ICFEM'02. LNCS 2459, pp 38 - 47. 2002

ABC/ADL: An ADL Supporting Component Composition

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

Abstract

Architecture Description Language (ADL) is one of the keys to software architecture research, but most attention was paid to the description of software structure and high-level analysis of some system properties, while the ability to support refinement and implementation of Software Architecture (SA) models was ignored. In this paper, we present the ABC/ADL, an ADL supporting component composition. Besides the capability of architecting software systems, it provides support to the automated application generation based on SA model via mapping rules and customizable connectors.

Keywords: Software Architecture, Architecture Description Language, Component Composition

1. Introduction

As an effective and practical way to solve the software development crisis, component-based software reuse is an important research area in current software engineering. Software component technology primarily includes three interrelated processes: component development, component management and component composition. Among these three processes, component composition is regarded as the most important and difficult [Cle-96a]. A systematic and integrated approach to guide the process is desired but still under research.

In nature, software architecture provides a top-down mechanism for component-based software reuse. Originating from the consensus of the importance of the overall software structure, research on SA aims at making the architecture of a system explicit, dealing with the high-level design issues such as gross organization and control structure, assignment of functionality to computational units, and high-level interactions between these units [All-94]. All these facilitate the component composition process. But current SA research pays most attention on how to effectively describe system structure and reason the behaviours of software, ignoring how to guide the development of applications. Therefore, as the basis of software architecture research, most ADLs lack the ability to help refinement and implementation of the high-level design model.

In order to utilize SA more effectively and efficiently in the component-based software development, we propose the architecture-based component composition (ABC) approach that 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 [Hon-00]. An ADL, called ABC/ADL, is also defined to support component composition, which is essential for the ABC approach.

In addition to basic abilities to architect software systems, ABC/ADL provides other features that have value in component-based software development, e.g., explicit differentiation between type and instance, customizable connectors and pluggable styles, etc. Moreover, mapping rules from ADL description to implementation on COTS middleware were established and a supporting toolkit, ABC Tool, has been implemented.

The rest of this paper is organized as follows:

section 2 introduces the primary features of ABC/ADL, section 3 describes the constructs of ABC/ADL, section 4 illustrates the supporting toolkit, section 5 discusses some relate work, and the last section concludes this paper.

2. Features of ABC/ADL

This section explains the basic thoughts on ABC/ADL, including the component model and design principles, and introduces its primary features that are valuable for component-based software development.

In order to explain ABC/ADL clearer, an example of a distributed scheduling system from [Tru-01], shown in Figure 1, is used in the rest of this paper.

In this system, each agenda is on behalf of a client. When a client wants to make a meeting with others, he should send the request via his agenda to Dating Manger. Then the scheduling manager will carry out a negotiation among invitees via their agendas. Before the client requests services of the scheduling manager, he should be authenticated and authorized. Moreover, the client can refer to the rule manager that provides a computer-aided decision-making to arrange his appointments.

2.1 Component model

A component model is the kernel in software component technology. A development process usually involves diverse kinds of personnel and can be divided into several sub-processes that treat components from different perspectives and at different detailed levels. As a result, different models should be provided to meet these needs. [Mei-01] proposed classification of current component models according to their usage: model for component description/classification; model for component specification/composition; model for component implementation. In ABC, the component model is defined in Figure 2 to meet the requirements of composition.

This component model is divided into two parts: external interfaces and internal specification. External Interfaces describe services that a component provides to other components and dependencies that are requested by the component

