Visual Component Assembly and Tool Support Based on System Architecture

Seungyun Lee, Oh-Cheon Kwon, and Gyu-Sang Shin

Component-based development leverages software reusability and reduces development costs. Enterprise JavaBeans (EJB) is a component model developed to reduce the complexity of software development and to facilitate reuse of components. However, EJB does not support component assembly by a plug-and-play technique due to the hard-wired composition at the code level. To cope with this problem, an architecture for EJB component assembly is defined at the abstract level and the inconsistency between the system architecture and its implementation must be eliminated at the implementation level. We propose a component-based application development tool named the COBALT assembler that supports the design and implementation of EJB component assembly by a plug-and-play technique based on the architecture style. The system architecture is first defined by the Architecture Description Language (ADL). The wrapper code and glue code are then generated for the assembly. After the consistency between the architecture and its implementation is checked, the assembled EJB components are deployed in an application server as a new composite component. We use the COBALT assembler for a shopping mall system and demonstrate that it can promote component reuse and leverage the system maintainability.

Keywords: Component assembler, architecture, CASE tool, COBALT assembler, Enterprise JavaBeans (EJB).

Manuscript received Jan, 21, 2003; revised May 20, 2003.

This work was supported by the National Research Laboratory (NRL) Program of the Ministry of Science and Technology of Korea.

Seungyun Lee (phone:+82 42 860 1186, email: coral@etri.re.kr), Oh-Cheon Kwon (email: ockwon@etri.re.kr), and Gyu-Sang Shin (email: gsshin@etri.re.kr) are with Computer & Software Research Laboratory, ETRI, Daejeon, Korea.

I. Introduction

There has been considerable work on component-based development (CBD) in the software industry. Component models such as Enterprise JavaBeans (EJB) [1], component object model (COM+) and common object request broker architecture (CORBA) [2] component model (CCM) have emerged to reduce the complexity of software development and to facilitate reuse of components [3]. By using a component model, a piece of software can be plugged into many different systems. CBD allows integration of external, off-the-shelf software and supports a plug-and-play technique which makes building and replacing software easier [4], [5]. As a component model of the server side, EJB was developed with the intention of reusing components on various platforms without code modification or a re-compiling process.

However, it is difficult to assemble various EJB components provided by third parties using a plug-and-play technique for the following reasons. Firstly, since a component model has a wide variety of vendor specific methods for implementing an infrastructure or middleware, it is difficult to assemble the components generated from different infrastructures or middleware. Secondly, the plug-and-play assembly of the EJB components and the flexible reconfiguration of a system model built by the assembly process are difficult since methods have to be directly invoked for the interactions between the EJB components. This means that the method of a component should be invoked at a client level where the component is reused, and the client code should have method invocation logics for components' interactions [6].

In order to solve the first problem with assembling the EJB components, [7] suggested the enterprise application integration (EAI) technique. The EAI technique attempts to

integrate legacy systems into an enterprise system. However, since the system integration techniques developed by various vendors are different, the component models and EAI techniques are dependent on the vendors that develop the infrastructure and middleware for supporting the models. Although this critical defect can be overcome by connecting the various middleware and application server platforms [8], there still remains the problem of assembling the EJB components themselves.

For the second problem, the architecture technology is used for the component assembly by a plug-and-play technique at the abstract level. A well-defined architecture can be used to analyze, refine, and test functions of the software system to be developed. Accordingly, if the EJB components are assembled using the architecture, the assembled EJB components can be exactly connected and operated through modeling, analyzing, and refining the architecture. The architecture consists of components and connectors, and each component is regarded as an independent piece of software. A connector plays the role of controller in the system architecture, and each component communicates with other components through the connector. When the component is considered independent in the system, it can be assembled or replaced by plug-and-play without influencing other components. Furthermore, by expressing the interacting EJB components in the Architecture Description Language (ADL), which is independent of the code, the interactions among EJB components can be managed separately from the EJB implementation. In this way, EJB components can easily communicate with each other and a component can be replaced with a new component without changing the code. The architecture for component assembly is defined in the system design phase and any inconsistencies between system design and implementation can be eliminated at the implementation phase by assembling the components using the architecture [9].

We propose the component-based application development tool (COBALT) assembler to support the design and implementation for component assembly, enabling a visual description of the system's architecture and flexible reconstruction of the EJB component system. It assembles pre-built components into specific application software or a large-grained component (i.e., a composite component). The generated architecture is expressed by ADL corresponding to the C2 architectural style, which consists of components and connectors [10]-[12]. The components and connectors have defined top and bottom ports. The components communicate with each other by passing messages through the connectors. The message-based communication of the C2 architectural style makes components independent from other components and simplifies the problem of integration. The architecture description plays a role in generating wrapper and glue code for assembling the components. The COBALT assembler checks the consistency between the architecture and implementation and ensures the 'communication integrity' of the system [13]. Connecting a composite component to a client program implements the application system.

This paper is organized as follows. Section II outlines previous work. Section III presents the method for EJB component assembly and the overall structure and its implementation as a subsystem of the COBALT assembler. Section IV describes how our case study using a shopping mall system evaluates our tool. Finally, section V describes our conclusions and planned future work.

