# Modeling Diverse and Complex Interactions Enabled by Middleware as Connectors in Software Architectures

Yali ZHU, Gang HUANG*, Hong MEI
*School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China.*
*E-mail: zhyl@sei.pku.edu.cn, huanggang@sei.pku.edu.cn , meih@pku.edu.cn*

## Abstract

*Middleware enables distributed components to interact with each others in diverse and complex manners. Such interactions should be modeled at architecture level for controlling the complexity of incorporating middleware into the target system. This paper extends a traditional architectural description language for describing the diverse and complex interactions enabled by middleware as complex connectors and constraints on them in a model driven process. Such functions and qualities of connectors that satisfy the requirements of the target system are modeled without any consideration of middleware at first. Then the connectors and constraints on them are refined by the characteristics induced by middleware. All information of connectors produced in the two-step process can be described at three levels, including the connection, coordination and context. The language and process are illustrated and evaluated by applying them into J2EE (Java 2 Platform Enterprise Edition) applications.*

## 1. Introduction

Recognized as an important and practical approach for reducing the complexity and cost of the development and evolution of complex software systems, component based software engineering (CBSE) is receiving more and more attention from the industrial and academic communities. In CBSE, there are two main research fields – middleware and software architecture. Middleware, that resolves heterogeneity and facilitates communication and coordination of distributed components [26], acts as the runtime infrastructure for components. Software architecture, that describes the gross organization of the system as a collection of interacting components [4], acts as a blueprint for guiding the composition of prefabricated components [11][26]. Traditionally, software architectures look middleware as a set of, such as object oriented or event based, message passing primitives [19]. However, such way becomes difficult and inefficient to guide the component composition because of the proliferation of middleware today.

Firstly, the major functionalities of middleware (i.e., to support interoperability among components), are proliferating for meeting diverse requirements of multiple application domains. Besides the popular and matured interoperability protocols supporting intranet distributions like IIOP (Internet Inter-ORB Protocol) [14] and JRMP (Java Remote Method Protocol) [22], there are also emerging some interoperability protocols for internet distributions like SOAP (Simple Object Access Protocol) [24], WSCI (Web Service Choreography Interface) [25] and BPEL (Business Process Execution language) [2]. Secondly, some quality related considerations are also incorporated in the middleware layer. For example, IIOP provides the functions of client authentications, delegations and privileges to overcome the deficiencies of transport layer via SSL [14]. Thirdly, middleware changes the actual structures and behavior of the target system. For example, complex interactions among components may be traditionally be implemented by the codes scattered in the interacting components. But now, such complex interactions can be implemented as BPEL processes which are isolated from the interacting components and executed by an independent BPEL engine. Naturally, the composition of components has to cope with the changes of the structure of the target system.

The above discussion reveals that modern middleware puts much more and usually important impacts on the development and evolution of distributed software systems. Particularly, middleware enables distributed

* Corresponding author

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 IEEE

components to interact with each others in diverse and complex manners. However, such interactions enabled by middleware are usually represented as message passing primitives or simple connectors in traditional architecture description languages. As a result, the architects cannot record, analyze and evaluate the impacts of the interactions enabled by middleware on software architectures and then cannot directly and efficiently control the complexity of incorporating middleware into the target system.

This paper tries to investigate and deal with the impact of middleware on software architecture at the design phase using the notation of connectors, that is, the interoperability functions of middleware are represented as complex connectors and constraints on them. Firstly, we identify five characteristics of connectors induced by middleware. Then, we extend an architectural description language, called ABC/ADL [10], to describe connectors with these characteristics at three levels. In order to ease the work of modeling such complex connectors, we define platform independent architecture models and platform specific architecture models, both of which are similar with Platform Independent Model (PIM) and Platform Specific Model (PSM) in MDA (Model Driven Architecture) [16]. In the platform independent architecture model, connectors are designed in terms of the functionalities and qualities of system requirements. In the platform specific architecture model, the details of middleware, such as the underlying interoperability protocols, their support for the non-functional requirements, etc., are considered to make the connectors consistent with the target system. To demonstrate the solution, we apply the language and process into J2EE applications.

The rest of the paper is organized as follows. Section 2 discusses the characteristics of connectors induced by middleware and related work. Section 3 introduces a typical J2EE application which will serve as the case study in the following sections. Section 4 describes the two-step process for modeling the connectors. Section 5 presents the extended architectural description language. Section 6 summarizes the contributions and identifies the future work.

## 2. Connectors Induced by Middleware

### 2.1. Characteristics of Connectors Induced by Middleware

After a thorough and careful investigation on middleware, five representative characteristics of connectors induced by middleware can be identified as follows.

