Abacus: A Service-Oriented Programming Language for Grid Applications

Xiaoning Wang[1, 2, 3] Lijuan Xiao[1, 2, 3] Wei Li[1, 4] Haiyan Yu[1, 4] Zhiwei Xu[1, 4]
[1](Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100080, P.R. China)
[2](Graduate School of the Chinese Academy of Sciences, Beijing, 100039, P.R. China)
[3]{wangxiaoning, xiaolijuan}@software.ict.ac.cn [4]{liwei, yuhaiyan, zxu}@ict.ac.cn

Abstract

This paper presents Abacus, a service-oriented programming language designed for the development of grid applications. Abacus considers that all the grid resources constitute a unified logical address space, where each memory cell holds a resource in the form of a service, and a grid application solves a problem by operating on these memory cells. Abacus allows programmers to concentrate on the logic of their applications, such as service implementation logic, service invocation logic, and glue logic. Low-level details such as resource distribution, resource binding, and service deployment are supported by the compiler and the runtime system. With such virtualization techniques, Abacus helps to enhance the productivity of programmers in developing grid applications.

Keywords: grid computing; service-orientation; programming language; grid application

1. Introduction

In recent years, there has been much research work about implementation technology of grid computing. However, it is still difficult and inefficient to develop grid applications. Most current programming environments do not provide a service abstraction at language level, and programmers have to consider many low-level details when building application codes, such as how to distribute resources in the network, how to deploy services, how to discover and bind services, how to communicate and collaborate, etc. Basic concepts used in current programming environments (e.g., object, inheritance, thread) only provide ways of service implementation, but less useful when specifying service semantics. Moreover, the existence of these concepts weakens the readability of grid applications. The development of grid applications is divided into many separate steps, such as service implementation, service deployment, programming for service invocation logic, etc. It is troublesome to debug and maintain the whole grid application.

We argue that new service-oriented programming language constructs are needed to help build robust grid applications. In recent years, there have been several projects dealing with grid programming, involving programming environment, paradigm and language. Globus Toolkit [5] and WebSphere [3] focus on building a middleware for grid applications. Jini [7], Openwings [9], Microsoft's .NET [8] and CoolTown [6] conform to the SOP (Service-Oriented Programming) paradigm [4]. The XL programming language [2] is designed for the implementation of services in XML Web applications. WSFL [15] and BPEL [1] focus on dealing with service composition and workflow. GSML [16], which is another grid programming language that our group has proposed for non-professional users, focuses on interaction and integration of various resources.

Different from the approaches above, we propose a service-oriented programming language for grid applications. This language, called Abacus, abstracts a services as a language construct. Abacus allows programmers to concentrate on the logic of applications, such as service implementation, service invocation, and glue logic, without caring low-level details. The compiler and the runtime system of Abacus provide supports for virtualization and dynamical binding of services.

With Abacus, a grid is viewed as a virtual supercomputer, and all the resources in the grid constitute a unified logical address space [12]. In this space, each memory cell holds a resource in the form of a service, and primitive operations on each memory cell include writing and executing. Writing operation means deploying a service, while executing operation means invoking a service. A grid application could

IEEE COMPUTER SOCIETY

manage full lifecycle of services based on this high-level logical view of grid services.

This paper is organized as follows. Section 2 introduces the grid address space model, which is the basis of our programming model. Section 3 discusses design considerations of Abacus. Section 4 describes syntax and semantics of the Abacus language. Section 5 describes implementation and evaluation of Abacus. Section 6 discusses the conclusions and the future work of Abacus.

2. Grid Address Space Model

When designing Abacus, we use a grid address space model [12, 13] shown in Fig. 1 as the basis of our programming model.

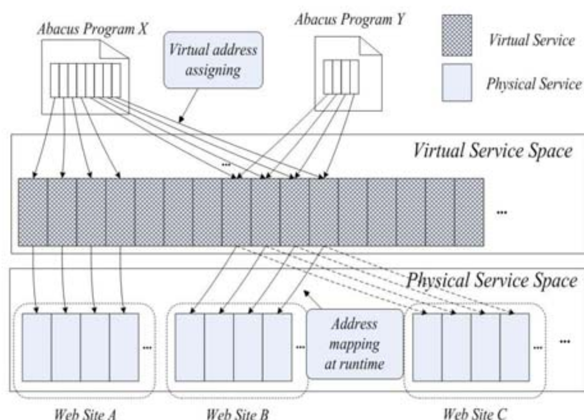

Figure 1. Grid address space model in Abacus

In this model, a grid is constructed on a memory space, with each grid service located in a memory cell. Using this method, a uniform logic view of grid is given to users, and we can access heterogeneous distributed resources in a uniform fashion. This unified resource access interface makes it possible to use general-purpose expressions in Abacus to indicate operations on grid resources.

