An Adaptive Software Architecture Model Based on Component-Mismatches Detection and Elimination

Shan Tang, Xin Peng, Yiming Lau, Wenyun Zhao, Zhixiong Jiang
Computer Science and Engineering Department, Fudan University, Shanghai 200433, China
{tangshan, pengxin, 051021050, wyzhao, 051021049}@fudan.edu.cn

Abstract

Commercial-off-the-shelf components (COTS) are widely reused at present and black-box composition is the unique way to integrate them into the target system. However, various mismatches among components often hamper the integration of COTS. While the existing approaches to modeling software systems often neglect the issue of COTS component-mismatch detection and elimination. Aiming at solving this problem, this paper proposes an adaptive software architecture model. Based on this model, we first analyze and conclude the mismatches among heterogeneous components, and then we propose the corresponding solutions to eliminate these mismatches and provide a seamless integration for COTS components. At the same time, we employ an example of web-based application system to illustrate the efficiency of our approach.

1. Introduction

It has become more and more difficult to construct software systems in the last few years. The complexity of software systems increase rapidly, resulting in higher development costs, which grow exponentially in case of large scale software systems. Component-Based Software Development (CBSD) provides a high efficient and low cost way to construct software systems by integrating reusable software components. Although CBSD has already become a widely accepted paradigm, it is still beyond possibility to assemble components easily from COTS components into one application system. The difficulty of integrating COTS components mainly stems from the mismatches among interacting components. Existing approaches to modeling software systems often neglect the issue of component-mismatches identification and resolution. This paper aims to solve this problem.

2. Software architecture model

A software architecture describes the overall organization of a software system. It is described in terms of components and connectors.

2.1. The component model

Software components interact with each other via their interfaces, through which messages are sent to and received from a potential collaboration mate component. In order to know whether a particular component interface is a valid mate of another (that is to say the two components will work properly together if connected), we need adequate semantic information about components' behavior. Due to the black-box nature of COTS components, this information should be provided in the component's interfaces. So we adopt the methodology presented by [1] and extend the interface description with a behavior specification. Therefore, the specification of a component consists of two important parts: the signature and its behavior specification. While, the signature consists of a set of provide ports and /or a set of request ports; the behavior specification consists of a set of interaction protocols which specify the legal orders of sending/receiving messages. A component is specified as follows:

Component ***componentName*** = {
  Signature: ***provide ports***;
  request ports;
  Behavior: ***interaction protocols***;
}

Furthermore, a set of input actions is a set of messages received by the provide port; a set of output actions is a set of messages sent by the request port. Both input and output actions may have parameters, representing the data exchanged in the communication.

2.2. The connector model

Several connector models have been proposed in recent years, but how to support correct coordination for heterogeneous COTS components is still a troublesome subject because of the difficulties of

0730-3157/08 $25.00 © 2008 IEEE
DOI 10.1109/COMPSAC.2008.13

detecting and eliminating mismatches among components. Aiming at this point, we propose a connector model which consists of the following elements:

Role: the roles of a connector identify the logical participants in the interaction represented by the connector, and specify the expected behavior of each participant in the interaction.

PortMap: map a request port of the request component to a provide port of the server component if these two ports have the equivalent semantic, or else we think that they mismatch at semantic level and they cannot connect with each other.

Message Interceptor: intercepts the message which is sent from the request component, and then disassembles the message content into atomic data items, and finally forwards these atomic ingredients to data buffer.

Data Buffer: stores the atomic data items filled by message interceptor and waits for data assembler take them away. In this model, data buffer plays the role of data intermediate center.

Data Assembler: organize the atomic data items which are taken from the Data Buffer into legal structure that the destination component could accept.

Adaptor: according to the reachability graph of the compatible IAN which will be described in section 5, Adaptor generates the correct behavior protocol to conduct the interacting components in a deadlock-free way.

3. Mismatches detection

According to the component model as we present at 2.1, we claim that the precondition for proper components interaction is that we demand both signature matching and behavior matching. Recent years many works have researched the mismatches exhibit among interacting components [2, 3]. Based on their research results, we analyze and conclude the common inducements which cause the mismatches at two levels: signature level and behavior level.

1) Signature level. On the signature level the main task is the specification of ports. Each port is specified by its implement method. The signature mismatches may be produced by that one component cannot obtain the expected data from its collaboration mate component. ***Name conflict*** and ***Data Block discrepancy*** may result in this type of mismatches. ***Name conflict*** is that the identifiers of the same data entities in the input message are different from those in the corresponding output message. ***Data Block discrepancy*** is that there may be discrepancies between the data blocks sent by one component and expected by another component. Namely, these data blocks have different data organizations. (i.e., the sizes of the data blocks transmitted between components are different).

2) Behavior level. Mismatches at this level make the interaction protocol unable to run. In this paper, an interaction protocol is a restriction on the orders of exchanged messages. This type of mismatches is mainly caused by ***deadlock***. (Deadlock refers to generic deadlock as in most circumstances.)

