

# GSiB: PSE Infrastructure for Dynamic Service-Oriented Grid Applications

Yan Huang

Department of Computer Science  
Cardiff University  
PO Box 916  
Cardiff CF24 3XF  
United Kingdom  
[Yan.Huang@cs.cardiff.ac.uk](mailto:Yan.Huang@cs.cardiff.ac.uk)

**Abstract.** This paper describes the Grid-Service-in-a-Box(GSiB) system – a visual environment for both service providers and service clients who wish to deploy, manage, and use services in a Grid environment, without having to be experts in Grid and Web service technologies. GSiB provides a variety of easy-to-use graphical user interfaces for service providers to deploy, undeploy, update, compose, and monitor services, and for service clients to generate service-composite applications, and to track and monitor submitted jobs. In this paper, these interfaces are described and several implementation issues are addressed.

## 1 Overview

Building on Grid and Web service technologies, the Open Grid Services Architecture (OGSA) proposes a new generation of Grid infrastructure, in which services act as the building blocks of the Grid system. OGSA abstraction [1, 2] is quite a new concept and presents a lot of challenges for the scientists who wish to use it. In general, these scientists cannot be expected to keep abreast of the rapid development of Grid technologies, so easy-to-use tools and environments are needed to allow scientists to take full advantage of service-oriented Grids, such as those based on OGSA, without them having detailed knowledge of the underlying infrastructure. This is the main motivation for the Grid-Service-in-a-Box (GSiB) system. The GSiB project is developing a visual problem-solving environment for both service providers and service clients. For the service provider GSiB provides a visual environment for creating and publishing Grid services from existing software and libraries, browsing, monitoring and querying Grid services, and composing higher-level Grid services from other services. For service clients, GSiB presents a visual environment for browsing and querying Grid services and creating Grid applications by composing Grid services. It also provides an execution environment for running Grid applications.

GSiB is designed to function in conjunction with OGSA-compliant Grid systems, as well as with other similar service-oriented architectures. In this context we refer generically to a Grid service as any service that is consistent with this

architecture. We assume that all Grid resources are accessible as services, and that there are standard mechanisms for service publication, service querying and service invocation. In [5] and [4], we have explored techniques for building such a system and a brief description of these will be given in section 3.

## 2 GSiB Interfaces and Functionality

GSiB provides an easy-to-use graphical user interface for service providers and clients that allows users to take advantage of Grid services even though they may have little knowledge of Grid computing and Web services. The details of the underlying technologies are largely hidden within the GSiB system. GSiB consists of two packages: the Service Provider GUI (SP-GUI) package, and the Service Client GUI (SC-GUI) package. The interfaces and functionalities of these opackages are discussed in the following subsections:

### 2.1 The Service Provider GUI Package

The SP-GUI package contains a variety of graphical interfaces for service providers to easily and quickly create Grid services from existing legacy software and libraries, to update, monitor and manage Grid services, and to compose more complex Grid services based on existing Grid services. GSiB provides mechanisms for supporting the updating of Grid services and fault-tolerant lifecycle management. The SP-GUI package contains the following graphical interfaces and components:

- *Login Interface* for users to log in as authorized service providers.
- *Service Deployment/Undeployment Interface* for deployment and undeployment of services. This provides the following functionality:
  - Automatic deployment of existing software routines or libraries as OGSA-compliant Grid services. Java, C, and C++ are supported as the native languages of the service deployed by this mechanism.
  - Visual Service Composition Environment (VSCE) for service providers for building and deploying a service composed of existing services by visually drawing a diagram representing the workflow of the composite service.
  - Updating of existing Grid services. When a service is updated, its implementation or/and interface may be changed, so it is important to consider how to avoid interrupting running applications and other high-level services that are built on top of the updated services.
  - Undeployment of an existing Grid service. This also involves consideration of how to avoid interrupting a running application and disrupting high-level services.
- *Service Configuration Interface* for setting up or changing a service's security policy and lifetime.
- *Service Monitoring Interface* for dynamically monitoring the distribution and workload of Grid services.

- Other interfaces include an *Authorization Interface* for a user to log in as an authorized service administrator; an *Authentication Interface* for setting up a Grid service so that it is accessible by only authorized users for a required or assigned time; and, a *Lifecycle Management Interface* for dealing with any failures, and reclaiming service and state associated with failed operations.

