Architectural design of the Online Auction System with AOSAD

Wen Jing ,Ying Shi, NiYouCong, Zhang LinLin
State Key Lab of Software Engineering, Wuhan University W*uhan, China P.R.*
jingwen_1982@yahoo.com.cn, yingshi@whu.edu.cn, nyc@hubu.net

Abstract

Crosscutting behaviors and features of architectural units have always been a tricky issue in software architecture design. If not well treated, they may cause some unnecessary coupling among architectural units and hamper maintenance, evolution and reusability of software products. Appropriate modeling approaches and expressions contribute to the solution of these problems. This paper proposes an Aspect-Oriented Software Architecture Design approach AOSAD to design the software architecture of systems. AOSAD employs a special kind of architectural component called Aspectual Component to encapsulate crosscutting behaviors and features to improve modularization, and introduces a special kind of connector called Aspectual Connector to make the complicated interaction more controllable. Additionally, this paper proposes a new Aspect-Oriented architecture description language AC2-ADL for formal specification of the software architecture of systems. The whole designing process of the approach is discussed systematically through a case study of Online Auction System in e-business domain.

1. Introduction

In distributed environments, large-scale applications run in an open context, several concerns must be considered and separated, including fault tolerance, data consistency, remote version update, runtime maintenance, etc.These concerns always are some crosscutting concerns that affect several units of an application,if not appropriately separated and modularized,leading to implementations where the code is said to be tangled and scattered. To regain these properties,Aspect oriented software development (AOSD) [1] is emerging as an important new approach to software engineering.Until now, work in aspects has been fairly limited to the implementation phase of software development. However, few works have tried to generalize the concept and apply it to the earlier phases of the software life cycle, in particular, the architectural design level. Nowadays, Software Architecture (SA) [2] has become an area of intense research in the software engineering communities,which actually set the early design decisions and have a large impact on the whole system. However,conventional software modeling approaches lack abstractions to support the modular representation of crosscutting features and behaviors in SA design stage. These crosscutting features and behaviors cut across the boundaries of other modules, leading to high coupling between components and connectors, and complex description of these modular units. Obviously, coping with aspects at the Software Architecture phases as such is a primary issue. The expected benefits are improved comprehensibility, ease of evolution and increased potential for reuse in the development of complex software systems.

Aiming to address above those problems, we propose a new Aspect-Oriented Architecture Design Approach AOSAD. This approach introduces a new kind of component called Aspectual Component[3] to describe and encapsulate the crosscutting features and behaviors, and extends conventional connectors called Aspectual Connector[4]to model the crosscutting interaction among architectural elements.In addition, AOSAD defines architectural joinpoint to provide assistance for the composition between the components and aspectual components.We take advantage of this approach to design architecture of the Online Auction System (OAS)[5].From the requirements of OAS, we specify and model as separated architectural elements those non-crosscutting and crosscutting concerns. Then, we describe these separated architectural elements, using a new Aspect-Oriented architecture description language AC2-ADL,which provides corresponding formal support to describe an Aspect-Oriented SA. Finally, Section5-6 contains a general discussion and an assessment AOSAD and presents our conclusions and future work.

2. The Online Auction System

This section begins by a simplified description of this system, then analyzes the main crosscutting concerns of this system, and presents Aspect-oriented software architecture of OAS.

978-0-7695-3395-7/08 $25.00 © 2008 IEEE
DOI 10.1109/ICEBE.2008.41

2.1. Analyzing Crosscutting Concerns

The Online Auction System (OAS) allows people to negotiate over the buying and selling of goods in the form of English-style auctions. In addition, besides the given local market, the customers can be allowed to participant in the remote auction markets. These local market servers are connected to each other by the Internet or by a high-bandwidth private network, with a communication infrastructure that enables them to conduct B2B (business-to-business) transactions. Thus, a collection of interconnected local markets constitutes the global market. Based on Separation of Concerns principle, the main crosscutting concerns of the auction system are listed as follows:

- Distribution and scalability: OAS would be used by bidders from different parts of the world with diverse market procedures and customs, and it must be designed to cater for variations in market mechanisms.
- Concurrence and Parallel: OAS is highly concurrent :clients bidding against each other in parallel, and a client placing bids at different auctions and increasing his/her credit in parallel.
- Data Consistence and Persistence: Persistent data is usually constructed and shared by the auction market of users and distributed in deferent locations. Therefore a major issue is the preserve the consistency of data in the presence of concurrency and failure.
- Security and Anonymity: Security and anonymity are crucial in OAS design. Unlike a traditional auction, the participants in an online auction are not physically present. OAS presents many opportunities for cheating.
- Dependability and Availability: A server or a processor can fail, usually in various ways, and must be built reliable using internal redundancy so that the auction service remains available.

