Symphony: A Java-based Composition and Manipulation Framework for Distributed Legacy Resources

Ashish Shah
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
ashah@microsoft.com

Dennis Kafura
Virginia Polytechnic Institute and State University
Department of Computer Science
Blacksburg, VA 24061
kafura@cs.vt.edu

Abstract

Symphony is an open and extensible Java-based framework for composition and manipulation of distributed legacy resources. Symphony allows users to compose visually a collection of programs and data by specifying data-flow relationships among them and provides a client/server framework for transparently executing the composed application. Additionally, the framework is web-aware and helps integrate web-based resources with legacy resources. Symphony uses Sun Microsystems' JavaBeans component architecture for providing components that represent legacy resources. These components can be customized and composed in any standard JavaBeans builder tool. Executable components communicate with a server, implemented using Java Remote Method Invocation mechanism, for executing remote legacy applications. Symphony enables extensibility by providing abstract components which can be extended by implementing simple interfaces. Beans implemented from the abstract beans can act as data producers, consumers or filters.

1 Introduction

Symphony is a component-based client/server framework for composing and manipulating distributed legacy resources. The framework consists of two parts: composable client components that represent resources such as data, programs and tools, and the Symphony server which executes remote legacy programs. The client components are implemented as Java beans and the Symphony server is implemented as a Java remote object on which client beans make remote method invocations. Also implemented are utility beans (such as an annotation bean for including comments) and abstract beans who simple interfaces can be extended to add new types of components.

Personalized applications can be built by instantiating, customizing, connecting, and executing the Symphony components using the facilities of any environment that supports the JavaBeans standards. The composing tool or environment provides a means for the user to select from a pallete or tool bar the type of component to be instantiated in the workspace, where it may be further manipulated. Customization refers to the act of specifying the attributes of an instantiated component. The attributes of a Symphony bean describe a specific resource. For example, a Symphony bean that represents a program can be customized to refer to a specific executable file in a specific directory on a specific machine that is runnable using specific authorization information.

The data-flow paradigm was chosen as a way of connecting components and specifying the execution sequence of related programs. This paradigm has been popularized by visualization systems such as AVS [1] and Khoros [8]. A visual program is described as a directed graph, where each node represents an operator or function and each directed arc represents a path over which data flows. In Symphony, such an integrated collection of components is termed a **meta-program**. A meta-program is a set of linked program and data components implemented as a data-flow graph that defines how each program accepts data from a previous computation step and produces data for further processing. Once a meta-program is built, it is possible to ensure its structural integrity and completeness, save it for future reuse, and execute it from the workspace.

Symphony is both platform independent and tool independent. Platform independence comes as a byproduct of using Java which is an architecture neutral programming language. Since JavaBeans is an open, published API and is supported by a large number of Java development tools and Java runtime environments, beans that conform to the API can be composed and manipulated within any such beans container. Thus, Symphony can be used in any tool that conforms to the JavaBeans standard. The current Symphony implementation has been executed on Windows 95, Dec and Sparc systems and tested in the BeanBox reference container developed by Sun, and in the Sieve workspace [6], an experimental, collaborative environment.

Our work on Symphony, while applicable to any set of legacy resources, was motivated by a larger interest at Virginia Tech in problem solving environments (PSEs) for science and engineering applications [2]. A **problem solving environment (PSE)** can be defined as a computer system that provides all computational facilities necessary to solve a target class of problems efficiently. The term problem solving environment has a very broad meaning, possibly including word processing software, which can be viewed as a PSE for formatting documents, as well as a system for assisting engineers solving various types of partial differential equations. Some properties shared by all PSEs are that they allow a user to formulate a problem solution in a language suitable for the target class of problems and to view or assess the correctness of the solution through analysis or visualization tools [5]. Depending on the problem domain, different features are desired in a PSE. Some of these features are:

- Collaboration - Allow multiple users to simultaneously take part in the problem-solving session
- Integration - Hide heterogeneity of individual problem-solving components
- Persistence - Allow saving and reproducing of problem-solving sessions
- Distribution - Handle local as well as remote computational tasks
- Security - Provide user and server security
- Intelligence - Make automatic or semi-automatic selection of solution methods by consulting an associated knowledge base

The field of problem solving environments is a relatively new discipline of computer science and the general understanding of the architecture, technology and methodologies for PSEs is still immature. In fact, no existing PSE or PSE-like system, including our own, has all of the features described above.