- Diversity of Interoperability Protocols: The core of middleware is the interoperability protocol. Typically, CORBA has IIOP, RMI has JRMP and COM has RPC. Recently, web services define SOAP to support the interactions across Internet. The diversity of interoperability protocols brings us flexibility as well as complexity. For instance, an EJB (Enterprise Java Bean) can be directly invoked through IIOP or JRMP to interact with CORBA components or other Java-based components within the boundary of enterprise, or be released as a web service to be indirectly invoked through SOAP which can penetrate the firewalls. At the same time, the proliferations of them also promote the establishment of a common glossary among software developers.
- Enhancement of interoperability protocols: Besides the functionality of facilitating communication among distributed components, middleware has also incorporated some application-level non-functional aspects in the interoperability protocols. For examples, an extension of SOAP messaging framework is “SOAP feature” including reliability, security, correlation, routing, and message exchange patterns etc. [26].
- Pub/Sub among multiple components: A Pub/Sub system is the event-based system that establishes implicit connections between the publishers who produce topics and subscribers who register their interests in the same topics so that both of them are not aware of the connections. CORBA event service [17] and JMS [24] are the typical middleware services supporting Pub/Sub in the Internet-based setting, providing the common vocabulary among components that solves the mismatches due to assumption conflicts [5].
- Choreography between two components: Choreography means that a component works well only when its services are invoked in a given order. The business logic contained in some components, i.e. booking airline ticket before making hotel reservation, makes the connection an inherent choreography to satisfy the temporal dependencies among activities. WSCI describes the behavior of a web service in terms of choreographed activities in the context of different message exchanges.
- Execution flow among multiple components: Execution flow specifies the potential execution order of operations from a collection of components. From the perspective of SA, it handles the coordination between multiple components. Typically, BPEL allows specifying

business process and identifying required web services in the process.

The above characteristics have great influences on the design of the target system including the structuring elements as components, connectors, and topology i.e. Pub/Sub among multiple components, the behavioral aspect of the system i.e. control flow among components by execution flow among multiple components, as well as the quality attributes of the system i.e. security, modifiability and flexibility by enhanced interoperability protocols. In order to shorten the gap between architecture based design and implementation as well as to make proper and sufficient use of current middleware technologies, middleware should be incorporated into earlier design phase instead of just considered as pure message passing primitives.

## 2.2. Existing Approaches to Modeling Connectors Induced by Middleware

Several existing Architectural description languages are studied and discussed on their support for middleware-induced styles in [13]. As for connector, the author argued that the definition is associated with a too restrictive semantics. Though it can be described as components, the architectural definitions are more readable and clear when the special purpose of these architectural elements for component interoperability is made explicit [13]. The explicit definition can also drive middleware into early consideration which may in return save the labor the developer due to the powerful capabilities supported by middleware infrastructure.

Though different middleware infrastructures have their own features, but the five characteristics mentioned above cover the most important aspects of connector including its main content (protocol), non-functional requirement, its effect on topology of the system, its relationship with components and its own behaviors. So to incorporate the middleware related information for later composition and deployment, connectors in architecting should have the following capabilities: explicit description or identification of interoperability protocols; support for non-functional properties provided by enhanced protocols; ability to model implicit, event-based connection; record of choreography requirement of the components for further validation of the connection; specification for temporal dependences among connected components.

According to these requirements, we examine the existing ADLs, including Wright [1], Armani [19], UinCon [22], C2 [10], Rapide [15], and Darwin [9] – with the purpose to investigate their modeling abilities to middleware-based systems and the result is shown in Table 1. We can see that since these ADLs matured before middleware proliferation, none of them fully support all the characteristics of connectors induced by middleware. Firstly, none of them takes the interoperability protocols explicitly in the definition of connectors. ADL like UniCon that supports predefined connector types has not covered all the functionalities provided by current interoperability protocols. While ADL like Wright that defines connectors using formal method can only describe interoperability protocols in such a perplexing way while specific interoperability protocol by itself contains certain syntactic and semantic information which is shared by software developers without ambiguity. Secondly, only UniCon and Armani partially describe non-function properties in their connector definition counting on the support of certain underlying environments. Thirdly, event-based connection is supported by the most ADLs except UniCon. But such connection is indirectly addressed with the different semantics. Fourthly, ADLs equipped with formalization tools can record the choreography

**Table 1 ADL support for middleware-induced connectors**

| | Wright | Armani | UniCon | C2 | Rapide | Darwin |
|---|---|---|---|---|---|---|
| explicit description of interoperability protocol | none | none | none | none | none | none |
| non-functional properties for interoperability protocols | none | property and invariant | attribute list | none | none | None |
| implicit, event-based connection | support by CSP | support | none | potential support by C2 style | support | Support |
| choreography requirements of components | support by CSP | property in connector using Wright | none | partial support by message filters | support by POSET | support by π-calculus |
| temporal dependences among components | The glue process | property in connector using Wright | none | none | none | none |

requirements of the components, but lack the related information in the connector definition except in Wright where the role in connector and player in component are modeled in the same way using CSP. Lastly, only Wright has the notation *glue* for the temporal dependence for the workflow among components. So in order to better support the architecting of middleware-based system, the description for connector must be modified or extended in ADL.