The grid address space model adopts a two-level address space for service deploying and locating, as shown in Fig. 1.

The bottom level is a URL-based space, where each service is a standard web service, addressed by a uniformed resource locator (URL). This single space is called *physical service space*, for this space is managed by autonomous physical sites. This space provides various physical resources, and each resource exists in the form of a physical service.

The upper level is a UUID-based space, where a service can be addressed by its virtual address, a 32-byte long Universally Unique Identifier (UUID) [10]. This space is called *virtual service space*, for it is maintained implicitly by the Abacus compiler and runtime system. This space provides a location-independent feature for Abacus programs. When an Abacus program moves from one site to another, other programs can still access its services by their virtual addresses. For example, in Fig. 1, when program *Y* moves from web site *B* to *C*, program *X* can still access the services provided by *Y* by their virtual addresses.

At compiling time, the Abacus compiler will assign a 32-byte long virtual address for each shared service declared in Abacus source files. At runtime, when deploying a service, the runtime system will first allocate resources and assign a physical address in the physical service space for the service, and then build a map between a virtual address and the corresponding physical address. With this mapping, other programs can locate and access physical services by their virtual address.

Using this model, programmers can focus on the functionality and quality of services to solve their application problems, without worrying about the location and dynamic properties of grid resources, which will be hidden by the virtual service space. At runtime, many low-level tasks such as service deploying, locating, destroying, resource scheduling, and error recovery will be transparently managed by a substrate runtime system, which frees programmers from the burden of dealing with these issues.

3. Language Design Principles

The Abacus programming language aims to provide a more effective way to develop robust grid applications, and to reduce the development cost. The design of Abacus focuses on how to abstract a service as a language construct and how to support management of the full life cycle of a service. The following outlines design considerations of Abacus.

Service abstraction. The core concept of Abacus is the service construct. A service is a first-class object in the language. In an Abacus program, a grid resource is encapsulated into a service, and services with the same construction are abstracted into a language construct. The declaration of a service construct is composed of a series of declarations of data and interfaces, and the construct encapsulates properties of resources and all the operations on resources. Due to the intrinsic loosely-coupled nature of a service, Abacus does not support inheritance and polymorphism features of object-oriented programming languages.

Service management. Abacus provides a flexible way to support management of the full lifecycle of a service, including service deploying, service invoking and service undeploying. In Abacus, the value held in a

IEEE COMPUTER SOCIETY

variable of a service type is the virtual address of a service, which makes service invocation independent of physical properties of the service. Service virtualization guarantees that applications can be performed normally when resources change. In addition, dynamically deploying function, provided by the runtime system of Abacus, can deploy a service into the grid at runtime.

Application development. Abacus should support both service implementation and service invocation. In most current programming environments, service implementation and service invocation are separated even if they are both parts of the same application, which makes the development complex. In Abacus, a declared service construct is a data type in the language. Multiple variables can be declared with this service type to hold different service instances. Abacus supports primitive operations on a variable of a service type, including service deploying and service invoking at language level. This way, a grid application could program both service invocation and service implementation, and the whole development process can be completed within three steps, namely, coding, compiling, and executing.

Compatibility and partial failures. Abacus should provide mechanisms to be compatible with existing applications. There are a lot of applications before grid and they are not designed for grid. If these applications can be integrated into the grid conveniently, grid computing will be more prevalent. Moreover, Abacus should offer programmers a powerful and flexible way to handle various faults that happen during the runtime.

4. Design Issues of Abacus Language

In the Abacus language, a compilation unit is composed of one or more Abacus source files, which contains service declarations. The method named main declared in some service construct is the execution entrance of a grid application, and all the implementations of application logic in the Abacus programs are done through a series of operations on services.

