AC2-ADL: Architectural Description of Aspect-Oriented Systems

Wen Jing¹,², Ying Shi¹, Zhang LinLin¹, Ni YouCong¹
¹(State Key Lab of Software Engineering, Wuhan University Wuhan, China P.R.)
²(Electrical Engineering and Computer Science The Henry Samueli School of
Engineering University of California Irvine)
{jjingwen_1982, zllnadasha}@yahoo.com.cn, yingshi@whu.edu.cn

Abstract

The traditional Architectural Description Languages ADLs lack the ability to describe the
crosscutting concerns and crosscutting interactions in the software architecture, which result
in the design of the system difficulty in comprehension, evolution and reuse. This paper
proposes a new Aspect-Oriented ADL — AC2-ADL to provide a formal basis for
representation of the tangling and scattering concerns and establish the software architecture
with higher dependability.

1. Introduction

Since the beginning of 1990s, Software Architecture (SA) has become an area of
intense research in the software engineering community. Architecture description
languages (ADLs) have been proposed as modeling notations to support architecture-
based development [1]. However, during the increasingly in-depth study and widely
application of ADLs, there is a gradual recognition that conventional ADLs lack
concepts to support the modular representation of crosscutting features and behaviors in
software architectural design and development stages. These crosscutting features and
behaviors always tend to cut across the boundaries of other modules, leading to high
coupling between them, and complexity in description of these modular units.

Nowadays, with the increasing maturity and widespread application of AOP [2]
(Aspect-Oriented Programming) technique which define the notion of aspect as being a
software entity that implements a crosscutting concern, and with the emergence of early
aspect [3] technique which has tried to generalize the aspect concept and apply it to the
earlier phases of the software life cycle, it is considerably valuable to research on how
to integrate corresponding Aspect-Oriented (AO) concepts to the design phase of
software architectures. The expected benefits are improved comprehensibility, ease of
evolution and increased potential for reuse in the development of complex software
systems.

Unfortunately, because the concept at software architecture level is much higher
abstract than that at the implementation level, It is difficult to directly and
automatically introduce the conventional aspect-oriented concepts, approaches and
techniques, which have almost exclusively focused on identifying the aspects in
programming-level artifacts, into the software architecture design process. This paper
proposes AC2-ADL, a new Aspect-Oriented ADL, to provide formal support to describe
SA of an Aspect-Oriented application or system in the different abstract degrees. In
AC2-ADL, Designers can model architectural aspects by using Aspectual Component (AC) to describe and encapsulate the features and behaviors only related to a specific crosscutting concern, and then employ a special kind of connector called Aspectual Connector (AC) to capture the crosscutting interaction of certain architectural elements, finally, define architectural joinpoints to accomplish the composition between the components and aspectual components. Section 2 presents the conception framework of AC2-ADL. Section 3-5 shows respectively how to design and describe Aspectual Component and Aspectual Connector and Architecture Configuration in AC2-ADL. Section 6 takes an Online Action System as a case study, illustrating how to apply AC2-ADL to resolve various problems occurring during the process of Aspect-Oriented architecture design. Finally, Section 7 and 8 presents the related work and our conclusions.

2. Outline of AC2-ADL Structure

Compared with traditional ADLs, AC2-ADL, in addition to general units such as components, connecters, configurations and so on, introduces the conceptions of Aspectual Component and Aspectual Connector, as the first-class citizenship. A partial syntax of the AC2-ADL types is specified in BNF. These notations provide abundant types for designing AO architectures.

Abstract syntax of types

| Type ::= baseType | abstractType |
| baseType ::= simpleType | complexType |
| simpleType ::= Any | Natural | Integer | Real | Boolean | String |
| complexType ::= tuple[Type,...,Type] | location[Type] | sequence[Type] |
| set[Type] | bag[Type] |
| abstractType ::= component | aspectComponent | aspectConnector | connector |
| architecture | interface | role |
| interface ::= providedInterfaces | requiredInterfaces | crosscuttingInterfaces |
| role ::= baseRoles | crosscuttingRoles |

In addition, based on the project linear temporal logic theory, AC2-ADL, integrates a special kind of temporal logic language XYZ/E [4] for supporting behavior and protocol description specifications for components and connecters. These specifications conform to the rules of XYZ/E, whose statements are called condition elements (c.e.). The form of condition elements is as follows:

\[ LB=y \land R \Rightarrow SOv=e \land SOLB=z, \]  
\[ LB=y \land R \Rightarrow @(Q \land SOLB=z). \]

The formula (i) directly defines the states transition relationships. Here, LB is used to represent the program control and “LB= y” denotes that program control is in state “y”. “R” is pre-conditions; "SO" is a kind of temporal operator indicating the next time point; “v=e” is some evaluation expression where variable or a list of variables “v” can be endowed with a value or a list of values of expression “e”; “LB=z” denotes a forward state. The formula (i) can be explained that under the pre-condition R satisfied, the state “LB=y” is transitioned into the next state “LB=z”, with the variable “v” being
The formula (ii) presents more abstract c.e specifications. Here, “@” can be "$O$$,” or “◇” which is another kind of temporal operator denoting the ultimate time point. “Q” can be the first-order logic expressions for describing the post-conditions.

