A Systematic Approach to Composing Heterogeneous Components

HUANG Gang, MEI Hong, WANG Qian-xiang, YANG Fu-qing
Dept of Computer Science & Technology, Peking University, Beijing 100871
{huanggang, meih, wqx, yang}@cs.pku.edu.cn

Abstract: As the Component-Based Software Development (CBSD) becomes the prevalent approach to building large-scale distributed systems, the infrastructure of CBSD -- middleware -- is proliferating. Not only is the existing middleware evolving, but also new middleware has emerged. At the same time, the proliferation of middleware brings a new challenge to CBSD, i.e., how to compose heterogeneous components deployed in different middleware. This issue has received increasing attention from the academia and industry, but the existing solutions lack a systematic view of the CBSD processes and do not provide an open way to extend new mechanisms for interacting with new types of heterogeneous components. In this paper, a systematic approach to composing heterogeneous components is presented. The composition can occur at different lifecycle phases and is automated via a set of CASE tools and runtime mechanisms. The feasibility and applicability of the systematic approach are shown by the implementation and case study of composing EJBs (Enterprise JavaBeans) with other heterogeneous components.

Key words: CBSD; Reuse; Middleware; EJB

1. Introduction

As a bottom-up approach to reuse, the Component-Based Software Development (CBSD) focuses on building large software systems by integrating prefabricated software components. CBSD can improve quality, support rapid development and reduce the maintenance burden [1]. As CBSD becomes a prevalent approach to building the large-scale distributed systems, middleware, as the infrastructure of CBSD, is proliferating. The existing middleware is evolving. For example, RMI (Remote Method Invocation) evolves to EJB (Enterprise JavaBean) with a component model, CORBA (Common Object Request Broker Architecture) evolves to CORBA v3 with the quality of service, Internet enhancement and component model, DCOM (Distributed Component Object Model) evolves to .NET with XML. In addition, new kind of middleware has emerged, such as the infrastructure of Web Services. However, the proliferation of middleware brings a new challenge for CBSD, i.e., how to compose heterogeneous components, which are deployed in different middleware.

Although middleware provides deployed components with transparency of the heterogeneous programming languages, operating systems and networks, it cannot deny the heterogeneity between different middleware because of two main factors as follows:

- **Heterogeneous interoperability architectures:** The middleware provides powerful interoperability via separating the details of programming languages, operating systems and networks from application programs. The interoperability architecture of middleware usually consists of interoperability protocol, interface definition and naming service. For example, CORBA interoperability architecture consists of GIOP/IIOP (General Inter-ORB Protocol /Internet Inter-ORB Protocol), IDL (Interface
Definition Language) and CORBA Naming Service, RMI consists of JRMP (Java Remote Method Protocol), Remote Interface and RMI Registry, and the infrastructure of Web Services consists of SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language) and UDDI (Universal Description, Discovery and Integration).

- **Heterogeneous orchestrations**: In order to achieve desired results from the target objects, a client has to perform a series of coordinated steps, called orchestration or choreography. Different middleware has different orchestration. For example, in order to invoke CORBA objects, a client has to retrieve IORs (Interoperable Object References) of the target objects from CORBA Naming Service or stringified references at first. But in order to invoke EJBs, a client has to create a JNDI (Java Naming and Directory Interface) context, then to lookup the references of Homes of the target EJBs, and finally to create instances of EJBs via corresponding operations of EJBHomes.

The above heterogeneities imply that components deployed in different middleware cannot interact with each other directly, which degrades the reusability of components and decreases the number of reusable components in CBSD. Because it is impossible for any middleware to meet requirements of every domain and it is difficult – in fact, next to impossible – for a large enterprise to standardize on a single middleware [2], the issue to compose heterogeneous components becomes inevitable and critical in the practice of CBSD.

