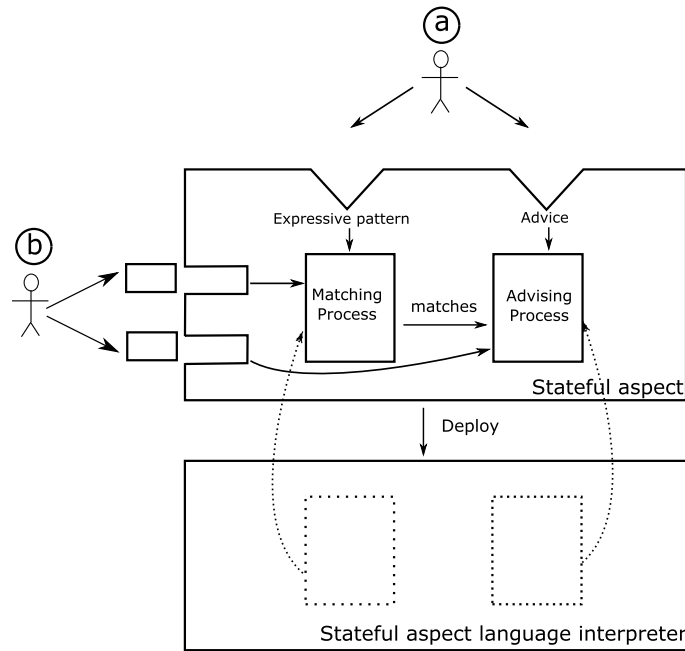


Fig. 10. Requirements for Expressive stateful aspect language: a) an expressive pattern language and b) customizable semantics per stateful aspect.

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 A shows these details).

ESA overview To satisfy the requirements of Section 4.3, this description allows developers to implement a stateful aspect language, whose pattern language is Turing complete and the semantics of each stateful aspect is customizable. 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, we use *open implementations* [17] – useful programming guidelines that suggest intuitive default semantics for a program. For customizations, 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.

In the following two sections, we first introduce the ESA pattern language, and then explain how ESA allows developers to customize stateful aspect semantics. Although stateful aspect languages commonly use 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 as they are sufficient for describing 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 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

Where a pointcut is a function $Pc: JoinPoint \rightarrow Boolean$, a pattern is a function with the type $Pattern: JoinPoint \rightarrow Boolean \cup Pattern$. 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 the next join points that should be matched by the pattern.

Unlike most existing approaches where a pattern is a specification (e.g., regular expression), the ESA pattern language approach uses a function that can take actions (e.g., match). This approach is inspired by *continuation-passing style* [36]. As an example of our approach, the implementation of a pattern to match a call to the up function in a touch device is:

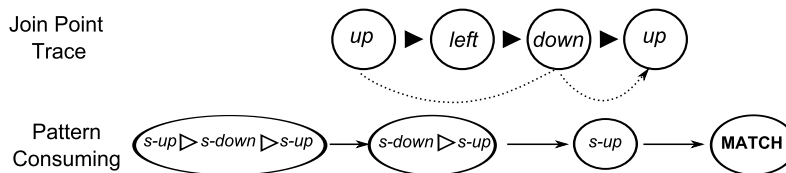


Fig. 11. How a pattern is consumed during its matching.

```
(: 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.

```
1 (: seq (Pattern Pattern → Pattern))
2 (define (seq left right)
3   (λ (jp)
4     (let ([result (left jp)])
5       (cond
6         [(Pattern? result) (seq result right)]
7         [(eq? result #t) right]
8         [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 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. Fig. 11 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 $s\text{-down} \triangleright s\text{-up}$. With the down call, the pattern changes to only $s\text{-up}$. Finally, once the user touches 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 timed-toggle-airplane-mode, 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 fold function, also known as reduce and accumulate, processes a list of patterns in the left order to return a new pattern.

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

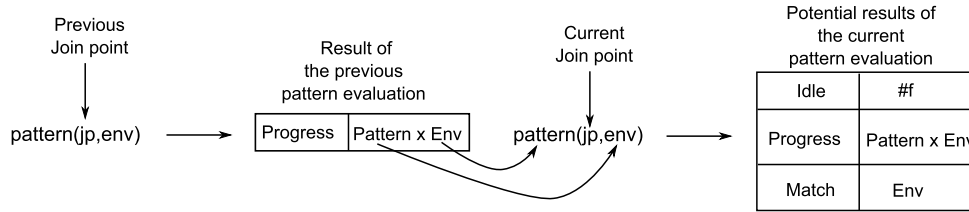


Fig. 12. Evaluation of a pattern and its potential kinds of results.

```
(: 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 be able to gather bindings. The standard and complete vision of the pointcut-advice model establishes a pointcut as a function $Pc: JoinPoint \rightarrow Env \cup False$. 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:

$$Pattern : JoinPoint \times Env \rightarrow Env \cup False \cup Pattern \times Env$$

A pattern now also takes an environment as a parameter. As Fig. 12 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 another pattern and an environment if the evaluated 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* is 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)]
```

