Towards Software Architecture at Runtime

Authors Names
Department of Computer Science and Technology
Peking University, Beijing, PRC, 100871
+86 10 62757801-1
{ }@ cs.pku.edu.cn

Abstract

1. Introduction

Software architecture (SA) is the gross structure of a software system as a collection of components, connectors and constraints [Perry 92, Shaw 96]. As a bridge between requirements and implementations, SA simplifies the comprehension of large systems, supports reuse at multiple levels, guide the construction of the systems, reveals the dimensions of the evolution, and facilitates analysis and management [Garlan 00].

Currently, almost all of the research and experiment focus on SA at design phase. Such condition makes it hard to achieve the architecture-based engineering [Garlan 00], i.e., to develop, manage and evolve software systems based on SA systematically. At the same time, the fact that Internet and wireless network become the dominated runtime environment makes software systems much more dynamic and open. The architectures of such systems may be frequently restructured at runtime, which need to deal with components evolved dynamically [Garlan 00]. Thus it is necessary to research SA at runtime. There is few work on SA at implementation and runtime. ArchJava extends Java to code architectural information in the implementation and enforce communication integrity [Aldrich 02]. [Oreizy 98] studies how SA can support corrective, perfective and adaptive evolution at runtime and experiments on C2, a layered, event-based architectural style. [Rosenblum 00] investigates the architectural concerns in the component interoperability framework and combines the JavaBean model with C2 style.

This paper presents a novel technique, called Runtime Software Architecture (RSA), which has close relation with SA at design phase (DSA), describing the structure of the system more concretely and accurately. RSA can keep consistency and traceability of SA between the development and runtime and help to manage and evolve SA at runtime. This paper addresses three fundamental issues towards RSA:

- In order to represent and manipulate RSA at runtime, RSA should be incarnated as a runtime entity, which encapsulates all of the architectural information of the runtime system and allows access and modification of such data.
- In fact, the data encapsulated in RSA is retrieved from some part of the states of the runtime system. And the modification of RSA is to modify the states and thus change the execution semantics of the runtime system. So there exists casual-connection between RSA and runtime systems, i.e., the changes made in RSA are immediately mirrored in the actual state and behavior of the runtime system. In a word, the casual-connection enables the management and evolution based on RSA.
- RSA should be integrated with DSA to introduce current achievements and experiences of SA from the design phase into runtime, such as formal notations, graphical representation, reason and validation. Moreover, such integration facilitates to apply SA technology to the systems that do not take architecture as an explicit design activity in the development.

The rest of the paper is organized as follows. Section 2 incarnates RSA through defining what data encapsulated in RSA and how to manipulate RSA. Section 3 describes what is and how to implement the casual-connection between RSA and the runtime system. Section 4 presents the tool for RSA and how to integrate RSA and DSA. Section 5

discusses related work and Section 6 summarizes the contributions and identifies the future work of the paper.

2. Incarnation Of RSA

It is a natural way to incarnate RSA in Object-Oriented paradigm, i.e., to encapsulate architectural data in a set of objects and investigate operations on these data.

2.1 Data Encapsulated in RSA

As DSA contains the main part of RSA and to keep the consistency, it is reasonable to define the content of RSA based on DSA. In order to represent and analyze architectural design, an ADL (Architecture Description Language) provides formal notations [Garlan 00], which usually define the complete contents of SA. Based on ABC/ADL, the ADL of the Architecture-Based Component Composition (ABC), some classes are derived to incarnate RSA, shown in Figure [*]. In fact, the elements of ABC/ADL have the similar definitions with those of the ADL Classification and Comparison Framework presented in [Medvidovic 00]. And the details of ABC/ADL can be found in [Mei 02]. The incarnation of RSA includes the following entities:

- *RuntimeSoftwareArchitecture*: A RSA consists of the definitions of a set of components, connectors and architectures.
- *ComponentDef* and *ConnectorDef*: A component or connector can be defined from five aspects. The Interface Aspect defines players that incorporate the instance-method, because we notice that some methods are bound to the component type and may be not executed by any instance, such as creation of instances, while others are executed by instances. The Attribute Aspect designates the attributes the component will use in the interaction with others. The Property Aspect describes additional information of the component, e.g. security, version, throughout limit. Most of the excess architectural information of RSA over DSA is represented in the Property Aspect of the component, connector and architecture. The Dependency Aspect describes the relationship of dependency between the players. And the Semantic Aspect uses formal methods to model, or, at least, use natural language to describe, the behaviors and features of the elements.
- *Architecture*: An architecture (it is also called architectural configuration in [Medvidovic 00]) is a group of interconnected component and connector instances that comply with the constraints of architectural styles. It consists of the declarations of all instances of components and connectors used in the system and the topologic layout of these instances.