Problem solving environments have predominantly focused on science and engineering applications [7, 9]. In this paper too, the term PSE will be interpreted with this application domain in mind. A generally accepted goal for a scientific PSE is that it should ease the burden of advanced scientific computing and should enable more people to solve problems more rapidly without requiring detailed knowledge of the underlying hardware, software, or algorithms, although knowledge about the specific problem domain addressed by the PSE is always required. The need for a PSE increases with the complexity and heterogeneity of the application.

Most existing PSEs are focused on providing problem-solving facilities for narrow application domains, such as solving partial differential equations (PDEs), data visualization, numerical analysis and others [3]. These PSEs are built around software libraries which are either modified or rewritten to adapt to the architecture of the PSE. Although these PSEs function very well in their own domain, they do not attempt to provide a generic framework for solving general-purpose science and engineering problems. In practice, many such problems involve the use of legacy software which is difficult to modify and/or port and may be distributed on geographically distant machines. Existing PSEs provide little support for solving such problems within a generic framework.

The specific shortcomings of the current implementation practice for PSEs for science and engineering, that this research aims to address are as follows:

1. **Inability to easily compose distributed components:** Most PSEs do not allow the integration of programs and data distributed on different machines. Given a set of legacy scientific computing resources developed by a diverse group of people on different platforms (possibly located in different geographical locations) an environment is needed for constructing integrated applications out of these resources. For example, the design of a modern aircraft requires the use of numerous, perhaps tens or hundreds, separate programs; such multidisciplinary design and optimization requires a much higher level of integration than is available in existing PSEs.

   Composition of distributed resources is becoming increasingly important with the growth of the World Wide Web. There are scores of applications on the Web which can be accessed at the click of a button (e.g., Java applets and servlets, CGI applications, and Web wrappers for legacy applications), but there is no single tool which can provide seamless integration of these Web-based applications with other legacy applications. There is also a need for an environment where legacy applications can be provided a graphical user interface for accepting input data and seamlessly integrated with analysis and visualization tools for processing the results.

2. **Lack of support for legacy software:** Most scientific PSEs provide little support for stand-alone legacy software applications. These are applications which are run from the command-line, have limited user interaction, and communicate using specially formatted files.

Support for legacy codes is extremely important because there exists millions of lines of legacy code, most of it difficult to understand and modify, yet very useful.

There seem to be two main reasons for this drawback. First, scientific and engineering PSEs are generally built around software libraries which provide encapsulated problem-solving power for some particular problem-domain. Thus the architecture of a PSE is inextricably linked to the structure of the underlying software library. Second, PSEs are generally built for a particular platform and the PSE software is typically not platform independent. These reasons, in the context of providing support for legacy applications, basically entail modifying the application or porting it to a different platform.

Rewriting legacy code to fit the architecture of the PSE or porting it to the platform supported by the PSE is not a feasible solution because of several problems. First, legacy code is usually difficult to understand and modify and the cost involved in such an attempt could be quite high. Second, the underlying software or hardware facilities assumed by the application may not be available on the particular platform for which the PSE has been developed. Finally, if the performance of the legacy application has been tuned to a particular type of architecture, porting it to a different architecture may take the performance advantages away.

3. **Lack of portability:** Very few existing PSEs are built around a client/server architecture and there is no clean separation of the PSE client interface from the the server-based functionality. Hence, for making the PSE available on another platform, the entire PSE software must be ported, instead of just the client functionality, as for a client/server system.

