An Automatic Connector Generation Method for Dynamic Architecture

Yiming Yang , Xin Peng and Wenyun Zhao
Computer Science and Engineering Department
Fudan University, Shanghai 200433, China
{051021056, pengxin, wyzhao }@fudan.edu.cn

Abstract

In a component-based system components are basic computation units implementing specific business functions, and their interactions are explicitly represented by connectors. If the system is required to be adaptable with dynamic architectural evolutions, the connectors must have the capability of being adapted at runtime to different interaction context. Aiming at this requirement, we propose an automatic connector generation method in this paper. In the method, connectors are generated on analysis of behavior and data specifications of two components to be assembled. First a state machine for connector behavior is created, and then the connector can be executed on the state machine, data adaptation scheme and predefined stub templates for various component types. Mismatches on data are resolved by model transformation. An example of payment query in e-business is referred throughout the paper to illustrate our method.

1. Introduction

Dynamic reconfiguration has been an essential feature for critical and long-running applications [1]. In these systems, the structure and behavior can be adapted at runtime to changing physical or business environment. Component-based system provides an architectural way for dynamic reconfiguration, in which interactions between components are explicitly represented by connectors and reconfigurations can be performed on the macro-architecture and be non-intrusive on the components [1]. Therefore, in order to achieve architectural dynamic evolution, the connectors must have the capability of being adapted at runtime to different interaction context. These contexts include component behavior protocol, communication protocol, parameter relationship and data format. So an automatic connector generation method for dynamic architecture is needed to makes it possible to efficiently re-compose components.

In this paper, we will study how to generate dynamic connectors automatically. Our Approach is based on Wright[2] and model transformation. The main contribution of this paper is to propose an approach for architecture connector generation. We conclude two main unadapted circumstances that connector should resolve to ensure components' interaction ---- data mismatch and behavior protocol mismatch. In our approach, we use ECORE to describe the data structure of components. Then we declare the transformation rules between components' parameters, and apply model transformation to resolve data mismatch problem. We introduce Wright component model as our component model and define a connector model to implement the four basic services a connector should provide (communication, conversion, coordination, facilitation). On the analysis of behavior and data specifications of components to be assembled we reason out the glue specification and organize it into a state machine. We use UML to construct state machine model. With the progression of state machine, connector finishes the glue function.

The article is organized as follows: in section 2 we first review related works, in Section 3 we shows our motivation example, Section 4 introduce the component and connector model we've used, in section 5 we argue our approach in detail , section 6 concludes this article and gives some future work.

2. Related Works

Our work is mostly related to the following researches.According to Metha et al. [3] connectors can be classified into four groups depending on the services they provide: communication, conversion, coordination, and facilitation.

Communication: Communication services support transmission of data among components.

Conversion: To enable heterogeneous components conversion of interaction required by a component to that provided by another component is essential.

Coordination: Coordination supports transfer of control among components.

31st Annual International Computer Software and Applications Conference(COMPSAC 2007)
0-7695-2870-8/07 $25.00 © 2007 IEEE

Facilitation: This type of service mediates and streams line components interaction, it facilitates and optimizes the interactions among components.

The aforesaid classification helps us to understand the four main services of connectors. Our connector model is based on this classification. And most of the units in our connector model focus on realizing the four functions.

In CBSD, a component can be viewed as a black-box entity which provides and/or requires a set of services. Component enters an organization in one of the three ways: it can be built in-house, purchased from a commercial vendor, or commissioned through a third party to be built [4]. These characteristics decide component should be adapted by connector before interaction. Many researches [5] [6] [7] show the mismatches between components' interactions. We conclude the two common mismatches:

Data Mismatch. The data mismatch result in the service component cannot obtain the expected data from the request component, the meaning of data and parameters are the same in most circumstances. Name conflicting and Data Block discrepancy may produce these mismatches. Name conflicting is the same data entities have the different identifiers. Data Block discrepancy is the same data entities have different date organizations.