2.2 Operations on RSA

According to the characteristics of the encapsulated data, the operations provided by RSA can be categorized into the following:

- Access to data: All of the data encapsulated in RSA can be accessed through the operations named as

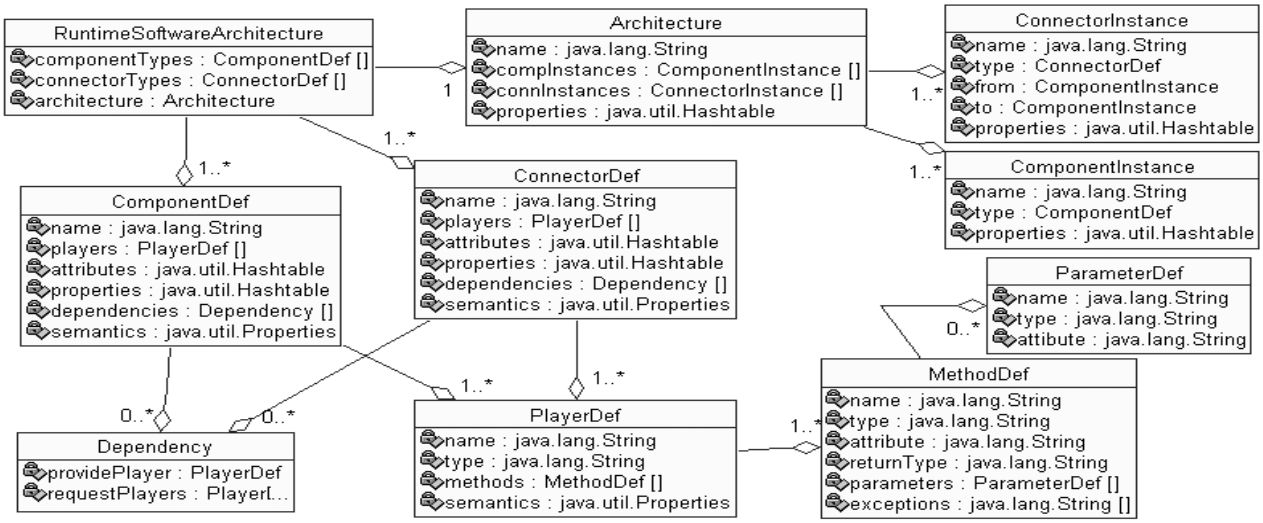

Fig. 1 Classes to incarnate the Runtime Software Architecture in UML

type of the player (provide or request) and several method specifications, i.e., *MethodDef*. There are two kinds of methods in every player: type-method and "*get<Data>*", such as *RuntimeSoftwareArchitecture. getArchitecture().getConnInstances()* to explore all of the connector instances.

- Modification of the data without changing the state of the runtime system: Some data of RSA is derived from DSA to keep consistency and cannot be retrieved from the runtime system, such as the Semantic Aspect. And RSA may modify such data to refine DSA, such as describing the semantics of the players more concretely and accurately.
- Modification of the data that leads to change the state of the runtime system: Almost all of the data encapsulated in RSA is casually connected with the state and behavior of the runtime systems. For example, when changing the name of the implementation class of a component as *ComponentDef.properties.put("ejb-class","...")*, the current implementation class will be replaced by the new one identified by the second parameter.
- Invocation of the instances of components: It allows to invoke one of the *Architecture.compInstances* with any methods defined in the *PlayerDef.methods*.

In practice, the above operations require the ability of casual-connection provided by the runtime system.

3. Casual-Connection between RSA and PKUAS

In order to keep consistency between RSA and the underlying runtime system through the casual-connection, it has to explore which states of the runtime system can be casually connected with the data encapsulated in RSA and what behavior of the runtime system can be casually connected with the operations on RSA.