   A problem solving environment is a complex system by nature and porting the entire PSE software to some other machine or even just installing a copy of the software on another machine may be a tedious task. Consider the example of Parallel ELLPACK (//ELLPACK), which is a problem solving environment for partial differential equations (PDEs) [4]. The //ELLPACK system consists of about one million lines of C, Lisp, and Fortran code. It's easy to see how complex it must be just to install a copy of the PSE on a new machine. If a system like this were to be built around a client/server architecture, only the client functionality would need to be ported to other platforms, the code for which would be only a small percent of the entire PSE software.

   These considerations, in general, limit the availability of the PSE to platforms for which they are developed and many times, to the user being present at the particular machine on which the system is installed. There is need for a PSE architecture that follows the client/server model and where problem specification and analysis of solution can be decoupled from the task of producing a solution.

This research tries to address the above issues, either partially or completely.

2 Overview of Symphony

Symphony is a platform-independent framework for specifying and transparently executing compositions of distributed resources, including legacy resources. The Symphony framework provides an ability to visually compose a collection of distributed program codes, data, and visualization components by specifying data-flow relationships among them. It also enables the user to execute the composed application in a manner that respects the data-flow requirements of individual programs in the composition. Execution transparency in this context means that all system level operations of program execution and of moving data across geographically distributed locations must be largely transparent to the user. Additionally, the Symphony framework is extensible, open and independent of any application domain. It provides an ability to compose Web-accessible resource with legacy resources and allows storing the meta-program in persistent storage so that it can be modified or used later, potentially on a different machine than the one on which it was built. It also provides the necessary system security required for execution of remote applications.

The following is a list of all the beans currently implemented in Symphony and a short description of their purpose, along with an example of how they may be connected together into a simple meta-program (Figure 1):

- **Program Bean:** This bean represents an executable resource on a remote machine or on the client machine. Based on the location, input-output requirements, and the manner in which it can be accessed, the program bean represents two broad categories of programs: HTTP-accessible programs, such as CGI scripts, and regular command-line programs.

- **File Bean:** This bean represents a local or remote data file used as an input to a program or produced as output. A file can be an HTTP accessible file, an anonymous FTP accessible file, or a private user accessible file on any machine connected to the Internet.

- **Socket Bean:** A socket bean encapsulates an input or output stream for communicating through TCP/IP sockets.

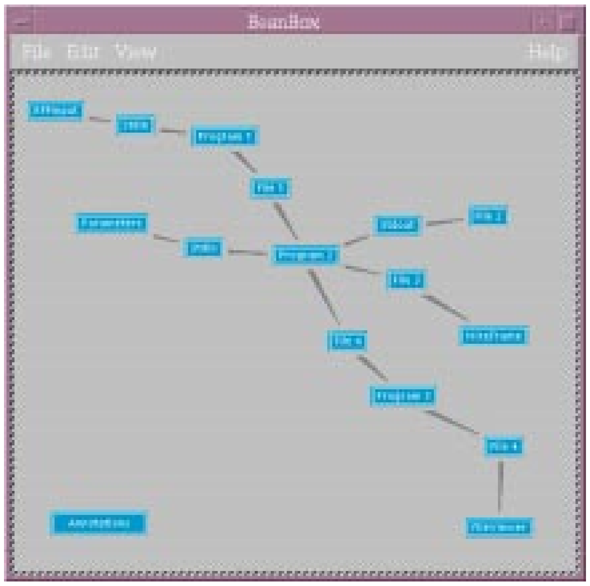


Figure 1. The Composing Environment for Meta-programs

- **Standard Stream Beans:** There are three different beans representing the standard streams of a program: standard input, standard output, and standard error. A standard input bean provides a way of redirecting data into a program's standard input stream. The standard output and error beans provide means of redirecting the respective streams from a program to other beans for processing.

- **Producer Beans:** A producer bean is an abstract bean that can be extended by implementing a simple interface to define new beans types that act as producers of data. One Symphony bean that has been implemented by extending the Producer bean is a Parameters bean. The Parameters bean reads, from a URL, a textual description of the set of parameters expected by a legacy program, and creates a graphical interface to solicit those parameters from the user. The parameters entered by the user are passed onto the next bean in the sequence for further processing during execution.

