Information and Software Technology 53 (2011) 1165–1182

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Specifying aspect-oriented architectures in AO-ADL

Mónica Pinto *, Lidia Fuentes, José María Troya

Dpto. de Lenguajes y Ciencias de la Computación, University of Málaga, Campus de Teatinos, s/n, E29071 Málaga, Spain

ARTICLE INFO

Article history:
Received 21 July 2010
Received in revised form 17 March 2011
Accepted 15 April 2011
Available online 23 April 2011

Keywords:
Software Engineering
Software Architectures
Languages
Aspect-Oriented Software Development
Metrics

ABSTRACT

Context: Architecture description languages (ADLs) are a well-accepted approach to software architecture representation. The majority of well-known ADLs are defined by means of components and connectors. Architectural connectors are mainly used to model interactions among components, specifying component communication and coordination separately. However, there are other properties that cut across several components and also affect component interactions (e.g. security).
Objective: It seems reasonable therefore to model how such crosscutting properties affect component interactions as part of connectors.
Method: Using an aspect-oriented approach, the AO-ADL architecture description language extends the classical connector semantics with enough expressiveness to model the influences of such crosscutting properties on component interactions (defined as 'aspectual compositions' in connectors).
Results: This paper describes the AO-ADL language putting special emphasis on the extended connectors used to specify aspectual and non-aspectual compositions between concrete components. The contributions of AO-ADL are validated using concern-oriented metrics available in the literature.
Conclusion: The measured indicators show that using AO-ADL it is possible to specify more reusable and scalable software architectures.

1. Introduction

Software architecture can be considered quite a mature discipline, which focuses on a high-level representation of the software system structure. Software architecture helps developers to think about a system's complexity, in order to obtain the architectural configuration that facilitates the best possible evolution and maintenance management of the final system [1]. Because of this, recent approaches in software engineering consider the description of the software architecture as an important part of the software development life cycle. Architecture Description Languages (ADLs) were proposed to represent the software architecture of a system, providing precise descriptions of the constituent computational components and of their interactions [2]. A detailed discussion of the benefits of using ADLs can be found in [3]. Due to the benefits of describing the software architecture in the early stages of the software life cycle, a large number of ADLs have been proposed [3]. These languages usually describe the software architecture of a system in terms of *components* and *connectors*. Components encapsulate computation and connectors represent patterns of communication and interactions between components.

This separation of concerns between computation and communication helps to increase the reuse of software artifacts. However, some other concerns (e.g. security, persistence and synchronization) normally crosscut several components, also affecting component interactions. These crosscutting concerns cannot be easily located using traditional ADLs, either inside individual components or in connectors (none of the ADLs described in [3,4] provide explicit support to separate crosscutting concerns). For example, persistence is a crosscutting concern that deals with recording the state of different software components in a database. Using traditional ADLs, the interaction between persistent components and the database will be scattered among the different component interfaces, and tangled with the components base functionality. This scattered and tangled behavior frequently results in poor architecture descriptions with highly coupled components, preventing reusability and evolution management of the affected components. Recently several empirical studies reveal that crosscutting concerns degrade system quality because they negatively impact internal quality metrics such as coupling, separation of concerns, defects or reusability [5].

Coping with scattering and tangling of crosscutting concern problems is the main goal of Aspect-Oriented Software Development (AOSD)[1] modeling crosscutting concerns as 'aspects'. AOSD promotes the principle of separation of concerns throughout all

* Corresponding author. Tel.: +34 952132796; fax: +34 952131397.
E-mail addresses: pinto@lcc.uma.es (M. Pinto), lff@lcc.uma.es (L. Fuentes), troya@lcc.uma.es (J.M. Troya).

[1] http://www.aosd.net.

0950-5849/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2011.04.003

the phases of the software life cycle, including software architecture design (Aspect-Oriented Architectural Design, AOAD). In [6] a quantitative comparison of aspect-oriented (AO) and traditional architectures is presented, revealing that an AO architecture normally improves separation of concerns and component cohesion, enhancing the evolution management of the system. On the other hand, since crosscutting concerns are normally hard to find, understand and work with, separating and specifying them at the architectural level will provide a means of understanding how to make changes to system concerns (update/add/remove concerns) correctly and consistently before moving to implementation. Despite the need for and benefits from specifying and documenting crosscutting concerns, there is a notable lack of support for them in existing ADLs (none of the ADLs described in [3,4] provide explicit support to specify and document crosscutting concerns).