## 3. The Sample of J2EE Application

As regarded as the technical environment which will influence and actually change the software engineering culture for the first decade of the twenty-first century [1], J2EE is selected as the typical middleware for our study on middleware-induced connectors. In this section, we will first examine a typical J2EE application server – PKUAS to have a concrete image of J2EE technologies, and then introduce a J2EE-based application as the case study for the paper.

### 3.1. A Typical J2EE Application Server - PKUAS

PKUAS (PeKing University Application Server) [13] is a J2EE application server, which is the platform including J2SE (Java 2 Standard Edition), common services and one or both of Web Container and EJB Container. In general, a system based on PKUAS has typical 3-tier architecture, as shown in Figure 1.

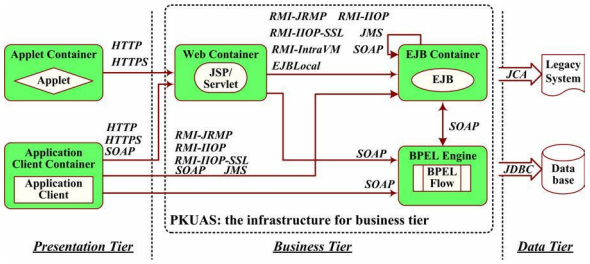


**Figure 1. Interoperability mechanisms in PKUAS**

Via its customizable and extensible interoperability framework [5], PKUAS provides many interoperability protocols which have all characteristics of the connectors induced by middleware. In details, PKUAS provides Internet and Web protocols include TCP/IP, HTTP, HTTPS (HTTP over SSL) and SOAP and Intranet protocols including IIOP, RMI-IIOP, RMI-IIOP-SSL and RMI-JRMP. To optimize the interactions among the components collocated in the same virtual machine, two private protocols, EJBLocal (all parameters are passed by references without marshalling) and RMI-IntraVM (all parameters except interfaces are passed by values without marshalling) are provided. JMS supports Pub/Sub interactions. JDBC supports the interactions with the backend databases. JCA (Java Connector Architecture) enables J2EE components to interact with some special enterprise applications, like ERP (Enterprise Resource Planning) and CRM (Customer Relationship Management).

### 3.2. A Typical J2EE application – Java Pet Store

The Java Pet Store (JPS) is a sample application by the Java Blue Prints program at Java Software, Sun Microsystems. It is a typical e-commerce application: an online pet store enterprise that sells animals to customers.

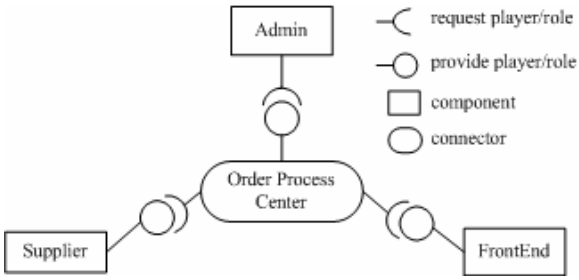


**Figure 2. Configuration view of JPS**

As shown in Figure 2, JPS can be divided into four main parts – FrontEnd, OPC (Order Process Center), Admin and Supplier – that asynchronously communicate. The OPC controls the main workflow of the whole application. In detail, it receives orders from the FrontEnd, if the total price is more than 500 dollars, then it send the unapproved order to the Admin for validation, and wait until receiving the approved order from Admin; otherwise it approves the order directly. Once the order is approved, it is sent to the Supplier to ship the products for the customer. And the OPC is waiting for the invoices from the Supplier. In the problem space, OPC is treated as a complex connector among the other three components and will be modeled step by stop in the rest of paper.

## 4. Modeling Connector Induced by Middleware

### 4.1. Overview of ABC

The architecture-based component composition

(ABC) approach employs SA descriptions as frameworks to develop components as well as blueprints for constructing systems, while using middleware as the runtime scaffold for component composition [12]. In ABC approach, SA runs through the whole life cycle of software applications including requirements analysis, architecting, composition, deployment, maintenance and evolution [5]. ABC also provides a set of supporting tools for different engineering phases, including an architecture description language called ABC/ADL, a deployment and runtime environment PKUAS, etc.