- **Consumer Beans:** This is an abstract bean which is useful for implementing new beans that act as consumers of data, e.g., visualization beans and viewer beans. Symphony beans that have been implemented using this bean are a FileViewer bean which displays a text file in a window, and a WireFrame bean which reads a stream of specially formatted data and creates a rotatable 3D wireframe graph.

- **Filter Beans:** The filter bean is an abstract bean that allows the user to implement different kinds of beans for filtering the data flowing through the system. The simplest filters can be text filters analogous to the Unix filters. More complex filters include image processing filters and file format converters.

- **Annotations Bean:** This is a utility bean that allows the user to add annotations to the meta-program being constructed. Annotations can be added and viewed at any time.

- **Properties Bean:** This bean represents common properties such as remote host name, user name, password, etc., that are read by all other beans in the environment for customization. This is a utility bean that decreases the amount of work the user has to do for customizing the beans in a meta-program. The user customizes the properties in the properties bean and the values are propagated to all the other beans which have these properties.

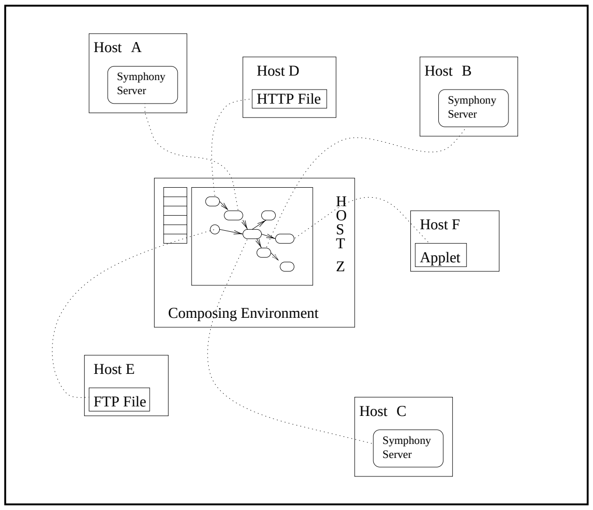


Figure 2. Symphony Architecture

Figure 1 shows how some of the above-mentioned beans can be composed to form a meta-program. Each Program bean can be connected to a set of input and output beans. For example, the RFPInput bean is actually a Producer bean which solicits parameters from the user during execution. Data from this bean is redirected to the standard input stream of Program1 which creates the file represented by the File1 bean when executed. Program2 takes input from the file represented by the File1 bean and also on its standard input. It creates files represented by beans File3 and File4 and the standard output from Program2 is redirected to File2. After File3 is created by the program represented by the Program2 bean it is read by the WireFrame bean which creates a 3D wireframe graph from the file data. The Annotations bean is used to add time-stamped annotations to the meta-program.

The Symphony server is a daemon process running on all host machines which serve programs to remote builder clients. The server is only needed for executing remote programs, not for accessing remote files. It is written in Java for portability. Client program beans communicate with application servers on various hosts by making remote method invocations (RMI) on objects residing in the server. Figure 2 shows the general architecture of the Symphony system. The Symphony meta-program on the client host machine includes three program beans that represent remote programs on hosts A, B, and C. Hence, the Symphony server is needed on these host machines. Another program bean in the meta-program refers to an applet accessible through a URL. The program to be executed on host C needs a file to be present on host E and the programs on host A needs a HTTP-accessible file from host D.

Finally, the name Symphony is representative of the fact that constructing meta-programs that provide access to distributed resources is similar to composing a complex and harmonious musical piece. The user who acts as composer, director, as well as audience, composes the musical score and, hopefully, appreciates the results.