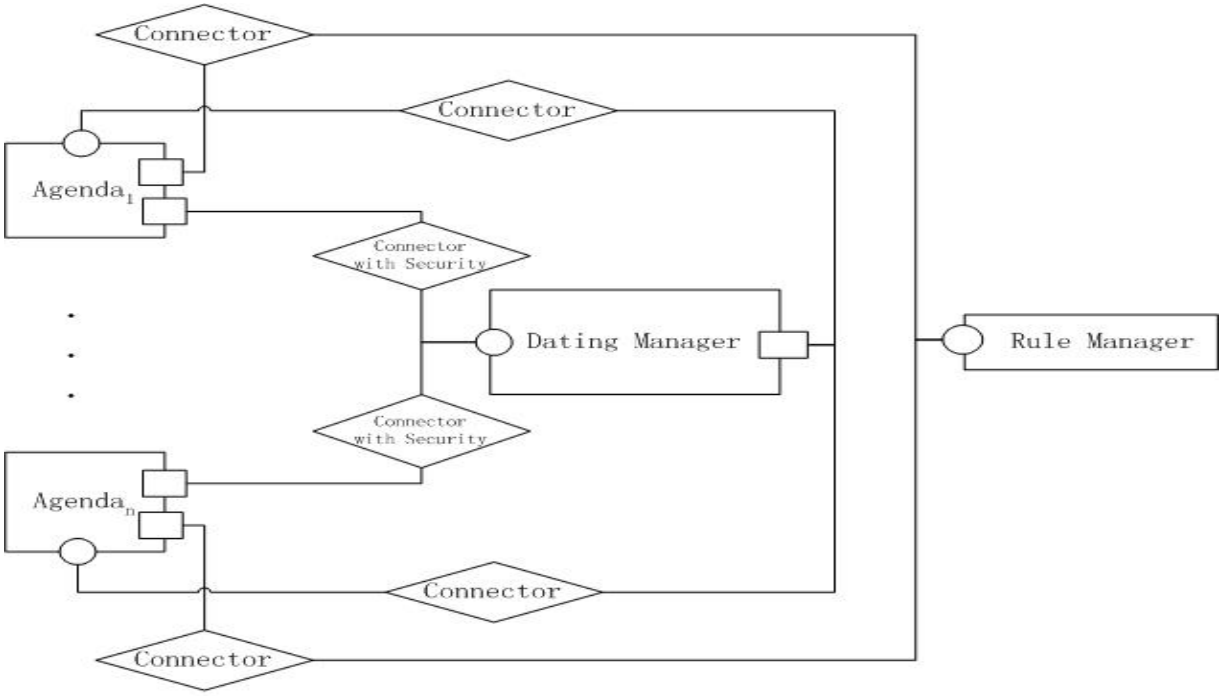

Figure 1: Architecture of distributed scheduling system

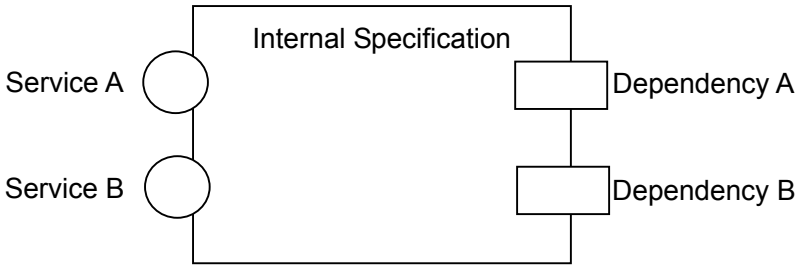

Figure 2: Component Model in ABC

itself. Moreover, external interfaces also define the component's contract with its environment and communication protocol with other components. Internal specification specifies constraints on component's interior structure, semantic model of the component and some nonfunctional features, e.g., security, throughput limitation.

Based on this model, ABC/ADL depicts the component at three layers: Basic layer includes syntactic descriptions of components and connectors, which primarily describes the operations in component and connector interfaces. Behavior layer includes semantic specifications and constraints on component functions, behaviors and nonfunctional features. Protocol layer includes definitions of contracts for the interaction between components and their environment, and communication protocols between components presented by connectors.

2.2 Design principles

ABC/ADL follows some sound design principles:

- ✓ Balance between simplicity and understandability - e.g., we adopt a natural language like syntax to facilitate understanding with a visual modeler to ease design.
- ✓ Concision – no more constructs that do not serve for the component composition exist in order to keep the language simple.
- ✓ Open framework for extension - we provide an open syntax and semantic framework that can be extended with existing or emerging programming and specification languages, as well as an API for 3rd party tool venders to access specification information.