## 2.2 The Service Client GUI Package

The SC-GUI package presents a variety of graphical interfaces for a service client to browse and query Grid services, create Grid applications, submit jobs, monitor the execution of jobs, and retrieve results. The SC-GUI package contains the following interfaces and components:

- *Login Interface* for users to log in as authorized service users.
- The *Visual Service Composition Environment* for service clients is similar to the VSCE for service providers, except that it produces a composite Grid application rather than a composite higher-level Grid service. A VSCE allows a user to create a representation of a Grid application by drawing its workflow. The graphical model created using VSCE can be transformed into an XML-based workflow description, and can also be sent to a Workflow Engine to be executed. This is discussed further in Section 4.
- The *Workflow Engine Component* provides an execution environment for Grid applications that are described in an XML-based workflow description language.
- The *Job Tracking and Monitoring Interface* allows users to track and monitor the jobs they have submitted.
- A *Service Browser* for browsing and querying services.

## 3 Implementation Aspects

### 3.1 Service Deployment

Usually additional code, such as a wrapper, is needed to make a piece of software available as a Web service. This can result in a lot of repeated work, especially when a large library of routines is to be wrapped as Web services. A legacy problem exists when the implementation language of the services and the language supported by the service hosting environments are different. Thus, service deployment is one of the main issues addressed by the SP-GUI package.

As mentioned above, we assume that Grid services are Web service compatible, which means that they are accessible by general Web service users regardless of how they are implemented. The public interfaces and bindings of a Web service are described in an XML document, and it is registered in a service registry to allow service discovery and querying [8]. The SP-GUI package can be used by service providers to publish existing software and libraries as Web services. Although Java is used to build the system framework, the implementation language of the services supported by the SP-GUI is not limited to Java – it can

be Java, C, or C++. In the implementation of the SP-GUI package, a Java-C Automatic Wrapper (JACAW) is used to automatically generate the Java JNI wrapping code for the legacy software [3]. This JNI wrapper provides a Java interface for the wrapped legacy code and invokes the original routine which is compiled into a dynamical library. Thus, the wrapped legacy code can be invoked and run in the same way as any other Java code. To deploy a Java program as a Web service the Axis Java2WSDL utility is used to generate the Web Service Description Language (WSDL) document from the Java code. The service is then published into a UDDI (Universal Description, Discovery and Integration) registry server by using UDDI4J which provides an Java API for interaction with a UDDI registry [7]. Figure 1 illustrates this service deployment scenario.

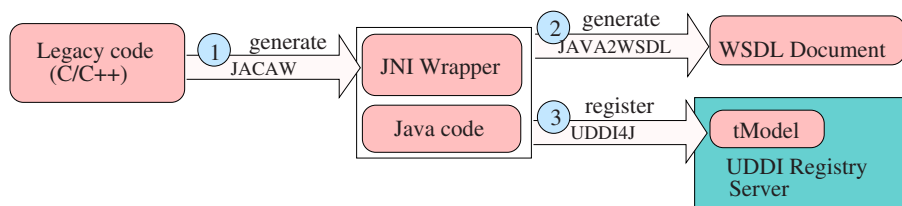


Fig. 1. Service deployment in the SP-GUI package.









### 3.2 Service Composition

It is useful to be able to build new services and applications through the composition of existing services, and this task can be made easier by using visual programming tools. GSiB provides a Visual Service Composition Environment (VSCE) for both service and application composition that allows a user to create a new composite service or application by drawing its workflow graph. The VSCE lists all the known service instances matching some set of requirements specified by the user, and allows the user to choose from these based on dynamically discovered metrics such as speed, workload, and network latency of the service instances<sup>1</sup>.

In the workflow representation, or flow model, an *activity* represents a task to be performed as a single step within a process, and is implemented by one of the operations provided by a Web service. There are three kinds of activities: normal activity, assignment activity and control activity. A normal activity is implemented by an operation of a Web service; an assignment activity specifies an assignment normally between a variable and the output of an activity; a control activity specifies a control process such as a loop or conditional construct. The workflow representation is a directed graph, similar to a task graph, and

<sup>1</sup> In the future this task will be done automatically by a resource manager.

the activities correspond to nodes in the graph. Activities are wired together through *control links*. Each control link is a directed edge of the graph, and together they represent the control flow of the flow model. A *data link* is a second kind of directed edge of the graph and specifies the flow of data between a source activity and a target activity. A *data map* that describes how a field in the message part of the target's input message is constructed from a field part of the source's output message is defined for each data link. A flow model itself also has input and output: a *flow source* is a source of the data links targeting activities in the flow model, and a *flow sink* is a target of the data links of the flow model. Figure 2 shows the graphical symbols used in displaying a flow model.