The heterogeneity induced by the proliferation of middleware has received increasing attention in the research and industry community. The existing solutions can be divided into four classes:

- **Transformation**: it transforms the exchanged messages from one middleware to another middleware, e.g., transforming IIOP messages into DCOM RPC messages in order to interoperate CORBA objects and DCOM objects. OMG (Object Management Group) has already observed the necessity to enable CORBA to interoperate with other middleware and has defined a series of interoperability specifications to standardize the transformations, e.g. CORBA to DCOM, DCE, RMI and SOAP [3]. The interoperability specifications cover all required details, from data types to message formats, and the runtime cost of the transformation is very high. And it is difficult to put the transformation approach into practice. Recently, OMG advances MDA (Model Driven Architecture) as the solution to the proliferation of middleware. The philosophy of MDA is to separate the details of middleware from the business models in order to reuse the artifacts at the design phase. When the middleware is upgraded or substituted, the developers can implement the business models in the new middleware again with some CASE tools [2]. But if the existence of multiple middleware is inevitable, MDA has no way other than the transformation.

- **Middleware for middleware**: it employs a special middleware for interactions between components deployed in heterogeneous middleware. Because Web Services gain the most supports in industry, reference [5] suggests the infrastructure of Web Services as middleware for middleware. As an emerging Internet technology, Web Services are not so mature to fulfill the responsibilities of middleware for middleware, such as security and transaction. Moreover, it is unpractical to determine the heterogeneous components in the development of components, i.e., programmers would be confused as to when and where to use Web Services.

- **Hybrid**: it allows components to invoke heterogeneous components directly via the interoperability architecture and orchestration of the heterogeneous middleware. For example, an EJB could invoke a
CORBA object directly through COS Naming and IIOP. The developers using the hybrid style have to face the heterogeneity. And the reusability of the components is decreased because the component can only invoke the given heterogeneous component.

- Bridging: it provides a bridging object that delegates the invocation from one middleware to another middleware, e.g., an EJB could invoke a CORBA object indirectly via another EJB, which uses the hybrid style to invoke the target CORBA object. The bridging style hides heterogeneous components via bridging objects. Because of the bridges, the performance resulted from bridging is poorer than that from the hybrid style. Compared to the transformation approach working at the message level, the bridging approach works at the invocation level.

From the applications of Transformation, Hybrid and Bridging, it can be concluded that a given middleware has to extend new mechanisms to enable its components to interoperate with other components deployed in another heterogeneous middleware. But none of them provides an open way to the implementation of new interoperability infrastructures. Moreover, because the component composition can occur not only at the implementation and runtime, but also at the design phase, the above solutions do not pay attention to the systematic methodology to guide the CBSD process, especially the component composition at higher abstract levels.

In this paper, a systematic approach to composing heterogeneous components is presented. The composition could occur at different lifecycle phases and is automated via a set of CASE tools and runtime mechanisms. Moreover, the runtime platform has an extensible framework that provides an open way to implement new mechanisms to interact with other types of heterogeneous components. The feasibility and applicability of the systematic approach are demonstrated by the experiment of composing EJBs with other heterogeneous components based on PKUAS, a J2EE (Java 2 Enterprise Edition) compliant application server [7].

The rest of the paper is structured as follows: Section 2 provides a brief introduction to ABC. Section 3 synopsizes our approach to composing heterogeneous components. Section 4 describes the implementation of the CASE tools and runtime mechanisms automating the approach. Section 5 illustrates the applicability of the approach with a case study. Section 6 brings a conclusion and discusses future direction of the work.

2. Overview of ABC

How to compose prefabricated components is a key issue in component-based reuse. Research on SA and CBSD provides two hopeful solutions from different perspectives. SA provides a top-down approach to realizing component-based reuse, but doesn’t pay enough attention to the refinement and implementation of the architectural descriptions, thus not fully able to automate the transformation or composition to form an executable application. CBSD provides a bottom-up way by using existing middleware infrastructures, but this technology is not able to guide systematically the CBSD process, especially the component composition at higher abstract levels.

