Journal of Zhejiang University SCIENCE
ISSN 1009-3095
http://www.zju.edu.cn/jzus
E-mail: jzus@zju.edu.cn

A distributed software architecture design framework based on attributed grammar*

JIA Xiao-lin (贾晓琳)†, QIN Zheng (覃 征), HE Jian (何 坚), YU Fan (虞 凡)

(*School of Electronics and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China*)

†E-mail: Xlinjia@mail.xjtu.edu.cn

Received Apr. 5, 2004; revision accepted Oct. 18, 2004

Abstract: Software architectures shift the focus of developers from lines-of-code to coarser-grained architectural elements and their overall interconnection structure. There are, however, many features of the distributed software that make the developing methods of distributed software quite different from the traditional ways. Furthermore, the traditional centralized ways with fixed interfaces cannot adapt to the flexible requirements of distributed software. In this paper, the attributed grammar (AG) is extended to refine the characters of distributed software, and a distributed software architecture description language (DSADL) based on attributed grammar is introduced, and then a model of integrated environment for software architecture design is proposed. It can be demonstrated by the practice that DSADL can help the programmers to analyze and design distributed software effectively, so the efficiency of the development can be improved greatly.

Key words: Software architecture, Attributed grammar, Distributed software, Component
doi:10.1631/jzus.2005.A0513 **Document code:** A **CLC number:** TP311.52

INTRODUCTION

Software architecture research is intended to reduce the cost of developing applications and increases the potential for commonality between different members of a closely related product family. To support architecture-based development, formal modeling notations and analysis and development tools that operate on architectural specifications are needed. Some software architecture description languages (ADLs) and their correlated toolkits have been proposed by Shaw *et al*.(1995) as the solutions. Loosely defined, an ADL for software applications focuses on the high-level structure of the overall application rather than the implementation details of any specific source module. A number of ADLs have been proposed for modeling architectures within a particular domain as well as general-purpose architecture modeling languages. Distributed software has characters of heterogeneity, autonomy and interoperability; the architecture of distributed software is quite different from that of other software. We think that a distributed software architectural description language should describe or support the following concepts to some degree:

· Components;

· Connections;

· Hierarchical composition, where one component contains an entire sub-architecture;

· Computation paradigms, i.e., semantics, constraints, and nonfunctional properties;

· Underlying formal models.

To support those concepts, we extend the attributed grammar (AG) and implement a distributed software architecture description language (DSADL), which provides means to software engineers to define and analyze the software system including component, interface, connector, instantiation, condition, and distribution. In this paper, the extended AG and the

*Project (No. 2000K08-G12) supported by Shaanxi Provincial Science and Technology Development Plan, China

key elements of DSADL are discussed, and an integrated developing environment (IDE) for architectural designers is introduced at last.

FORMAL BASES

A number of formal models have been proposed to describe and analyze the interacting systems; Medvidovic and Taylor (2000) classified and compared some ADLs. Z is suited for formalizing the architecture style, but lacks union semantic model. Meanwhile, CHAM based on the chemical abstract machine discussed by Berry and Boudol (1992) is strongly capable of analyzing and verifying properties of software architecture, but it has no general method for describing software architecture. Hence, we use the AG used by Fang (1998) in computer network protocol specification to describe the architecture of distributed software.

Extended AG

After AG was proposed by Knuth in 1964, it has been widely used in the auto-generation of compiler, syntax edit, interchanging system. We have found that it is easy for us to describe the architecture of distributed software by extending the AG. In this way, it absorbs the features of semantic localization, supports the abstract and encapsulation of information, and permits the complicated interaction and parallel operation. We extend the AG as follows:

(1) Introducing the parallel description mechanism to describe the parallel operation and the overlapped time of distributed software.

(2) A special terminal is introduced to describe the dependence of component and time sequence.

(3) Conditional production, to simplify the definition, and extend the capacity of description.