Graphic Symbol	JSFL Symbol
	Data input port
	Data output port
	Control input port
	Control output port
	Data Link
	Control Link
	Normal and assign activity
	Control activity

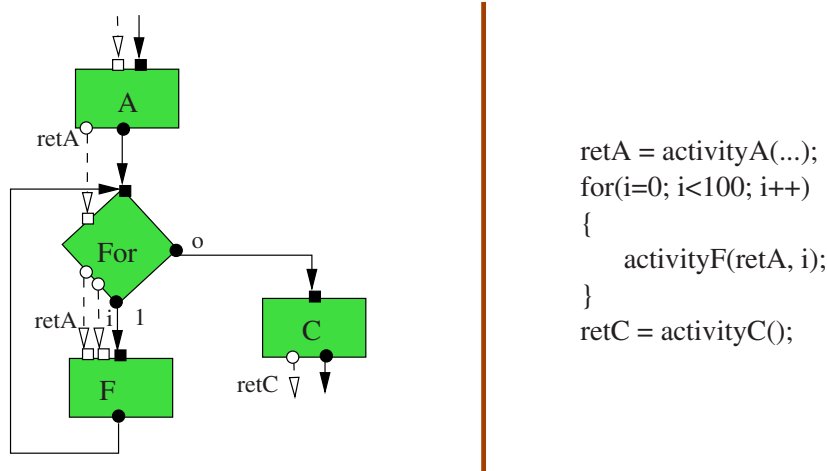
**Fig. 2.** Graphical symbols for constructing a flow model.

Figure 3 gives an example of the directed graph of a job flow model, together with the corresponding Java code.

After the graphical flow model representing an application has been created using VSCE, GSiB can convert it into an XML description document, and then (if the user requests it) the application can be submitted for execution by passing its this XML document to a workflow engine, as described in Section 5.

**3.3 Service Workflow Language**

One of the main aims of GSiB is to provide a visual composition and editing tool to create and submit service-based applications, described in an XML document and based on a standard job description language. Figure 4 shows how a user can create and run a service-based application through a drag-and-drop interface. In this scenario, the workflow engine is a service that accepts an XML document



**Fig. 3.** The graphical workflow model of a *for* loop example.

describing an application, automatically produces a corresponding Java code, compiles the code, and executes it.

In building a Visual Service Composition Environment, choosing and defining a job description language is one of the critical issues. A VSCE must be able to convert a graphical job flow model into an XML-based description document, and then from this XML description document into a distributed Java code. It would be more convenient if two-way conversion is available because, for example, after saving a job as a XML document, the user might want to modify the job again. Thus, the mapping between the XML job description language and job's graphical flow model, and the mapping between the XML job description language and the Java program are the main requirements in defining a job description language. Therefore, an XML job description language that allows documents to be readily converted to both a graphical representation and a Java program meets our requirements.

Web Service Flow Language (WSFL) warrants attention because it can describe a composite job in a graphical form. Its activities can be mapped into nodes of the graph, and its data links and control links can be mapped onto directed edges of different kinds. However WSFL has its limitations, such as a lack of a one-to-one mapping between Java programming constructs and WSFL processes, inflexibility in how the variables that are used as parameters in the boolean expression of the transition and exit conditions are specified, and difficulties in displaying WSFL control processes in a clear and understandable graphical form [5]. We have, therefore, extended WSFL to create the Service Workflow Language (SWFL), which overcomes the limitations of WSFL and