4. Mismatch elimination

4.1. Eliminate name conflict

To resolve the name conflict, we adopt domain ontology to provide a semantic basis for components description and matching. Domain ontology is the explicit specification of domain conceptualization. Usually, people use ontology language to write down this specification. We adopt OWL [4] as the ontology description language. In OWL, classes are the focus of most ontologies. Classes describe concepts in the domain. In the following, we introduce an example to use the ontology to solve the semantic mismatching problem of the interacting components.

For example, in the paying process of the online pet shopping system, when the customer submits an order, the client component ***C*** requests the server to process the payment action ***DoPay***. Then, the connector finds a server component ***S*** which can provide a service ***DoAccount***. If there exits an ontology specification snippet as follows containing the relation operator "owl:equivalentClass" that defines ***DoPay*** as the equivalent class as ***DoAccount***, then the component ***S*** can response the request by the component ***C***, or else these two components cannot interact.

```
<owl:Class rdf:about="DoPay">
    <owl:equivalentClass rdf:resource="#DoAccount"/>
</owl:Class>
<owl:Class rdf:about="DoAccount">
    <owl:equivalentClass rdf:resource="# DoPay"/>
</owl:Class>
```

4.2. Eliminate data block discrepancy

There are three matching relationships between data blocks of input message and data blocks of output message. Let Dp denotes the data set of input message of a server component P, and Dr denotes the data set of output message of a request component R.

1) $Dp = Dr$. Data set Dp and Dr are equal. Namely, component R's output meets the requirement of component P's input exactly. In this case, messages can be sent out directly without processing;

2) $Dp \bigcap Dr = \varnothing$. Data set Dp and Dr are disjoint. Namely, component R's output does not match

component P's input at all. Thus, conclusion can be reached that no further processing is necessary because these two components cannot be integrated at all;

3) $Dp \cap Dr \neq \varnothing$. Data set Dp and Dr have some common data elements. Then these data elements which are suitable for component P should be extracted and reorganized from component R. We will detail the corresponding solution for case 3) in the following section.

Specifically, we use disassembling/reassembling strategy to resolve the ***data block discrepancy***. The corresponding work flow of the functional elements in connector is described as follows: ***Message interceptor*** first intercepts the data blocks sent from source component (request component), then disassembles them into atomic data items and forwards these atomic ingredients to ***data buffer*** ; next ***data assembler*** takes the atomic data items from ***data buffer***, and then reassemble them into the data blocks that the destination component (server component) can accept; finally, ***data assembler*** transmits the data blocks to the destination component.

For example, in a register procedure of a web-based application system, there are two communication components ***Client*** and ***Server***. The ***Client*** provides users to input his/her "username", "email" and "password" by its ***register*** port to register himself/herself to the server once a time in order to reduce the number of request for connections. Whilst the ***Server*** performs like this to cut down the server's workload: first accepts a message only containing "username" and "email" through its ***usr*** port to verify that whether the "username" and "email" conflict with existing user information stored in the database system, if there is no conflict, then it goes on to accept the message that contains "password" by its ***pwd*** port and then returns an ok message by its ***notify*** port; else returns a failure information by its ***notify*** port. The specification of these two components is described as follows: ("?"("!") appended to the name of port denotes that the port is an provide(request) port, here we only discuss the mismatches caused by ***data block discrepancy*** at signature level, we omit the behavior description for short of space)

Component ***client*** = {
 Signature:***register!***("username", "email", "password");
 reply?("result");
 Behaviour: …
}
Component ***server*** = {
 Signature:***usr?***("username", "email");
 pwd?("password");
 notify!("result");
 Behaviour: …
}

The mapping rule is specified as follows:

M_{cs} = {
 register!("username", "email", "password") < > usr?("username", "email"), pwd?("password");
 reply?("result") < > notify!("result");
}

Figure 1 shows the corresponding adaptation process of exchanged message containing "username", "email" and "password". (username, email and password are respectively abbreviated as u, e, p.)

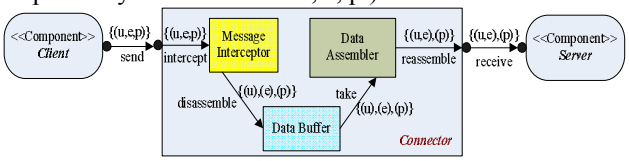

Fig.1. the adaptation process of exchanged message

4.3. Adapt incompatible behavior protocols

In this paper, we adopt Interface Automaton (IA) and Interface Automaton Network (IAN) to resolve the mismatches of interacting components at behavior level. For the detailed definition of IA, see [5].

A component-based software system can be regarded as a components network. Then its behaviors are the product of its components behavior. So, we propose to use interface automaton network (IAN) to model component-based software system. IAN consists of a set of interface automata which represent the abstract behaviors of software components. For the formal definition of IAN and further information about IAN, readers can refer to [6].

Sometimes, there may be some contradictions between the environment assumptions of two automata which have shared actions. Then, those corresponding composed states of IAN may cause ***deadlock***, we call these composed states ***illegal states***. In this paper, we refer as the IAN which only contains ***deadlock-free*** composed behaviors of the composed system to ***compatible IAN***.