Formally, the extended AG is modeled as a septet (N, T, P, S, A, V, R), in which N is a set of non-terminal char, presenting an expression or a sub-expression; T is a set of terminal char, e.g., constant, component name; P is a set of production; S is the start char, presenting the initial state; A is a set of attributes; V is the range of attribute value, which is related to each syntax symbol; R is a set of rules including the computation of the attribute values, the description of the exchanging messages and the synchronous relations among components.

For convenience, we use the process model to describe the interaction among components of distributed software, where each process is in accord with a component. Each grammar symbol <X> is connected with a definition of a process, namely process X. For example, production: <X>$\rightarrow$<Y><Z>, defines a process X which invokes process Y and process Z alternately.

Each <X> can have its attribute values: α_1, α_2, …, and be symbolized as <X>.α_1, <X>.α_2, … The I/O parameter of process X is composed of the attribute values of symbols X.

Semantic description

The semantic description of process behaviors and distribution is as follows:

1. Parallel

Abstractly, we consider the behavior of an architectural configuration as consisting of each of the behaviors of the individual component, while each operation is independent. The computation of each component forms a part of the overall behavior, where the order in which the computations occur, and transfer of data from one to the other is coordinated by the connectors.

The basic technique used in the extended AG to model the combination of coordinated processes is parallel composition, which uses a pair of the operator "||" to enclose the right part of the production.

For instance, in a session connection, process can send (or receive) data in both fast and ordinary speed at the same time, so the production that represents the parallel action is:

<TRANS>$\rightarrow$||<SEND_RD><SEND_ED>
<RECV_RD><RECV_ED>||

2. Distribution

In a distributed system, a component can be located in any place within a net, so a special terminal "@" is introduced to describe the component location. We can locate a component using a "@" and an integer.

3. Timer

To describe time sequence (one behavior occurs after some time), a special symbol "<TIMER>" is used to represent the timer. <TIMER> has an attribute

"timeout" and can be set with an integer. Once <TIMER>.timeout is set, the process's behavior can only occur when timer equals to <TIMER>.timeout. For example, consider the following process declaration:

<*X*>→<TIMER>.α
{<TIMER>.timeout := l0;
(other attribute rules)}

The process named TIMER is created when process *X* starts, the <TIMER>.timeout is set with ten time slices. The process TIMER sleeps for ten time slices, then creates the sub-process "α", other rules are executed finally.

4. Conditional production

There are some constraints that affect a process to be invoked. Conditional production is introduced to describe those constraints. Conditional production is placed in front of a process to judge whether the process can be invoked or not. A conditional production is a logic expression which is inducted by a symbol "when", and is marked as "<*X*>→COND α". For example,

<*X*>→when <*X*>.input = <*Y*>.output
{<compute>
(attribute rules)}

Defines a process that can be invoked only when the attribute value of <*X*>.input equals to the attributed value of <*Y*>.output, and then the semantic tree can be extended, which means that process *X* is activated, then process 'compute' is created and finally the other attribute rules can be executed.

In order to implement the syntax analyzer effectively, we adopt the LL(1) grammar as the basic grammar where the dependency satisfies the L-attribute condition (namely L-AG).

KEY ELEMENT OF DSADL

DSADL does not only support simple expression of describing connections among simple modules, but also configuration of subsystems into system. The key element of DSADL is as follows:

1. Components

A component describes a localized, independent computation. Components may be as small as a single procedure or as large as an entire application. Each component may require its own data or execution space, or it may share them with other components. The component which needs to invoke other component's service is considered as a client, and those that provide services to clients are servers. A component may be a client of some components, or may be a server for others as well. In GUI (Graphic User Interface), a component is presented as a rectangle.

Component types are abstractions that encapsulate functions into reusable blocks. A component type can be instantiated many times in a single architecture or it may be reused across architectures. There are two kinds of components, one is atomic component which implements a function and cannot be decomposed, the other is composite component which is composed of other components (or atomic component).