II. Previous Work

There has been little work on component assembly using architectural styles and supporting tools. Rosenblum put architectural concerns in JavaBeans component interoperability and implemented the tool, ARABICA [14]. It supports JavaBeans component assembly according to the C2 architectural style. Using a wrapping method, ARABICA maps JavaBeans components to C2 components. It also provides an editor for handling JavaBeans' attributes, methods, and events. However, this approach has a weakness in that the JavaBeans component model supports not server side components but GUI components. JavaBeans components can be assembled with other GUI components without adding business logic to its defined properties. In the case of assembling EJB components at the server level, however, each component must handle the business logic. Furthermore, multiple beans can be packaged together. In this case, the properties of beans and their interoperability in the EJB Jar need to be considered when EJB Jar components are assembled. ARABICA supports JavaBeans component assembly by a plug-and-play style, but it limits the scope to the components at the GUI level, not at the server level.

Hong proposed the architectural style, DSIAS, to integrate COTS components in distributed environments and built a guideline for the system design using DSIAS [15]. DSIAS supports wrapping and adapting technology to reuse COTS components in various platforms. It defines two coordinators, the UI generator and task coordination controller. Since the coordinators are in charge of a communication port between users and COTS components, they simplify interactions of components. However, this work does not provide a guideline for connecting system designs to system implementations. Since the system designs based on DSIAS do not guide an implementation process or a technique, assembling COTS components is still difficult at the implementation phase.

Although this work proposes an architectural style, it does not describe a well-defined architecture using ADL. Its weakness lies in defining, analyzing, refining, and testing the architecture.

For developing new systems by assembling EJB components, an assembly environment by plug-and-play is required. To achieve this goal, we propose the COBALT assembler, a tool that supports the assembly of EJB components and describes the architecture by the ADL according to the C2 architectural style and generates a wrapper and glue code for assembling components.

III. COBALT Assembler

This section describes the COBALT assembler, which supports the design and implementation of the EJB component assembly process. This tool makes it possible to describe the system architecture and component structure at the abstract level using ADL and to generate code for the assembly that conforms to the defined architecture.

1. Method for Component Assembly

To develop an application system by assembling components, information about domain and user requirements is necessary. A domain analysis phase supports domain modeling and component identification for the system development. Component identification has recently been done by the intuition and experience of domain experts. A systematic method was proposed and implemented in the COBALT constructor that was developed in parallel with the COBALT assembler and provides many features such as component identification, component modeling, component implementation, component deployment and testing [16]. The identified components make up the application system architecture. The design phase is divided into two steps: one is for the system architecture design, which shows the big picture of the system and the interaction of the components; the other is for the component structure design, which supports plug-and-play assembly and provides the basis for an implementation technique. ADL describes the system architecture and component structure. The descriptions of these structures are used to generate the wrapper and glue code in the implementation phase of component assembly after verifying architectural properties. Figure 1 shows the process of the component assembly.

System architecture defines the complex interaction among components at the abstract level. A connector manages the interfaces of the components. The connector supports plug-and-play assembly by keeping components in the system independent. The component structure design acts as an

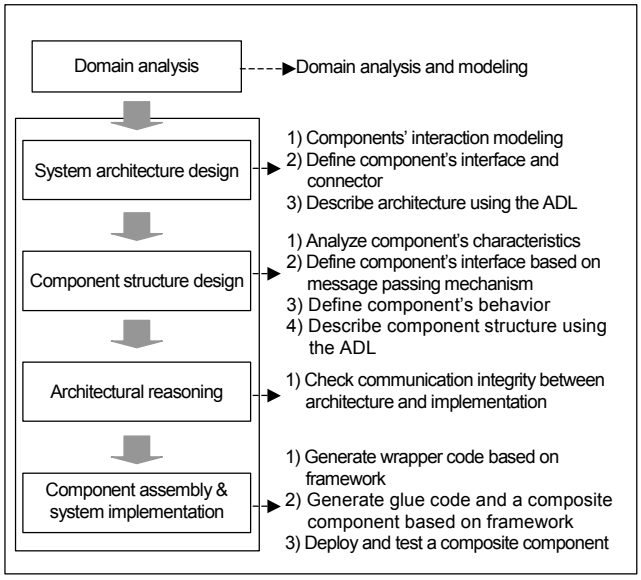

Fig. 1. Process of component assembly.

intermediary for implementing the interface of message-based components. The plug-and-play assembly of EJB components cannot be done by method invocation. For easy and flexible assembly, method invocations among components should be changed into messages by a message passing mechanism. Using information of an EJB component's code, a component structure design defines a component's ports and expresses a component's interfaces, methods, and behaviors at the abstract level. After describing a system architecture and a component structure, communication integrity among the assembled EJB components should be verified. The wrapper code is generated from a flawless component description. The glue code is generated from the interaction of the sound components. The composite component is implemented for reusing the system architecture and tested in the run time environment.

2. Architectural Description for EJB Component Assembly

Since interactions among EJB components are made by method invocations, the plug-and-play component assembly and the flexible reconfiguration of a system model are very difficult. Therefore, components' interaction should be defined at the abstract level, not at the implementation.

To abstract the definition of EJB components, architectural concerns should be implied in the component assembly and system development. The system architecture consists of components and connectors. Components communicate with each other by passing messages through connectors. For each component, internal methods and interfaces are defined. When a component gets a message from the outside through the interface, it performs the action stated in its behavior. This makes it possible for clients not to invoke a component's

internal method directly. Since components' interaction through the interface is implemented by the connector, it enables easy substitution and independent execution of components.