ABC takes views as the mechanism for separating concerns, and as for connector, we draw three views – type, configuration, and flow view. The *Type* view concerns the types of the sub-components and sub-connectors that directly take part in the fulfillment of providing services with the intention to draw the developers' attention on current developing stage and effectively support stage-by-stage refinement and creation. The *Configuration* view pictures the structural dependencies among sub-elements presented in the type view with the aim for configuring a runnable connector and providing a possible plan for further deployment. The *Type* and *Configuration* view exist for complex connectors that need refinement. The *Flow* view is to express the execution flow among roles in the connector for the sake of coordination, is checked for valid against the choreography requirements of participants, and is optional if there is no coordination requirements.

Connectors are treated as the first-class entities as components in problem space, so explicit connection specifications should be documented in the requirement specification. Connector in problem space will be refined and implemented in design phase [12]. In this section, we will illustrate the modeling process of connector using OPC as the example. The modeling process of connectors is centered on the creation of the three views in two steps – platform independent model and platform dependent model.

## 4.2. Modeling Views in Platform Independent Models

The modeling of connector as well as the whole system begins with the modeling of the three views. Following we will take the OPC as the example to illustrate the modeling process of connector.

First of all, we must determine whether OPC is a complex connector or a primitive one. If it is a primitive one that means it is totally supported by the underlying infrastructure, only the flow view may be defined if it has the coordination responsibility. Otherwise, the other two views should also be defined to support refinement. Obviously, OPC has the responsibility to connect the other three components and ensure the temporal relationship between them according to the business logic. If the other three components are released as web services, then OPC can be seen as a primitive connector supported by some work flow engine in the middleware infrastructure. All the architect should do is to describe the work flow in certain work flow language like BPEL or WSFL etc. The details associated with the concrete control and invocation of certain activities is shield in the primitive connector. The flow can be derived from the following steps. Firstly, we analyze the parties that participate in the interaction mediated by the connector. There are five parties in OPC – one for sending order from the front end named *PurchaseOrderSender*, one for receiving the unapproved order for validation named *Administration*, one for sending the approved order from Admin named *OrderApprovalSender*, one for receiving the order from OPC to ship animals to customers named *SupplyApprove,* and one for sending invoices to the OPC named *InvoiceSender.* These parties are modeled as roles in ABC indicating that some components play the role in the interaction. As described in section 3.2, the *Flow* view of OPC is shown in Figure 3.

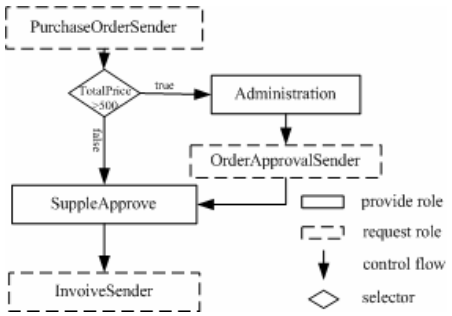


**Figure 3 Flow view of OPC**

If the other three components are not released as web services, then the OPC must be a complex connector that needs refinement. Then the three views must all be decided in the development. Firstly, the five roles mentioned above is recorded in the *Type* and *Configuration* views and can be seen as the start point of the modeling process and must be mapped to the roles of primitive connectors for implementations or be refined in the lower level. So now we must refine our design by determining the sub elements of the complex connector in the configuration view as shown in **Figure 4**. Two kinds of primitive connectors – *RemoteRrocedureCall* with two roles *caller* and *callee* and *Message* with two roles *sender* and *receiver* are employed as sub-connectors to fulfill the concrete task of linking the outer roles. In detail, *InvoiceSende*r,

*PurchaseOrderSender*, and *OrderApprovalSender* are mapped to the *sender* role of *Message*, while *SupplyApprove* is mapped to *receiver* role of *Message*, and *Administration* is mapped to *callee* role of *RemoteProcedureCall*. The roles of the primitive connector must be connected with the players (the functional parts in the component). Driven by that principle, we can get more components in the *Configuration* view until it is valid. By valid we mean that the outer role of complex connector is mapped and there is no pending roles or players of the sub-elements. Eventually, we get the *Configuration* view of OPC in Figure 4, while the type view is omitted here.

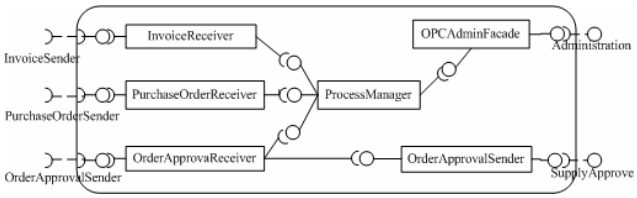


**Figure 4. Configuration view of OPC**

As shown in Figure 4, *InvoiceReceiver*, *PurchaseOrderReceiver*, *OrderApprovalReceiver* are to receive message from outside the connector, *OrderApprovalSender* is for sending the order to the supplier. The admin can directly manipulate the OPC through the *OPCAdminFacade*. All the activities in the OPC are managed by a workflow manager – *ProcessManager*, which records the status of current processing order and controls the temporal order of the five roles. By far, the structural aspect of the connector is achieved, and then the behavioral aspect of the connector should be considered.