Behavior Protocol Mismatch. According to [8], the composition needs not only signature level specification but also protocol level specification. In our context, a protocol is a restriction on the ordering of incoming and/or outgoing messages, including blocking conditions and guards on the methods.

We will explain our approach to solve these interaction mismatches in the following sections.

3. Motivating Example

Consider the simple online shopping system shown in Figure 1. It consists of an OnlineShop component and a BankService Component interacting via an ob-connector. Such a system is easy to describe in a "boxes and lines" diagram.

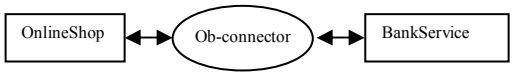


Figure 1. Static Topology: Simple Online Shopping System

In the Online-Shopping System, users can query the list of payment details under the Online-Shop Front. OnlineShop component will collect users' input, pack the data together and send it to the bank's system. The BankService component authenticates users' ID and password, and returns the current detail and history detail. Such a system is easy to describe in an ADL such as Wright [2, 9].

Figure 2 opens with a definition of the two components. Components have interfaces, which in Wright are called ports. Component may have multiple ports, each port defining a logically separable point of interaction with its environment. Here, OnlineShop send the query request and wait for the reply. BankService has a port for authentication. Because the payment details are stored in two different databases in bank's system, everyday a thread will put the current details into the history details database, BankService provide two ports for current details and history details query. The two query ports of BankService can be used correctly only after authentication. In the real-world scenario, the interaction between the two components may be more complex. We found it difficult to generate connector from the description only based on Wright. Wright focuses on describing the role specifications on a connector, and overall behavior specifications, such as the connector's glue. The role specifications localize one aspect of the interaction behavior to simplify consistency-checking and other analysis. But if we want to do connector generation, we must have more details that can solve the two mismatches we mentioned above. For example, in Figure 2, Wright uses a letter to define the input and output data of the process, but we should know the data structure of the input and output data, then define the data relationship between the roles. For another example, OnlineShop component may use method call to interact with the ob-connector, and the ob-connector may use a web service to interact with the BankServcie component, it should be pre-defined before the connector generation.

```
Component OnlineShop
  Port ClientQuery= query!q→(return?r□fail?f)→ClientQuery□ §
  Computation=ClientQuery.query!q→((fail?f→Computation)□ return?r → § )
Component BankService
  Port Authenticate=auth?a→ (ack!s∏ nack!f)→ (Authenticate□§)
  Port QueryCurrent=receive?r→(return!list)→ §
  Port QueryHistory=receive?r→(return!list)→ §
  Computation = S1 where
      S1 = Authenticate.auth?a→ ((Authenticate.ack ! s → S(i+1)) → ∏
       (Authenticate.nack ! f → S1) )   when i=1
        Q when i<>1   Q = QueryCurrent □ QueryHistory …..
Connector ob-connector
Glue = …..
```

Figure 2: Describing an online-shopping system's component using Wright.

4. Components and Connector Model

4.1 Component Model

A component describes a localized, independent computation. In our approach , our component model is based on Wright component model.

In Wright[2] , the description of a component has two important parts, the interface and the computation.

0-7695-2870-8/07 $25.00 © 2007 IEEE

An interface consists of a number of ports. Each **port** represents an interaction in which the component may participate. The **computation** section of a description describes what the component actually does.

The behavior and coordination of components is specified in WRIGHT using a notation based on CSP. CSP is a notation for specifying patterns of behavior and interaction[2].

Events : The basic unit of a CSP behavior specification is an event. An event represents an important action. An important property of events is that they can carry data. If a process supplies data, this is considered output, and written with an exclamation point. If a process receives data, this is input, and written with a question mark.

Processes : Given the basic element of behavior, the event, it is possible to construct patterns of behavior, or processes. Processes are described by combining events and other simpler processes.