ABC/ADL uses a three-layer structure to support the extensible framework. The meta-language layer provides abstract constructs to define templates and architectural styles. These constructs are embodied in language definition, and can only be extended by language designer. The definition layer provides concrete language constructs to define components, connectors, and generic architectures. All the constructs in definition layer must be defined by the constructs in the meta-language layer. The instance layer provides abstractions to define the interconnection of component and connector instances, which must be defined by constructs provided in definition-layer. To extend the ABC/ADL, users can define new constructs of definition layer based on constructs in meta-language layer.

2.3 Type and instance

ABC/ADL distinguishes component definitions and instances clearly. The definition defines common features of a type of component or connector and the relationship between types, while instances are instantiated from definition and used to construct the system. 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.

Besides, the methods in the component's interfaces are divided into two groups: type related methods and instance related methods. Because we notice that some methods are bound to the component type such as creation and finding, while others should be executed by instances. This classification depicts the component more

accurately, and helps in understanding and developing applications. It also facilitates transforming ADL description into Enterprise Java Bean (EJB) [EJB-99] and CORBA Component Model (CCM) [CCM-00] models. To release the users from repeated work, ABC Tool can automatically add some common type-methods when generating new components.

2.4 Architecture, Composite Component and Component Evolution

In ABC, architecture 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 composing components. A component can have its own interior architecture. Such components are called composite components (in fact, an application in ABC/ADL is a composite component). With this concept, we can refine the architecture gradually and make the design process more controllable. Moreover, composite component can be reused and composed as well, that is to say, we can reuse and compose design artifacts at a high-level.

In order to enhance the capability of system refinement and evolution, there are two kinds of component type relationships in ABC/ADL. The first is subtyping (a new component inherits and extends the old one's interfaces) and the second is refinement (the new component and the old one are identical in interfaces while different in interior architecture).

2.5 Pluggable style

Style is another important concept brought by SA. An architecture style is determined by the following [Bas-98]: a set of component types that perform some function at runtime, a topological layout of these components indicating their runtime interrelationships, a set of semantic constrains, and a set of connectors that mediate communication, coordination, or cooperation among components.

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. Second, it provides a unified semantic base through which different stylistic interpretations can be compared [Abo-93]. 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. For example, in [Bas-98], a set of architectural styles was cataloged and some empirical rules figured out for choosing styles. In some other efforts, the styles in use are limited to simplify system reasoning and facilitate code generation, e.g., Unicon [Sha-95] and C2 style [Tay-96]. As a more general solution, based on its open framework, ABC/ADL allows users to define their own styles according to their experiences and specific requirements.

2.6 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 between server and client. To model such interactions, ABC/ADL introduces complex connectors, which are the connectors that provide some functionality and 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. In the dating system, the connector between agendas and the scheduling manager may be considered as a complex connector that has the function of authentication and authorization.

2.7 Aspect component

Recently, research on advanced separation of concerns has become an attractive topic, e.g., aspect-oriented programming (AOP) [Kic-97] and

subject-oriented programming (SOP) [SOP-98]. Aspect is a way to encapsulate and modularize crosscutting concerns that used to be scattered over the whole system, such as security, logging, etc. [Kic-97]. Implementations can be more modular, easier to understand and better aligned with requirements with the application of aspect. AOP and aspect-oriented framework (AOF) [Tru-01] [Pin-01] were proposed and have had some successful applications. Application servers such as J2EE have implemented some common crosscutting features as system services, including transaction, security, logging, and so on. In substance, such services can be best expressed as aspects. We introduce aspect into ABC/ADL as a special kind of component, and a special kind of composition, named weaving, is also defined. For the scheduling system, the connector between agenda and dating manager is the connector with a security aspect.

3. Constructs in ABC/ADL

In this section, we discuss the basic constructs in ABC/ADL more detailedly, using the example of the scheduling system.

3.1 Component and connector

Components and connectors are building blocks in SA. In ABC/ADL, a component or connector must be based on a type of architecture style template to extend to its own specification. Architecture style offers addition constraints on components and connectors to avoid mismatch. The style of component determines the style of interfaces provided by the component. To accommodate different requirements, a component can integrate several styles.

