Science of Computer Programming 102 (2015) 108–141

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.

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).

http://dx.doi.org/10.1016/j.scico.2015.02.001
0167-6423/© 2015 Elsevier B.V. All rights reserved.

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.