4.1. Service Declaration

Service is the most important construct in the Abacus programming language. Its declaration syntax can be expressed in the following BNF:

ServiceDeclaration:
  service *identifier ServiceBody*
  static service *identifier ServiceBody*
ServiceBody:
  { *ServiceBodyDeclarations* }
ServiceBodyDeclarations:
  ServiceBodyDeclaration ServiceBodyDeclarations
  ServiceBodyDeclaration
ServiceBodyDeclaration:
  FieldDeclaration
  ConstructorDeclaration
  MethodDeclaration
MethodDeclaration:
  PublishedMethodDeclaration
  PrivateMethodDeclaration

The keyword “static” denotes the deploy scope of a service. The service declaration with “static” modifier defines a service that maintains state information between message calls, while the service declaration without “static” modifier defines a service that does not maintains state information. A stateful service can be used for encapsulating and maintaining resources.

A field declaration declares all the data and properties of resources. A service constructor plays a similar role to the constructor in an object, which initializes a service instance, but it has no parameters. Published methods declare interfaces that other services or applications can use. Private methods can only be used in the current service.

Services declared in this form will be deployed dynamically during the runtime and managed by the application.

4.2. Service Operations

In a grid application, the primary operations of a service include service deploying, service binding, and service invoking.

In the Abacus programming language, the operation of service deploying can be done through a service creation statement:

NewExpression:
  New *identifier*();

The string *identifier* is the name of the service declared as above which is considered to be a data type in the Abacus programming language. The service creation statement will dynamically deploy a service to some memory cell in the grid address space, and return its virtual address. This operation hides the details of service deployment and service instance creation. At the same time, it implicitly binds the application with the service just deployed.

The syntax of service invocation is similar as the method invocation of an object in Java Language:

MethodCall:
  identifier1.identfier2(*parameterlist*);

Here *identifier1* is the name of a service variable, *identifier2* is an interface name of the service, and

IEEE COMPUTER SOCIETY

parameterlist lists all the necessary parameters of the service invocation.

An important issue about services is sharing. In the Abacus programming language, services are shared through the declaration of global variables and extern variables. These two kinds of variables are declared outside of any service construct declaration.

Service providers share their services by declaration of global service variables, and the services holden in global variables are their shared services. Global variables have the same declaration syntax as common variables:

GlobalDeclaration:
  ServiceType identifier;

Here *ServiceType* is the type of the variable, and *identifier* denotes the name of the variable.

When a grid application uses a service that has been deployed in the grid, it shares this service through declaration of an extern variable, with the format:

ExternDeclaration:
  extern *ServiceType identifier1.identifier2*;

The declaration of an extern variable binds the application with a shared service. *ServiceType* is the type of the variable, *identifier1* is the library file name provided by a service provider, and *identifier2* declares the name of the variable that has been declared at the provider application.

A *library* mentioned above is a file that is a kind of output of Abacus complier at service providers, which records information of global service variables that an application shares and the description of all the available published interfaces. Service customers can not use services provided by others without corresponding library files.

4.3 Other Issues

The type system in the Abacus programming language includes three kinds of types: primitive types, array types and service types. Primitive types include **boolean**, **byte, char, short**, **int**, **long**, **float**, and **double**. An Array type is a kind of extend type in the Abacus language, and its corresponding value is a reference to an array entity. Besides, a service type can be used to declare service variables, and also can be used as parameter type or return result type of a method.

The Abacus programming language retains the commonly used control statements (such as if, switch, for, while, and do) and operators (such as addition, subtraction, multiplication and division) in traditional imperative programming language. It adopts a similar syntax to C and Java.

Finally, exception handling is rudimentary in the current version of Abacus design and implementation. Exceptions are not directly supported at the language level (e.g., by assigning a type). Instead, a simple way is adopted that every exception can be thrown along with a string carrying its information. Procedures implementing operations in services can throw exceptions. Service invokers must try and catch exceptions.

5. Implementation and Evaluation

5.1. Abacus Implementation

The runtime environment of the Abacus programming language is provided by Vega GOS [17]. A compiler based on this system for the Abacus has been implemented.

Vega Grid Operating System (Vega GOS) is a grid system software package developed by Institute of Computing Technology, Chinese Academy of Sciences. The most useful features of Vega GOS for Abacus are translation from a virtual service to a physical service and the management of service life cycle.

In Abacus, the value of a service variable is the virtual address of a service. Resource binding occurs at runtime. Therefore, translation from a virtual service into a physical service is required to be supported by the runtime system of Abacus. In Vega GOS, resource routers [14] and name servers can accomplish this translation and locate the physical service. The resource binding at runtime is transparent to programmers and users.