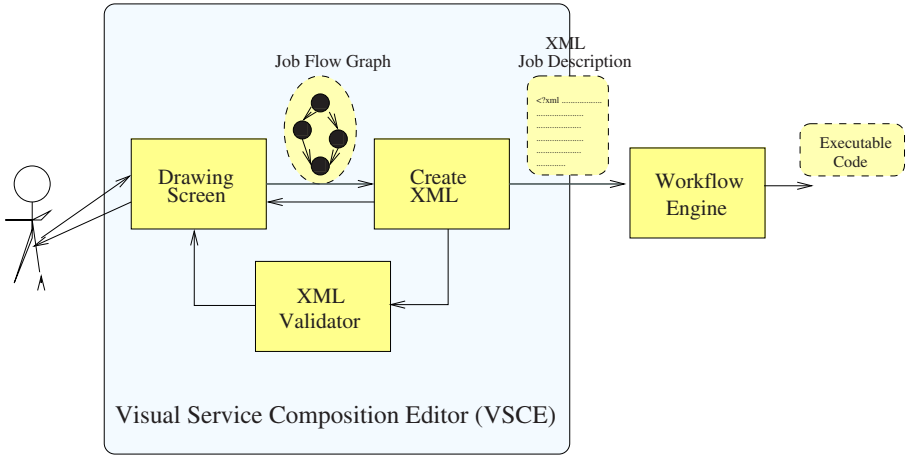


Fig. 4. Visual service composition.

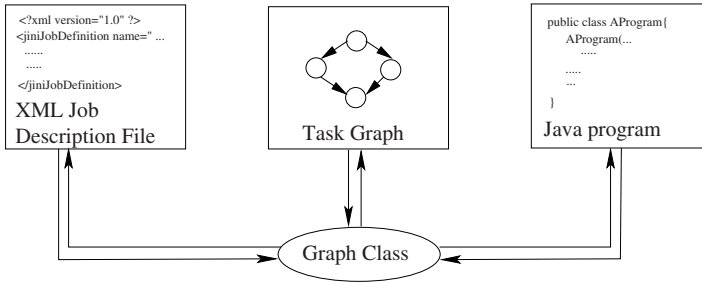
meets the above requirements for a job description language. The details of SWFL is available in [5].

## 4 SWFL2Java

We have developed a tool, called **SWFL2Java**, that converts the description of a job in SWFL into executable Java code. The details of **SWFL2Java** are discussed elsewhere [5], however, we will give here an overview of the implementation. In **SWFL2Java** an SWFL document is not translated directly into a Java program but is stored in an intermediate form as a Java *FlowModel* object. This is made up of two Java *Graph* objects: *DataGraph* and *ControlGraph*. The former stores the data flow structure of the flow model, and the latter stores its control flow structure. One reason for storing the job description in this intermediate form is to be able to interact readily with Java-based graphical tools, such as VSCE, for the visual composition of Web services. The graph created with VSCE can be stored as a *FlowModel* object and converted to and from SWFL, as well as into a Java program. Another reason for using the *FlowModel* form is to reduce the overhead when the same job is used many times and scheduled on different resources. In such cases it is easier to generate the Java code from the intermediate form rather than starting from the original SWFL. Figure 5 illustrates the importance of the *FlowModel* representation of a composite application.

## 5 Workflow Engine

After it is generated by VSCE, a composite service-based application may then be submitted to a Workflow Engine to be executed. A Workflow Engine provides



**Fig. 5.** Using a *FlowModel* object as an intermediate representation in *SWFL2Java*.

an execution environment for such applications described in an SWFL document. We have implemented such a Workflow Engine as part of our Jini-based Service Grid Architecture (JISGA) *citeThesis*.

In JISGA, the *WorkflowEngine* service supports both blocking and non-blocking submission of applications, as well as their sequential and parallel processing. For blocking sequential jobs, the Workflow Engine processes the job by generating the Java harness code using *SWFL2Java*, executing it, and sending the result to the job submitter. In the case of non-blocking and/or parallel jobs, Jini's *JavaSpaces* plays a key role in the job processing.

*JavaSpaces* provides object storage services through a tuple-space abstraction, and supports the sharing of objects between distributed applications [6]. In JISGA, *JavaSpaces* is used in the following ways:

1. As a shared memory, allowing message passing between different services and processes.
2. To store the *result* objects of non-blocking jobs so that job submitters can retrieve their job results later.
3. To store *message* objects used in communication between sub-job processes, thereby allowing job parallel processing.
4. In maintaining the two job queues of the JISGA system so that jobs and sub-jobs can be processed in an FIFO (First In First Out) order.

