COMPSAC'02. IEEE press, pp. 450 - 455. 2002

An Architecture-Based Approach for Component-Oriented Development

Feng Chen, Qianxiang Wang, Hong Mei, Fuqing Yang
Department of Computer Science and Technology, Peking University, Beijing 100871, P.R.China
{chenfeng, wqx, meih, yang}@cs.pku.edu.cn

Abstract

Component-based reuse is a hopeful solution to the software crisis. Research on software architecture (SA) has revealed a component-based vision of the gross structure of software and provides a top-down approach to direct the component-oriented development process. But the gap between SA design and final implementation prevents it from playing a fundamental role in the process. On the other hand, the component-based software development (CBSD) technology such as Java 2 platform enterprise edition (J2EE) and Common Object Request Broker Architecture (CORBA) provides a feasible bottom-up way to construct systems from standard components, forming an implementation basis for an integrated component-oriented development process. In this paper, we propose an architecture-based component composition (ABC) approach, which uses SA model as the blueprint of development and COTS middleware as the run-time platform to support an automated component-oriented development process.

1. Introduction

As a hopeful solution to the software crisis, component-based software reuse is a key issue in software engineering research. Many achievements have been achieved in component-oriented software development, but there has not yet been an integrated and systematic approach to guide the whole process.

The importance of SA has been well acknowledged: SA makes it possible to describe, observe, and reason about the system's behaviour and features at a high-level, and greatly helps to design and implement more reusable architectures, components, and connectors.

Until now, however, SA has some problems in practical application. First, most research is following practice, not leading it [1]. Second, lack of programming language support causes a gap between SA design and the final implementation. Some attempts have been made to shorten this gap. For example, the C2 style [2] predefines a message passing connector with a set of style rules to support development of some software families. ArchJava [3] tries to introduce component and communication integrity into programming language. But they are too specific in some aspects, especially in the definitions of connectors, and their ability to architect is limited.

CBSD technology such as CORBA and EJB has recently become prevalent. Supported by middleware, it provides a bottom-up way to construct systems from standard components. CBSD mainly aims at specifying the runtime component model and component interoperability in order to enable COTS components production. However without a systematic method to guide the developing process, its practical effects are evidently restricted. Moreover, it is of no help for component composition at higher abstraction levels, e.g., to compose design models.

On the basis of existing languages, CBSD makes components runtime entities and provides the foundation for the component-oriented development. On the other hand, SA could play a directive role in the component-oriented development process to make development more effective and efficient. We combined them and proposed the architecture-based component composition (ABC) approach in this paper, 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.

The rest of this paper is organized as follows: Section 2 gives an overview of ABC to show its philosophy and language. Section 3 introduces transformation rules in ABC and a supporting toolkit, ABC-Tool, to illustrate how to apply ABC in practice. The last section draws conclusions and discusses some future work.

2. Overview of ABC

Figure 1 shows the development process in ABC:

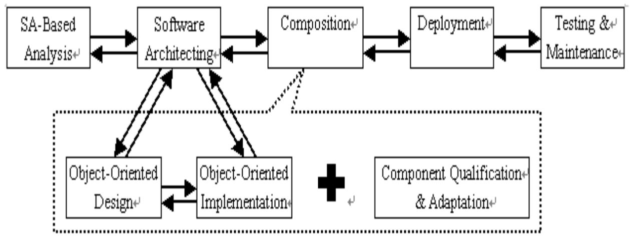

Figure 1: ABC Development Process

In ABC, SA model is used as the critical artifact through the whole process, while OO is used as the complementary means when developing components. The discussion in this paper focuses on the phases from architecting to deployment.

An example of a distributed scheduling system, shown in figure 2, is used in the rest of this paper: (The representation of component refers to figure 3.)

