Toward an Aspect Oriented ADL for Embedded Systems

Sihem Loukil, Slim Kallel, Bechir Zalila, and Mohamed Jmaiel

ReDCAD Laboratory, University of Sfax
B.P. 1173, 3038 Sfax, Tunisia
sihem.loukil@redcad.org, slim.kallel@fsegs.rnu.tn,
{bechir.zalila,mohamed.jmaiel}@enis.rnu.tn

Abstract. Managing embedded systems complexity and scalability is one of the most important problems in software development. To better address this problem, it is very recommended to have an abstraction level high enough to model complex systems. Architectural description languages (ADLs) intend to model these systems and manage their structure at a high abstraction level. Traditional ADLs do not provide appropriate formalisms to separate any kind of crosscutting concerns. This frequently results in poor descriptions of the software architectures and a tedious adaptation to constantly changing user requirements and specifications. AOSD (Aspect Oriented Software Development) deals with these problems by considering crosscutting concerns in software development. The effectiveness of AOSD appears when aspect concepts are considered throughout the software's life-cycle.

In this paper, we propose a new aspect language called AO4AADL that adequately manipulates aspect oriented concepts at the software architecture level to master complexity and ensure scalability.

1 Introduction

Implementing and managing software embedded systems are tedious tasks, due to the complexity and strict requirements of such systems. A possible solution to manage this complexity is to model these systems at architecture level. Architecture description languages [1,2] are an important tool for early analysis and feasibility testing. They can also support code generation and allow easier management of the configuration and the deployment of systems.

Traditional ADLs provide formalisms to describe functional concerns (what the system does) and non-functional concerns (the quality of service and the conditions under which the system correctly operates). They lack of appropriate formalisms representing crosscutting concerns (behavior that cuts across the typical divisions of responsibility). This lack frequently results in poor descriptions of the architectures and a tedious adaptation to the constantly changing user requirements and execution context. The specification of non-functional concerns is not well-modularized, as it is *tangled* with the specification of each component's core functionality or *scattered* across the specification of different

M. Ali Babar and I. Gorton (Eds.): ECSA 2010, LNCS 6285, pp. 489–492, 2010.

components. This results in an increase of the model complexity. Furthermore, when the designer modifies one of the concerns, he should manipulate all parts of the model related to that concern which is challenging as these parts are mixed with elements of other concerns. AOSD deals with these problems by considering crosscutting concerns in software development.

In this paper we propose AO4AADL, an aspect oriented language for AADL [2], a well known ADL. This language considers aspects as an extension of AADL using "annexes", an intrinsic mechanism to extend the AADL language. We consider that aspects can be specified in a language other than AADL, and then integrated in AADL models as annexes. The remainder of this paper is organized as follows. Section 2 overviews the syntax and the semantics of a new aspect oriented language for AADL in terms of pointcut and advice. Section 3 gives the related works and Section 4 concludes the paper and presents ongoing work.

2 The AO4AADL Language

Considering aspect concepts at the beginning of software life cycle is considerably valuable: it improves comprehensibility, evolution and reuse in the development of complex software systems. For this purpose, we extended AADL. Many reasons led us to this choice. AADL is a concrete ADL in which all elements correspond to concrete entities that allows describing both hardware and software parts of the system. It introduces two extension mechanisms: properties and annexes. These mechanisms make the language much easier to enrich. Moreover, they offer a good foundation for additional capabilities in analysis, automated system integration, distribution, and dynamism. Based on the annex extension mechanism, we propose to enrich AADL specifications with aspect concepts.

An AO4AADL aspect consists of two parts: (1) *pointcut_specification* determines the conditions under which the aspect is invoked by the corresponding functional components and (2) *advice_specification* encapsulates the behaviour of the aspect depending on its location. If the aspect influences only one component, it should be declared as an annex inside this component. If the aspect influences the behaviour of more than one component, it should be declared as an annex library in an AADL package outside the components.

2.1 Pointcut Specification

A pointcut is defined as a set of joinpoints which are used to accomplish the composition between the aspect description and the base description of the software system. Pointcut definitions consist of a left-hand side containing the specification of the pointcut name and parameters (the data available when the events happen) and a right-hand side consisting of the pointcut itself.

A joinpoint specifies a well-defined point of the aspect behaviour execution. AO4AADL explicitly defines the architectural joinpoints as places where the effect of aspect annex can occurs. They include: (1) the subprograms already declared in the AADL specifications (2) the outgoing data flow emerging from an AADL component and (3) the incoming data into an AADL component

with either a call or an execution primitive. Moreover, a joinpoint can expose instance of checks and control to specify when the arguments are instances of specific types or the types of the specified identifiers.