In Abacus, all the services are deployed at runtime. Dynamic Deploy Service (DDS) provided by Vega GOS could receive a service implementation file (in jar format) from a grid application and automatically deploy it into a service container during the runtime. A service creation statement at language level expresses the semantics of this action.

A grid application programmed in Abacus is made up of one or more service definition files. All these files will be translated by Abacus compiler into respective service execution files (jar files and corresponding WSDD files) that can run on the Vega GOS. The performing of a program is also service invocation, and this executing entrance is the main method defined in this service. Fig. 2 shows the compiling and running process of an Abacus program.

IEEE COMPUTER SOCIETY

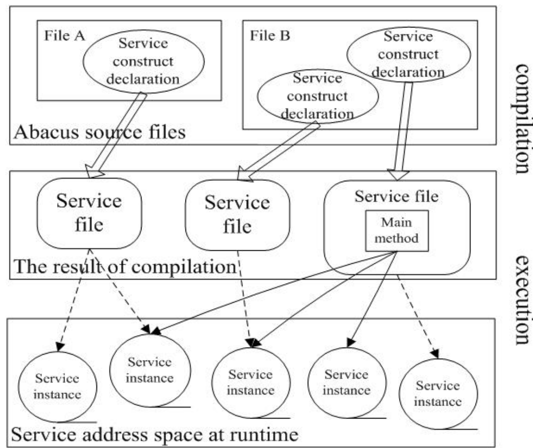

Figure 2. Compiling and running of an Abacus program

In Fig. 2, the Abacus program is composed of two source files, named *A.abacus* and *B.abacus*. The file *A.abacus* defines one service construct, and the file *B.abacus* defines two service constructs. The wide arrow in the figure means compilation by Abacus compiler. The program can be compiled by the command:

```
>AbacusC A.abacus B.abacus
```

After compilation, each service construct is compiled to respective service files named in terms of corresponding service name. If the main method is defined in the service named *Test*, the program can be executed using the command:

```
>Abacus Test
```

As a result, the program will execute from the starting of the main method in the service of *Test*. During the runtime, the program may dynamically deploy some services into the grid address space, or may use some services existing in the grid address space. The grid address space is global, which is shared by all the Abacus programs. In the figure above, the arrow with real line represents directly invoking an existing service, and the arrow with broken line represents dynamically deploying a service with service files. From the figure, it is clear that the development of a grid application can be completed within three steps, which are coding, compiling, and executing.

5.2. Abacus Evaluation

In this part, we evaluate the Abacus programming language, focusing on its impact on the development (including maintenance) process of grid applications. The evaluation criteria of a programming language include readability, writability, reliability, and cost [11]. Although such criteria are impossible to get even two computer scientists to agree on the value of some given language characteristic relative to others, most computer scientists would agree that these criteria are important.

Before the Abacus programming language is evaluated, a simple example of a grid application programmed in Java based on axis plus tomcat framework is given at first. The example is about querying stocks information.

A provider who gives a service about querying stocks information need implement corresponding functions by constructing a class or any other way.

```
package xmltoday_delayed_quotes;
class Quote{
        ……
        public float getQuote(String stock){
        //the implementation code of the method
        ……
        }
        ……
}
```

After being compiled, all the class files those are related service implementation need to be added into a jar file. Besides, a WSDD file, which holds all the information about service deployment, is needed.

```
<deployment
xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/provider
s/java">
    <service name=" QuoteService"
provider="java:RPC">
    <parameter name="className" value="
xmltoday_delayed_quotes.Quote"/>
    <parameter name="allowedMethods" value="*"/>
    <parameter name="scope" value="request"/>
    </service>
</deployment>
```

Moreover, developers have to deploy this service manually using the command:

```
>java org.apache.axis.client.AdminClient deploy.wsdd
```

A client who wants to get some information about the given stoke must know the physical address of the service (i.e. the URL of a stock querying service) before he is able to use this service. As a result, he gets the WDSL file of the service. In the next step, he has to build stubs on his local node for the service and get corresponding locator class and stub class. After all these are done, a client program can be written in Java.