In the context of software architecture design, the behaviors and features corresponding to theses crosscutting concerns tangled with other concerns, as well as the behaviors or features which have to be scattered throughout the system to be able to achieve a global effect, always make more difficult to reason and act upon them.

2.2. OAS Software Architecture

In AOSAD, the traditional component and connectors are also reserved to capture the core functionality of system and their interactions. Fig.2.2-1 represents a partial description of software architecture of OAS with AOSAD, which includes the key modules of this system.

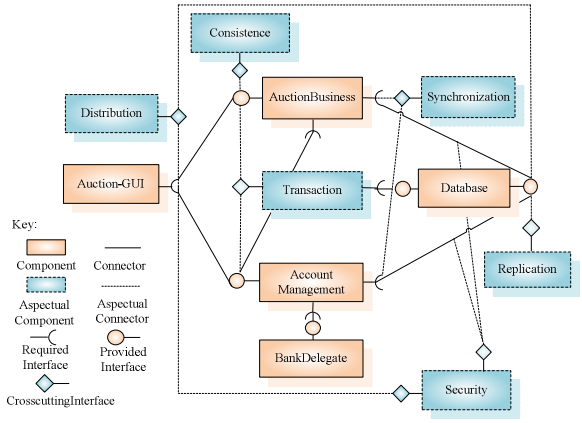


Fig2.2-1. Aspect-oriented Software Architecture of OAS

The *Auction-GUI* component represents the user graphical interfaces, which can interact with those components encapsulating the main business logic of an OAS, such as *AuctionBusiness* and *AccountManagement* components. *Database* components store the information manipulated by the system. Furthermore, *AccountManagement* and *BankDelegate* respectively deal with users' account and corresponding bank business.

On the other hand, aspectual components also play an important role in OAS. The *distribution* aspectual component treats with the remote location of the two principal components in the architecture (*AuctionBusiness* and *Database*). For example, when the guests, through the required interface from the GUI component, request the services from *AuctionBusiness* component, this required interface will be effected by a crosscutting interface from the *Distribution* aspectual component by means of an aspectual connector. *Transaction* aspectual component uses the transactional operations (begin_transaction, commit_transaction, and rollback_transaction) provided by the *transactionService* interface of the *Database* component, supporting the persistence crosscutting concerns to achieve placing bid action or deposit action. *Consistency* aspectual component is applied, during the interaction among the *Auction-GUI* and the main business components, every time OAS has to check the consistency of the received data. Additionally there is an aspect for synchronizing the bid, account, and goods object states with the corresponding database, when the software uses persistent data management, and/or with the corresponding remote server, when the software is distributed, in order to ensure consistency. In order to provide availability, the *Replication* aspectual component stores a copy of the data that OAS sends to

Database. *Security* aspectual component encapsulates the various security protocols to make secure the data sent over the Internet, and it also needs to guaranty users' validity. During implementation of those aspectual components, it was necessary to define auxiliary *Exception Handling* aspectual component which is left out here for the sake of simplicity.

In order for formal description of these architectural modules, the following section outlines the proposed Aspect-Oriented software architecture language AC2-ADL.

3. AC2-ADL

AC2-ADL, according to AOSAD, provides corresponding formal support to describe the Aspect-Oriented SA, which introduces two special kinds of units, named **AspectComponents** and **AspectConnectors** (so called AC2 for short) as the first-class citizenship. AC2-ADL supports the definition of three distinct facets of architecture:①the organization structure of a system together with its constituent parts. ② types and styles for defining classes and families of architecture. ③behavior and protocol description specifications for components and connecters.Fig.3-1shows a partial syntax of the AC2-ADL types specified in BNF[1].