Besides the behavior specification described by Wright, we also need some additional information for our component to support the automatic generation process. We need meta-models of parameters and operation semantic dictionary. The data schema of parameter would be detailedly illustrated in section 5.2. The operation semantic dictionary here now is just a table contain three columns ---- symbol (corresponding to event in CSP), operation semantic (we have a global dictionary to unify symbol into the same meaning) and condition. In our feature works, we plan to re- construct this table based on ontology. For example, table 1 is the OP table of OnlineShop.

Symbol	OP	Condition
Query	queryToday	q.stateDate=q.endDate =$TODAY
	queryHistory	q.endDate < $TODAY
	queryToday& queryHistory	q.endDate=$TODAY &&q.stateDate<$TODAY
…	…	…

Table 1: OP Table of OnlineShop

4.2 Connector Model

To realize the four main functions of connectors mentioned in Section 2, we propose a connector model designed as follows:

Parameter Adaptor: The parameter adaptor resolves the parameter mismatch problems. It uses Atlas Transformation Language to transform the input data to the data structure that the service port could accept.

Data Buffer: Data Buffer of the connector stores output and input messages . It uses a Hash map to organize data. When stub receives messages , it put data into data buffer . Parameter adaptor get data from data buffer , and put data back after transformation.

Request Controller: The request controller resolves the Behavior Protocol Mismatch problem. A state machine controls the interactions between components.

Stub: The stubs process the communication between component and connector. Stub is based on some pre-defined code template. Because we use xml to describe parameters , the stub supports transforming heterogeneous format parameter into xml automatically. For example , if the components use SOAP to communicate, we apply XSLT to transform. If the components use Java Method Call to communicate , we apply Java Reflection to resolve this.

4.3 The Meta-model of State Machine

A state machine is composed of a set of states. A state in the state machine represents the status in the process of interactions mediation. A state has some transitions. When a transition is fired, state machine changes current state from one state to another. If a state has two or more than two outgoing transitions, each transition from the state should have a guard condition to help event management decide which event would happen. A triggered event will cause the firing of the transition.

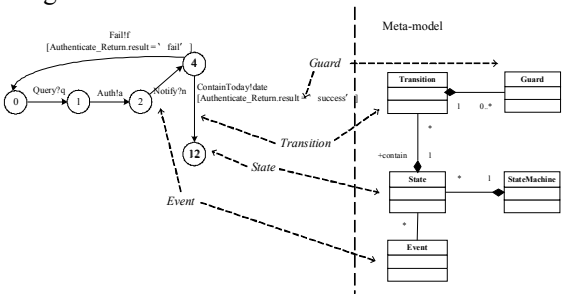


Figure 3: Part of the meta model of state machine

4.4 Execution of the connector state machine

Beyond states, events and transitions, we also have to care for the implementation of the state machine engine, i.e. the method that makes it progress by selecting which transition must be triggered according to events and the current state. Our state machine progression model is illustrated in Figure4. Class StateMachine is the facade of request controller. Method step() is a run-to-completion [10] procedure. We describe the operational semantics of the run-to-completion step as it shown in Figure 5. In our approach the time model is asynchronous. Under the asynchronous hypothesis, time is discrete; we apply an observer pattern to handle this situation. In the connector, each event is either a send-request procedure or a receive-result procedure. When an event

occurs, Stub will notify StateMachine. After being notified , StateMachine call step() to progress.