Table 1 shows part of the ABC/ADL description of the dating manager component based on Blackboard style defined in Table 5:

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

Table 1: Description of dating manager

Interface specification is composed of players that incorporate head declaration and several method specifications. Head declaration defines the type of the player (provide or request) and the template style on which the player must be based. Introducing the type of players facilitate automated transformation from SA to realization, that is to say, supporting tools could use this information to know whether the interface is providing services or requiring services, and then build up proper relationships between callees and callers while generating systems. Each player consists of two kinds of methods: type-method and instance-method, as discussed in section 2.3.

On the definition of method, we refer to the definition of CORBA/IDL for the purpose of compatibility and facilitating the generation of glue code to construct and deploy the system based on COTS middleware. In method specification, a method is described as comprising of three parts, namely prototype, kernel and exception. Prototype defines whether the method is synchronic, asynchronous or one-way; kernel part defines the return type, method name and parameters; exception describes the type of exceptions that can be thrown by the method.

Attribute section designates the attributes the component will use in the interaction with others. Property section describes additional information of the component, e.g. security, version, throughout limit. A property is composed of property name, property type and property value, which are constrained by the component style. Dependency section describes the relationship of dependency

between the methods in the provide players and the methods in the request players. And semantics description section describes the semantics information of the component. (Refer to section 3.4 for details.)

The specification of connectors has a similar structure with components, but usually simpler. The ABC/ADL description of a connector of dating system is shown in table 2.

```
Connector J2EEConnector is DEFAULT.Connector{
Interfaces {
  provide player Callee is Connector.Callee{*}
  request player Caller is Connector.Caller{*}
}
Properties{
  Platform = J2EE;
}
Dependencies{
  Callee depends on Caller;
}
SemanticDescription {
  Caller includes Callee;
}
}
```

Table 2: description of a connector of dating system

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.

Besides, users can define the aspect components and attach them to components and connectors. Aspects are special components in ABC/ADL, so the definition is the same as the definition of components. But it should be attached to target entity via weaving composition. Table 3 shows the connector with the security aspect:

```
Component SecuredAspect is DEFAULT.Aspect{
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.Caller {*}
}
Weaving {
  SecuredAspect.authorize weaves Callee.*;}
}
```

Table 3: description of Connector with Secured Aspect

3.2 Architecture

Table 4 shows the ABC/ADL architecture description of the Dating System.

```
Architecture DS_Architecture{
uses{
 Component agendas : Agenda[];
 Component datingManager : DatingManager;
 Component ruleManager : RuleManager;
 Connector agendaToDatingManager :
            SecuredConnector[];
 Connector agendaToRuleManager :
            DefaultConnector[];
 Connector datingManagerToAgenda :
            DefaultConnector[];
 Variable i : int;}
Config main{
 agendas[i].DatingManager connects
          agendaToDatingManager[i].Callee
 agendaToDatingManager[i].Caller connects
          datingManager.DatingManager
 agendas[i].RuleManager connects
          agendaToRuleManager[i].Callee
 agendaToDRuleManager[i].Caller connects
          ruleManager.RuleManager
 datingManager.Agenda connects
          datingManagerToAgenda[i].Callee
 datingManagerToAgenda[i].Caller connects
          agendas[i].Agenda}
  SemanticDescription{ }
}
Component Dating_System is System{
  Structure {architecture DS_Architecture}
  mapping {self.makeMeeting to
          datingManager.makeMeeting}
}
```

Table 4: Description of Dating System

Architecture specification comprises two sections. In *uses* section, all instances of components and connectors used in the system are declared. These instances must be instantiated from either the types defined in the specification of components and connectors or built-in types. C*onfig* section depicts the topologic layout of instances in the system, that is, the system structure is described here. Each item in *config* section describes a relation between a component and a connector, designating which component player links to which connector player. The relation

between components and connectors must conform to the style constraint. Moreover, to improve flexibility and adaptability, ABC/ADL allows a system has multiple configurations, so in the architecture specification, multiple *config* sections can exist. Users can designate a configuration at the late phases of the composition process according to requirement.