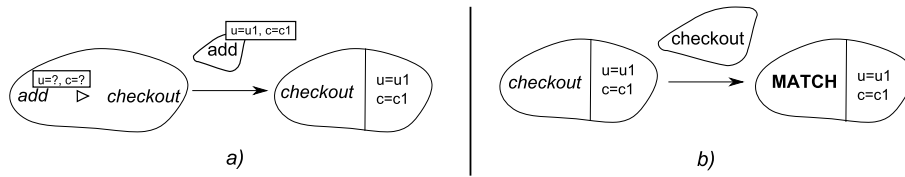


Fig. 13. a) The left cell creates a cell that expects to match the next join point and gathers bindings. b) When a cell matches the last join point specified by a pattern, the cell creates a *match cell*.

```

(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.

5.2. Semantics

In most stateful aspect languages, aspects of a language share the same exact semantics [6,8,11–14]. 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* [37], a flexible algorithm to match program execution traces. In this section, we first explain the previous algorithm, and we then use to open it the semantics of a stateful aspect.

5.2.1. MatcherCells

To flexibly match join point traces, *MatcherCells* use *self-replicating algorithms* [38], algorithms that emulate the reactions of a set of biological *cells* to a trace of *reagents*. To persist in the environment, the reaction of a cell to a reagent can be the creation of an identical copy of itself with a small variation, 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 of first cells (*a.k.a.* seeds) and the second one is the set of rules that governs the evolution of these cells.

MatcherCells terminology As *MatcherCells* is inspired by biology, this section uses biological terms to explain it. For example, this section uses terms as *reagents* to refer to join points, *cells* for different states of a potential match of a pattern, and *match cells* for matches of the pattern. After this section, we will keep AOP terminology.

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, Fig. 13a 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. Fig. 13b 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, we illustrate the matching of a pattern in *MatcherCells*. Fig. 14 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, each subsequent *up* join point will trigger a new match of the pattern (an unwanted semantics for *toggle airplane mode*).

MatcherCells allow developers to add rules to control the evolution of a set of cells. Fig. 15 shows four different evolutions, where each one has a different set of rules that are used to customize the matching semantics of a pattern. Although evolutions have different rules, we appreciate that all evolutions have the *apply reaction* rule, which only applies the reaction of each cell to a join point. Fig. 15a shows that the *kill creators* rule kills the cells that create a new cell. Adding this rule, a pattern cannot match multiple times anymore. Fig. 15b 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. Fig. 15c shows that the *keep seed* rule always keeps a seed to permit the starting of a new potential match of a pattern at any moment. Finally, Fig. 15d shows that the *life-time for a trace* rule kills all cells whose period of time of a join point trace has exceeded a determined period. This rule allows developers to only match traces of join points that occur at a period of time.

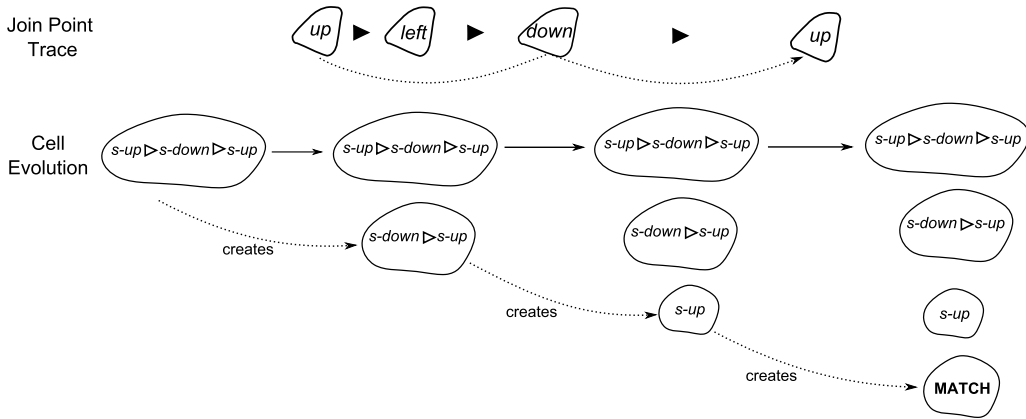


Fig. 14. Evolution of a seed during the matching of a pattern.

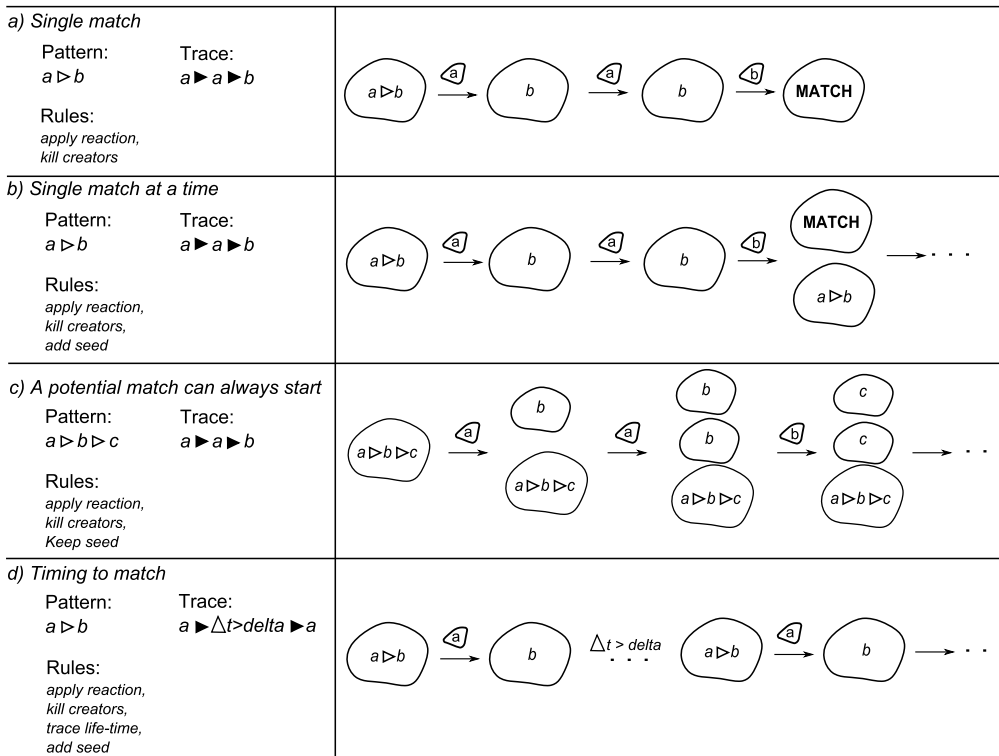


Fig. 15. Different matching semantics to match a pattern (figure adapted from [37]).

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 (Fig. 15a) 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.

5.2.2. Matching process

Unlike existing proposals, every ESA stateful aspect has its own matching process that can be customized by developers (see Fig. 10). 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.

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

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 inside of the smatch, this smatch really becomes a match. The last and optional parameter ([.]) of react is a function that allows developers to add information to a smatch in its creation. This is explained in more detail at the end of this section.

rule : List < SMatch > × JoinPoint → 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:

```
(: 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-seed³ 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 Fig. 15 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 all smatches to contain specific context information. For example, trace-life-time needs all smatches to contain the time in their environments when a potential match starts:

³ Notice that add-seed is in fact a higher-order rule designator, parameterized by the original pattern or a new one.

```
(: 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))
                    delta))
                matches))))))
```

To add context information to matches, 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 matches. In other words, these attributes crosscut through the matching of a trace [39]. The feature of customized information of matches allows developers to modularize the treatment of these intrinsic attributes.

As a conclusion, we can say the definition of the matching process of a stateful aspect language is crucial because a small change in the semantics of this process may strongly affect the (number of) matches of a pattern. Thereby, a wide range of semantics is possible. Through simple compositions of rules, the MatcherCells algorithm allows developers to explore this wide range of semantics. In ESA, MatcherCells allows developers to easily customize the matching process semantics per stateful aspect.

5.2.3. Advising process

Like the ESA matching process, the advising process is open to customization. When one or more matches 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:

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

where

$$\text{Advice} : \text{JoinPoint} \times \text{Env} \rightarrow \text{AdviceReturn}$$

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 matches. The type of 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 matches). 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. Conversely, 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 advantage of 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.

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

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 the 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 generate 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, understanding these features is complex for developers because they need to take into consideration 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 follows:

```
(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:

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

```
(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. The modular definition of these components allows developers to reuse and compose patterns (problem *p1*) with a Turing complete language (problem *p2*). In Appendix B, we use ESA pattern language to implement a string rewriting system, a Turing complete language used in JavaMop [29], in order to show the completeness of our pattern language. The semantics of stateful aspects can be customized (problem *s1*) by each stateful aspect (problem *s2*). The design of these components in ESA are modular. For example, 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 four problems of existing stateful aspect languages illustrated in Section 4.1. Apart from ESA-JS, we developed a version of ESA for ActionScript, named ESA-AS3. Although the former implementation is not described here because of space reasons, ESA-AS3 is available on the ESA Web site.

A brief overview of *AspectScript* ESA-JS is currently implemented as a seamless extension of *AspectScript* [40], 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 [19].

```

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

```

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 implementation is a correct implementation of the feature because of ESA default semantics: *single match at a time* semantics for the matching process and *single advice execution* semantics for the advising process.

6.1. Pattern language

Section 4.1.1 illustrates consequences related to the lack of reusing and composing of patterns (problem *p1*) and the limited expressiveness to define them (problem *p2*). As ESA can address previous problems, we can reuse patterns to define advanced ones:

```

var toggleAirplaneModeWithTime = {
  pattern: seq(s-left, toggleAirplaneMode.pattern), //reuse of the toggleAirplaneMode pattern
  advice: function(jp, env)
    Device.toggleAirplaneMode(10*60*60); //10 hours
  },
  kind: ESA.AFTER
};

```

The solution presented for *limited discount policy* in Section 4.1.1 has the drawback that its advice is *stateful*. This is because existing stateful aspect languages are insufficiently expressive at allowing developers to define patterns that gather a variable-size list of bindings (problem *p2*). Using ESA-JS, we can define patterns that gather this kind of list, implying a stateful advice is not needed 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 checks 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 and advising processes

In most stateful aspect languages, aspects share a fixed semantics (problem *s1*) and/or semantics can only be customized at a language level and not per aspect (problem *s2*).

In existing proposals, stateful aspects have fixed semantics for their advising process. As Section 4.1.2 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 because the computer with the greater number of customized pieces is only known when the user checks out (i.e., inside of the advising process). The *personalized discount policy* requires that the aspect only executes its advice with the environment that contains the computer with more customized pieces. When using ESA-JS, it is not necessary to intrusively modify the advice:

```

var personalizedDP = discountPolicy; //reusing the discountPolicy aspect

personalizedDP.advising = function(advice,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 see in the above pieces, 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 is executed with this modified environment.

ESA developers can create a particular aspect semantics for a given concern, meaning that each stateful aspect can have a different semantics. For example, the following ESA-JS aspect implements the *toggle airplane mode* feature with a *single match at a time* semantics:

```

var toggleAirplaneMode = {
  pattern: seqn([s-up,s-down,s-up]),
  advice: function(jp,env)
    Device.toggleAirplaneMode();
  },
  kind: ESA.AFTER,
  matching: singleMatchAtATime //customized matching semantics for toggleAirplane
};