The JISGA environment makes use of two job queues. One is the Parallel-Job Queue, containing parallel jobs to be processed. The other is the Sub-Job Queue, containing sub-jobs generated from the partitioning of parallel jobs. The *JSHousekeeper* performs housekeeping tasks in *JavaSpaces* serving in the system. Its duty is to collect *result* objects, update job queues, and maintain data messages. It is also a service, and provides an interface for job submitters to retrieve their job results.

The *WorkflowEngine* and *JobProcessor* services both deal with the processing of parallel jobs. After the parallel job is submitted, the *WorkflowEngine* service will leave the job for the *JobProcessor* service to process by adding the job to the Parallel-Job Queue. A *JobProcessor* is a service working internally in the



environment supported by a number of workflow engine services. Its main duty is to process parallel jobs, which includes partitioning a parallel job into sub-jobs, adding these sub-jobs to the Sub-Job Queue, and processing sub-jobs in the Sub-Job Queue. There are two threads running synchronously in a *JobProcessor*. One of the threads takes one job at a time from the Parallel-Job Queue and then partitions the job into sub-jobs which are then added to the Sub-Job Queue. Another thread takes one sub-job a time from the Sub-Job Queue and processes it. As many as *WorkflowEngine* and *JobProcessor* services as deemed necessary may exist in the environment they support. It is up to the system administrator to determine the number of *WorkflowEngine* services and *JobProcessor* processes. However, more than one *JobProcessor* processes are needed to allow the processing of parallel jobs.

## 6 Concluding Remarks

The main motivation for developing GSiB was to provide a user-friendly visual PSE to assist computational scientists to build and run composite service-based applications. A variety of visual interfaces are supported in GSiB to achieve this aim. The GSiB interfaces are divided into two packages: Service Provider GUI (SP-GUI) and Service Client GUI (SC-GUI). The SP-GUI package supports deploying, undeploying updating and monitoring services, as well as service composition. The SC-GUI package supports the creation of composite service-based applications, job tracking and monitoring, and service advertising and subscription.

Several issues in the implementation of the underlying infrastructure have been addressed in the paper. The legacy problem in deploying existing software as services is solved by using JACAW to automatically generate JNI wrappers for C legacy codes. By using VSCE users can generate composite services and applications through a simple “drag-and-drop” interface. The graphical representation of the flow model of a composite service-based application has been discussed, and SWFL, an XML-based language for describing such applications has been introduced. SWFL is an extension of WSFL that can describe Java-like conditional and loop constructs, permits sequences of more than one service within conditional clauses and loop bodies, and overcomes limitations inherent in WSFL’s data mapping approach. Given a SWFL job description the *SWFL2Java* tool can generate a representation of the corresponding data and control link structure in the form of a Java *FlowModel* object. An application created with VSCE and described in SWFL can be submitted to a Workflow Engine that provides an execution environment for such applications. By using *SWFL2Java*, code based on the SWFL document can be dynamically generated and run. The Workflow Engine supports blocking and non-blocking job submission, and provides both sequential and parallel job processing.

## References

1. I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed System Integration," [http://www.gridforum.org/ogsi-wg/drafts/ogsa\\_draft2.9\\_2002-06-22.pdf](http://www.gridforum.org/ogsi-wg/drafts/ogsa_draft2.9_2002-06-22.pdf).
2. S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, and P. Vanderbilt, "Grid Service Specification," <http://www.gridforum.org/ogsi-wg/drafts/draft-ggf-ogsi-gridservice-04.2002-10-04.pdf>.
3. Y. Huang, I. Taylor, D. Walker, and R. Davies, "Wrapping Legacy Codes for Grid-Based Applications," in Proceedings of the Fifth International Workshop on Java for Parallel and Distributed Computing, held as part of the International Parallel and Distributed Processing Symposium (IPDPS) in Nice, France, April 2003.
4. Y. Huang and D. Walker, "JSFL: An Extension to WSFL for Composing Web Services," UK e-science All Hands Meeting, 2-4 September 2002.
5. Y. Huang, "The Role of Jini in a Service-Oriented Architecture for Grid Computing," Ph.D Thesis, Department of Computer Science, Cardiff University, January, 2003
6. W. Keith Edwards and Tom Rodden, "Jini Example By Example," Sun Microsystems Press, 2001
7. "UDDI4J Overview," <http://oss.software.ibm.com/developerworks/opensource/uddi4j/>
8. "Web Services Architecture Requirements," <http://www.w3.org/TR/wsa-reqs>, April, 2002