3.1 Synopsis of PKUAS

PKUAS is an open J2EE-compliant application server, implementing all of the functionalities specified in [EJB 1.1] and the Local Interface in [EJB 2.0]. Shown as Figure [*], the internal functionalities of PKUAS are modularized as follows:

- Container System: A container is the runtime space for components deployed by applications, managing their lifecycle and runtime contexts [J2EE 1.2]. PKUAS implements EJB containers for stateless session bean, stateful session bean and bean-managed entity bean. One instance of a container holds all instances of one EJB. And a container system consists of the instances of the containers holding all EJBs in an application. Such organization of the containers facilitates the configuration and management specific to individual applications, such as security realm per application and architectural states of the application.

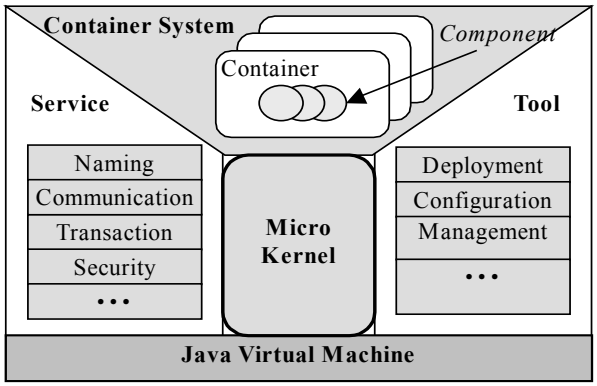

Fig. 2 Pluggable Architecture of PKUAS

- Service: It provides the common functions, e.g., naming, communication, security and transaction.
- Tool: it provides functions to facilitate the operation of PKUAS, i.e., deployment and management.
- Micro kernel: It provides a registry for the above modules and other management functions, e.g., class loading, relation, timer and monitor.

In the above entities, EJBs are the components in RSA, and the container holds the states related to the component data in RSA. The communication service provides connectors and holds the states related to the connector data in RSA. The services except communication hold the states related to the component properties, such as method permission and transaction attribute. And the container system holds the states related to the architecture data in RSA.

3.2 States Casually Connected with RSA

All of the states of PKUAS are derived from the deployment package, configuration and runtime context of the runtime system. Typically, the *ComponentDef* is casually connected with the states derived from the deployment package, the *ConnectorDef* is casually connected with the states derived from the configuration of the communication service configuration, and the *Architecture* is casually connected with the states derived from the runtime context and the configuration of the services except communication.

3.2.1 States Casually Connected with *ComponentDef*

A J2EE application is packaged and deployed as an archive with suffix of ".war", ".ear" or ".jar". Such archives typically contain interfaces and implementations of EJBs and deployment descriptors of EJBs and the application. The deployment descriptor describes the structure of EJBs and their external dependencies and the application assembly information, specifying how to compose individual EJBs into an application [EJB 1.1]. Most of the states derived from the deployment package is hold and operated by the EJB containers.

Table. 1 Casual-connection between RSA data and PKUAS states derived from the deployment package

RSA Data	Elements in Deployment Package
Name of *ComponentDef*	<ejb-name> in <module>
Name of the provide player of *ComponentDef*	<home> and <remote> or <local-home> and <local> in <session> or <entity>
Name of the request player of *ComponentDef*	<home> and <remote> in <ejb-ref>; <local-home> and <local> in <ejb-local-ref>
Attributes of *ComponentDef*	<env-entry>, <resource-ref>, <cmp-field> and <primkey-field>
Properties of *ComponentDef*	<ejb-class>, <session-type>, <persistence-type>, <prim-key-class>, <transaction-type>, <reentrant>, <security-role-ref>, <security-role>, <method-permission>
MethodDef of the provide player of *ComponentDef*	They can be easily retrieved from the interface classes of EJBs through the reflective mechanism of Java language, i.e., the JDK package of java.lang.reflect.

The *MethodDef* of the request player cannot be retrieved from the states derived from the deployment package. The data is very important to reason about RSA because component A can provide a player for another component B (or A can be connected to B through a connector) if and only if the *MethodDef* of the provide player of A contains the *MethodDef* of the request player of B. But the *MethodDef* of the request player can be retrieved from the data encapsulated by the log service, which can record all of the messages received or sent by EJBs. The outgoing messages contain the name and remote reference of the target EJB and the details of the invoked method. When all of the use cases of the runtime application are executed at least once, all of the *MethodDefs* of the request players are retrieved completely.