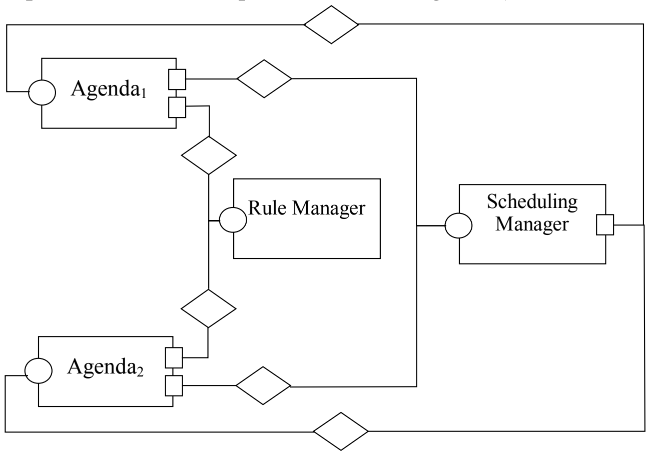

Figure 2: Architecture of Scheduling System

In this system, each agenda is on behalf of a client and the scheduling manager will carry out a negotiation if some clients want to make a meeting. Before the client requests services of the scheduling manager, it should be authenticated and authorized.

2.1 Basic ideas in ABC

This subsection discusses some basic ideas in ABC approach so as to show its philosophy.

Software architecture is a vital artifact in the software life cycle and should contribute to all phases of development. SA is well acknowledged as a very important high-level abstraction of software and enables more effective development, program understanding and high-level analysis. In ABC, SA guides the development process.

Component composition should be considered in different phases of development and at different level. Component Composition cannot be restricted only at the implementation level; rather it should cover every possible stage. In ABC, SA models, OO models, and executable components such as EJBs are all viewed as components to be composed.

Appropriate implementation mechanisms are needed to apply SA in practice. The current situation of decoupling implementation from architecture causes inconsistencies, confusion, and violation of architectural properties, and inhibits software maintenance and evolution. In ABC, CBSD technology is adopted to connect software architecture to implementation.

Mapping from SA to OO framework is necessary. Object-oriented technology provides both the conceptual basis and the practical tools for building and using components [4] and is dominant in current software development. The CBSD technology provides a bridge between the software architecture and OO implementation. Besides, a direct mapping from SA description to an OO framework is also needed to keep the traceability.

Tool-Supported automated mechanism is required. In order to preserve the architecture through the whole development process, the mappings, transformations and code generation should be done properly in relevant stages. Without supporting tools, the ABC approach will heavily burden the developers and make the development impractical.

2.2 Component Model in ABC

A component model is the kernel in software component technology. Diverse goals lead to different models. In ABC, the component model is defined in Figure 3 to meet the requirement of composition:

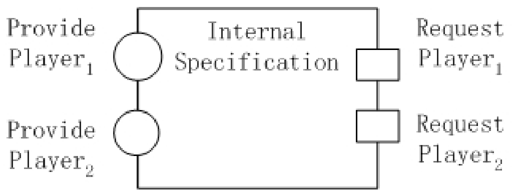

Figure 3: Component Model in ABC

This component model is divided into two parts. External Interfaces describe functions both provided and requested by the component. These functions are grouped by players that present the component's roles in different contexts. Moreover, these interfaces also define the component's contract with environment and communication protocol with other components. Internal Specification specifies constraints on component's interior structure, semantic model of the component, etc..

2.3 Introduction to ABC/ADL

Following the ABC component model, ABC/ADL is used as the description language. There exist many kinds of ADLs for different objectives, e.g., Wright [5], ACME [6] and Unicon [9]. ABC/ADL was designed to facilitate practical component-oriented software development.

ABC/ADL consists of three layers: meta-layer provides abstractions to define templates and architectural styles, and is embodied in language definition, which can only be extended by language designer; definition-layer provides languages construct to define components, connectors, and generic architectures, and all of the definitions must be defined by constructs provided by the meta-layer; instance-layer provides abstractions to define the interconnection of component instances. All of the instances must be defined by using constructs provided in definition-layer.