We described our system architecture for the EJB component assembly using ADL according to the C2 architectural style. The C2 architecture was developed by the University of California at Irvine. Figure 2 shows the block diagram that represents the C2 architectural style.

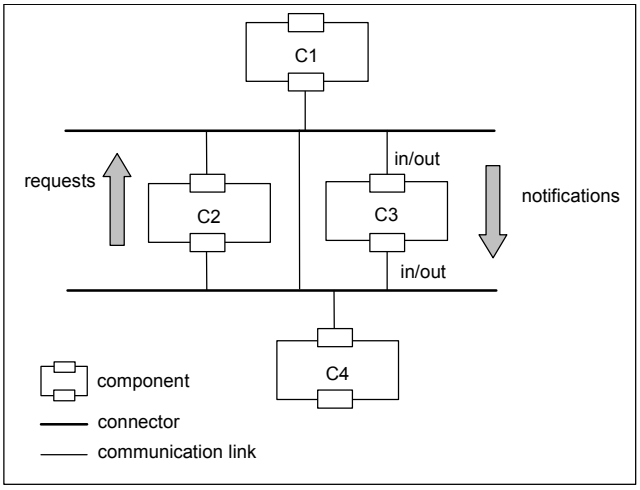

Fig. 2. C2 architectural style.

The C2 architectural style, as with most other styles of architecture, consists of components and connectors. The C2 components are connected to each other through connectors and have two ports: top and bottom ports. A C2 connector can have multiple top or bottom ports. The top of a component may be linked to the bottom of a single connector, and the bottom of a component may be linked to the top of a single connector.

When a C2 component receives messages incoming from the outside via the top or bottom ports, it may invoke its own methods and may also generate messages that go out to the outside via the top or bottom ports. There are two types of messages: a notification and a request. As shown in Fig. 2, a notification is sent downward through C2 architecture via the bottom out/top in port of a component, while a request is sent up via the top out/bottom in port. The style has no restrictions on the implementation language or the granularity of components and connectors. Its message-based communication simplifies the problem of control integration and facilitates interchangeability of components. These features make it possible to assemble EJB components by plug-and-play.

We redefined the C2SADL, which is an ADL developed by the University of California at Irvine, to achieve EJB components assembly based on architecture. Figure 3 shows the syntactic structure of our redefined ADL described in Unified Modeling Language [17].

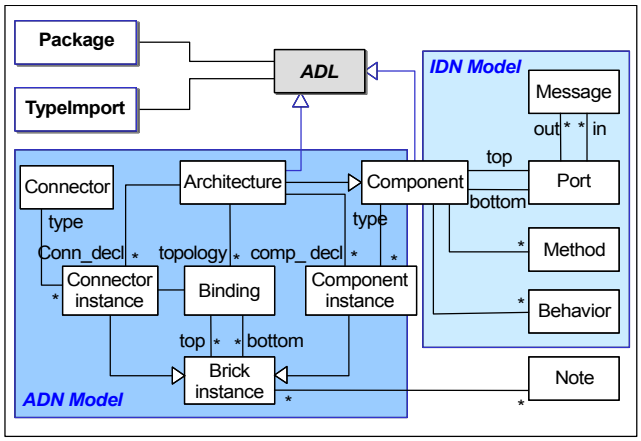

Fig. 3. Syntactic structure of the ADL.

As the figure shows, ADL consists of two types of notations: architecture description notation (ADN) and component interface definition notation (IDN). The two notations of the ADL, ADN and IDN, correspond to those of C2SADL. We specified the ADL in LL(1) grammar to support recursive-descent parsing. We redefined the C2SADL syntax and semantics to increase the expressive power of the EJB component specification. The redefined features of the ADL (IDN description) are as follows: 1) mapping relations between the formal parameters of received messages and the actual parameters of invoked methods or generated messages; 2) naming the component of an invoked method; 3) naming of the return value of an invoked method; 4) supporting conditional method invocation and message generation. Our ADL supports the mechanisms of packaging and importing Java programming language to our ADL.

3. Description of the System Architecture and Component Structure

To develop an application system by assembling components, a design phase is divided into two steps: one for the system architecture design, which shows the big picture of the system and the interaction of the components, and the other for the component structure design, which supports plug-and-play assembly and provides the basis for an implementation technique. The ADL mentioned in section III.2 describes the system architecture and component structure. The descriptions of these structures are used to generate the wrapper and glue code in the implementation phase of the component assembly.

A system architecture defines the complex interaction among components at the abstract level. Identified components and connectors can form a system. Components' interaction is

modeled to satisfy the system's requirements, and the operations of each component are defined. Through linking components to connectors, components' various interfaces are provided to the outside from the connector. The connector supports plug-and-play assembly by managing interfaces of components and bridging their mismatched interfaces. The system architecture is described by the Java-like ADL mentioned in section III.2. Figure 4 shows the syntax of the ADN, which consists of three system elements: components, connectors, and binding information.

```
<architecture> ::=
    <package>
    <import>
    architecture <arch_name> {
        ε |
        components {
            [ <context> <comp_name>
            <comp_inst_name_list> ; ]*
        }
        connectors {
            [ <conn_name> <conn_inst_name_list> ; ]*
        }
        topology {
            [ binding <conn_inst_name> {
                top = { <brick_inst_name_list> } ;
                bottom = { <brick_inst_name_list> } ;
              } ]*
        }
        notes {
            [ <brick_inst_name> = <note> ; ]*
        }
    }
```

Fig. 4. The syntax of ADN describing the system architecture.