Naturally another solution is to combine the above two approaches to realize component-based reuse, also identified in [8] and [9]. In [6], we presented an architecture-based component composition approach, abbreviated as ABC. In this approach, SA description is used as the blueprint and middleware technology as the runtime scaffold for component composition, using mapping rules and mini-tools to shorten the gap
between design and implementation. ABC presents an ADL, which supports users to define their own connectors according to requirements and has an extensible framework. Without paying more efforts to the reasoning of architecture, it attempts to use some existing techniques to reach this goal. For mapping a SA description into implementation, it is first mapped to an OO design model described in UML (Unified Modeling Language), then to final implementation. Although the architectural design is important, detailed design is still necessary, and gradual mapping is relatively easier and more possible to be done mechanically. In ABC, the architectural description can be mapped into source code (in our experiment, C++ and Java is chosen) or executable code by using some prevalent middleware (in our experiment, CORBA and J2EE are chosen). Fig.1 illustrates the process model of ABC.

ABC provides a feature-driven, SA-guided, middleware-based environment to automate the ABC process. The ABC environment consists of Feature Modeling Tool to help developers build the feature model of a system, ABCTool to help developers focus on the phases from Architecting to Deployment, and PKUAS, which is a J2EE-compatible application server and acts as the deployment and runtime platform of component-based systems developed by ABC.

3. ABC’s Approach to Composing Heterogeneous Components

3.1 Technical Framework of Middleware

As there exist many so-called middleware, this paper focuses on middleware that addresses the issue of interoperability and supports the concept of component, such as CORBA, RMI, DCOM and Web Services. The functional core of the prevalent middleware can be divided into the interoperability framework, the endpoint and the adapter, which together form a technical framework, shown in Tab.1.

<table>
<thead>
<tr>
<th>Middleware</th>
<th>Interoperability Framework</th>
<th>Endpoint</th>
<th>Adapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA</td>
<td>GIOP/IIOP</td>
<td>IDL</td>
<td>POA</td>
</tr>
<tr>
<td>RMI</td>
<td>JRMP</td>
<td>Remote Registry</td>
<td>Container</td>
</tr>
<tr>
<td>Web Services</td>
<td>SOAP</td>
<td>WSDL</td>
<td>UDDI</td>
</tr>
</tbody>
</table>

The interoperability framework enables components to interact with each other, which may be implemented by different programming languages and residing on different operating systems and networks. In fact, the interoperability framework implements the interoperability architecture described previously, which includes the communication protocol, interface definition and naming. The endpoint provides components
a uniform runtime space, which separates the details of the whole runtime environment from the components. The endpoint is responsible for the lifecycle management of components, including instantiation and release, and some system-level functions, such as security and transaction. And the endpoint usually implies the orchestration of components described previously. The adapter is the glue between the other two, enabling components contained by the endpoint to receive and send messages via the interoperability framework.

From the specifications and implementations of the prevalent middleware, it can be concluded that the technical framework of middleware is monolithic, i.e., the components running in the endpoint of one middleware can only receive and send messages via the interoperability framework of the same middleware. Moreover, the components have to use the three elements of the same interoperability framework together. In other words, the components cannot use the communication protocol of one middleware and use the interface definition or naming of another middleware. In CBSD, the implementation of components usually specifies the interface definition, naming and orchestration of the invoked components. Because of the monolithic technical framework of middleware, the components have to use the communication protocol and adaptor of the same middleware at runtime. The monolithic technical framework of middleware can be considered as the root reason why heterogeneous components cannot interoperate with each other directly.

3.2 Systematic View of Composing Heterogeneous Components

In the technical framework of middleware, the implementation of components specifies the interface definition, naming and endpoint. And such implementation doesn’t inherently require the communication protocol and adapter of the same middleware at runtime if the technical framework was not monolithic. Unlike the existing approaches, the systematic approach presented in this paper breaks up the monolithic technical framework of middleware via encapsulating the communication protocol and orchestration into the adapter. Based on the process model in Fig.1, the composition of heterogeneous components in ABC can occur at the following phases:

- **Software Architecting**: ABC/ADL (Architecture Description Language) is the formal language to specify components, connectors and architectural attributes in ABC. It covers almost all syntaxes and semantics of the interface definition languages of the prevalent middleware, and provides an open framework to support users to define new syntaxes and semantics. Because the ABC/ADL separates knowledge of middleware from SA, the SA in ABC can be considered as the Platform Independent Model of MDA [2].