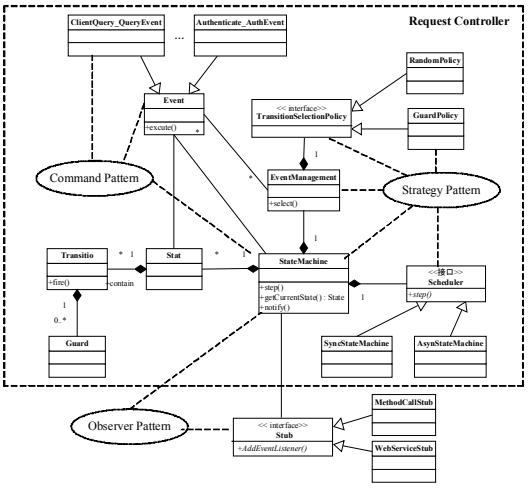


Figure 4: state machine progression

With respect to the state machine described above, this run-to-completion procedure looks like a GoF's Template Method [11], that is the skeleton of an algorithm in an operation, deferring some steps to subclasses. The steps we want to be able to redefine here are the following:

```
step()
{
   anEvent = getTriggeredEvent();                ①
   aTransition = anEvent.getTransition();        ②
   aTransition.fire();                           ③
   eventSet = currentState.getEvents();          ④
   nextEvent = eventManagement.choice();         ⑤
   execute(nextEvent);                           ⑥
}
```

Figure 5: operational semantics of the method step

1. First get a triggered event. This event would be a sending-message event or a receiving-message event, usually the stub notifies state machine the occurrence of events.
2. With this event, now we can get the transition.
3. Fire the transition, then the current state of the state machine will be changed.
4. Then we select all possible events on the current state.
5. The current state might return more than one event, In order to determine which event we want to process, we use an event management unit to choose the one we actually process (by certain event selection policy). We apply a strategy pattern on the event management unit, and our default event selection policy is shown as follows.
6. Finally execute the next event.

The Event Selection Policy

- If only one transition is available and the event on transition is a voluntary event (send output), fire this transition.
- If all events on transitions are passive events (wait for input), wait for the notification from stub and decide to choose which transition.
- If two or more than two transitions are available, the choice must depend on the guard conditions. The guard conditions are usually a group of values on the connector.

Figure 6 shows a single request and response function to help us understand the process flow inside the connector.

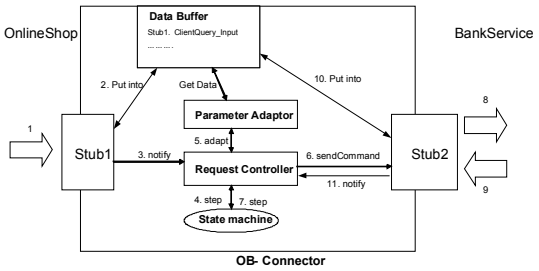


Figure 6: Process Flow of a single request and response function

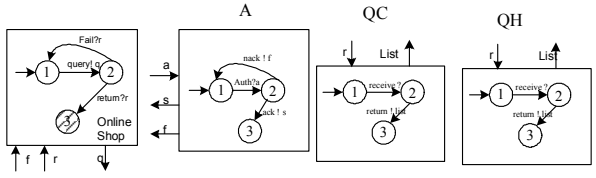
5. Connector Generation

5.1 Generation of State Machine

In component integration, an ideal situation is to deduce the connector's glue logic from the behaviors of components interact with the connector. There are two meanings of the elements on components' ports, one is the directive corresponding to the port, and another is the data corresponding to the directive, these two parts should be pre-defined. In the following algorithm, we will learn how to construct the state machine of connector.

1. According to the parameter relationships between request component and service components (transformation rules we declare in section 5) , first we reduce the number of available service ports by removing inappropriate service ports of which there exists a surjection from request parameters to service parameters.

2. Construct the state machine of components' ports according to their respective computation and port specification.

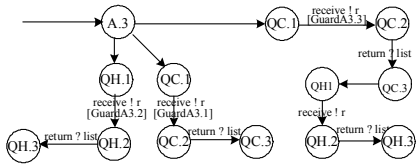


3. From computations' when condition expressions, establish the dependence relation of each port and service order. For example we can learn that

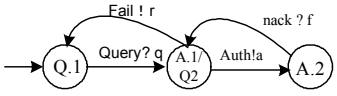
Authenticate should be process before query operation from condition (when i=1).