The component structure defines the interaction of EJB components as a message-passing style. The EJB components' plug-and-play assembly cannot be done through a method invocation due to the hard-wired composition at the code level. For easy and flexible assembly, method invocations among components should be changed into message passing. The component structure defines a component's ports (i.e., a component's interfaces), methods, and behaviors at the abstract level according to the C2 architectural style [18].

For describing the component structure, the information on each EJB component's implementation, such as classes, interfaces, methods, and attributes information, should be referenced for identifying the component's messages and methods. The component's interface is made of 'request' and 'notify' messages. The component's method is identified from the EJB component's method signature. After defining the

```
<component> ::=
    <package>
    <import>
    component <comp_name> {
        ε |
        port top {
            out { <msg_decl_list> }
            in { <msg_ decl_ list> }
        }
        port bottom {
            out { <msg_ decl_ list> }
            in { <msg_ decl_ list> }
        }
        methods { <method_decl_list> }
        behavior {
            [ startup { <invoked_methods> <generated_msgs> } ]
            [ cleanup {<invoked_methods> <generated_msgs> } ]
            [ received <received_msg> { <invoked_methods>
              <generated_msgs> } ]*
        }
    }
    <method_decl_list> ::= ( <type> <m_signature_1> ; )*
    <m_signature_1> ::= <id_list> ( <param_list_1> )
```

Fig. 5. The syntax of IDN describing the component structure.

component's messages and methods, the component's behavior describes how to act according to messages received from other components. Startup, execution, and cleanup behaviors are defined as message flows and method executions, (Fig. 5).

While the system architecture design provides the basic framework for components' plug-and-play assembly, the component structure design acts as an intermediary for implementing the message-based component's interface.

4. Architectural Reasoning for EJB Component Assembly

Based on the descriptions of the system's architecture and a component's structure, the code is generated for component assembly. The wrapper code enables EJB components to pass messages for communications. The glue code enables EJB components to connect to each other for implementing the system. The code should be generated from complete architecture descriptions so that the implemented system runs in a deliberate way. Architectural reasoning ensures the communication integrity between the architecture and implementation.

The wrapper code is generated from a flawless component description. This means that messages defined at a top/bottom port should be shown in the component's behavior description, recognizing received messages as the starting point of the behavior. EJB component's interface signatures should be selected and used for describing the component structure

M_{in}: messages defined as 'Top/Bottom In' messages

M_{out}: messages defined as 'Top/Bottom Out' messages

R: received message G: generated message

$r \in M_{in}, g \in M_{out}, \forall r \in R, g \in G$

Fig. 6. Architectural reasoning between the component structure and its wrapper code.

conforming to the constraints of the C2 architectural style. Figure 6 shows the rule for generating the wrapper code corresponding to the component structure.

The glue code is generated from the sound components' interaction. Interactions defined as the message passing style should not be lost during implementation. This means that all declared request messages outgoing from the top port of a component are included in the set of incoming messages of all components connected above the component, and that all declared notification messages outgoing from the bottom port of a component are included in the set of incoming messages of all components connected below the component. If all outgoing messages from the top and bottom ports of a component are not included in the incoming messages of the components connected to the component, the architecture has type errors and does not generate the glue code during the implementation phase. Figure 7 shows the rule for checking the type error in the architecture.

C_i: i-th component, C_j: j-th component,

C_k: k-th component

$M(T_{in})_i$: Incoming messages from the Top port of a component, C_i

$M(T_{out})_i$: Outgoing messages from the Top port of a component, C_i

$M(B_{in})_i$: Incoming messages from the Bottom port of a component, C_i

$M(B_{out})_i$: Outgoing messages from the Bottom port of a component, C_i

C_i is connected to C_j and C_k at the lower level, $\forall C_i, C_j, C_k$

$M(T_{out})_j \subset M(B_{in})_i$ AND $M(T_{out})_k \subset M(B_{in})_i$

$M(B_{out})_i \subset (M(T_{in})_j \otimes M(T_{in})_k)$

Fig. 7. Architectural reasoning for the communication integrity.

The COBALT assembler has a model checker that verifies the consistency of the system. Components in the architecture communicate with others through connectors. The message from a component should be passed to at least one of the components connected to it by a connector. A loss of messages in the architecture is not allowed so that the generated code can ensure the sound interaction of EJB components.

5. Component Assembly and System Implementation

Descriptions of the system architecture and component structure are necessary to implement the component assembly. The wrapper code is generated from the component structure description. It helps a component support plug-and-play assembly. The glue code is generated from the system architecture description. It binds the system architecture of the C2 architectural style to EJB component implementation.