3 Using Symphony

This section describes the procedure to initialize the BeanBox for composing Symphony meta-programs. It also describes important BeanBox operations at a higher level in terms of the steps required for composing a meta-program [11]. Two modifications have been made to the BeanBox provided by Sun purely for aesthetic purposes. First, the original BeanBox displays an empty property sheet when a bean that does not export any properties is selected in the workspace. In the modified BeanBox, the property sheet disappear when such a bean is selected. Second, the original BeanBox does not visually depict connections between beans, but the modified BeanBox does. This later modification is important for Symphony in order to be able to present the data-flow graph of a meta-program more naturally to the user. This, however, does not mean that the modified BeanBox has to be used for composing and executing meta-programs. Any bean builder tool can be used whether or not it provides these visual features.

3.1 Constructing a Meta-Program

There are five major steps to creating a meta-program. Symphony beans first need to be loaded into the BeanBox environment after which a visual representation of each bean appears in the tool box. Second, the required beans must be inserted into the workspace and, third, customized for the resources they represent. Fourth, the beans must be linked according to the desired data-flow patterns, and finally, the meta-program can be verified, executed, or saved for future use. This section also provided details for setting up the execution environment for executing a meta-program. Details about meta-program verification and execution are given in the next sub-section.

3.1.1 Loading Symphony Beans

Symphony beans are packaged in a single Java archive (jar) file. In order to use the beans they need to be loaded in the BeanBox from the jar file, which can be done in two ways. The jar file can be placed in the BeanBox's default jars directory (accessible from the directory in which the BeanBox is installed), in which case the beans are loaded automatically when the BeanBox application is started. Alternatively, the "Load Jar File" menu option of the BeanBox can be used to load the beans. Regardless of how they are loaded, if the load operation is successful, icons for Symphony beans appear in the tool box. Figure 3 shows Symphony beans already loaded into the environment. Notice, for example, that icons for the Program and File beans appear in the ToolBox window.

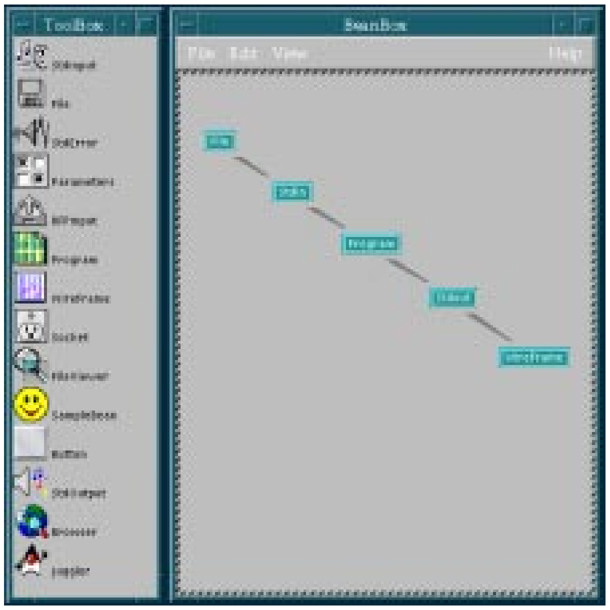
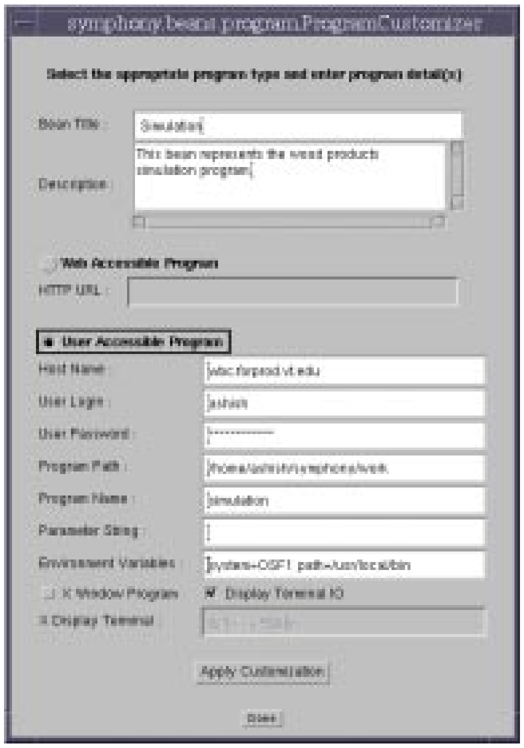