Moreover, XYZ/E provides corresponding Communication Mechanism. The message passed from one process to another is implemented by using channel operations. There are two channel operation commands: output command (write channel) and input command (read channel). Input and output commands are formulized as follows:

\[
LB=y \land R \Rightarrow ChNm?x \land SOLB=z \\
LB=y \land R \Rightarrow ChNm!e \land SOLB=z,
\]

Here, “ChNm” is a channel name, and x is a variable, e is an expression. “ChNm? x” means that the variable x will receive data through the channel ChNm from the output process to the input process; while, “ChNm! e” means that the value of expression e will be sent through the channel ChNm from the input process to the output process. In AC2-ADL, we use c.e formulae to describe the detailed computational behaviors of various components and, together with the input and output commands to describe the interaction protocols involved in each interface.

3. Aspectual Component in AC2-ADL

In AC2-ADL, Aspectual components have a new kind of interfaces. Unlike conventional interfaces (such as provided and required interfaces which respectively represent the published and customized services), this interfaces are called crosscutting interfaces[5] describing the crosscutting services provided by aspectual components. In this section, we provide a partial syntax of the prototype of the aspectual component and interfaces. An aspectual component is modeled in AC2-ADL as follows in fig3 -1:

```
aspectComponent ::= aspectComponent component_name is 
{ 
 parameters 
 {parameters name is simpleType;} 
 methods {methods} 
 crosscuttingInterface {crosscuttingInterfaces} 
 requiredInterface {requiredInterfaces} 
 [subArchitecture subArchitecture_name is{ architecture|null;}]
 [mappingDeclaration {mapping_expression}| null;]
 [internalProcess is 
 {process_expression;}| null;]
 [constrains is 
 {constrains_expression;}| null;] 
 }

crosscuttingInterfaces ::= crosscuttingInterface_name is 
{ 
 [add_Operation method | {method;}| null ]
 [replace_Operation 
 {method | {method;} | null ]
 [introduce_Operation 
 {parameters_name | methods| 
 providedInterfaces | 
 requiredInterfaces} | null}]

requiredInterfaces::= requiredInterface_name is 
{ 
 [Operation method | {method;}| null ]

providedInterfaces::= providedInterface_name is 
{ 
 [Operation method | {method;} | null ]

method ::= method_name is 
{direction (parameters_name) 
{ parameters_name;}|null} 

direction ::= in | out | inout
```
The keyword *aspectComponent* definition includes a set of interfaces (crosscutting and required), and methods of computation, internal computational process, data type definitions and design constraints, even sub architecture. If an aspectual component contains some sub-architecture, as a result of the elements in the sub architecture invisible to other external modules, this composite module must provide relevant specification support to indicate the mapping relationships between the internal system representation and the external interfaces of the module itself. In the next, AC2-ADL interfaces are specified as follows in fig3-2.

A crosscutting interface provides three kinds of operations: *add*-*Operation*, *replace*-*Operation* and *introduce*-*Operation*. Among them, *add/replace*-*Operation* contains a set of methods. While *introduce*-*Operation*, besides some methods, can include a set of fields and/or provided/required interfaces. Note that the AC2-ADL also supports communication scenarios in which a sequence of received notifications and/or requests may result in a sequence of outgoing notifications and/or requests.

4. Aspectual Connector in AC2-ADL

Inspired by the work of Alessandro Garcia[6], aspectual connectors involve two kinds of roles: base role and crosscutting role. They represent the participators who can join in the interaction described by this connector. Base role indicates a set of objects applied by aspectual components, such as component instances, connector instances or interfaces, and crosscutting role indicates a set of crosscutting objects, such as the crosscutting interfaces of aspectual component instances. Each role can contain one or multiple behaviors, which specify a list of activities of this role. Compared with aspectual connector, traditional connectors in AC2-ADL only have base role.

Factualy, in addition to the declarations of aspectual connectors and the definitions of roles and behaviors, most design of aspectual connector is mainly concentrated on interaction specifications which represent the crosscutting relationships among the crosscutting roles and base ones. AC2-ADL provides four kinds of the crosscutting interaction protocols listed by the keywords: *before*, *after*, *around* and *introduce*. The semantics of crosscutting interaction protocol types is similar to that of advice composition from AspectJ[2].