4. From the request port , mapping states and events from ports' state machine to connector's state machine. The output and input of events in connector is contrary to that in the components. While there is an output event in request ports , look for a suitable service port by dependence relation of service ports , then mapping the selected service port state machine to connector the same way as the request port did.

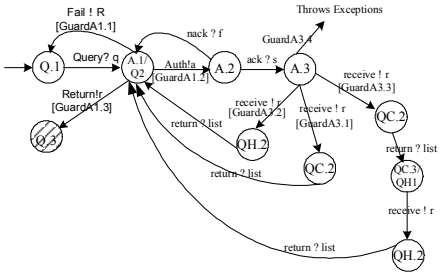
5. When there are more than one suitable service ports, look for the condition in component's computation specification and find all possible ports. We construct sets corresponding to these ports. For example, <QC, ϕ >,<QT, ϕ >. Then we do Cartesian product on these tuples, the result set {<QC, ϕ >,<QT, ϕ >,<QC,QT>,< ϕ , ϕ >} is a set of possible executable path of connector , a tuple represents a sub-state machine. We can automatically add guard condition to the sub state machine according to OP Tables illustrated in section 4.1. Because we have analyzed the dependence in step3, in this step we consider the ports here are independent and order-free, this is said that <QC,QT> is the same as <QT,QC>. If there is a external choice in computation, we argue a corresponding external choice in the connector decide this choice and the decision is according to the output and input message values in the connector. So the guard condition is a boolean expression of outputs and inputs of connector.



6. If all the state of request port's state machine has been mapped into the connector's, we merge the state between which the transition is empty.



7. Finally, the end of the algorithm.



5.2 Parameter Adaptor

Consider the example we show in section 3, the OnlineShop component and the BankServcie component have unadapted parameter on their clientQuery and Authenticate ports. The parameter mismatches are described as follows.

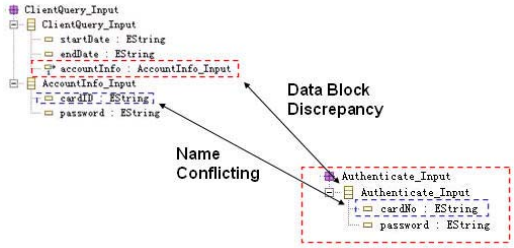


Figure 7: parameter mismatches between OnlineShop and BankServcie

The OnlineShop component sends an object ClientQuery_Input to the ob_connector. The ClientQuery_Input object has 3 attributes, but the Authenticate port of BankServcie only wants the information in the ClientQuery_Input's accountInfo attribute. So data block discrepancy appears. And obviously there is a name conflicting between cardID attribute in AccountInfo_Input and cardNo attribute in Authenticate_Input.

In our approach, parameter adaptor in the connector model aims at resolving the parameter mismatch problems. We use Ecore[11] as the metametamodel to define the input/output data's meta-model. The OMG MOF 1.4 Model has influenced the design of the EMF Ecore model. The Ecore model evolved in parallel with the MOF 1.4 model.

Then, we use ATL[12] to declare the transformation rules. ATL is the ATLAS INRIA & LINA research group's answer to the OMG MOF /QVT RFP. It is a model transformation language specified as both a metamodel and a textual concrete syntax. The transformation rules between the two parameters are described as follows.

```
module C2A;
create OUT : Authenticate_Input from IN : ClientQuery_Input;

rule C2A {
from
  cqi : ClientQuery_Input!ClientQuery_Input
to
  out : Authenticate_Input!Authenticate_Input (
    cardNo <- cqi.accountInfo.cardID,
    password <- cqi.accountInfo.password
  )
}
```

Figure 8: ATL transformation rules

Finally, we transform the ClientQuery_Input parameter to the parameter accepted by Authenticate port. These model transformations effectively solve Name conflicting and Data Block discrepancy mismatch and make it possible to generate the parameter adaptor's code.