- **Composition**: Emphasizing on the component-based reuse, ABC implements systems with the composition of the prefabricated component stored in the components repository. These reusable components have the facets to describe functions and other attributes in ABC/ADL, facilitating the component qualification with SA in ABC/ADL. If the component qualification fails, the required components can be developed at hand. ABC supports two approaches of components development. The components can be developed in Object-Oriented paradigm via the UML models transformed automatically from SA, or developed with the preferred middleware based on the programming framework automatically generated from SA. And either approach focuses only on one middleware.

- **Deployment**: At this phase, developers have to deal with heterogeneous components by utilizing the deployment tool to generate adapters, which encapsulate the communication protocol and orchestration. Given two heterogeneous components, they will communicate with each other via the
same protocol. And the adapter will transform one orchestration into another orchestration.

- **Runtime**: Besides the runtime mechanisms to send and receive messages through the specified communication protocol, it is necessary to deal with the heterogeneous naming mechanisms, i.e., different naming servers and heterogeneous addresses of components. For example, when an EJB wants to invoke a CORBA object, it has to deal with the IOR of the CORBA object as the PortableRemoteObject of an EJB. Moreover, it is necessary to provide an open and easy way to extend new mechanisms for new interoperability framework. So, ABC provides the extensible interoperability framework and naming federation to enable the composition of heterogeneous components at runtime.

### 4. Implementation

There are some important mechanisms supporting ABC to compose heterogeneous components from Software Architecting, Composition, and Deployment to Runtime, including ABC/ADL, automated generation of the adapters, and runtime mechanisms, shown in Fig.2. Reference [6] described ABC/ADL and its transformation to UML, C++ and CORBA in details. So this paper focuses on the other two. Because PKUAS only supports EJBs, the implementation presented in this paper focuses on composing EJBs with other heterogeneous components. The application of this approach in other middleware is under development.

#### 4.1 Automated Generation of Adapters

In current composition, ABC can generate adapters for EJBs, while other middleware is responsible for generating adapters for heterogeneous components. The runtime communication protocol and transformation of heterogeneous orchestration encapsulated in the adapters generated for EJBs have to conform to other middleware. For instance, a CORBA object can be invoked by a single request for the target operation, while an EJB can only be invoked after invoking the Home of the target EJB previously. If an EJB wants to invoke a CORBA object, the stub of the EJB has to transform two invocations into a single invocation. In other words, the stub returns itself and does not send any message to the server when the first invocation to EJBHome occurs. And only when the second invocation to EJB occurs, it sends a request to the server. Conversely, if a CORBA object wants to invoke an EJB via a single request, the skeleton of the EJB has to transform the single request into two invocations to the Home and EJB sequentially. The
automated generation of adapters for EJBs by ABC has some restrictions shown in Tab.2.

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
<th>Adapters</th>
<th>Communication Protocol</th>
<th>Orchestration</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJB</td>
<td>EJB</td>
<td>Stub/Skeleton</td>
<td>IIOP or Local</td>
<td>EJB</td>
</tr>
<tr>
<td>EJB</td>
<td>CORBA/</td>
<td>Stub</td>
<td>IIOP/SOAP</td>
<td>EJB-to-CORBA/</td>
</tr>
<tr>
<td></td>
<td>Web Services</td>
<td></td>
<td></td>
<td>EJB-to-WebServices</td>
</tr>
<tr>
<td>CORBA/</td>
<td>EJB</td>
<td>Skeleton</td>
<td>IIOP/SOAP</td>
<td>CORBA-to-EJB/</td>
</tr>
<tr>
<td>Web Services</td>
<td></td>
<td></td>
<td></td>
<td>WebServices-to-EJB</td>
</tr>
</tbody>
</table>

Now ABC deployment tool can generate adapters for composing EJBs with EJBs, CORBA objects and Web Services. The developers can implement and add new generators into ABC deployment tool for composing EJBs with new types of heterogeneous components.