3.2.2 States Casually Connected with *ConnectorDef*

There have three typical connector types for EJBs, including RMI-IIOP (Remote Method Invocation – Internet Inter-ORB Protocol), RMI-JRMP (Jave Remote Method Protocol) and EJBLocal. RMI is the default distributed object model of Java. The RMI-IIOP supports interactions between EJBs and other CORBA-compliant systems. And the RMI-JRMP support interactions between EJBs and other RMI-compliant Java systems. As RMI supports interactions between EJBs deployed in the same or different JVM (Java Virtual Machine) with the pass-by-value semantics, the EJBLocal supports interactions between EJBs collocated in the same JVM with the pass-by-reference semantics and much lower cost. Figure [*] shows the specification of RMI-IIOP and EJBLocal in ABC/ADL, which is discussed in the next section. In this specification, the use of "*" in the player definitions denotes that the player's methods are the same as the component player that connects to it.

```
Connector RMIoverIIOP is OO.Connector {
  Interfaces {
    provide player Callee is Connector.Callee {*}
    request player Caller is Connector.Caller {*}
  }
  Properties {
      WireProtocol = iiop;
      Version = 1.2; //version of IIOP
       ◦ ◦ ◦
}
Connector EJBLocal is OO.Connector {
  Interfaces {
    provide player Callee is Connector.Callee {*}
    request player Caller is Connector.Caller {*}
  }
  Properties {
      WireProtocol = javaLanguageDefault;
      Version = 2.0; // version of EJB
       ◦ ◦ ◦
}
```

Fig. 3 RMIoverIIOP and EJBLocal specified in ABC/ADL

3.2.3 States Casually Connected with *Architecture*

In the application deployed in PKUAS, the name of *Architecture* is equal to the name of the deployment package. And one *ComponentDef* retrieved from the deployment package must have the corresponding *ComponentInstance* with its multiplicity casually connected with the state in the EJB containers.

The <ejb-ref> and <ejb-local-ref> in the deployment descriptor identify the connector instances between two components. And the type of the connector instance is casually connected with the type of the communication proxy at the client side. For example, if EJB *A* is invoked by EJB *B* through RMI-IIOP and by EJB *C* through RMI-JRMP, there have two connector instances, i.e., *B2A* with the type of *RMIoverIIOP* and *C2A* with the type of *RMIoverJRMP*.

Most of the *Properties* are casually connected with the states derived from the configuration of the services except communication, such as the security realm and naming space.

3.3 Behavior Casually Connected with RSA

In the four types of the operations on RSA, only the modification of the data that leads to change the state of the runtime system is casually connected with behavior of the runtime system. These data is typically encapsulated either in the

ComponentDef and *ComponentInstance* or in the *ConnectorDef* and *ConnectorInstance*.

3.3.1 Behavior Casually Connected with Components

In order to add or remove functions other than the standard functions defined in [EJB 1.1] of the contain, we employ the design pattern of interceptor to dynamically insert extra functions before and after the invocation of the implementation of EJBs. Typically, the services used by the container should have corresponding interceptors, such as security interceptor, log interceptor and transaction interceptor, shown in Figure [*]. Moreover, the instance

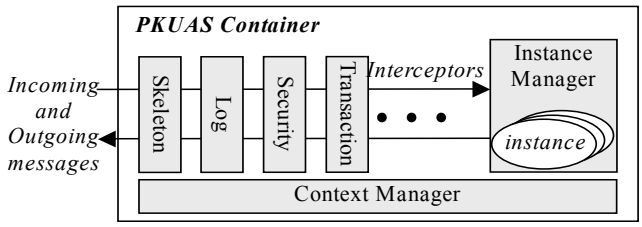

Fig. 4 Structure of PKUAS Container

manager is responsible for the lifecycle management of EJBs, such as creating, releasing, passivating and activating the instances. The context manager holds all of the data related to EJBs, including the states derived from the deployment package and runtime context.