In a complete system mode, an architecture description will not be stand-alone, but should be attached to some composite components using the *structure* section in component specification. In ABC/ADL, an application is a composite component with the overall architecture, such as the Dating_System component in the scheduling system shown in Table 4. A subsystem or a part of the system can also be a composite component to make the design more understandable and reusable. Besides, the interface of the composite component is determined by its interior components. The *mapping* section specifies how to connect the interface of the composite component with its internal components.

3.3 Style

As discussed in section 2.5, ABC/ADL provides an extensible framework that allows users to define their own styles instead of using built-in styles. Table 5 shows the definition of Blackboard style, which is used to express the Dating System.

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

Table5: Definition of Blackboard Style

Every definition of style includes component templates, connector templates and connection specifications. Component and connector templates describe the basic frameworks of the components and connectors, including players and some properties. And connection specifications restrict the relationship between components and connectors.

3.4 Semantic description

From above example codes, one can see that the semantic description is not a standalone element in ABC/ADL, but scattered over every specification of elements, using `SemanticDescription` key word to mark. The semantic description is trying to use formal methods to model, or, at least, use natural language to describe, the behaviors and features of the elements. Thus developers are able to know more beyond the interfaces and connection constraints, and construct the systems more precisely. Moreover, some automated system verification and validation can be achieved based on formal methods. In fact, the ability in system reasoning is one of the advantages of software architecture approaches.

In ABC/ADL, we do not want to prescribe the formal language to use. Instead, ABC/ADL provides an open framework for users to build up their own specification of semantics information with support of proper toolkits. Every semantic description section contains multiple segments, each of which uses a kind of formal language or natural language. Before the semantic segment, the name of used language must be designated, according to which the analysis tool will pass the content of that segment to corresponding module. In our ABC tool, it is easy to plug new analysis modules into the toolkit, as well as set up the relation between language name and its corresponding module. A semantics description of DatingManager is showing in Table 6, which use UML-OCL to describe the actions of Agenda.

```
Component Agenda is BLACKBOARD.Client {
```

```
……..
  SemanticsDescription
  {
  … …
    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
       )
      }
      … …
    }
  … …
  }
}
```

Table 6: Semantics Description of DatingManager

4. Tool Support

A prototype of ABC tool has been implemented to support the ABC/ADL. Figure 3 showing its structure.

By hiding details of language, graphic presentation is more understandable and able to increases designing efficiency. ABC tool allows users to design applications in a visualized way by offering visual representation for language elements.

It also accomplishes some transformation of ABC/ADL, e.g. mapping SA description into UML framework, generating IDL and Java code from ADL description. In addition, it can automatically construct applications from existing components based on some COTS middleware specifications, including CORBA and J2EE. Figure 4 shows its main windows.

Before generating applications, the system model will be validated. Because ABC/ADL provides both structural and semantic information, the system validating consists of three layers:

- ✓ Syntax layer: the SA model is checked to avoid syntax errors.
- ✓ Implementation layer: component implementations are checked to guarantee compatibility with the specified platform and type-matching check is also applied in component invocation.
- ✓ Semantic layer: basic constraints on components and connectors that are defined in style are taken into account, and some features, for example, deadlock-free, could be checked if proper formal models and correspondingly analysis modules are provided.