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
```

Fig3-1. Abstract syntax of types

From the fig.3-1, we can see the AC2-ADL type system be divided into two different levels of abstraction: **baseType** and **abstractType**. **baseType** includes some simple basic types and complex data construct types. For example, **Any** can indicate any kind of simple types. While, **abstractType** expresses more abstract types where: **aspectComponent** and **component** which respectively indicate the aspectual components and general components; **aspectConnector** denotes the crosscutting interconnections, and **Connector** denotes the general interconnections; **interface** (including required/ provided interface and crosscutting interface) are basic interaction points to send and receive appropriate messages and data between a component and its environment. **Role** is a special kind of interface of connector. Each role may be **crosscuttingRoles** or **baseRoles** which are two different kinds of agents of participators in interactions*;* **architecture** denotes the structure of a system or a composite module, which specify components and connectors how to be composed together.

In the following section, we will present the concrete demonstrations about how to utilize thses abstract syntax of types to design an AO architecture of OAS and its constituent parts.

4. AOSAD

This section discusses respectively the design and specification of different elements of software architecture, through taking OAS as a case study.

4.1 Modeling Aspectual Component

The motivation to define Aspectual Component is to be able to modularize the behaviors and features considered to be characteristic of crosscutting. Hence, an Aspectual Component should provide explicit interfaces to describe the crosscutting function and support heterogeneous aspect components design. In our approach, a new interface type called crosscutting interface is defined as sets of computational delegations that characterize the crosscutting behaviors of Aspectual Component with respect to their crosscutting features. Moreover, an Aspectual Component can be a composite one which consists of many architectural elements, and provides multiple different crosscutting interfaces, according to various crosscutting features. Thus, a valid aspect component should have the following abilities:

- Each Aspectual Component can contain one or more than heterogeneous crosscutting interfaces to affect multiple different entities
- Crosscutting interface can directly affect modules' interfaces. That means a crosscutting interface can modify the contract prescribed by each affected interface, but can not affect the internal behaviors of a module, thereby protecting the module encapsulation from invasive changes
- Crosscutting interfaces have power to modify the structure of a module by mixing in new elements like attributes, methods and even interfaces to it
- In addition, crosscutting interfaces can also crosscut connectors/aspectual connectors. In other words, aspectual component can affect the interactions

[1]In BNF, "**::=**" means "is defined as", "|" means "or", "[**...**]" denotes an optional item and "{**...**}" a repetitive item. For simplicity, single character Terminals, such as a semicolon, are not surrounded by quotes. Spacing and indentation are used solely for readability.

between the interface and certain interface connected with it

We provide a partial syntax of the prototype of the aspectual component and interfaces in Fig4.1-1 and fig4.1-2.According as Fig4.1-1 An **aspectComponent** defines a set of interfaces (crosscutting and required), operations inside the interfaces, and methods of computation, internal computational process, data type definitions and design constraints, even sub architecture. If a component or an aspectual component contains some sub-architecture, 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. From a black-box perspective, only interfaces and their operations and the data passing through operations are observable. At the same time, from a white-box perspective, internal behaviors and structure are also observable. Thereby, the internal computational process, sub architectures, the mapping relationships and constrains of an aspect component are optional.

```
aspectComponent ::=
  aspectComponent component_name is {
    parameters parameters_name is simpleType
    methods methods
    crosscuttingInterface crosscuttingInterfaces
    requiredInterface requiredInterfaces
    [subArchtecture subArchtecture_name
                          is{ architeture|null;}]
    [mappingDeclaration {mapping_expression}| null;]
    [internalProcess is {{process_expression;}| null;}]
    [constrains is {{constrains_expression;}| null;}]
  }
```

Fig.4.1-1 A partial syntax of aspectComponent

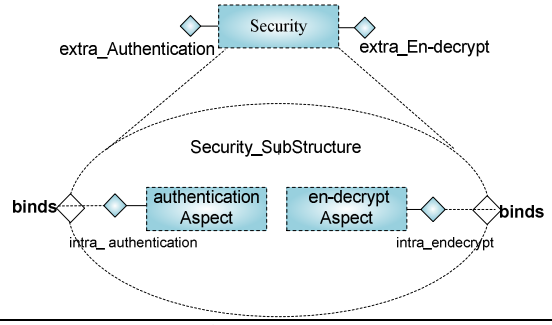
In the next, there are three kinds of interfaces of AC2-ADL specified in fig4.1-2. Crosscutting interface provides three kinds of operations: **add_Operation**, **replace_Operation** and **introduce_Operation**. Among them, **add/replace_Operation** is composed of a set of methods. While **introduce_Operation**, besides some methods, can include a set of fields and/or provided/required interfaces. These operations embody different crosscutting functions: Through **add_Operation,** a aspectual component can augment the target components/aspect components by adding the methods defined itself; while by using **replace_Operation,** a aspectual component can substitute the methods of the target components/aspect components with the ones defined in itself; finally, **introduce_Operation** can make an aspectual component mix own elements into the target components.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. Therefore, each method need specify the directions of its transmitting data. The kinds of the data transmission direction include incoming, outgoing or incoming-and-outgoing.