The modification of the *ComponentDef* and *ComponentInstance* will make PKUAS perform the following behavior:

- To add or remove *ComponentDef* will make PKUAS add or remove the corresponding EJBs.
- To modify the value of the "*ejb-class*" in *Properties* of *ComponentDef* will make PKUAS update the corresponding EJBs with new implementation classes, including blocking the incoming invocations, buffering the old instances until its execution or transaction is over, creating new instances of the new implementation and copying the unchanged attributes from the old instances.
- To modify the value of the elements in *Properties* of *ComponentDef* that relate to the usage of the services will make PKUAS add, remove or update the corresponding interceptors. For example, to change the "*transaction-type*" will change the behavior of the transaction interceptor.
- To modify the value of the elements in *Properties* of *ComponentInstance* will make PKUAS reconfigure the corresponding containers, such as the size of the instance pool and the rule to passivate the instances.

More details of adding, removing, updating and reconfiguring EJBs and interceptors in PKUAS can be found in [Wang 02]. And the modification of the *PlayerDef*, i.e., to update the interfaces of EJBs, is under development.

3.3.2 Behavior Casually Connected with Connectors

The communication service of PKUAS is implemented with a flexible interoperability framework that can add, modify and delete communication protocols embedded in PKUAS, shown in Figure [*]. PKUAS interoperability

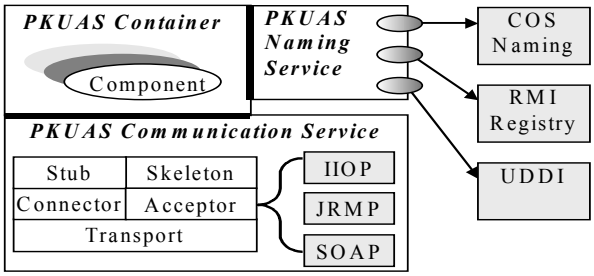

Fig. 5 PKUAS Interoperability Framework

framework separates the interoperability aspect from the container in order to enable EJBs to interoperate with other components through multiple protocols, such as IIOP, JRMP and SOAP. The stub and skeleton are responsible for the transformation between the invocations of RMI and the messages of the underlying interoperability protocol. The connector and acceptor transform the messages between the interoperability protocol and the underlying transport protocol and manage the connections. The transport is responsible for sending and receiving messages through the underlying transport protocol. And PKUAS naming service can publish and retrieve different interoperability addresses with the integration of the naming services specific to the interoperability protocols. Currently, PKUAS allows EJBs interoperate with other components through IIOP, JRMP, SOAP and EJBLocal.

Then, the modification of the *ConnectorDef* and *ConnectorInstance* will make PKUAS perform the following behavior:

- To add or remove the *ConnectorDef* will make PKUAS reconfigure the communication service with adding or removing the corresponding interoperability protocol. Moreover, the deletion of the *ConnectorDef* succeeds only when there is no *ConnectortInstance* of such type.
- To add the *ConnectorInstance* will make PKUAS allocate a new acceptor to the "*to*" component and publish the interoperability address of the new acceptor into the naming service. Then the "*from*" component can invoke the "*to*" component through the new connector. This operation is very useful in the integration or evolution. For example, to add a new *ConnectorInstance* as the type of RMI-JRMP allows the legacy Java program to invoke an EJB, and to add

a new *ConnectorInstance* as the type of RMI-SOAP enables an EJB to act as a Web Service.

- To remove the *ConnectorInstance* will make PKUAS reclaim the corresponding acceptor from the "*to*" component and remove its interoperability address from the naming service. Then the "*from*" component cannot invoke the "*to*" component through the corresponding connector any more.
- To modify the value of the elements in *Properties* of *ConnectorInstance* will make PKUAS reconfigure the corresponding acceptor or connector, such as the size of the message buffer, the transport semantics and the maximum number of the concurrency connections.

4. Integration of RSA and DSA

The integration between RSA and DSA means that RSA retrieved from the runtime system can be represented and analyzed in the architecting tools, and at the same time, SA constructed at design phase can facilitate the retrieval of RSA. In fact, the major challenge of the integration is the transformation between SA specified in ADL and the data encapsulated in RSA.