4.2 Open Communication Service

CORBA Extensible Transport Framework (XTF), derived from the Open Communication Interface, can plug and play multiple transport protocols into ORB. Modified from XTF, the open communication services of PKUAS can plug and play not only multiple transport protocols, such as TCP/IP, SSL and HTTP, but also multiple interoperability protocols, like IIOP and SOAP. Shown in Fig.2, the pluggable protocols can be achieved through a set of pluggable components as follows:

- **Stub/Skeleton**: They are adapters for the client and the server respectively. The stubs transform the method invocation specific to the client-side programming languages into the message specific to interoperability protocols (e.g., GIOP and SOAP), and vice versa. The skeletons transform the message specific to interoperability protocols into the messages defined by PKUAS, and vice versa. And the PKUAS-specific message would be transformed into the method invocation specific to the server-side programming languages by containers, which therefore could disinterest the heterogeneous interoperability protocols. The transformation includes not only the signature but also non-functional constraints of the given interface, such as the interface name, operation name, parameters, return value, exceptions and contexts, e.g., security and transaction. The stubs and skeletons are specific to interfaces, i.e., different interfaces have different stub and skeleton. For a given interface, the number of stubs and skeletons is equal to the number of interoperability protocols supported by the implementation of the given interface. For example, if an interface wants to be accessed by IIOP and SOAP, there would be one stub and skeleton for IIOP and another stub and skeleton for SOAP. Like most of the middleware, PKUAS can automatically generate stubs and skeletons from given interfaces.

- **Connector/Acceptor**: Both are responsible for sending and receiving messages specified by the given interoperability protocols, transforming the messages specific to interoperability protocols into the messages specific to underlying transport protocols (e.g., TCP and HTTP), and managing connections. Different interoperability protocols have different transfer syntaxes. For example, GIOP uses Common Data Representation to deal with messages as binary streams and SOAP uses XML to deal with messages as text streams. Before sending a message and after receiving a message, the connector and acceptor have to marshal and unmarshal messages with the given transfer syntax. Typically, the connector and acceptor manage connections via dealing with messages in different formats. For example, the connector and acceptor of GIOP manage connections through the seven types of messages, including Request, Reply, Cancel Request, Locate Request, Locate Reply, Close Connection and Message Error. Connectors can build connections, send requests and receive...
responses at the client side, while acceptors can listen at a given network address, receive requests and send responses at the server side.

- **Transport**: it is responsible for sending and receiving messages through the underlying transport protocol. The transport supports to send and receive in a blocking or non-blocking manner, deal with timeout and throw some exceptions to identify errors in network. Moreover, the transport encapsulates the network address, i.e., constructing and consuming the network address in the interoperability address, while the endpoint is responsible for the object key in the interoperability address. For example, an IOR consists of the IP address and port to identify the ORB runtime instance in the network and the object key to identify the CORBA object instance in the ORB runtime. Defining standard interface between Transport and Connector/Acceptor, PKUAS allows that one transport protocol can be employed by multiple interoperability protocols.

From the above components embedded in PKUAS, the heterogeneous endpoints can send and receive messages through the same communication protocol and without awareness of the heterogeneous orchestrations at runtime. Currently PKUAS provides implementations for IIOP, JRMP, SOAP and Collocated (it is used to communication between EJBs hosted in the same PKUAS instance to improve the performance). And the extensibility of PKUAS allows users to develop new components for other communication protocol on demand.

### 4.3 JNDI Federation

Different middleware employs different naming services and heterogeneous interoperability addresses, which is a critical challenge to support the composition of heterogeneous components at runtime. Separating the interoperability framework and endpoint and constructing the two parts of the interoperability address separately, ABC supports EJBs to manipulate addresses of heterogeneous components and publish itself as the heterogeneous address via the JNDI Federation implemented in PKUAS.

Current J2EE implementations always store all of the data on a central JNDI server, which may become the bottleneck of performance and the point of failure in the whole system. Differently, PKUAS stores the data on a hierarchical and distributed JNDI Federation, shown in Tab.3.