//where is singleMatchAtATime
var singleMatchAtATime = addSeed(pattern)killCreators(applyReaction));

```

```

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
}

```

Fig. 16. (Left) AspectScript aspect used for the experiment. (Right) ESA stateful aspect used for the experiment.

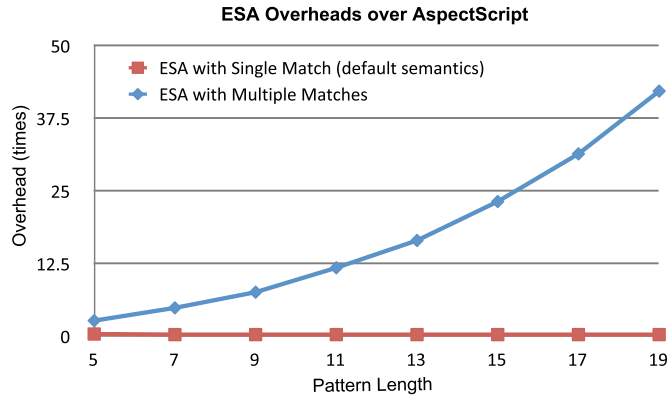


Fig. 17. Overhead of ESA with two different matching semantics over AspectScript.

6.3. Performance

We ran tests to evaluate performance of ESA-JS and results were compared to that of JavaMop. For this experiment, we used 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* compiler [41] (version 1.3.0) for JavaMop.

```

//PATTERN_LENGTH varies from 5 to 19
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 (Fig. 16). The AspectScript aspect keeps a counter of matches and the ESA-JS aspect with two different semantics: *single match* and *multiple matches* semantics. Finally, we execute the same experiment with an aspect of AspectJ and a stateful aspect of JavaMop. 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 19.

ESA-JS and JavaMop are aspect language extensions of AspectScript and AspectJ respectively. Figs. 17 and 18 show the increment of the overhead of ESA-JS and JavaMop over their aspect languages. In these figures, the x axis represents lengths of PATTERN_LENGTH and the y axis represents how many times slower the runtime of a stateful aspect extension is over an aspect language (e.g., ESA-JS over AspectScript). The overhead of ESA-JS is evidently less than JavaMop, and the JavaMop overhead quickly increases when the pattern is longer, which may be due to the index scheme used in long patterns [42]. In addition, Fig. 17 shows that the choice of the semantics of a matching process strongly affects performance of a ESA-JS stateful aspect: performance of the default semantics is quite similar to an aspect of AspectScript. Instead, the stateful aspect that uses the multiple match semantics differs from the aspect implementation in an exponential manner.

Although Figs. 17 and 18 show less overhead in ESA-JS, these results do not mean that ESA-JS has less overhead for JavaScript than JavaMop for Java. Following the same axis scheme of the previous figures, Figs. 19 and 20 show that the overhead of AspectScript is significantly greater than AspectJ over their base languages respectively. The AspectJ performance is very similar to that of Java (average of 1.8375) instead of the AspectScript performance (average of 1,582.8375).

Discussion The current implementation of AspectScript is very slow compared to AspectJ. This is because AspectScript reifies every join point of the program execution to know whether or not an aspect matches at runtime; no partial evaluation or other optimization is performed. Instead, AspectJ optimizes aspect weaving aggressively, therefore the additional layer introduced by JavaMop is comparatively more costly. In addition, the choice of the matching process significantly impacts

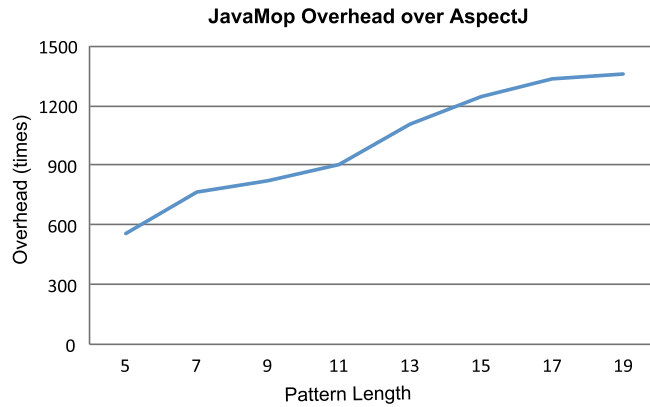


Fig. 18. Overhead of JavaMop over AspectJ.

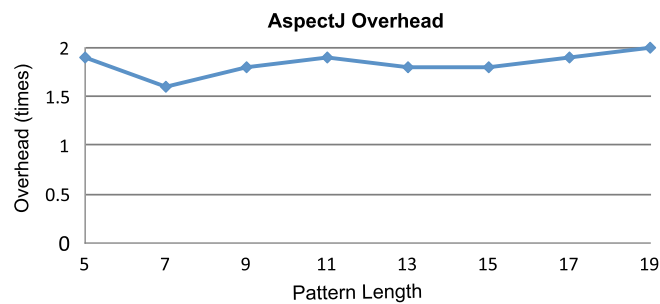


Fig. 19. The overhead of AspectJ over the Java language.