4.1 ABC/ADL and ABCTool

ABC/ADL is the architecture description language of the Architecture-Based Component Composition (ABC), which employs SA descriptions as frameworks to develop components as well as blueprints for constructing systems, while using middleware as the runtime scaffold for component composition [Chen 02]. Figure [*] shows part of the architecture of eShop, the sample J2EE application discussed in the next section, specified in ABC/ADL. We also provide the ABCTool to support the design of SA in ABC/ADL, shown in Figure [*]. More details of ABC/ADL and ABCTool can be found in [Mei 02].

```
Component ShoppingCart is J2EE.EJB {
  Interfaces {
    provide player ShoppingCartHome is EJB.Home {
      instance-method{    ShoppingCart create();
        。 。 。  }}
    provide player ShoppingCart is EJB.Object {}
    request player OrderHome is J2EE.EJB.Home {};
      。 。 。
  }}
Component Order is J2EE.EJB{}
 。 。 。
Architecture eShop_Architecture{
   uses{
     Component ShoppingCart : sCarts[];
     Component Order : orders[];
      。 。 。
      Connector RMIoverIIOP : shoppingCartToOrderHome[];
      。 。 。
      Variable i : int;
   }
   config main {
     sCarts[i].OrderHome connects
                shoppingCartToOrderHome[i].Callee;
     shoppingCartToOrderHome[i].Caller connects
                orders[i].OrderHome;
    。 。 。
}}
```

Fig. 6 Description of eShop in ABC/ADL

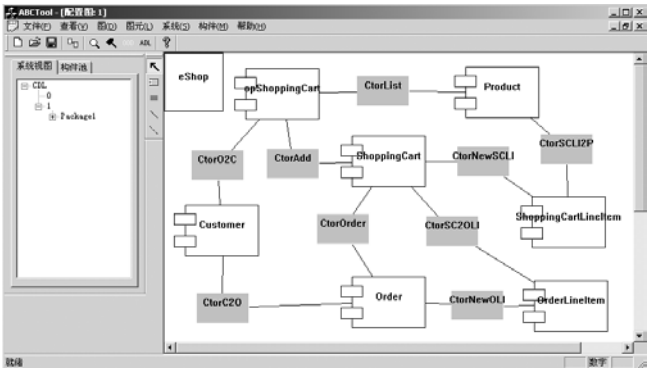

Fig. 7 Graphic Modeling of eShop using ABCTool

4.2 RSATool

The RSATool allows administrators to manage and evolve PKUAS and the deployed applications through the representation and manipulation of RSA. Figure [*] shows the main window of RSATool, representing RSA of eShop that is a simple B2C application allowing customers to purchase computer products from Internet. The location of the visual elements is very important to make the representation clearer and more understandable. Such location information cannot be retrieved from the runtime system and should be complemented through the drag and drop of the visual elements by the administrator. After the "*Save View*" command in the "*View*" menu is executed, RSATool will save the location information and retrieve it at the next time.

4.3 Transformation Between RSA and DSA

One of the main reasons to incarnate RSA based on ABC/ADL is to make the natural transformation between them. Because all of the elements of ABC/ADL are mapped into the data encapsulated in RSA, the description of SA in

ABC/ADL can be directly retrieved from the operation of generateADL() of *RuntimeSoftware-Architecture*. Moreover, PKUAS can retrieve information from the ABC/ADL description embedded in the deployment package to construct RSA faster. Currently, we develop a RSA tool independent of ABCTool. And the two tools are integrated through the description of SA in ABC/ADL, shown as Figure [*].

ABCTool *ABC/ADL* *RSATool*

Fig. 9 Integration of RSA and DSA

5. Conclusion and Future Work

Acknowledgments

This effort is sponsored by the State 863 High-Tech Program, National Natural Science Foundation of China, and Major Program of Ministry Of Education of China.

References

[1] Jonathan Aldrich, Craig Chambers and David Notkin, ArchJava: Connecting Software Architecture to Implementation, Proceedings of 24th International Conference on Software Engineering, ACM Press 2002.

[3] Feng Chen, Qianxiang Wang, Hong Mei, Fuqing Yang, An Architecture-Based Approach for Component-Oriented Development, Proceedings of COMPSAC 2002.

[4] David Garlan, Software Architecture: A Roadmap, The Future of Software Engineering 2000, Proceedings of 22nd International Conference on Software Engineering, ACM Press 2000.

[5] Nenad Medvidovic and Richard N. Taylor, A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transaction On Software Engineering, Vol. 26, No. 1, p70-93, January 2000.