As shown in Fig.4-1, these crosscutting interaction types are embedded in corresponding crosscutting interaction expressions, constituting the resulting crosscutting protocols. Some constrains for connectors may be enforced by roles and among roles, for example, the priority of roles. For the same reasons as aspectual component, the specifications of constrains are omitted.

```plaintext
aspectConnector::=  
aspectConnector connector_name is {  
  baseRole baseRoles  
  crosscuttingRole crosscuttingRoles  
  crosscuttingProtocols  
    crosscuttingProtocol_expressions |  
    {crosscuttingProtocol_expressions;} | null;  
  {constrains is {constrains_expressions;} | null;}  
}  
baseRoles::={Role_name:;}{{Role_name behaviors behavior_names;}}
```
5. Architecture and Configuration in AC2-ADL

System architecture is a composite element representing all modules of this system, as well as the configurations (topologies) of these modules’ instances. A partial syntax of the prototype Aspect-Oriented architectural specification can be displayed in Fig.5-1.

Besides typical composition among components in architectural configurations, AC2-ADL must add more support to specify the compositions between two kinds of components. Thus, AC2-ADL explicitly defines the architectural joinpoints: They are some places where the effect of aspectual components can occur. These places include: ①the component /aspectual component instances, ②the various interfaces from these instances, ③the operations from interfaces, ④and even the connectors’ instances. Thereby, four different kinds of the architectural join point are proposed to denote the semantic of joinpoints which are respectively defined as: component_JP, interfaces_JP, operation_JP, and connector_JP

In real architectural configuration, a base role of an aspectual connector may be played by several architectural joinpoints that may be affected by aspect components. Hence, an architectural pointcut designator (pcd) should be defined as a formula that specifies certain joinpoint or the set of joinpoints to which the crosscutting interface of an aspect component is applicable. In order to express the architectural Quantification Mechanism[7], we introduce the operations and, or and not, as well as Wildcards such as ‘*’ to describe sets of joinpoints playing the same base role. The grammar of pcd’s is given in the Fig.5-2.
The architectural configuration, whose partial syntax is represented in Fig5-3, is defined by listing a set of connections between pcd and base roles of connector instances, and a set of connections between crosscutting interfaces and crosscutting roles. In addition, in order for more explicit and accurate expressions of the attachments between the different operations and the corresponding behaviors, configuration specification provides a piece of attachments, as an optional item, for each connection. Consequently, the compositions between components and aspect components can be more clearly described in the architectural configuration.

6. Case Study

In order to show the contributions of our AC2-ADL, we use the Online Auction System (OAS) model problem, taken from the software architecture literature.

Based on the AO architecture design of OAS, the following sub-sections present two classical case studies to illustrate how to formally describe the heterogeneous aspectual components and deal with intertwined aspectual components.

6.1. Second-order headings

An example of heterogeneous aspectual components[5] is the ExceptionHandling aspectual component. There are different exceptional situations in OAS that require different solutions, such as communication problems, data storing and retrieving problems, invalid data problems and so on. All these problem categories can be represented as sub-aspectual components constituting the composite ExceptionHandling as shown in Fig.6-1. Each sub-aspectual component provides corresponding crosscutting interface with encapsulation of the different exception handling features.
These internal crosscutting interfaces can be bound to the external crosscutting interfaces of composite ExceptionHandling to affect other external components according to the thrown exception events. By using the choose statements of XYZ/E, we can describe the internal computing process of composite ExceptionHandling. From Fig.6-1, we can see the keyword internalProcess describes the computational behaviors, where ExceptionHandling, through external crosscutting interfaces, receives the corresponding exception event, and then calls the relevant handler methods, finally sends the error information from the internal crosscutting interface to the external one. The mark “!!” denotes these processes can be repeated.