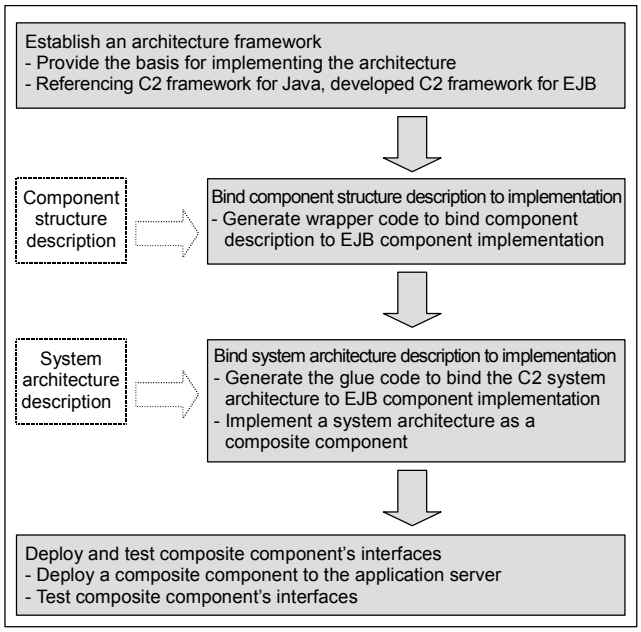

Fig. 8. Process of component assembly and system implementation.

Figure 8 shows the detailed process of the component assembly and system implementation; we explain each step below.

1) Establish an architecture framework—A framework is necessary to bind the design phase and implementation phase. This paper references and modifies the C2 framework for the implementation of EJB components. It provides elements, such as a component, a connector, component's interfaces, a method, a behavior, etc., to be implemented in the architecture.

2) Bind a component structure description to the implementation—The wrapper code is generated from the component structure description to make components independent and support plug-and-play assembly. It binds the component structure of the design phase to the reusable component, converting the component behavior from the message invocation style to the message passing style.

3) Bind a system architecture description to the implementation—The glue code is generated to bind EJB components to the C2 based system architecture. Using the C2

framework, a composite component is generated corresponding to the system architecture. The composite component enables component users to reuse the architecture itself as a new component.

4) Deploy and test a composite component's interfaces—A composite component that is generated in step 3 can be deployed to the application server as a new EJB component. Its operation corresponds to the system architecture, and its interfaces are tested for the system validation.

6. Structure of the COBALT Assembler

This section shows the structure of the COBALT assembler. The COBALT assembler supports the architecture design and implementation of EJB components' plug-and-play assembly. Before the system architecture is designed, information about the domain and user requirements are required. Components for the system development are identified by domain modeling, and identified components make up the application system architecture. The COBALT assembler assumes that the developer has already analyzed the system requirements and selected candidate components for the system implementation.

Components identified from the domain analysis are assembled at the abstract level using an architecture editor. Commercial off-the-shelf EJB components can be assembled by drag-and-drop in the architecture editor. Components and connectors are bound to each other. The topology information of the architecture is saved in the system architecture description file. A model checker verifies whether the architecture is well defined according to communication integrity within the constraints of the C2 rule. An architecture model is manipulated by the model checker in two kinds of internal representations: syntactic and semantic models. The syntactic model represents the parse tree of the architecture model as a result of syntactic analysis. The semantic model is generated for use in style checking such as communication integrity.

A component information analyzer provides the component's characteristics. It analyzes the deployment descriptor file of an EJB component and provides useful information to describe the component structure. Information such as operations and properties is used for describing the component structure using a component specification editor. The description file of a component structure is generated after the component specification editor edits the component's interfaces, methods, and behaviors. The model checker analyzes and checks the component structure description, as in the architecture description.

We developed a new application system using a system architecture description file and a component structure description file. A wrapper generator creates the wrapper code using the component structure and component implementation information. The wrapper code replaces method invocations of the EJB component with messages passing through the top and bottom interfaces. The message handling logic of the wrapper code is formed from the behavior description of the component structure. Composite component/glue code generators create code for binding EJB components to the C2-based system architecture. A new composite component is a new stateless session bean which is formed by a combination of messages included in the bottommost component of the architecture. The composite component provides the functionality of the whole architecture. The composite component is deployed to the application server and its interfaces are tested [19], [20].

Figure 9 shows the relationships between subblocks of the COBALT assembler and their artifacts. For the component assembly and system development, several artifacts are generated: the architecture description, the wrapper code, the glue code, etc. Those artifacts are generated by submodules of the COBALT assembler.

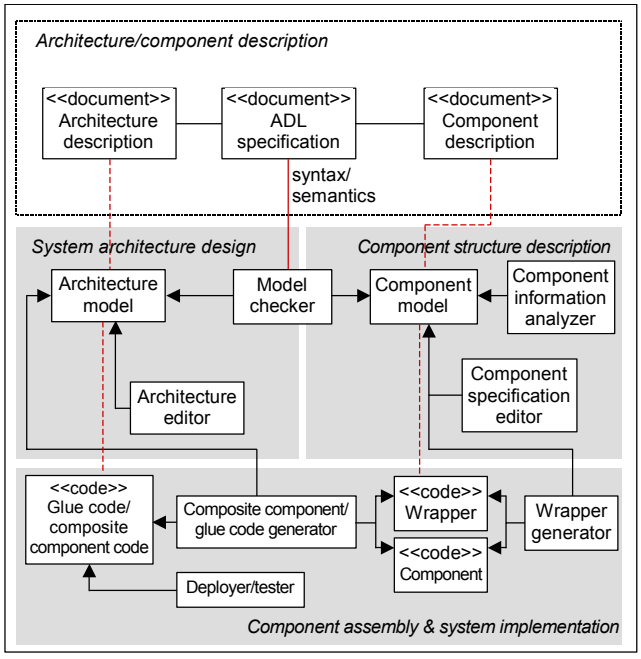

Fig. 9. Subblocks of COBALT assembler and their artifacts.