In real architectural configuration, aspect behaviour may be executed by several architectural joinpoints. Hence, an architectural pointcut should be defined as an expression that specifies the set of joinpoints to which the behaviour of an aspect is applicable. In order to express the architectural quantification mechanism, we introduce the operators "and" ("&&") and "or" ("||") as well as wildcards such as "*" to describe sets of joinpoints invoking the same advice.

Listing 1 shows an example of an aspect code, `CheckCode`, described in AO4AADL. It belongs to the software part of an automated teller machine (ATM). It specifies that the client has exactly three authentication attempts. Each time it gets an incorrect code, the system prompts for the code again. If it reaches the third time, the card will be rejected and an explanatory message is displayed to the customer. The example shows also the interaction between the AO4AADL code and the corresponding AADL entities (here, the out port `RestoreCode_out` specified in the AADL thread containing the annex).

Listing 1. Example of AO4AADL aspect

```
1  aspect CheckCode{
2  pointcut Verification():call outport RestoreCode_out (..);
3  advice around ():Verification() {
4    variables{counter : Integer_Type; message : String_Type;}
5    initially{counter:=1; message:="Card Rejected!";}
6    if (counter = 3){
7      RejectedCard_out! (message);
8      counter := 1;}
9    else{
10     proceed();
11     counter := counter + 1;}}}
```

2.2 Advice Specification

The advice defines the crosscutting relationships among the aspect behaviour and the place where to inject this behaviour (joinpoint). AO4AADL provides three kinds of crosscutting interactions listed by the keywords: `before` (the advice action runs before the joinpoint), `after` (the action runs after each joinpoint) and `around` (the action runs before and after of each joinpoint. The joinpoint itself can be executed by calling `proceed`). To each pointcut, we can associate one or more crosscutting behaviour which is expressed in an advice section allowing one or more advice sections can be associated to the same pointcut.

The syntax used to specify the action performed by the advice action on the functional component is inspired from the AADL Annex Behavior [2] with some modifications to express other requirements[1].

In our aspect `CheckCode` (Listing 1), the advice is presented in lines (3 – 12). We use an `around` advice to execute the joinpoint only if the user has remaining attempts. In the other case, the card will be rejected.

[1] The full AO4AADL grammar can be found on `www.redcad.org/projects/AO4AADL`

3 Related Work

There are several points of view on how to represent aspects at architectural level but most of existing Aspect-Oriented architectural approaches agree on that the semantics of the composition should be somehow extended in order to ensure the connection between the aspects and the basic components.

As stated by [3], extending a component based formalism to AOSD is performed either symmetrically or asymmetrically. Some existing implementation of AOSD in an ADL used the asymmetric approach [4,5]. They use two different formalism to describe the model and the aspect. Some other implementations use the symmetric approaches [6]. They use components to model both functional components and aspects. In our case, we integrated the aspect code in the model (in the same document) while keeping out model compatible with tools that do not support AO4AADL. Therefore we used the AADL annex extension mechanism. This allows us to have a whole new formalism to describe the aspects (benefit of the asymmetric approach) while keeping a single model which can be reusable among different tools (benefit of the symmetric approach).

4 Conclusion and Future Work

In this paper, we presented AO4AADL, an aspect-oriented ADL, which extends the AADL language using the annex extension mechanism to capture crosscutting concerns at architectural level. We defined a rigorous grammar that supports most of aspect concepts. We are currently working on the implementation of the code generator from AO4AADL aspect to AspectJ aspect. Future work include using AO4AADL for defining an approach for managing "at runtime" configurable (adaptive) embedded systems.

References

1. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. 26, 70–93 (2000)
2. SAE: Architecture Analysis & Design Language (2004), `http://www.sae.org`
3. Harrison, W.H., Ossher, H.L., Tarr, P.L., Harrison, W.: Asymmetrically vs. symmetrically organized paradigms for software composition. Technical report, Research Report RC22685, IBM Thomas J. Watson Research (2002)
4. Navasa, A., Pérez-Toledano, M.A., Murillo, J.M.: An ADL dealing with aspects at software architecture stage. Inf. Softw. Technol. 51, 306–324 (2009)
5. Jing, W., Shi, Y., LinLin, Z., YouCong, N.: AC2-ADL: Architectural description of aspect-oriented systems. In: Proc. of the ASEA, pp. 147–152. IEEE, Los Alamitos (2008)
6. Pinto, M., Fuentes, L.: AO-ADL: An ADL for describing aspect-oriented architectures. In: Early Aspects: current challenges and future directions, pp. 94–114 (2007)