We discuss ABC/ADL's most significant features in the following subsections.

2.3.1 Description of Component and Connector

Table 1 shows part of the ABC/ADL description of the scheduling manager component based on Blackboard style defined in Table 3:

Table 1: Description of Scheduling manager

```
Component SchedulgManager is BLACKBOARD.BlackBoard{
Interfaces {
   provide player SchedulingManager is BlackBoard.Entry {
      type-method{
        SchedulingManage findByPrimaryKey (Object id);
      }
      instance-method{…}
   }
   request player Agenda is BlackBoard.Notification { … … }}
   Attributes {…}
   Properties {…}
   Dependencies {…}
   SemanticDescription{…}
```

There are two kinds of methods in every player: type-method and instance-method, because we notice that some methods are bound to the component type such as creation and finding, while others are executed by instances. This classification depicts the component more accurately, and helps in understanding and developing applications.

2.3.2 Architecture and Composite Component

In ABC/ADL, we distinguish component definitions and instances clearly. This separation enables us to handle architectural issues both at the definition level like constraints (what types of components and connectors can connect to each other.) and at instance level like multiplicity (one server can be connected by 0...n clients) and dynamic.

An architecture definition in ABC/ADL is a group of interconnected component and connector instances that comply with the constraints of architectural styles. It models the application's overall structure and is the blueprint for the composer. A component can have its own interior architecture. Such components are called composite components. With this concept, we can refine the architecture gradually and make the design more controllable. Moreover, composite component can be reused and composed as well: we can reuse and compose design artifacts at high-level.

Table 2 shows the ABC/ADL description of the Scheduling System.

Table 2: Part of the Description of Dating System

```
Architecture DS_Architecture{
uses{
   Component agendas : Agenda[];
   Component schedulingManager : SchedulingManager;
   Connector agendaToSchedulingManager :
                  SecuredConnector[];
   Connector schedulingManagerToAgenda :
                  DefaultConnector[];
   Variable i : int;}
Config main{
   agendas[i].SchedulingManager connects
                agendaToSchedulingManager[i].Callee
   agendaToSchedulingManager[i].Caller connects
               schedulingManager.SchedulingManager
   schedulingManager.Agenda connects
                schedulingManagerToAgenda[i].Callee
   schedulingManagerToAgenda[i].Caller connects
                agendas[i].Agenda}
SemanticDescription{
      OCL{
            Self.timetable is Sequence of TimeSlice
            Invariants {
                  Self.timetable->ForAll(t1, t2 |
                  t1 <> t2 implies t1.starttime >=
                  t2. endtime or t2.starttime >= t1.endtime
            )}
}
Component Dating_System is System{
      Structure {architecture DS_Architecture}}
```

2.3.3 Pluggable Styles

Style is another important concept brought by SA. An architecture style is determined by [7]: a set of component types, a topological layout of these components, a set of semantic constrains, and a set of connectors.

A number of engineering benefits can be obtained by introducing style: First, it provides a template to formalize architectures in a uniform way and establishes the vocabulary used in describing systems, thereby simplifying the communication among designers [1]. Second, it provides a unified semantic base through which different stylistic interpretations can be compared [8]. Third, the study of architectural styles can guide developers to choose proper architecture in practice since different styles possess different features. Some efforts towards the style handbook have been made. In some other efforts, the styles in use are limited to simplify system reasoning and facilitate code generation, e.g., Unicon [9] and C2 style [2]. As a more general solution, ABC/ADL provides an extensible framework that allows users to define their own styles according to their experiences and specific requirements. Table 3 shows the definition of Blackboard style, which is used to express the Dating System.