```
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 | in|out
```

Fig4.1-2 A partial syntax of crosscuttingInterfaces

Given the above syntax, we can model the elements of OAS in AC2-ADL. Here, we describe the structures of *SecurityAspect* aspectual component from Fig2.2-1,where the internal sub architecture *security_SubStructure* is transparent for external entities, but visible only for *SecurityAspect* itself. *security_SubStructure* defines two new types of aspect components: *authenticationAspect* and *en-decryptAspect*, and constructors their instances: *authentication_Instance* and *en-decrypt_Instance.* On one hand, *authenticationAspect* provides an *authentication* crosscutting interface, which, through the **add_Operation** operation, adds *verification* method into the target modules whose users need to be verified. On the other hand, *en-decryptAspect* provides another crosscutting interface *en-decrypt* whose **add_Operation** contains two methods: One is the *encoder* method, which receives the users' password to achieve the encryption feature and send the encrypted data to Internet. The other is *decoder* method, which receives the encrypted password from Internet to achieve decryption feature, and then deliveries this password to *Database* component. Finally, the keyword **mappingDeclaration** indicates that the internal interface *intra_authentication* from *authentication_Instance* and *intra_en-decryption* from *en-decrypt_Instance* are respectively bound to the external crosscutting interface *extra_authentication* and *extra_En-decryption*. For simplifying the descriptions, the internal computational behaviors and constraints have been left out here. AC2-ADL promotes designers to model an aspectual component in a more abstract manner in the earlier stage of software development. Meanwhile, these models provide powerful clues for mapping from design to

implementation.




```
aspectComponent SecurityAspect is {
    parameters username is String;
    parameters password is String;
    crosscuttingInterface extra_Authentication is {}
    crosscuttingInterface extra_En-decrypt is {}
    subArchtecture security_SubStructure is {
        aspectComponent authenticationAspect is {
            ...
            methods verification is in(username)
            CrosscuttingInterface intra_ authentication is {
                add_Operation verification }
            ...}
        aspectComponent en-decryptAspect is {
            ...
            methods encoder is in/out (password);
            methods decoder is in/out (password);
            CrosscuttingInterface intra_en-decrypt is {
                add_Operation encoder;
                              decoder }
            ...}
        aspectComponent_instances
            authentication_Instance is authenticationAspect;
            en-decrypt_Instance is en-decryptAspect}
    mappingDeclaration
        authentication_Instance. intra_authentication
                              binds extra_ authentication;
        en-decrypt_Instance. intra_en-decryption
                              binds extra_En-decryption
    ...}
```

Fig4.1-2. Model of SecurityAspect

4.2 Aspectual Connector

Inspired by the work of [3], as the first-class entity in software architecture, aspectual connector also has interfaces. The interface of connector is a set of roles. The type of each role may be **baseRole** or **crosscuttingRole**. They represent the participators who will join in the interaction described by this connector. **baseRole** type indicates a set of objects applied by some aspectual components, such as component instances, interfaces or connector instances, while **crosscuttingRole** type indicates a set of crosscutting objects, such as aspectual component instances or crosscutting interfaces. In addition, each role can contain one or multiple behaviors, which need to be followed by the actors who play this role. Note that: during designing traditional connectors with AOSAD, roles in traditional connectors only have one type that is **baseRole.**Factually, 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 specification which can represent the crosscutting relationships among the aspectual components and traditional ones. AC2-ADL provides four kinds of the crosscutting interaction protocols listed by the keywords: **before**, **after**, **around** and **introduce.** As shown in Fig.4.2-1, 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.