Now, let us discuss a simple example to illustrate the use of IA and IAN for describing the interactive behavior of components. Consider the simplified Online Pet Shopping System (OPSS) in Figure 2. It consists of three business components: Audit Component (AC), Customer Component (CC) and Order Process Component (OPC). For more information about these three components, see our previous work [7]. In our example, we assume that Order Process Component doesn't support Booking business between online-shop and online customer. So it doesn't going to synchronize on Booking operation with Customer Component. And this will cause a deadlock in the overall composed system.

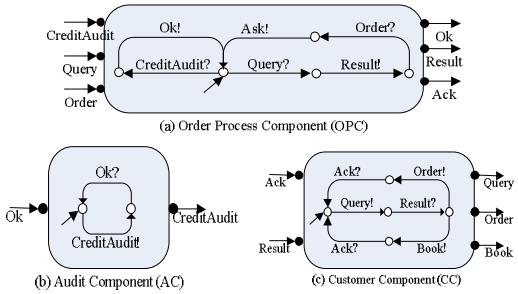

Fig.2. the IAN of the Online Shopping System

("?"("!") appended to the name of actions denotes that the action is an input (output) action. An arrow without source denotes the initial state of the interface automaton.)

The corresponding original composed behaviors and deadlock-free behaviors model of the above example are represented in figure 3. In the original IAN, state 5 is an illegal state, which will lead system OPSS into a deadlock. Because after performing an output action Book!, Customer Component engages in synchronizing on the shared action of Ack with Order Process Component, but in fact Order Process Component is not expected to share this action under such circumstances. In other words, according to Order Process Component's designed behavior, it is not assumed to support such booking business. Therefore, in order to derive the deadlock-free composed behaviors of AC, CC and OPC in OPSS, i.e., compatible IAN, state 5 and its backward transitions in the original IAN should be removed.

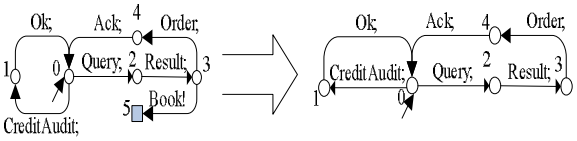

Fig.3. the original composed behaviors (left) and deadlock-free behaviors (right) of the IAN

(";" appended to the name of actions denotes that the action is an internal action. The other symbols are same as fig.2.)

For a given compatible IAN, we can easily construct a corresponding reachability graph. Our previous work [7] has presented the formal definition of reachability graph and detailed how to construct reachability graph. Here we omit these contents for short of space. Actually, the reachability graph is the deadlock-free behavior graph of its corresponding composed system. Figure 4 shows the corresponding reachability graph of the above example.

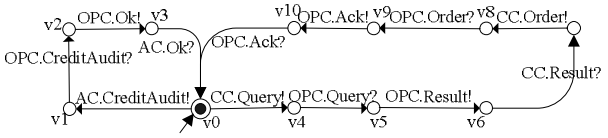

Fig.4. Reachability Graph of compatible IAN

Finally, according to the reachability graph of compatible IAN, we can easily generate the behavior graph of the adaptor which is used to coordinate these composed components to interact in a deadlock-free way by inverting the input and output actions of the reachability graph of IAN.

5. Conclusions

In this paper, we focus on the issue of detecting and eliminating mismatches among heterogeneous COTS components. The main contributions of our work include: 1) We propose an adaptive software architecture model; 2) Based on this model we analyze the mismatches that might occur at multiple levels and we present the corresponding solutions to eliminate these mismatches.

6. Acknowledgement

This work is supported by the National High Technology Development 863 Program of China under Grant No.2007AA01Z125,2006AA01Z189, the National Basic Research Program of China (973 Program) granted No.2006CB3003002.

7. References

[1] A. Bracciali, A. Brogi, and C. Canal, "A formal approach to component adaptation", *Journal of Systems and Software*, 2005, 74(1):pp.45-54.

[2] Min H, Choi S, Kim S, "Using smart connectors to resolve partial matching problems in COTS component acquisition", CBSE 2004, pp.40–47.

[3] Wenpin Jiao and Hong Mei,"Dynamic Architectural Connectors in Cooperative Software Systems", ICECCS 2005, pp.477-486.

[4] http://www.w3.org/TR/owl-guide/

[5] de Alfaro, L. and T. A. Henzinger,"Interface Automata", ESEC/FSE 2001, pp.109-120.

[6] Jun Hu, Xiaofeng Yu, Yan Zhang, Tian Zhang, Xuandong Li, Guoliang Zheng, "Checking Component-Based Embedded Software Designs for Scenario-Based Timing Specifications", EUC 2005,pp.395-404.

[7] Yiming Lau, Wenyun Zhao, Xin Peng, Zhixiong Jiang, Liwei Shen, "Coordination-Policy Based Composed System Behavior Derivation", APSEC 2007,pp.151-158.