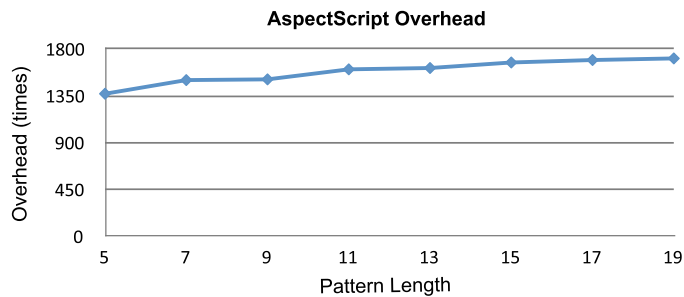


Fig. 20. AspectScript overhead over JavaScript.

Table 1

Time used to execute the ESA-JS runtime in the Tetris game.

Aspects/Tetris game tasks	ESA-JS runtime	Idle
Aspect deployed	8.65%	90.12%
No aspect deployed	4.69%	80.30%

on the ESA-JS overhead. Finally, raw ESA-JS overhead is high, JavaScript is more widely used for interactive applications, that may even include remote communication. For instance, we tested ESA-JS on a JavaScript Tetris game,⁴ finding no noticeable difference in the execution. To quantify the previous point, we use Google Chrome's Developers Tools to measure and compare the time used to execute the ESA-JS runtime in the Tetris. Table 1 shows that an Tetris execution uses 8.65% of time for ESA-JS runtime while 90.12% is *idle*. Although the ESA-JS runtime consumes a significant amount of time, its impact becomes almost unnoticeable due to the interactive nature of the Tetris application.

⁴ <http://www.pleiad.cl/esa/wiki/tetris>.

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 [14] and Halo [13]. For each stateful aspect language, we show the necessary customizations of matching and advising processes.

7.1. Alpha

Alpha uses Prolog, thereby, a pattern is 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 corresponds to a computation history and the set of answers corresponds to the matches of a pattern. 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. For example, if a pattern is simultaneously matched twice and these two matches gather the environments $\boxed{x=1, y=1}$ and $\boxed{x=1, y=2}$ respectively, the aspect advice is executed twice because both environments have a different binding for 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:

```
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);
  });
};
```

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 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. Fig. 21 illustrates this difference, where we evaluate Halo and Alpha with the same pattern and two join point traces. With the first join point trace, we 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 $\boxed{v=1}$, a match is discarded. With the second join point trace, we can instead observe a different number of matches. In Alpha, there are two matches of the pattern because its advising process actually filters matches with the same environment, and in this example, two different join points trigger the matches. With Halo, there is only one match because the filter starts inside of the matching process instead 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:

```
1 var differentBindings = function(rule) {
2   return function(smaches, jp) {
```

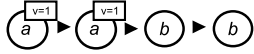


Pattern $a^{[v=?]} \triangleright b \triangleright b$		
Join Point Trace	Alpha	Halo
	1	1
	2	1

Fig. 21. 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.

```

3  var nextSmatches = rule(smatches, jp);
4  var newSmatches = difference(nextSmatches, smatches);
5
6  //filtering new smatches with the same environment
7  var filteredNewSmatches = newSmatches.filter(function(newSmatch) {
8    var oldSisters = sisters(newSmatch, smatches); //get old sisters of newSmatch
9
10   return oldSisters.some(function(oldSister) {
11     return equalEnv(newSmatch, oldSister);
12   });
13 });
14
15 return difference(nextSmatches, filteredNewSmatches);
16 } };

```

The `differentBindings` rule designator allows developers to filter 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. Summary

Through the customization of matching and advice processes, we can instantiate different stateful aspect languages. In addition, any instantiation of ESA can use 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. Design considerations

The main purpose of the ESA design is to give expressiveness and software-engineering properties (e.g., reusability) to the three components of a stateful aspect language: pattern language, matching process, and advising process. In addition, ESA is precisely described using a typed functional language. In this section, we discuss questions and consequences that arise from these decisions.

8.1. An ESA implementation for an object-oriented language?

Although ESA is described through a typed functional language, this proposal has been implemented for two object-oriented paradigms: JavaScript, a prototype-based language, and ActionScript, a class-based language. In both implementations, aspects, join points, and environments are objects. However, these implementations use first-class functions to express patterns and `MatcherCells` rules (i.e., matching processes). This section outlines implementations of previous two components in a class-based language without the support of first-class functions such as in Java. Finally, we show how these components are integrated to create an object-oriented version of ESA.

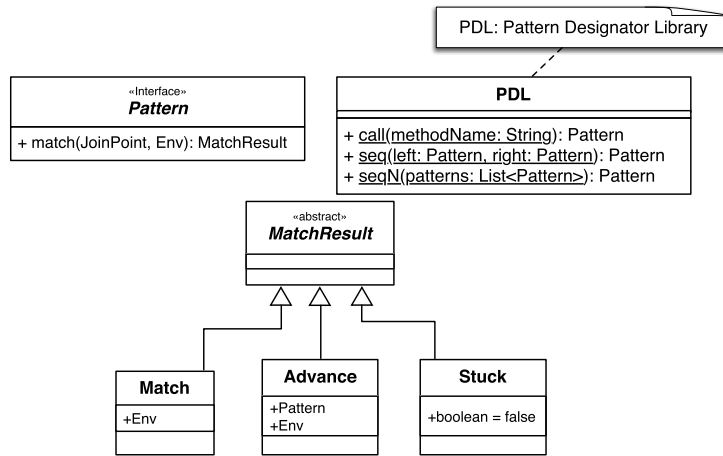


Fig. 22. An object-oriented design for the ESA pattern language.