Finally, no matter it is the primitive connector or the complex one, all non-functional requirements derived in the analysis are recorded and will be dealt with in the platform specific architecture model. In OPC, one of the main non-functional requirements is security, which required the message exchanging between OPC and other parts of the system to satisfy the integrity and confidentiality requirements. These requirements must be supported by underlying platform for primitive connector, implemented by the specific component, or be implicit achieved by the certain architectural solutions i.e. topology for complex connector.

### 4.3. Refinement in Platform Specific Model

After the platform independent architecture model is achieved, the logic entities in the model should be implemented by selecting, qualifying and adapting components, connectors and constraints in the reusable assets repositories. In other words, the designed architecture model is refined with the details of the implementation which is closely coupled with middleware. Typically, the underlying interoperability protocol for the primitive connectors should be selected in this phase. Assume that there are two logical components interacting with each other. If one is implemented by an EJB and another is implemented by a CORBA object, the connector between them should take IIOP as the underlying interoperability protocol. More complex, assume that both are implemented by EJBs, the underlying interoperability protocol can be RMI-IIOP if they are distributed or be EJBLocal if they are collocated in the same host.

Here for OPC, BPEL can be selected as the interoperability protocol if it is the primitive one. In the second situation, we can select RMI-IIOP as the interoperability protocol for *RemoteProcedureCall* and JMS for *Message*. The inner components are also mapped to corresponding implementations i.e. the *PurchaseOrderReceiver* is mapped to a Message Driven Bean in the sample application.

At the same time, non-functional properties of connectors should be enforced by utilizing the mechanism provided by middleware. For examples, both RMI-IIOP-SSL and HTTPS can transfer messages in a secure way. If the transaction property of an EJB is *Mandatory*, *Required* or *Supported* [25], the connectors associated with the EJB should be able to transfer transaction contexts. If an EJB defines the method permission [25], its connectors have to be able to transfer security contexts.

Here for OPC, in order to satisfy the security requirements identified above, the transport-layer encryption by SSL is required and then modeled as the properties for the connector *RemoteProcedureCall* as we can see in section 5.2.3.

## 5. The Extension of Description for Connectors in ABC/ADL

ABC/ADL, as the basic tool for ABC (Architecture-Based Component Composition) method [10], is defined to support component composition. It has the ability not only to describe the system structure but also to help refinement and creation of software system and support automatic composition and verification. As a modularly extensible language, XML is used in ABC/ADL to support an extensible framework.

## 5.1. Three levels of Connector Model

The connector model in ABC concerns three levels to cover all currently known circumstantialities induced by middleware.

The connection level is the basic level that focuses on the connecting aspect of the connector, including the explicit description of interoperability protocols, the syntactic description of its interior structure in case of complex connection and the connection points with the components. In essence, it addresses the structural aspect of the connector.

The coordination level deals with the coordinating functionality description of the connector based on connection, including satisfaction of choreography requirements of components and the execution flow of the connected multiple components. From other perspective, it captures the behavioral aspect of the connector.

The context level pays attention to the desired environmental requirements for the connector to fulfill the connection and coordination functionalities. Typically, non-functional requirements as load balance, security, and response time etc., which are probably supported by middleware infrastructures are taken into consideration. In other words, it is aimed to guarantee the non-functional requirements of the connector.

**Table 2. Relationship between ABC connector model and connectors induced by middleware**

| | Connection (structure) | Coordination (behavior) | Context (non-functional quality) |
|---|---|---|---|
| diversity of interoperability protocol | + | | |
| enhanced interoperability protocols | + | | + |
| Pub/Sub among multiple components | + | | |
| choreography between two components | | + | |
| execution flow among multiple components | | + | |

Table 2 shows the relationship between the three levels of ABC connector model and the five characteristics of connectors induced by middleware mentioned in section 2.

## 5.2. Definition of Connectors

All the information derived in the modeling process is recorded in ABC/ADL. The definition of connector shown in Figure 5 has two parts – *VisiblePart* and *InvisiblePart*. The *VisiblePart* describes the elements that visible to the accomplishment of the functionalities provided by the connector, including *Role*, *Protocol* and *Property*. The *InvisiblePart* describes the *InnerArchitecture* of the complex connector with the *Mapping* from inner roles to outer ones to support refinement.