IV. Case Study Using Shopping Mall Components

To evaluate the method of EJB component assembly and its implementation with the COBALT assembler, we did a case study on a shopping mall system. We identified seven components for the system development by EJB component assembly. The components' functions are as follows.

– Mail: sends or receives the mail.

- Category: manages the category information of the product.
- Member: manages customer information.
- Payment: manages customer's payment information.
- Company: manages the information of the third party company providing shopping mall products.
- Product: manages the product information.
- Order: manages product orders.

1. System Architecture Design Using the COBALT Assembler

The first step of the system architecture design is to model the interactions of the components. The interfaces of the components are defined through modeling the components' interactions, and a connector can manage the interfaces. Figure 10 describes the system architecture using the COBALT assembler.

The COBALT assembler provides the graphical and textual editors to build the system architecture. To assemble EJB components, the tool user can drag and drop the EJB components (EJB Jars) from the composite palette, which browses EJB jars in the directory and shows the files in it.

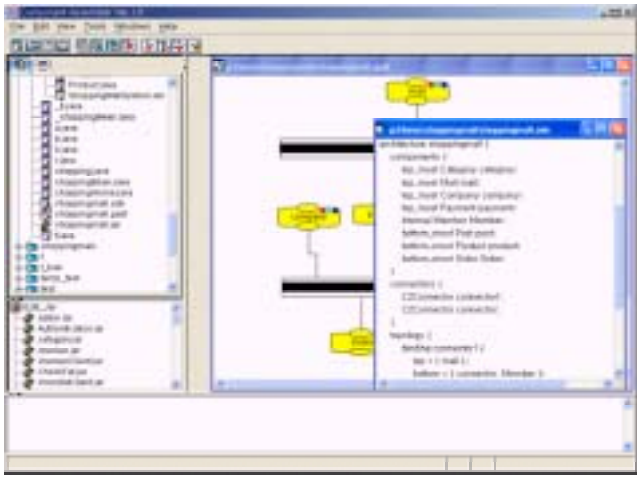

Fig. 10. System architecture of the shopping mall system described using COBALT assembler.

2. Component Structure Description Using the COBALT Assembler

A component structure is built using the component specification editor. Information about a component's characteristics is extracted from the component information analyzer. For users' convenience, the component specification editor automatically converts the component's methods to interfaces of the message type. The EJB component is described in the C2 architectural style. It has top and bottom

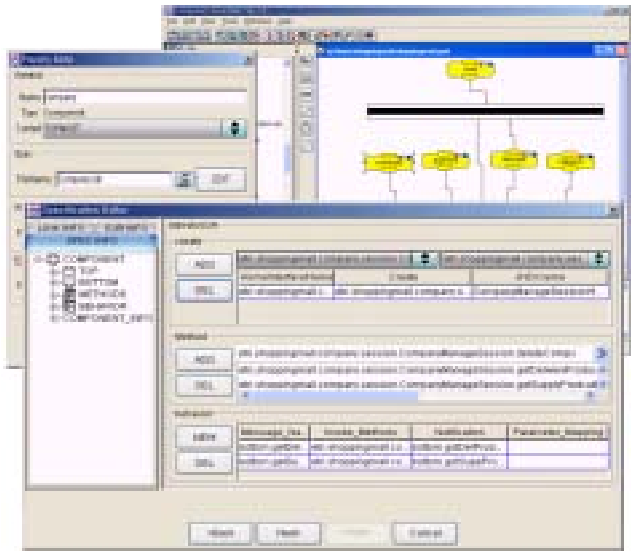

Fig. 11. Company component's structure description using COBALT assembler.

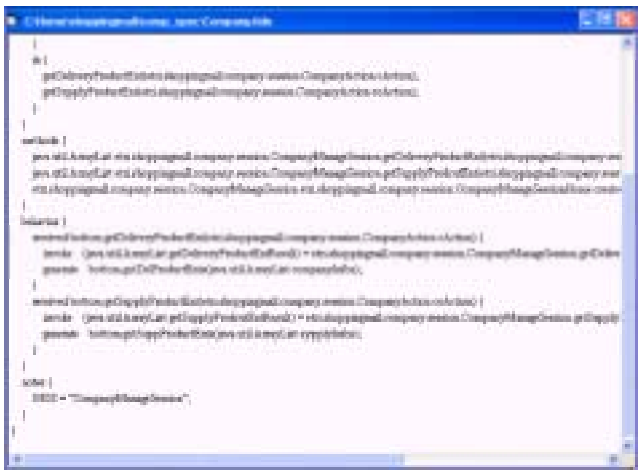

Fig. 12. Company component's structure file generated by COBALT assembler.

interfaces, methods, and behavior. A component's behavior is formed from its incoming messages. Figure 11 shows the process of the component specification editing to define the 'company' component's behaviors. Figure 12 shows the generated component structure file.