In this paper we present AO-ADL, an aspect-oriented ADL (AO ADL) based on XML. The goal of AO-ADL is to provide a language support flexible enough to separate and inject crosscutting concerns in a non-intrusive way at the architectural level. Following a symmetric approach, AO-ADL models crosscutting concerns (i.e. *aspectual components* in AOAD terminology) using the classical component. So, components model both crosscutting and non-crosscutting concerns (i.e. *base components* in AOAD terminology). As crosscutting concerns influence base component interactions, it seems reasonable to model their composition with base components as part of connectors. AO-ADL extends the classical connector semantics with enough expressiveness to model the influence of crosscutting concerns on component interactions by means of *aspectual compositions*. Aspectual compositions in AOAD terminology is defined as the weaving between base and aspectual components at architectural level. In order to preserve the building blocks that define the architectural structure in classical ADLs, connectors in AO-ADL encapsulate any kind of component interaction, either aspectual or non-aspectual, in a homogenous way. This facilitates the understanding of how a set of aspectual components affect a given interaction between base components.

So, with AO-ADL it is possible to retain base structural properties of architectural models while at the same time being able to insert "aspects" into the model simply by modifying or adding new connectors. The resulting architectural description will be more cohesive and less coupled, since base components do not contain any reference to aspectual components, which only encapsulate one concern. This means that the evolution and maintenance management is greatly improved since crosscutting concerns are localized in separate modules and their composition is defined separately inside connectors. This will have a positive impact on several qualities of the software system as shown in this paper (e.g. evolvability, traceability, composability and reusability[2]). Finally, describing AO-ADL architectural specifications is supported by a toolkit that facilitates the work of software architects, being also part of an AO integrated development environment (IDE).[3]

The organization of the rest of the paper is as follows. The following section presents the AOSD background, firstly describing AO architectural design concepts and, secondly, providing a detailed discussion about the shortcommings of current AO ADLs. Then, the case study used to illustrate AO-ADL concepts and the main features of the language are presented in Section 3. Since the main building blocks of AO-ADL are components and connectors we dedicate a section to each of them in Sections 4 and 5. Then, in Section 6 we describe the AO-ADL Tool Suite that supports the specification of AO-ADL descriptions, and outline how these tools are integrated in a development methodology. As a proof osf concept, in Section 7 we validate our approach by presenting how specifying software architectures in AO-ADL may have a positive impact on several qualities, enhancing the system evolution management. Finally, in Section 8 we present our conclusions and further work.

2. Related work on aspect-oriented architecture design

In this section we first describe common AOSD terminology, specific to the architectural-level. Secondly, we provide an overview of existing aspect-oriented ADLs.

2.1. Aspect-oriented architectural design

AOSD aims to achieve separation of those concerns that are scattered among multiple software artifacts, which are known as crosscutting concerns. The main motivation for using AOSD is the impossibility of traditional software technologies, like object-oriented programming or component-based software development, to appropriately modularize crosscutting concerns. AOSD copes with this limitation by introducing the concept of *aspect*.

Although AOSD covers all the phases of the software life cycle, the terminology used at architectural level is strongly influenced by the one introduced in Aspect-Oriented Programming (AOP) [7]. As defined in [8], an *architectural aspect* is a unit for modularising an otherwise crosscutting concern. Non crosscutting concerns are considered *base components*. Inside an architectural aspect, the *advice* specifies the operations that will transform (augment or diminish) the architecture at certain *architectural join points*. Architectural join points are well-defined points in the specification of the software architecture, where aspect composition takes place. For each architectural style, a specialized set of join points can be defined, such as the invocation of services, publishing/subscribing of events, etc. [9]. Usually the aspect behavior (or architectural advice) can be inserted or woven before, after or around a join point. *Architectural pointcuts* refer to collections of architectural join points. An architectural pointcut is an expression that specifies every place where an aspect behavior is injected.

Nevertheless, defining a new artifact called aspect (or architectural aspect in AOAD) introduces some asymmetry into AO architectures, causing some problems to arise. Asymmetric architectural models mean that concerns modeled as aspects cannot be (re) used as standalone architectural components, reducing their reusability possibilities. Also, this asymmetry prevents the same component from playing an aspectual or non-aspectual role depending on the requirements of each system. In order to overcome these limitations, symmetric models do not distinguish between components and aspects (as is defined in [10–12]), but between aspectual or non-aspectual compositions. An *aspectual composition* specifies the interaction between base components influenced by other components playing an aspectual role, according to an architectural pointcut definition. When a component plays an aspectual role it could also be named at architectural level an *aspectual component*.

2.2. AOAD related works

A large number of AO approaches have emerged in recent years at each level of the software life cycle. A detailed comparison between the most representative AOAD approaches can be found in the survey [13]. The results of this survey illustrate that only a few AO proposals were defined at the architectural level, including our previous work DAOP-ADL [14]. After this survey was published, some new AO ADLs have appeared. In this section we will compare several AO ADLs and remark on some shortcomings that

[2] These properties are defined as qualities of an architectural configuration description in [3].

[3] http://www.aosd-europe.net/ In Research → Atelier → IDE Framework → Tools → IDE Tools with Achieved Integration.