Table 3: Definition of Blackboard Style

```
Style BLACKBOARD_STYLE{
COMPONENT_TEMPLATE Blackboard {
  PROVIDE_PLAYER_TEMPLATE Entry {multicity=n};
  REQUEST_PLAYER_TEMPLATE Notify {multicity=n};}
COMPONENT_TEMPLATE Client {
  PROVIDE_PLAYER_TEMPLATE Notify {multicity=n};
    REQUEST_PLAYER_TEMPLATE Entry {multicity=1};}
//Here is no connector definition because this style
//uses default connectors
CONNECTION {
    Client.Entry :: DEFAULT.Connector.Callee
    DEFAULT.Connector.Caller :: Blackboard.Entry }
CONNECTION {
    Blackboard.Notify :: DEFAULT.Connector.Callee
        DEFAULT.Connector.Caller :: Client.Notify }
}
```

2.3.4 Complex Connector

Although the connectors are viewed as the first-class entities in SA, they are simple and have no interior structure in most SA study. However, in practice, communication between components may be quite complex especially at high-level of abstraction, e.g., FTP protocol. To model such interactions, ABC/ADL introduces complex connectors, which are the connectors that have interior architectures, can be refined, and finally implemented just like composite components. Users can build up their own connector library and express their systems more effectively.

2.3.5 Aspect

Recently, research on advanced separation of concerns has been greatly advanced, e.g., aspect-oriented programming (AOP) [10] and subject-oriented programming (SOP) [11]. Aspect is a way to encapsulate and modularize crosscutting concerns that used to be scattered over the whole system, such as security, logging, etc. [10]. Implementations can be more modular, easier to understand and better aligned with requirements. AOP and aspect-oriented framework (AOF) [12] were proposed and have had some successful applications. In practice, application servers such as J2EE-compliant platforms have implemented some common crosscutting features as system services, including transaction, security, logging, and so on. Such services can be best expressed as aspects. ABC introduces aspect into its ADL, and a special kind of relationship, named weaving, is also defined. Table 4 shows how to use aspect to express the connector between agenda and scheduling manager.

Table 4: description of Connector with Secured Aspect

```
Aspect SecuredAspect {
   Interfaces {
      provide player PreInvocation{
         instance-method {BOOL authorize()}
   }
}
Connector SecuredConnector is DEFAULT.Connector {
   Interfaces {
      provide player Callee is Connector.Callee{*}
      request player Caller is Connector.Callee {*}
   }
   Weaving {
      SecuredAspect.authorize weaves Callee.*;}
}
```

3. ABC in Practice

ABC provides an architecture-based systematic approach to guide component-oriented software development, especially the development of distributed component-based applications. But without the appropriate transformation mechanisms and supporting tools, it cannot be effectively applied in practice.

3.1 Implementing Components

Currently, we pay primary attention on implementing components in OO paradigm, using UML as bridge. There is no a simple mapping from ADL to UML [13]. Instead, a 2-phase process is adopted in ABC. First, the component's SA description is mapped into a UML framework within which users can refine the design and finally implement it. Then, when constructing composite components, UML models of composed components should be fitted together to obtain the overall OO models. The mapping rules in the first step are shown in Table 5, and the second step will be discussed in the next subsection.

Table 5 Mapping rules from ABC/ADL to UML

ABC/ADL elements	UML basic types
Component, Complex Connector	Package and Classes in the Package
Provide Player	Interfaces
Request Player	Abstract Classes
Attribute	Attribute
Other description	Note

Mapping rules will be affected by the choice of underlying middleware. For instance, when J2EE compliant is used as the run-time platform, every provide player will be mapped into two interfaces according to J2EE specification: type-methods to home interfaces and instance-methods to remote interfaces respectively.

One more important fact is that Request Players are mapped to proxy classes, which conceal the glue code and will be implemented during the Component Composition stage. Figure 4 shows the J2EE-based UML framework of the scheduling manager.