**Tab. 3 Four Levels of the JNDI Federation in PKUAS**

<table>
<thead>
<tr>
<th>Level</th>
<th>Namespace</th>
<th>Storage</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Single Container</td>
<td>Per Container</td>
<td>Initial value of EJB properties; Autonym of the Refered EJBs</td>
</tr>
<tr>
<td>2</td>
<td>Single Application</td>
<td>Per Application</td>
<td>Shared properties in an application; Interoperability addresses of Local.</td>
</tr>
<tr>
<td>3</td>
<td>Single Server</td>
<td>Per Server</td>
<td>Shared properties in a server; Interoperability addresses of IIOP.</td>
</tr>
<tr>
<td>4</td>
<td>Global</td>
<td>Global</td>
<td>Global properties; Interoperability addresses of the accessible components.</td>
</tr>
</tbody>
</table>

Because the naming servers of other middleware cannot be modified at all, PKUAS integrates them via proxies. The JNDI Global server provides a global namespace and a set of proxies. At the initialization of the JNDI Global server, the proxies traverse the associated naming servers to retrieve all of the bindings and then register every available name bound with their own references, instead of the interoperability addresses of the target components, into the JNDI Global server. The JNDI Global server performs the
same actions to refresh the bindings periodically. When a name is found in the JNDI Global server, it does not return the binding at once but delegates the associated proxy to retrieve and then return the real interoperability address of the target component from the corresponding naming server. Because EJBs are only able to deal with the interoperability addresses as PortableRemoteObjects, PKUAS re-implements the narrow method of the PortableRemoteObject to encapsulate the heterogeneous address, which can be accessed by the stub to connect the heterogeneous component. On the other hand, an EJB may be accessed by other heterogeneous components. Like the generation of skeletons for heterogeneous components, PKUAS has to construct and publish multiple addresses into the JNDI Global server, which delegates proxies to publish the heterogeneous addresses into the specific naming servers. Then other heterogeneous components can retrieve the valid addresses from their own naming servers and invoke EJBs.

The JNDI Context hides the details of the JNDI implementations. And developers can implement new proxies to integrate other naming servers into PKUAS on demand.

5. Case Study

Drawn in ABCTool, Fig.3 illustrates the SA of a Web Home Heating System. The Controller reads the temperature via the Thermometer periodically. The periodical interval, \textit{beatFreq}, is mapped from the \textit{interval} set by users. If the temperature is lower than the user preferred temperature, \textit{controlTemp}, the Controller will open the Heater. Otherwise the Controller will close the Heater. Users can start up or shut down the Controller through Internet. Developers can compose these five components through ABC processes as follows:

- **Software Architecting**: Developers can achieve the SA shown as Fig.3, only considering the interfaces of components other than their implementations.
- **Composition**: Developers can select components from the components repository. The five components are heterogeneous, i.e., the Clock is a Java application, the Controller and Thermometer...
are EJBs, the Heater is a CORBA object, and the WebController is a client-side component of Web Services. But developers can qualify these components through their ADL descriptions without caring for the heterogeneous middleware, i.e., RMI, CORBA and Web Services.

**Deployment:** The five heterogeneous components have to be deployed into four platforms: Clock into J2SE, Heater into Orbix, WebController into a remote host, and Controller and Thermometer into PKUAS. ABC deployment tool automatedly generates five adapters for Controller and Thermometer according to the encapsulation rules, shown in Tab.4. Fig. 4 illustrates Controller’s stubs for Heater and Thermometer, giving detailed description of the encapsulation of interoperability protocols and orchestrations in the adapters.

**Tab. 4 Automatically Generated Adapters for EJBs of the Web Home Heating System**

<table>
<thead>
<tr>
<th>Adapters</th>
<th>Client</th>
<th>Server</th>
<th>Communication Protocol</th>
<th>Orchestration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stub</td>
<td>Controller</td>
<td>Heater</td>
<td>IIOP</td>
<td>EJB to CORBA</td>
</tr>
<tr>
<td>Stub</td>
<td>Controller</td>
<td>Thermometer</td>
<td>Local</td>
<td>EJB</td>
</tr>
<tr>
<td>Skeleton</td>
<td>Clock</td>
<td>Controller</td>
<td>JRMP</td>
<td>EJB</td>
</tr>
<tr>
<td>Skeleton</td>
<td>WebController</td>
<td>Controller</td>
<td>SOAP</td>
<td>Web Services to EJB</td>
</tr>
<tr>
<td>Skeleton</td>
<td>Controller</td>
<td>Thermometer</td>
<td>Local</td>
<td>EJB</td>
</tr>
</tbody>
</table>