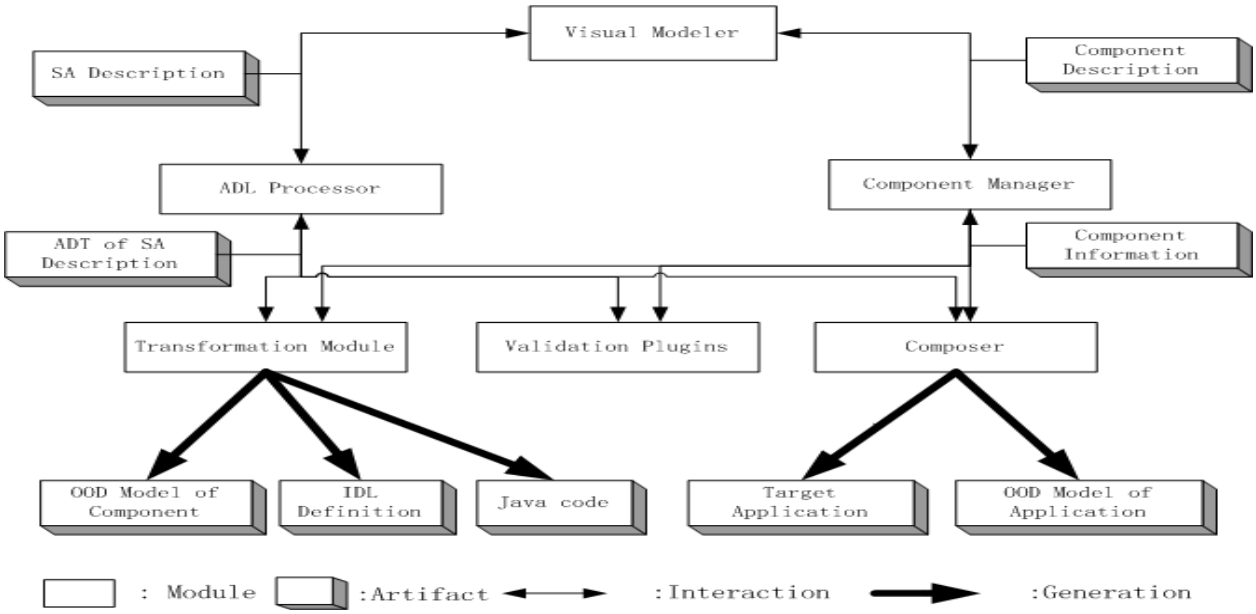

Figure 3 Structure of ABC Tool

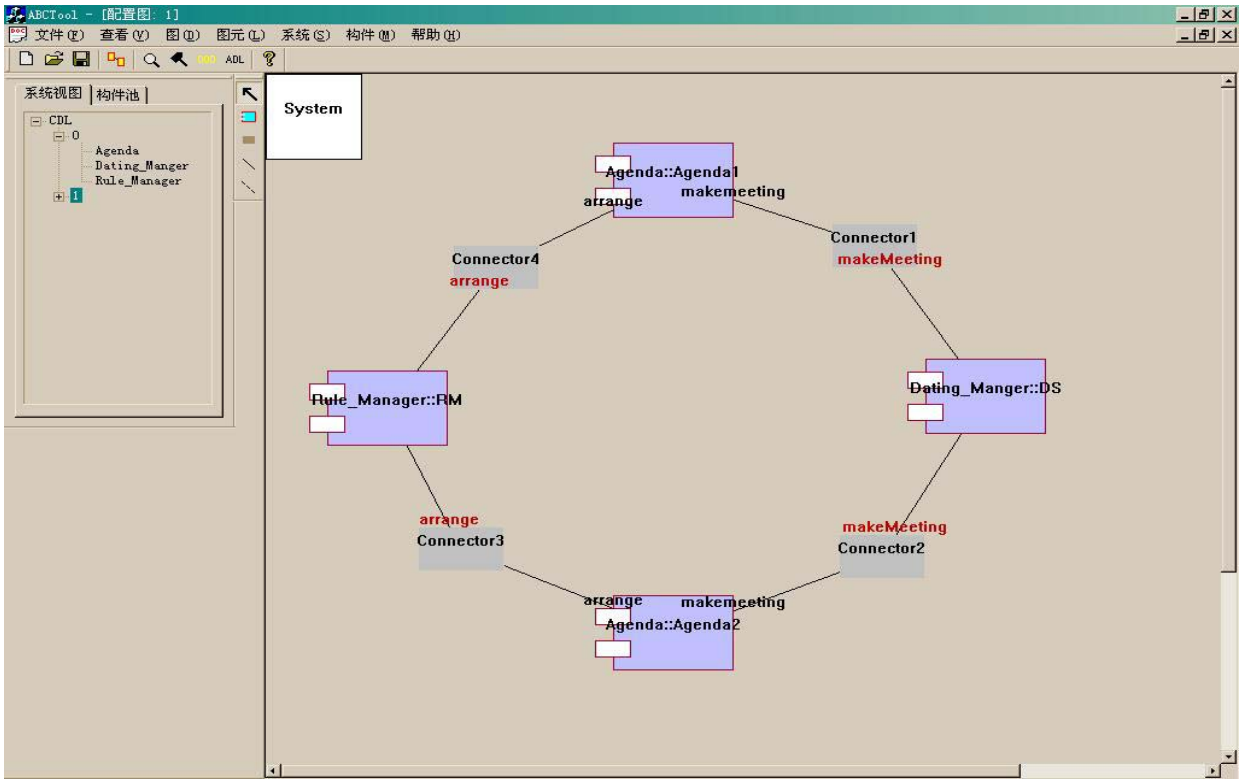

Figure 4: Graphic Modeling

5. Related Work

5.1 Other ADLs

There exist many kinds of ADLs for different objectives, e.g., Rapide[Luc-95], Wright [All-97], ACME [Gar-97] and Unicon [Sha-95].

Rapide is based on event-driven model in order to support component-based development for large-scale and multi-language systems. This ADL presents the capabilities of architecting, analysis, simulation and code generation, but doesn't regard connectors as first-class entities, which limits it ability to describe applications.

Wright is regarded as one of he most representative ADL. It adopts CSP to describe behaviours so as to formally verify some aspects of the architecture description. But, Wright is only a language for specification and don't support system development.

UniCon is a step toward the system realization, because it realizes a set of predefined connectors so that makes it possible to generate system from architecture. But it is limited for only the predefined connectors can be utilized.

ACME is an architecture description interchange language. Different ADLs provide complementary capability for architectural development and analysis, but their implementations are isolated and it is difficult to integrate them. ACME provides a structural framework for characterizing architectures, together with annotation facilities for additional ADL-specific information [Gar-97]. On the basis of ACME, different ADLs can share a set of kernel capabilities and set up their own capability via the open framework. ACME can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools. But ACME is not a practical ADL to model application. ABC/ADL benefits from the structure of ACME.

5.2 Advanced separation of concerns

Study on advanced separation of concerns (SOC) reveals a new vision to software architecture.