2. Interface

An interface is a set of interaction points between component and the external world. An interface consists of a number of ports, each port represents an interaction in which the component may participate. A port specification indicates two aspects of a component, one is component's behaviors, in this view, the port specification indicates the properties that the component must have if it is viewed through the lens of that particular port; two is the expectant systems with which a component interacts.

The use of interface is similar to how abstract data type declarations are used in a programming language. The abstract type declaration acts as an interface to the type, allowing consistency checks to be performed statically and to guide programmers' use of the type. An abstract data type declaration provides a means of simplifying checks that the type will be used appropriately throughout the program.

There are two types of interfaces, one is provide, the other is require. A provide interface describes a service or function which a component provides to others. A require interface represents a service or functionality which the component must get from other components. In GUI, a solid dot represents provide interface, and a hollow dot represents require interface.

3. Connectors

Connectors are architectural building blocks

used to model interactions among components and rules that govern those interactions. Unlike components, connectors may not correspond to compilation units in an implemented system, they may be implemented as separately compliable message routing devices, table entries, client-server protocols, SQL links a database and an application, and so forth. A connector provides, in effect, a set of requirements that the component must meet, and an information-hiding boundary that clarifies what expectations the component can have from its environment.

When a component meets another's requirement, it means that if the relationship between two components fits the client-server protocol, we link the two components together by defining a connector. We also can use connectors to make a composite component. In GUI, we use a direction line (start from a require interface to a provide interface) to represent a connection.

4. Instances

Because there may be more than one usage of a given component or connector in a system, each instance must be explicitly and uniquely named. In order to expand our descriptions from single systems to families of systems, we need to let the descriptions cover more situations, that is, they must be more flexible so that they can be used in more places in a description. We do this by providing static instance and dynamic instance. In static instance, the type of a port or role, a computation, the name of an interface, etc., must be configured when the system is compiled. A dynamic instance leaves "holes" in the description that will be filled in when the type is instantiated. So the type of a port or role, a computation, the name of an interface, etc., can be parameterized according to the IEEE Standard 1471 listed by Mark *et al*.(2001).

5. Distribution

In a distributed system, each component can be located in a computer, and in any host of the net. In order to describe the feature of physical distribution, when we declare an instance, we can use a "@" following an integer to make an image of the physical location of the instance. Distribution is an abstraction to describe the physical distribution.

6. Guard

As the ability to localize and reuse both interaction descriptions and analysis is critical to the practical characterization of architectural styles, guard is introduced to specify causal relationships between different components, and given constraints to behavior of a component. These constraints can be declared at the global configuration level, or in front of a behavior, for example, we can constraint a component by restricting the number of associations in which its players can participate. Besides we can constrain the implementation and usage of a component by specifying its guard attributes.

He *et al*.(2002a) discussed the syntax of SADL in detail. The extended AG makes DSADL have normative syntax and strict semantics, and the analyzer can formally design software architecture with DSADL. However, it is very hard for analyzers remembering the syntax of DSADL to edit the software architecture description, it is very useful to provide an IDE for architectural designers.

IDE FOR DSADL

We developed an integrated environment (namely, EDAD) for software architecture development, partly discussed by He *et al*.(2002b), which is a visual programming tool, and supports the design and construction of distributed software (Fig.1).

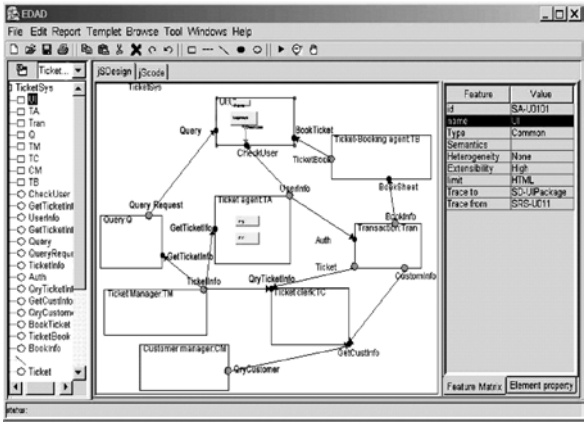


Fig.1 The user interface of EDAD

Introduction to EDAD

It provides a framework in which software design can be captured, viewed and modified easily and quickly. Intelligent assistance is provided throughout the design process, from the sketching of the design diagrams to the generation of compliable DSADL

code.