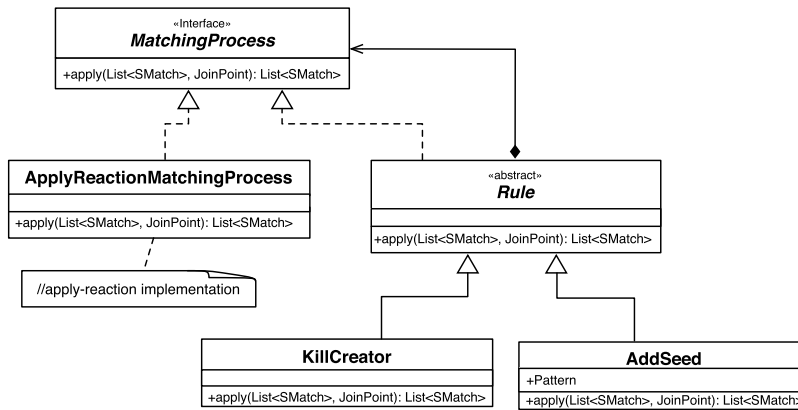


Fig. 23. An object-oriented design for the ESA matching process.

8.1.1. Pattern language

Fig. 22 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 4.1) 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 s-up-down-up = PDL.seqN(patterns);
    
```

8.1.2. Matching process

As mentioned in Section 5.2.1, rule composition of `MatcherCells` uses the Decorator pattern [16] to express a matching process in ESA. Fig. 23 shows an object-oriented solution that uses this pattern to implement matching processes of our proposal. The matching process is an interface, where the `apply` method must be executed for every new join point. A rule like `KillCreator` is a class that implements the interface and decorates the matching process that this rule receives in its creation. This solution allows developers to (dynamically) express all possible compositions of rules. For instance, the following declarations show three matching processes, where `singleMatch` and `singleMatchAtATime` use `applyReaction` as an initial matching process.

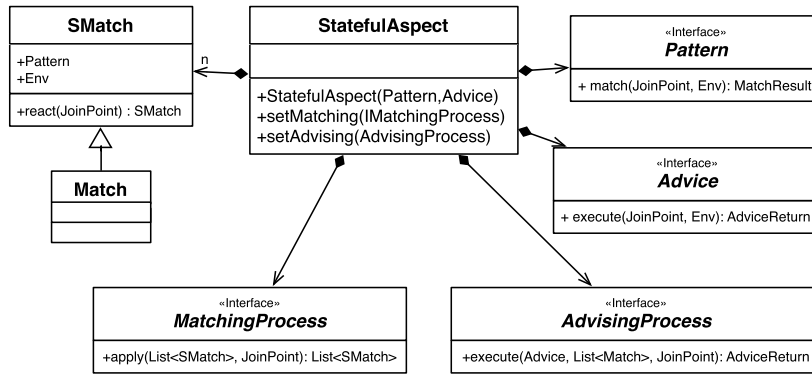


Fig. 24. Class diagram of an ESA stateful aspect.

```
MatchingProcess applyReaction = new ApplyReactionMatchingProcess();
MatchingProcess singleMatch = new KillCreator(applyReaction);
MatchingProcess singleMatchAtATime = new AddSeed(pattern, new KillCreator(applyReaction));
```

8.1.3. ESA integration

Fig. 24 shows the design of ESA stateful aspects. The creation of a stateful aspect requires two objects that implement the interfaces `Pattern` and `Advice` respectively. In addition, a developer can customize the matching and advising processes of a stateful aspect. The following piece of code exemplifies the creation of an aspect for *toggle airplane mode* in ESA:

```
//s-up-down-up as in Section 8.1.1
//singleMatchAtATime as in Section 8.1.2
```

```
StatefulAspect tam = new StatefulAspect(s-up-down-up, new ToggleAirplaneMode());
tam.setMatching(singleMatchAtATime); //customizing matching semantics
```

```
ESA.deploy(tam);
```

A stateful aspect internally controls a list of smatches (*i.e.*, potential matches). Following the weaving process described in Section 5.3.2, we present an object-oriented solution for this process, which evolves the smatch list and executes the advice for each match found:

```
void weaver(StatefulAspect sAsp, JoinPoint jp) {
    List<SMatch> tempSmatches = sAsp.matching(jp);
    List<SMatch> matches = sAsp.getMatches(tempSmatches);
    sAsp.updateSmatches(sAsp.getNoMatches(tempSmatches));
    if (matches.size() > 0)
        //execute advice with bindings of each match
    else
        //execute the join point proceed
}
```

8.2. Pattern language expressiveness

In existing stateful aspect languages, two different pattern language approaches have been commonly used: Turing complete languages as found in Halo [13] and domain-specific languages such as Tracematches [11]. For ESA, we take the first approach. Next, we discuss benefits and drawbacks of this option.

Benefits

- A Turing complete language allows developers to use the full power of a base language to define patterns.
- ESA only requires that developers follow a protocol to define a pattern (see Section 4.1). Similar to JavaMop, another specification of a pattern language can work for ESA if this specification satisfies the protocol.
- A particular benefit of ESA is the separation of the pattern language from the matching process. This separation allows developers to use different matching semantics (*e.g.*, *single match* or *single match at a time*) with the same pattern. This separation is not supported in most stateful aspect languages. For instance, the `rightToggleAirplaneMode` implementation, presented in Section 4.1.2, requires the use of `__RESET` in the advice declaration to match the pattern more than one time.

Drawbacks

- A Turing complete language gives few options to implement optimization strategies; in ESA, it is the use of higher-order functions to define patterns. For example, the pattern below does not allow pattern language developers to know which subpattern, left or right, is evaluated for the next join point:

```
var randomLeftRight = function(left, right) {
  return function(jp, env) {
    if (random() > 0.5)
      return left(jp, env);
    else
      return right(jp, env);
  }
}
```

- Although a general-purpose language is sufficiently expressive and flexible at defining patterns, this kind of language is not focused on the domain problem of pattern definitions [43]. As a consequence, a pattern language like ESA lacks concise and self-documenting abstractions, resulting that developers may introduce erroneous semantics. ESA, for instance, requires a developer to explicitly pass an environment of bindings between patterns.

8.3. Performance

One of the main concerns with existing stateful aspects is their performance. To address with this concern, expressiveness has been sacrificed. Conversely, ESA focuses on expressiveness and also on the modular separation of three main components of a stateful aspect language. As seen in Section 6.3, with this non-positive consequences, performance issues arise. In spite of these consequences, the modular separation between the pattern language and matching process allowed us to discover a new optimization opportunity: an eligible matching process semantics. This is because developers can define the minimal version of a matching process that satisfies requirements of a particular stateful aspect. For example, stateful aspects like `toggleAirPlaneMode` do not need to simultaneously match a pattern, therefore, it is not necessary to keep multiple and temporary matches of a pattern. Despite this optimization opportunity that ESA offers, we are aware that improving performance is one of the major challenges of ESA, and thereby, future work points to this direction.

9. Conclusion and future work

Because creating specialized stateful aspect languages or overburdening their aspects is a common task to address specific 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 of first-class patterns, offering the benefits of reusability and composition of patterns, and *b)* customize internal processes of each stateful aspect. Using this description, we developed ESA-JS, a concrete and practical implementation of ESA for JavaScript (Section 6). In addition, we assessed the expressiveness of this proposal by implementing the semantics of some existing stateful aspect languages (Section 7). To contrast ESA with existing proposals, we develop a reference frame of these proposals in terms of expressiveness (Section 3).

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 work of ESA is oriented towards addressing this concern:

Eligible 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 4.1). Similar to JavaMop [15], we plan to allow developers to select the pattern language expressiveness, e.g., regular expressions. With this, ESA improves performance according to specific features of the selected pattern language. As an additional benefit, the use of simplified domain-specific languages can prevent developers from introducing errors when patterns are defined.

Eligible matching process As mentioned in Section 8.3 and shown in Fig. 17, an appropriated selection of a matching process may prevent keeping unnecessary potential matches of a pattern, therefore, improving ESA performance. We will explore the real impact of this potential optimization.

Matching process Although we showed that the selection of the semantics of the matching process can improve performance (Section 6.3), we think some semantics (e.g., multiple matches) needs to improve its performance. Bodden et al. in [44–47] and Meredith’s dissertation [29] studied several proposals in this line of research, e.g., *dependency advices* [45]. Using these proposals, we plan to improve ESA performance and validate these improvements in ESA-JS.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback on this paper.

Appendix A. 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.

A.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 for knowing what a pattern evaluation returns. Notice that 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 `time-diff` 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))))

```

A.2. Adaptation of MatcherCells for ESA

We adapt the MatcherCells algorithm [37] 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 carried 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)

```

A.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) ))
```

A.4. Advising process

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

```
;;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.

A.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))))
```

Appendix B. A SRS implementation with ESA pattern language

Meredith in [29] implements String Rewriting System (SRS) [48], a Turing complete language, as a plugin for Java-Mop [15]. To show the Turing-complete expressiveness of our pattern language, we define two ESA pattern designators that are able to emulate SRS:

```
1 ;;A SRS pattern is created from rules: a set of key-value
2 (: (Listof (Pair String String)) → Pattern)
3 (define (SRS rules)
4   (SRS-inner rules ""))
5
6 ;;Apart from a rules, this function receives the string that is rewritten according to SRS rules
7 (: (Listof (Pair String String)) String → Pattern)
8 (define (SRS-inner rules str)
9   (λ (jp env)
10    (let* ([str (string-append str (to-string jp))]
11           [str (apply-rules-in-string str rules)])
12     (if (string=? str "@match")
13         env
14         (SRS-inner rules str))))))
```


SRS consists of a set of rules and a string, which is repeatedly rewritten using these rules until that they cannot be applied anymore. In ESA implementation, a SRS pattern begins with an empty string (Line 4) that increases with string representations of join points (Line 10). For every new join point, the pattern returned by SRS-inner applies the defined rules to rewrite the string (Line 11). After applying these rules, if the string is "@match" the pattern matches (Line 13); otherwise the pattern returns a new SRS pattern with the string rewritten (Line 14). This returned pattern will wait for the string representation of the next join point to apply the same rules again.