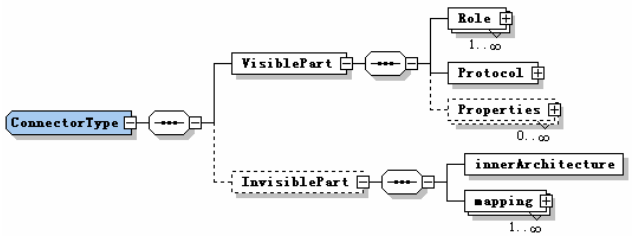


**Figure 5. Definition of connector in ABC/ADL**

**5.2.1 Role.** It defines the participant in the interaction, connected with the players of the components.

```
<Role name="PurchaseOrderSender" attribute="request">
 <Message>
   <sender>PurchaseOrderSender</sender>
   <receiver>OPC</receiver>
   <sendingTime/> <receivingTime/>
   <Priority>high</Priority>
   <MessageContent type="PurchaseOrder"/> </Message>
 <Property>
    <name>resource-env-ref-type</name>
    <value>javax.jms.Queue</value>
    <name>SSL</name> <value>required</value>
 </Property>
</Role>
```

**Figure 6. One of the roles in OPC**

Figure 6 shows the *PurchaseOrderSender* role. It is assigned the attribute *request* to identify the role of the connected player. The main part of the role is the description of the message sending from the *PurchaseOrderSender* to OPC, with the content type of *PurchaseOrder*.

**5.2.2 Protocol.** It describes the communication protocol implemented by the underlying language, operating system or middleware like JRMP, IIOP or SOAP etc. in the *predefined* field of the primitive connector. But for complex connector, it must describe the *userdefined* protocol with the detail shown in Figure 7.

The *description* is aim to model the control flow in the connector, which supports all the basic control patterns – sequence, parallel split, synchronization, exclusive choice, and simple merge [28] while maintains a relative simple definition. A *unit* corresponds to an independent role in the connector definition, which represent the simple activity in the flow; the *entry* describes the parallel execution of certain activities with the parallel split as the start point and synchronization as the end point; *predecessor* and *successor* are the pair to

model the consecutive steps in the flow; *if* shows the conditional execution of certain activities, *temporal4T* stands for the activities chosen in case the expression in the *if* block is satisfied, and *temporal4F* otherwise; *case* and *doWhen* extends the former one by adding more choices.

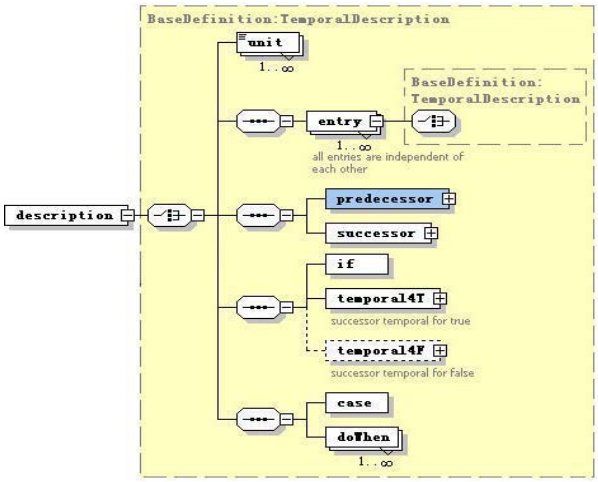


**Figure 7 User-defined protocol for connector**

```
<Protocol> <userdefined> <description>
 <predecessor>PurchaseOrderSender</predecessor>
 <successor>
  <predecessor>
   <if> <expression>
    <predicate xsi:type= “More”>
     <Parameter>TotalPrice</Parameter>
      <value>500</value>
      <unit>dollar</unit>
    </predicate>
   </expression> </if>
   <Temporal4T>
    <predecessor>Administration</predecessor>
    <successor>OrderApprovalSender</successor>
   </Temporal4T>
  </predecessor>
  <successor>
   <predecessor> SuppleApproval</predecessor>
   <successor> InvoiceSender </successor>
  </successor>
 </successor>
</description> </userdefined> </Protocol>
```

**Figure 8. User-defined protocol in OPC**

Figure 8 shows the userdefined protocol used by OPC to coordinate interaction among the connected components and it formally records the workflow in Figure 3 . The protocol shows that *PurchaseOrder* is the start point of the whole process. If the total price is more than 500 dollars, the order is sent to the *Administration* for validation until receiving the approved order from *OrderApprovalSender*. Otherwise, the order is directly approved by the OPC. Then, the approved order is sent to the *SupplyApproval* and OPC will wait until the *InvoiceSender* send the invoices.

**5.2.3 Property.** It shows the properties the connector should have, including non-functional properties i.e. load balance and security context etc. supported by underlying platforms with the *Parameter*s as the measurement for such requirements. Each Property is associated with an *Objective*, which stands for the non-functional requirement of the connector that causing the existence of the property.