```
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;}
crosscuttingRole_name::=
  {Role_name;}|{Role_name behaviors behavior_names;}
crosscuttingProtocol_expressions::=
    crosscuttingRole_name. behavior_name
    crosscuttingProtocol_types
     baseRole_name. behavior_name
crosscuttingProtocol_types::=
    before | after |around| introduce
```

Fig.4.2-1 A partial syntax of the prototype of aspectConnector

Here,we also depict an aspectual connector named *SecurityConnector* as an illustrative example in Fig 4.2-2. This aspectual connector encapsulates the crosscutting interaction protocols between *SecurityAspect* and other components in OAS. The description of *SecurityConnector* can be presented in Figure 4.2-2, where two base roles *Login* and *Transmission* and two crosscutting roles *Authentication* and *Secrecy* are contained. Hereinto, each role declares the primary behaviors according to the corresponding interaction activities. For example, the crosscutting role *Authentication* defines a *verifyLogin* behavior for the verification active, while the crosscutting role *Secrecy* defines two behaviors: *encrypt* and *decrypt* for the encryption and decryption activities.After that, the specifications of interaction among the roles are described in keyword **crosscuttingProtocal**. The semantics of crosscutting interaction protocol types is similar to that of advice composition from AspectJ[7]. For example, the first

expression in **crosscuttingProtocal** means that the behavior for verifying users' validity form *Authentication* role will occur before the users log in the OAS. When some crosscutting interfaces play certain crosscutting role, the crosscutting interfaces will invoke different operations, in accordance with the different crosscutting protocols to affect the base roles' players. To put it more specifically, **add_Operation** acts on the **before** or **after** crosscutting protocols to add itself methods, While, **replace_Opreation** is used to substitute the target entities' methods on account of the **around** crosscutting protocols. Finally, **introduce_Opreation** is responsible for **introduce** crosscutting protocol.

```
aspectConnector SecurityConnector is {
    baseRole
      Login behaviors logging;
      Transmission;
    crosscuttingRole
       Authentication behaviors verifyLogin;
      Secrecy   behaviors encrypt, decrypt
    crosscuttingProtocals
        Authentication. verifyLogin before Login. logging;
         Secrecy.encrypt before Transmission;
         Secrecy. decrypt after Transmission;
}
```

Fig 4.2-2 Model of SecurityConnector

```
Architecture ::=
   architecture architecture_name is{
        components component_list
        component_instances component_instance_list
        aspectComponents aspectComponent_list
        aspectComponent_instances
                          aspectComponent_instance_list
        [connectors connector_list]
        [aspectConnectors aspectConnector_list]
        [connectors_instances connectors_instance_list]
        [aspectConnectors _instances
                          aspectConnectors_instance_list]
        [architectural_Configuration Configuration]
   }
component_list ::={component;}
component_instance_list ::=
      instance_name is component_name
      [with (parameter_instantiation)];
aspectComponent_list ::=
      {aspectComponent;}
aspectComponent_instance_list ::=
      aspectComponentInstance_name is
      aspectComponent_name
      [with (parameter_instantiation)];
connector_list ::={connector;}
aspectConnector_list ::= {aspectConnector_name;}
…
```

Fig4.3-1 A partial syntax of Architecture

4.3 Aspect-Oriented Software Architecture and Configuration

As an integrated architectural design approach, the system architectural design and configuration play an absolutely necessary role.

Architecture Specification

An Aspect-Oriented architectural specification comprises the elements: components, aspectual components, aspectual connectors, connectors and configurations. In addition, each architecture declares a list of the defined elements instances. Fig4.3-1 displays a partial syntax of the AC2-ADL architecture.

Join points Specification

Besides typical composition among components in architectural configurations, our approach must add more support to specify the compositions between components and aspect components. However, it is a challenging problem, unlike any AO implementation language such as AspectJ, Jposs-AOP, where the aspect coding segments can be woven into the core coding segments according to a set of join points defined inside the execution process, the composition between components and aspect components through aspect connectors is described in a higher abstract way rather than using the weaving mechanism at the implementation level. Thus, at the architecture level, the definition and involved concept of the join point is supposed to be more abstract.

Considering these issues, our approach explicitly defines the architectural join point: They are some places where the effect of aspects will occur. These places include: ①the component /aspect component instances, ②the interfaces from these instances, ③the operations from the interfaces , ④and even the connector instances. Thereby, four different kinds of the architectural join point are proposed to denote the semantic of join points. These architectural join points are respectively defined as followed by: *component_JP*, i*nterfaces_JP, operation_JP*, and *connector_JP*.