3. Assembly and System Implementation of Shopping Mall Components Using the COBALT Assembler

Using the system architecture and component structure descriptions, the wrapper and glue code is generated. The seven components we used in this case study were not implemented according to the message passing mechanism, so the wrapper was needed for solving this problem. A component interacts with other components by passing messages through

the wrapper. The composite component is generated to reuse a system architecture. The glue code is generated to bind the system architecture to the seven components. Using the architectural reasoning mentioned in section III, the COBALT assembler checks whether the messages defined in the component structure are used in its behavior description and the system architecture before generating files. This validation is an important processing ensuring that the implemented files comply with architectural constraints.

Figure 13 shows the validated description of the company component. Its 'bottom in' message should be described in the 'product' component's description and should be found in the company component's behavior. Its methods can be found in its behavior.

4. Evaluation

We developed a shopping mall system with the COBALT assembler. Seven of the components were prebuilt EJB components that have binary code. We successfully assembled seven components giving attention to architectural concerns. We firstly designed the shopping mall system with seven components according to the C2 architectural style and generated the wrapper code and the glue code. The composite component could be reused as a single new EJB component. However, there were several difficulties in applying the COBALT assembler to the case study. Firstly, it was difficult to decide how to build an architecture using components and to determine which component should be connected to the top port of another component. We overcame this difficulty with the help of several experts on system architecture. Secondly, it was also hard to determine the methods required for the assembly. Although a component is distributed with its specification, it was not easy to understand the full functionality of the component. For this difficulty, the COBALT assembler provides the component information analyzer to help the user understand a component's properties. In addition to the component information analyzer, we found it necessary to use a component tester for simulating component functions during the assembly.

V. Conclusion and Future Work

To facilitate component reuse, this paper described the

```
package comp_spec;
component Company {
   port bottom {
      out {
         gotDelProductEnts(java.util.ArrayList companyInfos);
         gotSuppProductEnts(java.util.ArrayList supplyInfos);
      }
      in {
         getDeliveryProductEnt(CompanyAction cAction);
         getSupplyProductEnt(CompanyAction coAction);          // Product component's top out message
      }
   }
   methods {
      java.util.ArrayList getDeliveryProductEnt (CompanyAction m_0);
      java.util.ArrayList getSupplyProdcutEnt (CompanyAction m_0);     // Company component's method
      CompanyManageSession CompanyManageSessionHome.create();
   }
   behavior {
      …
      received bottom.getSupplyProductEnt (CompanyAction coAction) {
         invoke (java.util.ArrayList getSupplyProdcutEntResult) =
                        getSupplyProdcutEnt(CompanyAction m_0);
         generate bottom.gotSuppProductEnts(java.util.ArrayList supplyInfos);
         }
   }
}
```

Consistency checking

Consistency checking

Fig. 13. Valid information of company component's structure by COBALT assembler.

COBALT assembler, which supports flexible and plug-and-play EJB component assembly and system development based on the architecture. The COBALT assembler enables users to design the system and its components according to the C2 rule, generate the wrapper and glue code for the plug-and-play assembly, and create a new composite EJB component to reuse the assembled EJB components. In addition, the shopping mall components were applied to the COBALT assembler.

Our case study confirmed that component assembly by plug-and-play can be achieved by considering architecture. In other words, the architecture design is mapped to reusable components and their relationships are kept through the implementation. The shopping mall system architecture is defined at the abstract level and the interactions of EJB components are changed from a method invocation style to a message passing style. The wrapper and glue code is automatically generated to support plug-and-play assembly. The completeness and consistency of the system are checked according to the extended C2 rule. If the system is built based on the architecture at the abstract level, the extensibility and maintainability can be leveraged. To extend the system's functionality for searching for an address by the zip code, an additional EJB component, the 'post' component, can be easily linked to the system using a connector. In conclusion, this paper described how to apply architectural considerations to a component assembly and introduced its supporting tool, the COBALT assembler.

In terms of research perspectives, the COBALT assembler will strengthen the ability of architectural reasoning by checking with an extended ADL. ADL for system architecture needs to be revised to describe the system more precisely. The connector should manage messages coming from components as a coordinator. ADL should support the connector's message management. Furthermore, ADL needs to express dynamic changes of the architecture. The wrapper and glue code needs to be changed according to the architecture description. The component's information needs to be analyzed in detail so that the component's structure can be defined in a rich format. In addition to these revisions, the COBALT assembler will integrate a component identifier that identifies components through a domain modeling included in the COBALT constructor. The COBALT constructor has been developed together with the COBALT assembler in order to support component identification, design, implementation, packaging, deployment, and testing. This result will be the base for the new version of the COBALT assembler that supports a plug-and-play component assembly in the various platforms such as .NET and CORBA.

References

[1] Sun, *Designing Enterprise Applications with the Java™ 2 Platform*, Enterprise Edition, Version 1.1, Mar. 2001.

[2] Object Management Group, *CORBA Components*, http://www.omg.org, Mar. 1999.

[3] J. Andersson and P. Johnson, "Architectural Integration Styles for Large-Scale Enterprise Software Systems," *Proc. of 5th Int'l Enterprise Distributed Object Computing Conf.*, 2001, pp. 224-236.

[4] I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, Addison Wesley, 1997.

[5] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison Wesley, 1998.