Left to the EDAD is the hierarchical system view, which shows all the component types used in the program and the 'include' relationship between them. It is updated automatically whenever the program structure is modified in the sketchpad. It is useful not only as an indicator of the overall program structure but can also be used as a navigational aid for browsing the different parts of the system–selecting the appropriate type in this view will bring up the structure of that component type in the sketchpad. In addition, it serves as a 'where am I' indicator by highlighting the part of the tree diagram which corresponds to the component being displayed in the front-most sketchpad.

Central to the EDAD is the Configuration Window, which includes the jSDesign and jScode sketchpad. It is an intelligent diagram editor with built-in knowledge of the DSADL syntax. The jSDesign contains a sketchpad in which the program architecture can be mapped out using the appropriate tools from the tool palette. The sketchpad displays the graphical configuration view of the software architecture, and where all editing of the program structure takes place. Controls are provided for traversing the hierarchy within this view. The DSADL code corresponding to the diagram is drawn in jSCode. While each sketchpad allows the display and editing of a single component at a time, the associated hierarchical system view presents the entire program in a hierarchical tree structure.

Right to the EDAD is the property page for defining the features of architecture elements. Each architectural element has features, i.e., priority, status, difficulty, stability, cost, trace from and trace to, etc.

Application

During developing of the online ticket booking and selling system (TicketSys), we adopted EDAD to analyze and design the architecture of the TicketSys. The main functions of the TicketSys include:

1. Provide methods for users to query information, such as train schedule, ticket price, and booking information;

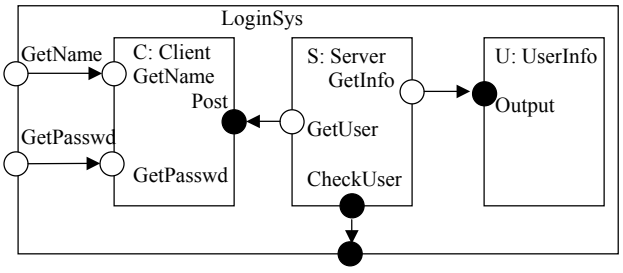
2. Sell the real-time ticket, which must be in three days before leaving; draw the full payment and print the ticket;

3. Book ticket, which must be between three to fifteen days before leaving. Customer can query the booking result.

jSDesign (Fig.1) illustrates the graphical configuration of TicketSys's architecture. We can see that TicketSys is composed of eight components, such as UI, Query, Ticket Manager, Customer manager, Ticket Agent, Ticket clerk, Ticket-Booking agent and Transaction. jSDesign also shows that the TicketSys has two composite components, one is UI component, which is composed of LoginSys and HTML forms. LoginSys is a login interface based on web, which gets user name and password, and verify whether the user is valid. The other composite component is Ticket agent component, which is composed of seller component and teller component.

We also generated the DSADL code of TicketSys. For example, the DSADL code for LoginSys is shown in Fig.2b.

LoginSys (Fig.2a) comprises three components: C (Client type), S (Server type) and U (Userinfo type), among which, C receives, submits user name and password, and S receives user name and password and verifies the user, and U stores user's registration information. These three components are deployed under the distributed environment.