```
pcd ::=operation_JP | interfaces_JP
        | component_JP | connector_JP
        | (and pcd pcd) | (or pcd pcd) | (not pcd)
operation_JP::=component_instance_name.interfaces_name.
                    [method_name| *]
interfaces_JP::=component_instance_name.
                    [interface_name|  *]
component_JP::= component_instance_name|*
connector_JP::= connector_ instance_name|*
```

Fig4.3-2 A partial syntax of pcd

In real architectural configuration, a base role of an aspect connector may be played by several interfaces of possibly different components. These interfaces represent structural join points that may be affected by aspect components. Hence, an architectural pointcut designator (pcd) should be defined as a formula that specifies certain join point or the set of join points to which the crosscutting interface of an aspect component is applicable. In order to express the

architectural Quantification Mechanism [8], we introduce the operations **and**, **or** and **not,** as well as Wildcards such as '*' to concisely describe sets of join points to play the same base role. For example, all the operations from certain component can be formalized as: *component_instance_name.*.**. The grammar of pcd's is given in the Fig.4.3-2.

Configuration Specification

The architectural configuration, whose partial syntax is represented in Fig4.3-3, is defined by listing a set of connections to bind pcd to base roles of connector instances, and a set of connections to bind crosscutting interfaces to crosscutting roles. In addition, in order for more explicit and accurate expressions of the attachments between the different operations and the corresponding behaviors, AC2-ADL 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.

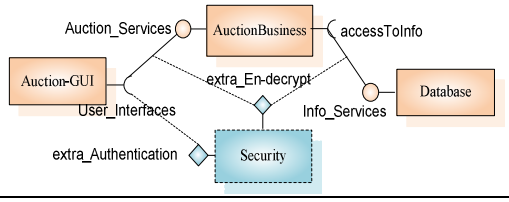
```
Configuration::=
   configuration is {
        ConfigurationBody| { ConfigurationBody; }|null }
ConfigurationBody::=
  pcd plays aspectConnector_Instance _name.baseRole_name
  | aspectComponent_Instance_name.
        crosscutting_Interfaces_name   plays
    aspectConnector_Instance_name. crosscuttingRole_name
    [attachment
      attachment_Expressions|
       { attachment_Expressions;}| null]
attachment_Expressions::=
    method_name attaches behavior_name
```

Fig4.3-3 A partial syntax of Configuration

Modeling OAS Software Architecture in AC2-ADL

For purposes of brevity and clarity, only a partial model of the OAS software architecture is given below in Figure4.3-4. This model mainly describes the configuration where the instances of components *Auction_GUI*, *Auction_Business* and *database* are affected by an instance of aspectual component *SecurityAspect* through an instance of aspect connector *SecurityConnector.* Furthermore, this architecture also defines some traditional connector which includes two roles *caller* and *callee* whose type both are **baseRole.**

In terms of the configuration, we can see: traditional connectors' instances c1 and c2 respectively connect *GUI_instances* with *Business_instances*, and *Business_instances* with *Database_instances.* Moreover these two connectors constituting a pcd, are affected by *SecurityAspect_instances* through the crosscutting interface *extra_En-decryption.* That means whenever, before *Business_instances* communicates with database_*instances* or *GUI_instances* communicates with *Business_instances,* the data sent into Internet need to be encrypted. While, after the data are received, it should be decrypted. In addition, *SecurityAspect_instances* also apply to the required interfaces *user_Interface* of *GUI_instances***.** Every time, users log on OAS by using the *users_logging* methods, they have to be verified in advance by the *verification* method from *SecurityAspect_instances*' another crosscutting interface *extra_authentication.*