Traditional development approaches divide applications into structural units, e.g., modules, objects or components. Recent SOC approaches such as adaptive programming, aspect-oriented programming, composition filter, hyperspaces, subject-oriented programming, etc., try to enhanced the traditional ones by providing separation of concerns along additional dimensions, beyond structural units. For instance, aspect encapsulates the crosscutting features in software to make the implementation more modular [Tzi-01]; subject-oriented design and programming align the design and implementation with the requirement, keeping a good traceability [Sio-99]. ABC/ADL also adopts aspect so as to architect applications more accurately.

5.3 Component based software development

CBSD (Component-Based Software Development) has become more and more prevalent in industry. Based on the middleware and specification of runtime component, it provides a practical bottom-up approach to construct systems from existing components. With the development of CBSD technology, there emerge some widely accepted runtime component models, e.g., enterprise java bean (EJB), CORBA component model (CCM), Microsoft's distributed component object model (DCOM) and the newly web service model. These models provide the foundation for component development and composition. ABC/ADL adopts some features of them such as type methods and instance methods to enhance the ability of description and narrow the gap to implementation. But CBSD primarily puts emphasis on the interoperability of components in implementation layer, and lacks a systematic methodology to guide the developing process. As a result, it's unable to help the component composition at the higher level of abstraction, which is just the strength of ABC/ADL.

6. Conclusion

This paper presents an architecture description language supporting component composition, ABC/ADL. ABC/ADL stresses on the capabilities of refinement and realization of architecture, trying to support component composition better. By separating run-time and design-time configurations, supporting composite components and complex connectors, introducing aspects, it effectively support the ABC approach, which employs SA descriptions as blueprints for constructing systems while using middleware as the runtime scaffold for component composition. Besides, it provides an open framework to allow user extend the language. A supporting tool, ABC Tool, has been implemented to visualize the modeling process, analyze the ADL description and automate the application generation.