IEEE COMPUTER SOCIETY

```
// import the service interface definitions and their
// helper classes
import xmltoday_delayed_quotes.GetQuote;
import
xmltoday_delayed_quotes.GetQuoteServiceLocator;
public class MyClient {
	public static void main(String [] args) throws
Exception {
// get the service from the service locator helper class
// and the physical address is record in this locator help
// class
	GetQuoteServiceLocator locator = new
GetQuoteServiceLocator();
	GetQuote getQuote = locator.getGetQuote();
	//invoke the service
	float result = getQuote.getQuote("Dummy, Inc.");
	System.out.println("got result: " + result);
	}
}
```

After being compiled, this program can be executed using java command.

If this example is implemented in Abacus, the codes of the provider program are as follows:

```
QuoteService quoteService;
service QuoteService{
	published float getQuote(String stock){
	//the implementation code of the method
	……
	}
	published void main(String[] args){
		quoteService = new QuoteService();
	}
}
```

A client program takes a library file named *QuoteService.lib* from the service provider, and the codes can be:

```
extern QuoteService QuoteService.quoteService;
extern IOService System.ioService;
service MyClient{
	published void main(String[] args){
	//invoke the service
	float result = quoteService.getQuote("Dummy,
Inc.");
	ioService.println ("got result: " + result);
	}
}
```

Both service provider program and service client program can be executed after being compiled without any other work.

From the example above, some advantages of the Abacus programming language are concluded as follows:

Improved productivity. The Abacus programming language simplifies the development of a grid application. Fig. 3 compares two kinds of development steps.

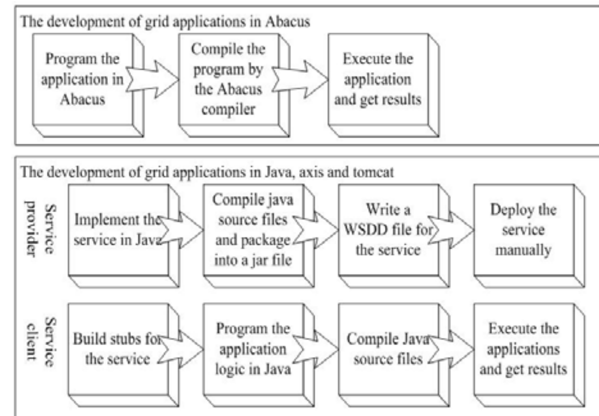

Figure 3. Comparison on development steps

A grid application contains some necessary elements and actions. For a service provider, service implementation codes, service deployment information, and the action of service deploying are necessary; and for a client, stubs or physical address information of services is necessary.

For the development in Java, service implementation codes are composed of some related Java classes; necessary information about service deployment is recorded in a WSDD file which programmers have to write; and the action of service deploying is completed by programmers manually. Also, a client has to build stubs before he can invoke services.

In a grid application programmed in Abacus, service implementation is expressed in service constructs in the language, and a service construct have many corresponding service instances at runtime; most of service deployment information has been hidden; the keyword “static” when declaring a service construct denotes the deploy scope of a service; and a service creation statement performs the behavior of service deploying. Besides, stubs are also transparent to clients.

Consequently, programmers have not to hold much detail knowledge at low-level when developing a grid application in Abacus. At the same time, the Abacus programming language could facilitate the development of a grid application efficiently.

IEEE COMPUTER SOCIETY

Increased resource utilization. The Abacus programming language makes grid applications flexibly and fully utilize grid resources.

In the programming environment composed of Java, Axis and Tomcat, the physical address information of services is a part of source files in a grid application. When the location of some service changes, the codes have to been modified and compiled again before being performed normally.

Abacus supports service virtualization. A program in Abacus utilizes virtual services to solve a problem, and service binding occurs at runtime. If the location of some service changes, the Abacus program does not have to be modified. When the number of available resources in the grid changes, the Abacus program can suit the change automatically.

Flexible support of service composition. The Abacus programming language supports primitive service composition. Abacus provides primitive types, variables, primitive arithmetical and logical operations, and basic control statements, which could express the logic of service composition. Besides, Abacus considers a service as a data type that can be declared to be several variables, and each service variable holds a service instance, which can be passed as a parameter or as a return value of an interface. This way, a more flexible way is provided to support service composition. Since Abacus is a general-purpose programming language, it does not support workflow of services.