Quality attribute scenarios are used in ABC method to specify non-functional requirement. A quality attribute scenario is a quality-attribute-specific requirement [3]. It uses six elements to identify a specific scenario: a *source* (some entity outside the system) generated a *stimulus* (a condition need to be considered when it arrives at a system) to some *artifact* (the stimulated artifact in the system) in a specific *environment* (the condition of the system when the stimulus occurs) and the artifact *responses* (the activity undertaken after the arrival of the stimulus) to the stimulus by some *measure* (the response should be measurable in some fashion so that the requirement can be tested).

The addition of *Objective* is to remain traceability of the non-functional requirement to certain properties of the architecture elements. An *Objective* associated with a complex connector can be associated with the Properties of its sub-elements (sub-connectors or sub-components) or the Property of sub-architecture as its topology or behavior. All the information can be recorded through the definition of Property with the same *Objective*. It can be seen as the rationale of the design that supports the verification and reuse of the architecture. It can also be used as the runtime detection for violation of certain quality requirement.

A security scenario in OPC is as follows: when a customer (source) requests an order from the FrontEnd (stimulus) in normal operation (environment), the OPC (artifact) should guarantee that the content of the order (response) are remain integrity and confidential (measure).

As shown in Figure 9, the OPC has the security requirements, that is, its four roles – *PurchaseOrderSender*, *InvoiveSender*, *OrderApproval-Sender*, and Administration must ensure the integrity and confidentiality. In the sample application, it is implemented by the SSL transport supported by the underlying application server as shown in Figure 10.

```
<Property name="Security">
 <Objective>
  <source> customer</source>
  <stimulus> invocation</stimulus>
  <environment> normal</environment>
  <measure>
    <response>integrity</respose>
    <value>required</value>
    <response>confidentiality</response>
    <value>requied</value>
  </measure> </Objective>
 <Parameter name="integrity"
              type="Choice">Required</Parameter>
 <Parameter name="confidentiality"
              type="Choice">Required </Parameter>
 <RelatedRole>PurchaseOrderSender</RelatedRole>
 <RelatedRole>InvoiceSender</RelatedRole>
 <RelatedRole>OrderApprovalSender</RelatedRole>
 <RelatedRole>Administration</RelatedRole>
</Property>
```

**Figure 9. Property of OPC**

```
<connector name="RemoteProcedureCall">
 <Role>Caller</Role> <Role>Callee</Role>
 <Protocol>RMI-IIOP</Protocol>
 <Property>
  <Objective>…</Objective>
  <Parameter name="SSL" type="Choice">Required</Parameter>
  <RelatedRole>Caller</RelatedRole〉
  <RelatedRole>Callee</RelatedRole>
 </Property>
</connector>
```

**Figure 10. ADL definition for RemoteProcedureCall**

**5.2.4 InnerArchitecture and Mapping.** They are the reference to the architecture anchor with the unique name and the corresponding relationship of inner and outer roles respectively.

```
<InnerArchitecture>OrderProcessCenter</InnerArchitecture>
<Mapping>
     <outerrole>OrderApprovalSender</outerrole>
     <innerrole> <connector>Message</connector>
          <role>Sender</role>
     </innerrole>
     …
</Mapping>
```

**Figure 11. InvisiblePart of OPC**

Figure 11 indicates that the inner architecture of OPC is defined by another architecture with the unique name of *OrderProceeCenter*, and one of the five roles – *OrderApprovalSender* is mapping to the *Sender* role of the primitive connector – *Message* in that architecture.

In summary, Table 3 presents the relation of the language elements in ABC/ADL with the three levels of the connector.

**Table 3. Relationship between connector model and description**

| | role | protocol | | property |
|---|---|---|---|---|
| | | predefined | User-defined | |
| connection | + | + | | |
| coordination | + | | + | |
| context | | | | + |

## 6. Conclusion and Future Work

In [5], Garlan points out that the world of software development and the context in which software is being used are changing in significant ways, and these changes promise to have a major impact on how architecture is practiced. The use of middleware is highlighted by Emmerich that it is not transparent for system design and that the issue should be addressed by existing design methods [4]. In industrial community, OMG proposes MDA to explicitly deal with technical details specific to middleware via the concept of Platform Independent Model and Platform Specific Model [18]. But as Nitto mentioned in [13] that the top-down approach adopted by the software architecture community in the development of languages and tools seems in many ways to ignore the results that practitioners have achieved (in a bottom-up way) in the definition of middleware. Enlightened by that, we argue that the involvement of middleware related concerns in the architectural design will make proper tradeoffs in early stages of development.

In this paper, we provided a preliminary solution to model diverse and complex interactions enabled by middleware using the notion connector. Resemble to MDA, the modeling process includes platform independent architecture and platform dependent architecture design. At first, the functionalities and quality requirements of the connectors are modeled without consideration of middleware. Then such connector along with the constraint on them is refined using the characteristics supported by middleware. To record such modeling process and keep the traceability of target connector, an architecture description language named ABC/ADL is also extended to incorporate a three-level connector model including connection, coordination, and context. To demonstrate our approach, a J2EE application – Java Pet Store is also introduced with its core connector – Order Process Center (OPC) being modeled step by step.