Fig.6-1 Modeling Heterogeneous Aspectual Component

```java
aspectComponent ExceptionHandling is {
  crosscuttingInterface extern_RemoteException is {};
  crosscuttingInterface extern_IOException is {};
}

subArchitecture exceptionHandling_SubStructure is {
  aspectComponent remoteExceptionHandler is {
    ...
    Methods print_ErrorLog is out (string errorInformation;) 
    CrosscuttingInterface remoteException is {
      add_Operation print_ErrorLog();
    } ...
  } ...
  aspectComponent IOExceptionHandling is {
    ...
    CrosscuttingInterface IOException is {...}
  } ...
  ...
  aspectComponent_instances
    RemoteExceptionHandling_Instance is remoteExceptionHandling
    IOExceptionHandling_Instance is IOExceptionHandling ...
}

mappingDeclaration
  remoteExceptionHandling_Instance , remoteException binds extra_RemoteException;
  IOExceptionHandling_Instance , IOException binds extra_IOException;

internalProcess is {
  LB=star1 ||
  extern_RemoteException?RemoteExceptionEvent., SO LB=11
  LB=11=>print_ErrorLog, SO LB=12
  LB=12=remoteException!errorInformation., SO LB=Exit |
  extern_IOException?IOExceptionEvent., SO LB=12
  LB=12=>print_ErrorLog, SO LB=12
  LB=12=IOException!errorInformation., SO LB=Exit |
  ...
}
```
6.2. Intertwined Aspectual Components

The phenomenon of overlapping concerns often appears all over the complex system, resulting in multiple aspects relevant to these concerns intertwining with each other at the same joinpoint.

As shown in the Fig.6-2, Synchronization and Replication aspectual components arises the intertwining behaviors when data storing and retrieving activities occur. Synchronization must control multiple processes to access the identical storage, for example, several clients can simultaneously place bids. On the other hand, one of crosscutting services of Replication needs to check the overflow problems of storage. This section proposes a new solution for such kind of problems.

We employ aspectual connector \textit{synch\_Repli\_Connector} to represent the interactions among aspectual component \textit{Synchronization}, \textit{Replication} and component \textit{Database}. This aspectual connector has one base role \textit{accessStorage} with two behaviors \textit{dataStoring} and \textit{dataRetrieving}. In addition, two crosscutting roles are also involved: \textit{synch\_CRole} conations
two behaviors locking and unlocking, and replic_CRole includes storageChecking and storageReplicating behaviors. The crosscutting protocols represent the interaction relationships among these behaviors of roles. A partial description of synch_RepliConnector can be shown in Fig.6-2. At meantime, their architectural configuration is presented, where crosscutting roles synch_CRole and replic_CRole respectively are played by the crosscutting interfaces from the instances synchronization and replication, and the methods from these crosscutting interfaces also are attached to the corresponding behaviors. In the same way, the traditional component Database’s instance can be connected with the base role accessStorage.

7. Related Work

Nowadays, AOSD techniques have become an important frontline and hot topic in software engineering. Our work has been influenced by research in several areas: AO software design, AO software architecture and AO ADLs.

Chavez [5] has proposed that crosscutting interfaces as a conceptual tool for dealing with the complexity of heterogeneous aspects at the design level. Crosscutting interfaces have been incorporated by the aSideML modeling language in order to enhance aspect description at the design level. In process of resolving the crosscutting interaction problems, our approach mainly refers to Batista’s work [6], which proposes Aspectual Connector (AC) as the only necessary enhancement to an ADL ACME that supports the seamless integration of AOSD into software architecture. The another representative work is CAM/DAOP[8] that defines DAOP-ADL, a component- and aspect-based language designed to be interpreted by DAOP, a dynamic component- and aspect-oriented platform to specify the architecture of an application in terms of components, aspects and a set of plug-compatibility rules between them.

8. Conclusion

This paper has systematically addressed an aspect-oriented architecture description language AC2-ADL which introduces a special kind of aspectual component into the architecture level to capture kinds of crosscutting concerns. The aspectual connector provides the corresponding composition specifications to support crosscutting interaction descriptions. Architectural join points are not specified as part of the aspectual component but delayed to explicitly address in the configuration section, contributing to make aspectual components more reusable. Finally, this paper takes two examples of OAS, against heterogeneous aspects and concern overlapping problems, as demonstrations to resolve the modeling problems. The future research will pay more attention to aspect-oriented architecture dynamic evolution and reuse. In addition, we will enhance the expressive semantic capability, especially, formal description capability of component behaviors and interaction contracts, and even the overall system context.

9. Acknowledgements

We gratefully acknowledge supports from the National Natural Science Foundation of China (Grant No.60773006), and the Doctoral Fund of Ministry of Education of China (Grant No. 20060486045).

10. References


Authors

WEN JING, female, Joint Ph. D.Candidate of State Key Lab of Software Engineering and University of California Irvine. Research fields: Aspect-oriented software engineering, Semantic software engineering, Real-time computing system.

YING SHI, Male, Professor, Associate chairmen of State Key Lab of Software Engineering. Research fields: Software engineering, Service-Oriented architecture, Semantic software engineering.

Zhang Lin Lin
Female, Ph. D.Candidate of State Key Lab of Software Engineering, Research fields: Aspect-oriented software engineering, Service-Oriented Architecture.