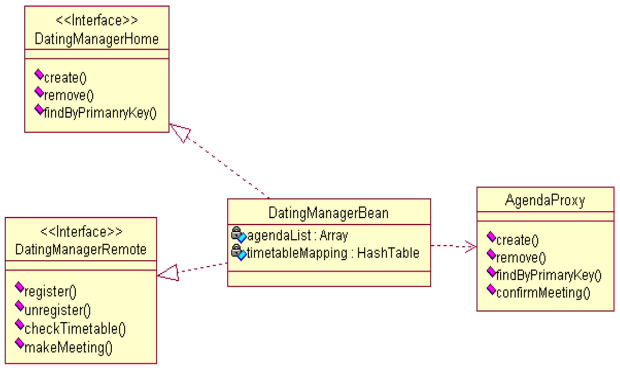

Figure 4: UML framework of Scheduling manager

ABC also encourages the thorough decomposition of systems and then generates component code frameworks directly from SA descriptions. There are two kinds of target languages supported in ABC now: the interface description language (IDL) for CORBA and OO programming languages such as Java. Code generation rules are similar to ABC/ADL-UML mapping rules in Table 5, except that the result is legal code in a designated language.

3.2 Composing components

Based on an application server, constructing a target application includes three steps: generating glue code, packing the application compliant to the given platform specification and finally deploying the application. These processes are platform-related, and the following briefly explain how to construct systems on CORBA and J2EE.

CORBA components interact with each other through an ORB (Object Request Broker). To compose components on ORB, the ABC Tool implements proxy classes by generating and compiling invocation code, and then registers the resulting executable components as CORBA servers in specified locations.

J2EE components, or called EJB, run in component containers and access other components or resources through containers. Containers use deployment descriptor files and the Java naming and directory interface (JNDI) to discover the requested component's physical location and control communication between components. When the ABC tool constructs a J2EE-compliant application, it implements proxy classes, generates deployment descriptor files, packs the components and descriptor files into an application, and deploys it to the platform.

UML models of components should be composed also. According to the collaboration between a component and its counterparts, dependencies will be established between the proxy classes and referenced interfaces. Figure 5 shows the class diagram of the scheduling system:

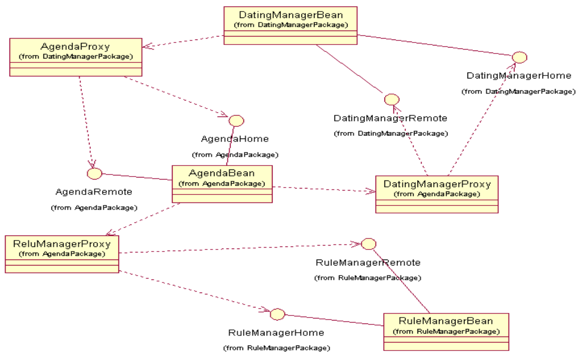

Figure 5: Class Diagram of the Scheduling System

3.3 Validating system

ABC/ADL provides both structural and semantic information. Three layers of system validation will be carried out before the system is generated: in syntax layer, the SA model is checked to avoid syntax errors; in implementation layer, component implementations are checked to guarantee compatibility with the specified platform and type-matching check is also performed; in semantic layer, basic constraints on components and connectors that are defined in style are taken into account, and some properties, e.g., deadlock freedom, can be checked if proper formal models are provided.

Instead of using a fixed formal language, ABC provides an extensible framework to integrate exsiting formal languages via an open framework of ABC/ADL.

3.4 Supporting Tool

We have implemented the ABC Tool based on J2EE and CORBA platform. In addition to the above core mechanism, some other necessary functions are also provided to facilitate software development, including graphic modeling ability and a simple component repository to store and manage existing components.

The simplified structure of the ABC Tool is shown in Figure 6.