Currently, our method works well on J2EE, one of the

most popular middleware based systems, and we plan to apply the method to other middleware, such as .NET. We will also do more case studies on realistic projects. On the other hand, we will apply the consideration of middleware in software architectures, which contains plentiful information of the target system, into the self-adaptation of middleware at runtime.

## Acknowledgements

This effort is partially sponsored by the National Key Basic Research and Development Program of China (973) under Grant No. 2002CB31200003; the National High-Tech Research and Development Plan of China (863) under Grant No. 2004AA112070; the National Natural Science Foundation of China under Grant No. 60125206, 60233010, 60403030, 90412011; and the IBM University Joint Study Program.

## References

[1] Allen R. J., "A Formal Approach to Software Architecture", Ph.D. Thesis, May 1997.

[2] Andrews T., F. Curbera etc, Business Process Execution Language, V1.1,May 2003.

[3] Bass L, P. Clements, R. Kazman *Software Architecture in Practice*, Addison Wesley Professional, 2003.

[4] Emmerich, W. "Software Engineering and Middleware: A Roadmap", ICSE ACM Press, 2000, pp. 117 - 129

[5] Garlan D., *Software Architecture: A Roadmap*, The Future of Software Engineering 2000, Proceedings of 22nd International Conference on Software Engineering, ACM Press, 2000, pp.91-101.

[6] Huang G., H. Mei and F.Q. YANG, "Runtime Software Architecture Based On Reflective Middleware". Science in China, Series F, 2004 47(5) pp.555-576.

[7] Huang G., H. Mei, Q.X. Wang, and F.Q. YANG, "A Systematic Approach to Composing Heterogeneous Components", Chinese Journal of Electronics, Vol. 12, No. 4, 2003, pp. 499-505.

[8] Luckham D.C., J. Vera, "An Event-Based Architectural Description Language", IEEE Transactions On Software Engineering, Vol.21, No.9, 1995, pp. 717-734.

[9] Magee J., N. Dulay, S. Eisenbach, and J. Kramer, "Specifying Distributed Software Architectures", in Proceedings of 5th European Software Engineering Conference, Spring Verlag, 1995, pp. 137-153.

[10] Medvidovic N., "ADLs and Dynamic Architecture Changes", Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, 1996, pp.24-27.

[11] Mei H., F. Chen, Q. Wang and Y. Feng. "ABC/ADL: An ADL Supporting Component Composition." ICFEM2002, LNCS 2495, 2002, pp. 38-47.

[12] Mei H., J.C. Chang, F.Q. Yang, "Composing Software Components at Architectural Level", In Proceedings of International Conference on Software-Theory and Practice, IFIP the 16th World Computer Congress, 2000, pp. 224-231.

[13] Mei, H. and G. Huang. PKUAS: An Architecture-based Reflective Component Operating Platform, invited paper, in Proceedings of 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS), 2004, pp. 163-169.

[14] Microsoft, Component Object Model, http://www.microsoft.com/com

[15] Nitto D., E. and Rosenblum, D.S. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In Proceedings of the 21st International Conference on Software Engineering, 1999, pp.13-22.

[16] OMG, Common Object Request Broker Architecture: Core Specification, v2.1, 2002.

[17] OMG, Event Service Specification, Version 1.1, 2001.

[18] OMG, MDA Guide, Version 1.0.1, 2003.

[19] Oreizy P., Nenad Medvidovic and Richard N. Taylor, "Architecture-Based Runtime Software Evolution", ICSE, 1998, pp. 177-186.

[20] Peltz C., Web Services Orchestration and choreography, Computer, Vol.36, No.10, 2003, pp.46-52.

[21] Shaw M. and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[22] Shaw M., R. DeLine, and D. V. Klein etc., "Abstractions for Software Architecture and Tools to Support Them", IEEE Transactions on Software Engineering, Vol. 21, No. 4, 1995, pp. 314 – 335.

[23] Sun Microsystems, Java Message Service, V1.1, 2002.

[24] Sun Microsystems, Java Remote Method Invocation Specification. http://java.sun.com/j2se,

[25] Sun Microsystems. Enterprise JavaBeans Specification, V1.1, Final Release. http://java.sun.com/j2ee, 1999.

[26] W3C, SOAP Version 1.2 Part 1: SOAP Messaging Framework.

[27] W3C, Web Service Choreography Interface, W3C Note 8 August 2002.

[28] Workflow Patterns, http://tmitwww.tm.tue.nl/research/patterns/index.htm.