```
architecture OAS is {
  component
    Auction_GUI;
    Auction_Business;…
  connector
    traditional_Connector is {baseRole caller; callee }…
  aspectComponent
     SecurityAspect;  …
  aspectConnector
     SecurityConnector ;…
  component_instances
    GUI_instances is Auction_GUI;  …
  aspectComponent_instances
    SecurityAspect_instances  is SecurityAspect;…
  aspectConnector_instances
    SecurityConnector_instances is SecurityConnector;…
  connector_instances
    c1   is   traditional_Connector;
   c2   is   traditional_Connector
  configuration is {
    GUI_instances.user_Interface plays c1. caller;
    Business_instances.Action_service plays c1. callee;
    Business_instances. accessToInfo plays c2. caller;
    Database_instances.Info_services plays c2. callee;
     Auction_GUI_instances.user_Interface plays
                    SecurityConnector_instances. Login
              attachment
                    users_logging attaches logging;
   c1 and c2 plays SecurityConnector. Transmission;
   SecurityAspect_instances.extra_authentication plays
                  SecurityConnector. Authentication ;
   SecurityAspect_instances.extra_En-decryption plays
                  SecurityConnector_instances. Secrecy
              attachment
                    encoder() attaches encrypt;
                    decoder() attaches decrypt; }
}
```

Figure4.3-4. AO Architecture Description

5. Related Work

Our work has been influenced by research in several areas: AO software design, AO software architecture and AO ADLs. Below we mainly look at related approaches along these dimensions.

Chavez [3] has proposed that crosscutting interfaces as a conceptual tool for dealing with the complexity of heterogeneous aspects at the design level. This kind of crosscutting interfaces has to expose extra information

relative to their crosscutting nature and, as a consequence, need to be extended. The crosscutting interfaces in our approach focuses on capturing the computational commitments associated with a given crosscutting feature but not crosscutting interactions. These crosscutting interaction relationships can be encapsulated by aspectual connectors, as results in independence between component and aspect, so the resulting aspects are more extensible and reusable in different contexts.In process of resolving the crosscutting interaction problems, our approach mainly refers to Batista's work [4], which proposes Aspectual Connector (AC) as the only necessary enhancement to an ADL ACME. However, because of an absence of specification to explicitly define the aspectual component, AspectualACME is unreadable to provide the convincing clues for designer and developers to analyze, design and code an application or a system with AOP techniques.The another representative work is CAM/DAOP [6]that defines DAOP-ADL,a component-and aspect-based language designed to be interpreted by DAOP, a dynamic component- and aspect-oriented platform. The DAOP-ADL language provides an XML Schema defining the structure of a DAOP application. Obviously, though the usage of XML facilitates the integration and interpretation of the information offered by the DAOP-ADL, it can not express the computational behaviors of components or aspects, resulting in difficulties in refinement and verification that whether the functions provided by components or aspects conform to their interface specifications.

6. Conclusion

In this paper the software architecture of the OAS has been modeled using our aspect-oriented software architecture design approach. The main contributions of this approach are three. First contribution is the definition of a systemic composition model, where crosscutting and non-crosscutting concerns are modeled by the different architectural blocks. Second contribution is the extension of the semantic of typical connectors with aspectual composition information. Last is the definition of a special kind of Aspect-Oriented Architectural description langue AC2-ADL, to enhance software architectural description.

The future research will pay more attention to design and develop an integrated aspect-oriented OAS with AOSD. In addition, we will enhance the expressive semantic capability, especially, formal description capability of elements' behaviors and interaction contracts, and even the overall system context.

7. Ackownledgment

We gratefully acknowledge supports from the National Natural Science Foundation of China (Grant No60773006), and the Doctoral Fund of Ministry of Education of China (Grant No. 20060486045).

8. References

[1]Araujo, J., Baniassad, E., Clements, P., Moreira, A. and Tekinerdogan, B. "Early Aspects: The Current Landscape, Technical note", *CMU/SEI and Lancaster University.* Feb 2005

[2]Medvidovic N, Taylor RN. "A classification and comparison framework for software architecture description language". *IEEE Transactions on Software Engineering*, 2000, 26(1):70~93.

[3] Thaís Batista, Christina Chavez, Alessandro Garcia, et al., "Aspectual Connectors: Supporting the Seamless Integration of Aspects and ADLs", *In Proc. of the ACM SIGSoft XX Brazilian Symposium on Software Engineering (SBES'06)*, Florianopolis, Brazil, Oct. 2006.

[4]Chavez, C., Garcia, A., Kulesza, U., Sant'Anna, C., Lucena, C. "Taming Heterogeneous Aspects with Crosscutting Interfaces". *Journal of the Brazilian Computer Society*, Jun 2006.

[5]P. Ezhichlevan and G. Morgan, "A dependable distributed auction system: Architecture and an implementation framework", *in proceedings of 5th International Symposium on Autonomous Decentralized Systems*, Dallas (TX), March 2001.

[6]M. Pinto, L. Fuentes, and J. M. Troya, "DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development," *presented at International Conference on GPCE*, Erfurt, Germany, 2003.

[7]Gregor Kiczales, Erik Hilsdale, Jim Hugumin, et al, "An Overview of AspectJ", *In Proc. of the 15th European Conference on Object Oriented Programming (ECOOP 2001)*, Springer-Verlag, 2001, pp. 327-353.

[8]Robert E. Filman, Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000.