[6] B. Felix, B. Len, B. Charles, C.D. Santiago, L. Fred, R. John, S. Robert, and W. Kurt, *Technical Concepts of Component-Based Software Engineering*, Pittsburgh, PA, Software Engineering Institute, Carnegie Mellon University, 2000.

[7] WebMethods, "Achieving Global Business Visibility with the webMethods Integration Platform: a Technical White Paper," Mar. 2002.

[8] R. Sharma, B. Stearns, T. Ng, and S. Dietzen, *J2EE Connector Architecture and Enterprise Application Integration*, Addison Wesley, 2002.

[9] R.S. Moreira, G.S. Blair, and E. Carrapatoso, "A Reflective Component-Based and Architecture Aware Framework to Manage Architecture Composition," *Proc. of 3rd Int'l Symp. on Distributed Objects and Applications* (DOA 2001), 2001, pp. 187-196.

[10] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor, "A Language and Environment for Architecture-Based Software Development," *Proc. of the 21st Int'l Conf. on Software Engineering* (ICSE 21), Los Angeles, CA, May 1999, pp. 44-53.

[11] N. Medvidovic, P. Oreizy, and R.N. Taylor, "Reuse of Off-the-Shelf Components in C2-Style Architectures," *Proc. of the Symp. on Software Reusability* (SSR'97), Boston, MA, May 1997, pp. 190-198.

[12] N. Medvidovic and R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Language," *IEEE Trans. Software Engineering*, vol. 26, no. 1, Jan. 2000.

[13] M. Moriconi, X. Qian, and R.A. Riemenschneider, "Correct Architecture Refinement," *IEEE Trans. Software Engineering*, vol. 21, no. 4, Apr. 1995, pp. 356-372.

[14] D.S. Rosenblum and R. Natarahan, "Supporting Architectural Concerns in Component-Interoperability Standards," *Proc. of IEE Software*, vol. 147, no. 6, Dec. 2000, pp. 215-223.

[15] Z.W. Hong, J.M. Lin, H.C. Jiau, and D.S. Chen, "DSIAS: a Software Architectural Style for Distributed Software Integration Systems," *25th Annual Int'l Computer Software and Applications Conf.*(COMPSAC 2001), 2001, pp. 291-296.

[16] W.J. Lee, O.C. Kwon, M.J. Kim, and G.S. Shin, "A Method and Tool Support for Identifying Domain Components Using Object Usage Information," *ETRI J.*, vol. 25, no. 2, Apr. 2003, pp. 121-132.

[17] D.I. Shin, S.W. Nho, T.W. Jeon, and S. Lee, "The Design and Implementation of an ADL Model Checker," *Int'l J. of Computer and Information Science* (IJCIS), vol. 4, no. 2, June 2003.
[18] R.N. Taylor, N. Medvidovic, and etc., "A Component and Message Based Architectural Style for GUI Software," *IEEE Trans. Software Engineering*, vol. 22, no. 6, June 1996, pp. 390-406.
[19] Y.H. Choi, O.C. Kwon, and G.S. Shin, "An Approach to Composition of EJB Components Using C2 Style," *Proc. of the EUROMICRO*'02, Sept. 2002.
[20] S. Lee, O.C. Kwon, and G.S. Shin, "COBALT Assembler: a Case Tool for Supporting EJB Component Assembly Based on Architecture," *KIPS*, June 2002, pp. 32-38.

Seungyun Lee received the BS and the MS degrees in computer science from Sogang University, Korea, in 1999 and 2001. She has been a Member of Engineering Staff in Computer & Software Laboratory at ETRI (Korea Electronics and Telecommunications Research Institute) since 2001. She is currently involved in developing software architecture based component technology and model driven architecture (MDA) based software development technology. Her current research areas include software architecture, component assembly and model-driven architecture.

Oh-Cheon Kwon received the BA degree from Inha University, Korea, in 1985, and the MS degree in software engineering from the University of Teesside, England, in 1994, and the PhD degree in computer science from the University of Durham, England, in 1998. He worked for SERI (Systems Engineering Research Institute)/KIST (Korea Institute of Science and Technology) from 1985 to 1997. He has been a Principal Researcher for ETRI since 1998. He was also a Visiting Researcher at IBM/RTP, North Carolina, USA, in 1991. He is currently involved in developing an Integrated Management System of Satellite Imagery Information. He has served as the editor of Transactions of the Korea Information Processing Society, and the assessor of qualifying the new S/W technology (KT Mark) sponsored by the Ministry of Science and Technology of Korea (MOST). His research interests include remote sensing, GIS, telematics, component-based development (CBD), and model driven architecture (MDA).

Gyu-Sang Shin received the BS degree in statistics from Sung Kyun Kwan University, Korea, in 1981, and the MS degree in statistics from Seoul National University, Korea, in 1983, and the PhD degree in computer science from Chungnam National University, Korea, in 2001. He worked for Systems Engineering Research Institute (SERI), Korea, as a researcher between 1983 and 1996. He has been a Principal Researcher for ETRI (Korea Electronics and Telecommunications Research Institute) since 1997. He has been engaged in the development of component-based development tool, real-time operating system, video streaming server, and object-oriented CASE tool. He is currently involved in developing software architecture based component technology and MDA (model driven architecture) based software development technology. His research interests include component-based software engineering, model driven software development, CASE tool and multimedia applications.