0-7695-2870-8/07 $25.00

5.3 Additional coordination

Connectors are coordinators for interacting components. In our method, connectors are generated to implement desired component interactions in various contexts. However, in some cases, the connector may be required to provide additional functions, such as data pretreatment and additional security aspects. To achieve these additional functions, in our approach we define some listeners on the state machine, in this listener we deal with the data in the buffer unit or finish other functions while an event is triggered. For example , to merge the QueryCurrent and QueryHistory action's result , we add a listener to State , a fragment from beforeFireTransitionListener is given below.Before a transition is fired , the listener get corresponding event from the current state . If the ID of the event is "Q.return!r" , we do a merge operation. There are also many other listeners like beforeFireTransitionListener.

```
public boolean beforeFireTransition(State state)
{
    if( state.getSelectedEvent().getID()
                            .equals("Q.return!r") )
    {
        Buffer.setValue("Q.r" ,
                merge( Buffer.getValue("QC.list")
                        , Buffer.getValue("QT.list"))) ;
    }
    return true;
}

public List merge(Object object , Object object2)
```

Figure 9: fragment from listener

6. Conclusion and Future Works

In this work we proposed an approach for automatic connector generation for Dynamic Architecture from components' specification. By referring to Section 2, we found two main problem of component integration that connector must solve ---- data mismatch and behavior protocol mismatch. In section 4&5 we learn how to generate state machine of connector and the progression of state machine, during mapping we only use the matching of data and a simple semantic table of operations to reduce the size of result set after Cartesian product in certain state. In our future works, we want to introduce ontology to specify component's operation semantic. Then we use a model-transformation approach solve data mismatch problem, in this approach, components should provide their meta-model description of their output and input messages. What we should do is to describe the transformation rules between these meta-model, and this is not a comfortable work. How to automatically find the relationship through reasoning based on the existing rules is one of our future works. Finally, more case studies are need.

7. References

[1] Xin Peng, Wenyun Zhao, Liang Zhang, Yijian Wu. "An Intelligent Connector Based Framework for Dynamic Architecture." CIT 2005.

[2] Robert J. Allen, "A Formal Approach to Software Architecture", Carnegie Mellon University, CMU Technical Report CMU-CS-97-144, May 1997

[3] N. R. Mehta and N. Medvidovic, "Understanding Software Connector Compatibilities Using a Connector Taxonomy", *In Proceedings of First Workshop on Software Design and Architecture 2002*

[4] Paul Clements and Linda Northrop , *Software product lines* , Addison-Wesley , 2002

[5] Wenpin Jiao, Hong Mei, "Dynamic Architectural Connectors in Cooperative Software Systems", *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems* 2005

[6] DANIEL M. YELLIN and ROBERT E. STROM . "Protocol Specifications and Component Adapters", *ACM Transactions on Programming Languages and Systems, Vol. 19, No. 2,* March 1997

[7] Hyun Gi Min,Si Won Choi and Soo Dong Kim , "Using Smart Connectors to Resolve Partial Matching Problems in COTS Component Acquisition" , *CBSE 2004*

[8] Vallecillo A, Hernandez J and Troya JM , "New issues in object interoperability ", *In: Proc. of the ECOOP 2000 Workshop on Object Interoperability.*

[9] Robert Allen and David Garlan , "A Formal Basis for Architectural Connection" , *ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 6 Issue 3*, ACM Press , July 1997

[10]Harel, David and Naamad, Amnon ,"The STATEMATE Semantics of Statecharts" *ACM Transactions on Software Engineering and Methodology, 5(4):293-333,* October 1996.

[11] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides John, *Design Patterns:Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995

[11] www.eclipse.org

[12] ATL User Manual, www.eclipse.org/m2m/atl/doc

0-7695-2870-8/07 $25.00 © 2007 IEEE