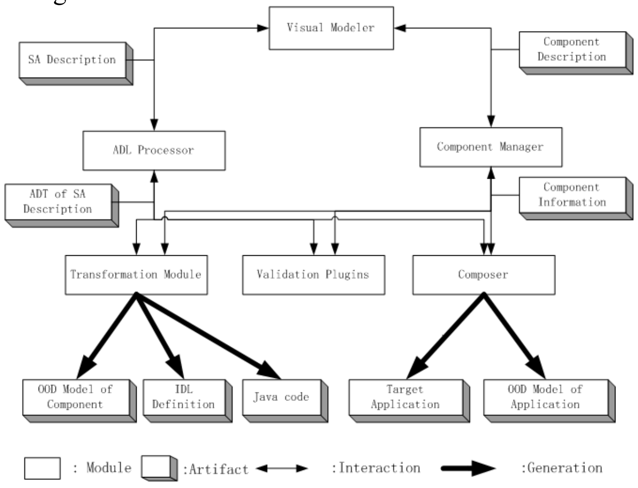

Figure 6: Structure of the ABC Tool

4. Conclusion

Software Architecture and CBSD technology both are focuses in recent software engineering research and practice. By combing them, this paper presents a SA-based approach for component-oriented software development, named ABC approach. Backed by CBSD technology, ABC uses SA to direct the development process, and automated transformation mechanisms are applied to shorten the gap between design and implementation with a supporting toolkit.

One area of future work is to establish a traceable transformation from requirement model to a SA design. There are several potential solutions to this problem: one is to connect SA to requirement analysis; another is to apply SA in domain engineering. Another important work is to strengthen the ABC/ADL's description ability, especially its semantic description.

Acknowledgements

This effort is sponsored by the State 863 High-Tech Program (SN: 2001AA110236), and Natural Science Foundation of China (SN: 60043002, 60125206).

References

[1] Clements P. and Northrop L., “Software Architecture: An Executive Overview”, *Technical Report CMU/SEI-96-TR-003*, 1996

[2] Taylor R., Medvidovic N., and Anderson K., “Component- and message-based architectural style for GUI software”, in *IEEE Transactions on Software Engineerin*g, June 1996.

[3] Aldrich J., Chambers C., Notkin D., “ArchJava: Connecting Software Architecture to Implementation”, to be in *Proceedings of ICSE 24*, 2002.

[4] Meyer B. and Mingins C., “Component-Based development: From Buzz to Spark”, in *IEEE Computer*, July 1999.

[5] Allen R. And Garlan D., “A formal Basis for Architectural Connection”, in *ACM Transactions on Software Engineering and Methodology*, July, 1997.

[6] Garlan D., Monroe R. and Wile D., “ACME: An Architecture Description Interchange Language”, In *Proceedings of CASCON'97*, November 1997.

[7] Bass L., Clements P. and Kazman R., “Software Architecture in Practice”, Published by Addison-Wesley in the SEI Series, 1998.

[8] Abowd G., Allen R. and Garlan D., “Using Style to Understand Descriptions of Software Architecture”, in *Proceedings of SIGSOFT'93: Foundations of Software Engineering, Software Engineering Notes* 18(5), 1993.

[9] Shaw M., Deline R., Klein D.V., Ross T.L., Young D.M. and Zelesnik G., “Abstractions for Software Architecture and Tools to Support Them”, in *IEEE Transactions on Software Engineering*, April 1995.

[10] Kiczales, G., et al., “Aspect-Oriented Programming”, In *Proceedings of the EuropeanConference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, Finland, 1997.

[11] IBM Subject-oriented programming research home page [online]. Available WWW <URL:http:// www.research.ibm.com/sop/sophome.htm>

[12] Pinto M., Amor M., Fuentes L. and Troya J., “Run-time coordination of components: design patterns vs. component-aspect based platforms”, in *Advanced Separation of Concerns workshop of the ECOOP,* 2001.

[13] David Garlan and Andrew J. Kompanek, “Reconciling the Needs of Architectural Description with Object-Modeling Notations”, in *UML2000*, 2000.