(a)

```
ComponentLoginSys {
  Import GetUserName @ "Client/GetUsername";
         GetPasswd @ "Client/GetPasswd";
  Export CheckUser @ "Server/CheckUser";
    Inst
          C: Client
          S: Server
          U: UserInfo
    Bind
          GetName–C.GetName;
          GetPasswd–C.GetPasswd;
          CheckUser–S.CheckUser;
          C.Post–S.GetUser;
          S.GetInfo–U.OutPut;
  }
```

(b)

Fig.2 The architecture description of LoginSys
(a) LoginSys; (b) The DSADL code for LoginSys

After architecture design, we used Unified Modeling Language (UML) (Booch *et al*., 1999) to build static and dynamic model for the objects and their relationship in the TicketSys. For example, class diagram describes the types of every object and their static relationship; interaction diagram describes the interactive relationship among the objects; state chart describes all the possible states the specific objects have and how the events impact the states of the objects. Fig.3 illustrates the class diagram for LoginSys. In this class diagram, CLoginFrm is an html interface for user login, and it is responsible for receive and submit user name and password. CServer running on server verifies user. CServer accesses the database by JDBC (adopting SUN J2EE platform). CUserTable stores user name and password of the registered users.

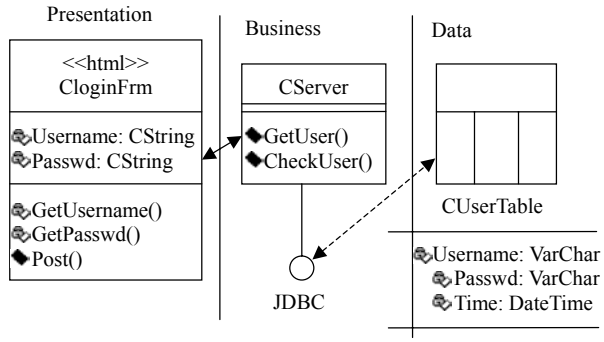


Fig.3 The class diagram for LoginSys

LoginSys has 3-tier structure (Presentation layer, Business layer, and Data layer).

At last, we implemented the TicketSys in J2EE platform. For example, the code for QueryAgent is as follows:

```
Public class QueryAgentBean implements QueryRequest,
SessionBean{
    String m_userID;
    Public void ejbCreate(Object userID) {
       m_userID=(String)userID;
       …
    }
    public TimeTable getTrainTimeTable(Object trainID) {
     TicketInfo
      t_ticketInfo=TicketInfo.getTicketInfo(trainID);
          Return t_ticketInfo.getTimeTable();
    }
  …
}
```

CONCLUSION

In this paper, we refine the characters of distributed software architecture, introduce the extended AG to describe and analyze those characters. We have designed a distributed software architecture description language (DSADL) based on the extended AG, and developed an integrated environment for software architecture design. We have adopted DSADL to analyze and design many large-scale software, such as TicketSys, mobile and embedded E-commerce system. It can be demonstrated by the practice that DSADL can help the programmers to analyze and design distributed software effectively.

References

Berry, G., Boudol, G., 1992. The chemical abstract machine. *Theoretical Computer Science*, **96**:217-248.

Booch, G., James, R., Ivar, J., 1999. The Unified Modeling Language User Guide. Addison Wesley Longman, Reading, MA.

Fang, D.Y., 1998. Computer network protocol specification with attribute grammar. *Journal of Software*, **9**(4): 296-300 (in Chinese).

He, J., Fang, D.Y., Qin, Z., 2002a. A Formal Approach to Distributed Software Architecture. IEEE TENCOM'02. Beijing, China, p.342-36.

He, J., Fang, D.Y., Wang, Z.M., Qin, Z., 2002b. Component based distributed software architectural description language. *Journal of Xi'an Jiaotong University*, **36**(6):612-615 (in Chinese).

Mark, W.M., Emery, D., Rich, H., 2001. Software architecture: introducing IEEE Standard 1471. *Computer*, **34**(4):107-109.

Medvidovic, N., Taylor, R.N., 2000. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, **26**(1):70-91.

Shaw, M., Deline, R., Klen, D.V., 1995. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, **21**(4):314-355.