One of the future works is to setup an XML-based framework for ABC/ADL. XML provides a standard way to define the ADL, facilitating understanding and transforming ADL. Besides, it is easy to extend languages based on XML, and there are numerous tools available to parse, analyze and manage XML-based languages. Another significant work is to map ABC/ADL into UML. As a high-level abstraction of applications, SA does not describe how to implements its components and connectors. UML is the most popular OO design language, so a good mapping between ABC/ADL and UML can greatly benefit the development process of ABC approach.

Acknowledgement:

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

References

[All-94] Allen, R. and Garlan, D., "Formalizing Architectural Connection", in *Proceedings of ICSE 16*,

1994.

[All-97] Allen R. And Garlan D., "A formal Basis for Architectural Connection", in *ACM Transactions on Software Engineering and Methodology*, July, 1997.

[Abo-93] 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.

[Bas-98] Bass L., Clements P. and Kazman R., "Software Architecture in Practice", Published by Addison-Wesley in the SEI Series, 1998.

[CCM-00] CORBA Component Model RFP [online], Available WWW URL: < http://www.omg.org >

[Cle-96a] Clements C. Paul, From Subroutines to Subsystems: Component-Based Software Development, 3-6. Component-Based Software Engineering: Selected Papers from the Software Engineering Institute. Los Alamitos, CA: IEEE Computer Society Press, 1996.

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

[EJB-99] Enterprise JavaBeans Specification [online], Available WWW URL: < http:// java.sun.com/ products/ejb/docs.html >

[Gar-93] Garlan D. and Shaw M., "An Introduction to Software Architecture", in *Advances in Software Engineering and Knowledge Engineering*, Volume 1, World Scientific Publishing Company, 1993.

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

[Hon-00] Hong Mei, Jichuan Chang, Fuqing Yang, "Composing Software Components at Architectural Level", IFIP WCC2000, Beijing, 2000.8

[Hon-01] Hong Mei, Jichuan Chang, Fuqing Yang, "Software Component Composition based on ADL and Middleware", Science in China(F), 44(2), 136—151, 2001

[Kic-97] Kiczales, G., et al., "Aspect-Oriented Programming", In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, Finland, 1997.

[Luc-95] Luckham D.C. and Vera J., "An event-based architecture definition language", IEEE Transactions on Software Engineering, Sept., 1995.

[Med-97] N. Medvidovic, Neno, "A Classification and Comparison Framework for Software Architecture Description Languages", Technical Report UCI-ICS-97-02, University of California at Irvine

[Mei-01] Mei, H. "A Component Model for Perspective Management of Enterprise Software Reuse", Annals of Software Engineering 11,219-236, 2001

[Mey-99] Meyer B. and Mingins C., "Component-Based development: From Buzz to Spark", in *IEEE Computer*, July 1999.

[Per-92] Perry D. and Wolf A., "Foundations for the Study of Software Architecture", in *ACM SIGSOFT Software Engineering Notes*, 17(4), 1992.

[Pin-01] 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.

[Qio-97] Qiong Wu, Jichuan Chang, Hong Mei, Fuqing Yang, JBCDL: An Object-Oriented Component Description Language, Proceedings of the twenty-fourth International Conference TOOLS ASIA, Beijing, 1997.

[Sha-95] 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.

[Sio-99] Siobhán Clarke, William Harrison, Harold Ossher, Peri Tarr, "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code", OOPSLA, 1999.

[SOP-98] IBM Subject-oriented programming research home page [online]. Available WWW <URL:http:// www.research.ibm.com/sop/sophome.htm>

[Tay-96] 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.

[Tru-01] Truyen, E.; Vanhaute, B.; Joosen, W.; Verbaeten, P.; Jorgensen, B.N.; Leuven, "Dynamic and selective combination of extensions in component-based applications", ICSE, 2001.

[Tzi-01] Tzilla Elrad, Mehemet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher, "Discussing Aspects of AOP", IEEE Panelists, 2001.