Improved readability. The Abacus programming language improves the readability of programs by encapsulating a service into a construct. In the example above, the program in Java invokes a service by some related classes, which does not express semantics of a service literally, in spite that its function is to invoke a service. However, the variable *quoteService* in the Abacus program represents a service literally, and supports all the operations on this service. The whole program expresses clear service invocation logic to programmers. Compared with grid programs in traditional languages, Abacus programs are easy to read and understood, which is good to maintenance of grid applications.

To sum up, the Abacus programming language could facilitate the development of grid applications, reduce the development cost, and increase programmers' productivity efficiently.

6. Conclusion

In this paper, we sketched design and initial implementation of the Abacus programming language, and gave a qualitative evaluation. Abacus abstracts services as a language construct and provides some primitive operations on service variables, hiding many low-level details, which frees programmers to concentrate on the logic of their applications. Simple concepts representing services in Abacus help enhance the readability of programs. Moreover, Abacus decreases development complexity, which is helpful to debug and maintain the whole grid application. Therefore, Abacus helps enhance the productivity of programmers in developing grid applications.

There is a lot of work to do in the future. Most importantly, how does Abacus integrate existing applications? Full lifecycle management of services is another research topic. For example, when should a service be undeployed, and how does Abacus support this operation? How does Abacus provide supports for managing and controlling physical locations of services? Other problems include service type casting, asynchronous invocation of services, exception handling, and runtime resource adaptation.

Our future development plan includes coding e-science and e-business applications in Abacus to test and evaluate the language. Another main development effort is to construct a full-fledged compiler that can implement Abacus on Vega GOS, Globus, and the UK e-Science OMII platforms.

Acknowledgement

This work is supported in part by the National Natural Science Foundation of China (Grant No. 90412010), the China Ministry of Science and Technology 863 Program (Grant No. 2002AA104310) and 973 Program (Grant No. 2003CB317000), and the Chinese Academy of Sciences Hundred-Talents Project (Grant No. 20044040). The authors are very grateful for these supports. Besides, the authors are also grateful to anonymous referees for their constructive comments.

References

[1] BPEL, http://www-106.ibm.com/developerworks/library/ws-bpel/.

[2] D. Florescu, A. Grunhagen, and D. Kossmann, "XL: A XML programming language for web service specification and composition", *Computer Networks*, Elsevier Science B.V., 2003, pp. 641–660.

[3] E. Bayeh, "The WebSphere Application Server Architecture and Programming Model", *IBM Systems Journal,* April, 1998, pp. 336-348.

[4] G. Bieber, J. Carpenter. "Introduction to Service-Oriented Programming", http://www.openwings.org/.

[5] Globus. http://www.globus.org/.

[6] HP Cooltown. http://cooltown.hp.com/.

[7] Jini. http://www.jini.org.

0-7695-2408-7/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

[8] Microsoft .NET. http://www.microsoft.com/net.
[9] Openwings. http://www.openwings.org.
[10] P. Leach, M. Mealling, and R. Salz, “A UUID URN Namespace”, draft-mealling-uuid-urn-05 (work in progress), December, 2004, http://www.ietf.org/internet-drafts/draft-mealling-uuid-urn-05.txt.
[11] R.W. Sebesta, *Concepts of Programming Languages (5th edition)*, Addison-Wesley, Boston, 2002.
[12] W. Li, and Z. Xu, "Model of grid address space with applications," *Journal of Computer Research and Development*, Science Press, Beijing, 2003, pp. 1756.
[13] W. Li, Z. Xu, L. Cha, H. Yu, J. Qiu, and Y. Zhang, "A Service Management Scheme for Grid Systems, " *The Second International Workshop on Grid and Cooperative Computing*, Springer-Verlag, Berlin, 2003, pp. 541-548.
[14] W. Li, Z. Xu, F. Dong, and J. Zhang, “Grid resource discovery based on a routing-transferring model”, *Proceedings of International Workshop on Grid Computing*, Springer, Berlin, November, 2002, pp. 145-156.
[15] WSFL, http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.
[16] Z. Xu, W. Li, D. Liu, H. Yu, and B. Li, "The GSML tool suite: A supporting environment for user-level programming in grids," *Proceedings of Parallel and Distributed Computing, Applications and Technologies*, August, 2003, pp. 629-633.
[17] Z. Xu, W. Li, “The Research on Vega Grid Architecture”, *Journal of Computer Research and Development*, Science Press, Beijing, 2002, pp. 923-929.

IEEE COMPUTER SOCIETY