/* fragment of Controller.java*/  
/*invoke thermometer*/  
ThermometerHome meterHome = (ThermometerHome)PortableRemoteObject.narrow(  
jndiInitialContext.lookup("thermometer" ), ThermometerHome.class);  
float temperature = meterHome.findThermometer().readTemp(); …  
/*invoke heater*/  
HeaterHome heaterHome = (HeaterHome)PortableRemoteObject.narrow(  
jndiInitialContext.lookup("heater" ), HeaterHome.class);  
heaterHome.findHeater().switch(TRUE); …

/* fragment of ThermometerHomeStubRMI.java*/  
public Thermometer findThermometer() {  
GIOPMessage msg = new GIOPMessage();  
/*RMI-IIOP is default for EJBs.*/  
… /*put data into msg, then send msg to target  
identified by ior via IIOP.*/  
connector.send(msg, ior); …}  

/* fragment of HeaterHomeStubIIOP.java*/  
public Heater findHeater( ) {  
/*One CORBA object acts as EJB Home and Remote.*/  
return new HeaterStubIIOP(ior, connector); }  

/* fragment of heaterHomeStubRMI.java*/  
public void switch(Boolean onOff) {  
GIOPMessage msg = new GIOPMessage();  
… /*put data into msg, then send msg to target identified  
by ior via IIOP.*/  
connector.send(msg, ior); …}

Fig.4 Controller’s stubs for Thermometer and Heater

**Runtime:** In the initialization of PKUAS, the interoperability addresses are registered into the corresponding naming servers via the JNDI Federation, shown in Tab.5.

**Tab. 5 The Interoperability Addresses of the Web Home Heating System**

<table>
<thead>
<tr>
<th>Addresses</th>
<th>Client</th>
<th>Server</th>
<th>Published by</th>
<th>Stored into</th>
<th>Retrieved from</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSDL</td>
<td>WebController</td>
<td>Controller</td>
<td>JNDI</td>
<td>UDDI</td>
<td>UDDI</td>
</tr>
<tr>
<td>UnicastRemoteObject</td>
<td>Clock</td>
<td>Controller</td>
<td>JNDI</td>
<td>RMI Registry</td>
<td>RMI Registry</td>
</tr>
<tr>
<td>Local Reference</td>
<td>Controller</td>
<td>Thermometer</td>
<td>JNDI</td>
<td>JNDI</td>
<td>JNDI</td>
</tr>
<tr>
<td>IOR</td>
<td>Controller</td>
<td>Heater</td>
<td>COS Naming</td>
<td>COS Naming</td>
<td>JNDI</td>
</tr>
</tbody>
</table>

6. Conclusion and Future Work

How to compose heterogeneous components is an important but unavoidable challenge in the practice of
CBSD. Current solutions lack a systematic approach to conducting the whole process and cannot provide an open way to the extension of new interoperability mechanisms. Observing the monolithic technical framework of middleware, this paper presents a systematic approach to composing heterogeneous components. And the composition can occur at different lifecycle phases and is automated by a set of CASE tools and runtime mechanisms. PKUAS provides an open way to extend new mechanisms to interact with other types of heterogeneous components. The implementation and case study demonstrate the feasibility and applicability of this systematic approach.

Based on PKUAS, current implementation only supports to compose EJBs with other heterogeneous components. The future work can focus on how to implement this systematic approach on other middleware, such as CORBA and Web Services, and the heterogeneous orchestrations besides RMI and CORBA.

Acknowledgements

This effort is sponsored by the State 863 High-Tech Program (Contract No. 2001AA113060), National Science Fund for Distinguished Young Scholars (Contract No. 60125206), National Natural Science Foundation of China (Contract No. 60043002 and 60103001) and Major Program of Ministry Of Education of China (Contract No. MAJOR0214).

References