[6] Hong Mei, Feng Chen, Qianxiang Wang, Yaodong Feng, ABC/ADL: An ADL Supporting Component Composition, accepted by 4th International Conference on Formal Engineering Methods (ICFEM2002).

[7] Peyman Oreizy, Nenad Medvidovic and Richard N. Taylor, Architecture-Based Runtime Software Evolution, Proceedings of 20th International Conference on Software Engineering, ACM Press 1998.

[8] D. E. Perry and A. L. Wolf, Foundations for the Study of Software Architecture, ACM SIGSOFT Software Engineering Notes, vol. 17, no. 4, pp. 40-52, October 1992.

[9] David S. Rosenblum and Rema Natarajan, Supporting Architectural Concerns in Component Interoperability Standards IEE Proceedings – Software Special Issue on Component-Based Software Engineering, 2000.

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

[11] Sun Microsystems, Java 2 Platform, Enterprise Edition Specification, V1.2, Final Release, http://java.sun.com/j2ee, December 1999.

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

[13] Sun Microsystems, Enterprise JavaBeans Specification, V2.0, Final Release, http://java.sun.com/j2ee, August 2001.

[14] Qianxiang Wang, Feng Chen, Hong Mei, Fuqing Yang, Using Application Server To Support Online Evolution, International Conference on Software Maintenance (ICSM2002), 3-6 October 2002, Montréal, Canada.

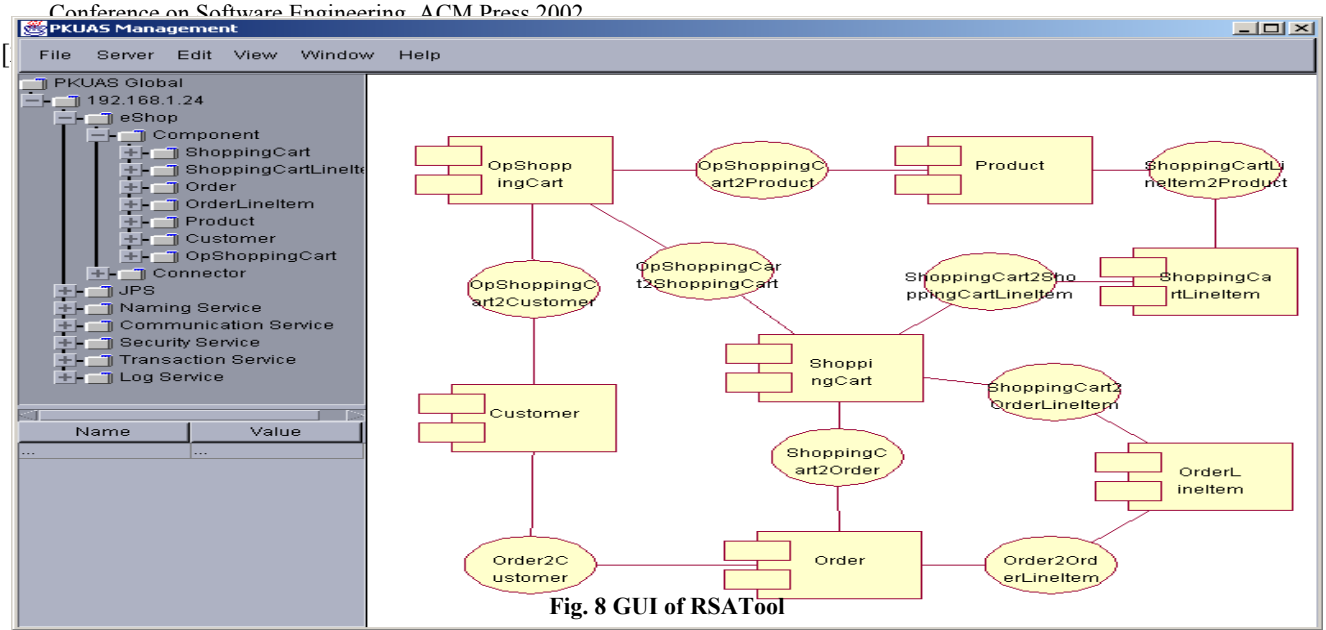

Fig. 8 GUI of RSATool