References

- [1] D. Parnas, On the criteria for decomposing systems into modules, *Commun. ACM* 15 (1972) 1053–1058.
- [2] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Lopes, C. Maeda, A. Mendhekar, Aspect oriented programming, in: Max Muehlhaeuser, et al. (Eds.), *Special Issues in Object-Oriented Programming*, 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*, Budapest, Hungary, in: *Lecture Notes in Computer Science*, vol. 2072, Springer-Verlag, 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*, in: *Lecture Notes in Computer Science*, vol. 2622, 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*, Leuven, Belgium, in: *Lecture Notes in Computer Science*, vol. 5346, Springer-Verlag, 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: *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2005*, San Diego, California, USA, in: *ACM SIGPLAN Not.*, vol. 40, ACM Press, 2005, pp. 365–383.
- [7] P. Augustinov, J. Tibble, O. de Moor, Making trace monitors feasible, in: *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2007*, Montreal, Canada, in: *ACM SIGPLAN Not.*, vol. 42, ACM Press, 2007, pp. 589–608.
- [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*, Genova, Italy, in: *Lecture Notes in Computer Science*, vol. 5653, Springer-Verlag, 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] S. Malakuti, M. Akşit, Event modules: modularizing domain-specific crosscutting RV concerns, in: *Transactions on Aspect-Oriented Software Development XI*, in: *Lecture Notes in Computer Science*, vol. 8400, 2014, pp. 27–69.
- [11] C. Allan, P. Augustinov, 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: *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2005*, San Diego, California, USA, in: *ACM SIGPLAN Not.*, vol. 40, ACM Press, 2005, pp. 345–364.
- [12] S.F. Goldsmith, R. O’Callahan, A. Aiken, Relational queries over program traces, in: *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2005*, San Diego, California, USA, in: *ACM SIGPLAN Not.*, vol. 40, ACM Press, 2005, pp. 385–402.
- [13] C. Herzeel, K. Cybels, 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*, Cambridge, United Kingdom, ACM, 2007, pp. 1–14.
- [14] 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*, in: *LNCS*, vol. 3586, Springer-Verlag, 2005, pp. 214–240.
- [15] P.O. Meredith, D. Jin, D. Griffith, F. Chen, G. Roşu, An overview of the MOP runtime verification framework, *Int. J. Softw. Tools Technol. Transf.* 14 (2011) 249–289.
- [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*, Boston, Massachusetts, USA, ACM Press, 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*, San Francisco, CA, USA, ACM Press, 2008, pp. 395–406.
- [19] É. Tanter, Expressive scoping of dynamically-deployed aspects, in: *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development, AOSD 2008*, Brussels, Belgium, ACM Press, 2008, pp. 168–179.
- [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*, London, UK, Springer-Verlag, 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*, Massachusetts, USA, ACM, 1986, pp. 1–15.
- [24] C.L. Forgy, Rete: a fast algorithm for the many pattern/many object pattern match problem, *Artif. Intell.* 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*, Bonn, Germany, ACM Press, 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*, Colorado, USA, *Electron. Notes Theor. Comput. Sci.* 89 (2) (2003) 108–127.
- [28] F. Chen, G. Roşu, MOP: an efficient and generic runtime verification framework, in: *Object-Oriented Programming, Systems, Languages and Applications, Montreal, Canada*, in: *ACM SIGPLAN Not.*, vol. 42, ACM Press, 2007, pp. 569–588.
- [29] P.O. Meredith, Efficient, expressive, and effective runtime verification, Ph.D. thesis, University of Illinois at Urbana-Champaign, 2012.
- [30] S. Malakuti, Event composition model: achieving naturalness in runtime enforcement, Ph.D. thesis, University of Twente, Enschede, the Netherlands, 2011.
- [31] S. Malakuti, M. Akşit, Evolution of composition filters to event composition, in: *Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC’ 12*, Trento, Italy, ACM, 2012, pp. 1850–1857.
- [32] S. Malakuti, Complex event processing with event modules, in: *Reactivity, Events and Modularity Workshop, REM’ 13*, Indianapolis, USA, ACM, 2013.
- [33] P. Leger, É. Tanter, Towards an open trace-based 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, University of Central Florida, 2010, pp. 25–30.

- [34] 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, SBC, 2010, pp. 123–138.
- [35] P. Leger, É. Tanter, R. Douence, Modular and flexible causality control on the web, *Sci. Comput. Program.* 78 (2013) 1538–1558.
- [36] G.J. Sussman, G.L. Steele 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.
- [37] 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, Potsdam, Germany, ACM Press, 2012, pp. 27–32.
- [38] J.V. Neumann, Theory of Self-Reproducing Automata, University of Illinois Press, Champaign, IL, USA, 1966.
- [39] 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, Porto de Galinhas, Brazil, ACM Press, 2011, pp. 241–252.
- [40] 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, Rennes and Saint Malo, France, ACM Press, 2010, pp. 13–24.
- [41] P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, ABC: an extensible AspectJ compiler, in: Transactions on Aspect-Oriented Software Development, in: Lecture Notes in Computer Science, vol. 3880, Springer-Verlag, 2006, pp. 293–334.
- [42] E. Bodden, Personal communication, July 10, 2012.
- [43] A. van Deursen, P. Klint, J. Visser, Domain-specific languages: an annotated bibliography, *SIGPLAN Not.* 35 (6) (2000) 26–36.
- [44] E. Bodden, P. Lam, L. Hendren, Clara: a framework for statically evaluating finite-state runtime monitors, in: 1st International Conference on Runtime Verification (RV), Paris, France, Springer, 2010, pp. 74–88.
- [45] 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, Charlottesville, Virginia, USA, ACM Press, 2009, pp. 3–14.
- [46] E. Bodden, Specifying and exploiting advice-execution ordering using dependency state machines, 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, University of Central Florida, 2010, pp. 31–42.
- [47] M. Parzonka, A library-based approach to efficient parametric runtime monitoring of Java programs, Master's thesis, Technische Universität Darmstadt, Darmstadt, Germany, 2013.
- [48] R.V. Book, F. Otto, String-Rewriting Systems, Texts and Monographs in Computer Science, Springer, 1993.