Figure 3. BeanBox Windows

3.1.2 Inserting Beans in the Workspace

A bean is inserted in the BeanBox workspace by clicking on the bean label or icon in the tool box, dragging the mouse pointer to the desired insertion point in the workspace. When a bean is inserted it becomes the currently selected bean in the workspace and a hatched border appears around the been. Any bean can be selected by clicking the left mouse button on the bean or on its perimeter.

3.1.3 Customizing Bean Properties


symphony.beans.program.ProgramCustomizer

Select the appropriate program type and enter program detail(s)

Bean Title :	Simulation
Description :	This bean represents the wood products simulation program
○ Web Accessible Program	
HTTP URL :	
● User Accessible Program	
Host Name :	wbc.forprod.vt.edu
User Login :	ashish
User Password :	************
Program Path :	/home/ashish/symphony/work
Program Name :	simulation
Parameter String :	
Environment Variables :	system=OSF1 path=/usr/local/bin
☐ X Window Program	✓ Display Terminal IO
X Display Terminal	[illegible]

Figure 4. Program Bean Customizer

Every bean publishes certain properties which can be discovered by the BeanBox at run-time and customized by the user. All Symphony beans provide property editing through specialized customizer dialogs. If, for example, a user selects the Program bean shown in Figure 3 and chooses the Edit $\rightarrow$ Customize... menu-item, the BeanBox will display the Program bean customizer shown in Figure 4. Notice that among the properties of the program bean are the program type, host name, login name, password, and other relevant properties for a program.

3.1.4 Linking Beans by Events

Symphony beans communicate through events. A particular bean can generate a variety of events depending on the bean type and purpose. For example, the most important event generated by a button bean is the action event, which occurs when the button is pushed. On the other hand, a bean which has an explicit user interface for interacting with the user may generate events corresponding to mouse and keyboard actions. When a bean is selected in the workspace, the types of events it generates appears as a sub-menu of the Edit $\rightarrow$ Events menu item. Items in this sub-menu represent classes of events such as mouse events, keyboard events, window events, etc. Each item opens another sub-menu which shows the actual events generated, such as the mouse clicked event, mouse dragged event, key pressed event, and so on.

An event generated by a bean can be linked to a method call in another bean, such that the chosen method in the target bean is invoked automatically when the chosen event is generated in the source bean. Each Symphony bean that takes part in the data flow, except for a Consumer bean, exposes only one type of event (connection $\rightarrow$ createConnection) and every bean, except for a Producer bean, exposes only one target method (`eventSend`). In order to connect two Symphony beans, the connection event from a `data source` bean must be connected to the eventSend method of a `data sink` bean. It must be noted that a connection between two Symphony beans is always made in the direction of the desired data flow.

Since the implementation of the source bean knows nothing about the target method in the target bean, the BeanBox generates a standard adapter class for forwarding the event notification from the source bean to the target bean. The BeanBox also takes care of registering the adapter object with the source bean. If the adapter generation and registration are successful, an arrow depicting the connection appears between the source and target beans.

3.1.5 Saving the Workspace

A meta-program may be saved in a persistent state in the form of a Java serialized file. This file can be loaded into the BeanBox at a later time to reproduce the meta-program and modify or execute it.

The BeanBox uses object serialization to save and restore the current contents of the workspace (the beans in the workspace, their state, and connections) [10]. On selecting the File $\rightarrow$ Save menu item in the BeanBox, a file dialog box appears, which can be used to save the current workspace to a named file. In order to retrieve the saved beans, select the File $\rightarrow$ Load menu item and select the required file name in the file dialog that appears. The current contents of the BeanBox workspace will be replaced with the contents of the serialized file. A serialized file is machine and architecture independent and can be transported to any other site and loaded in any bean container, if support for static serialization is provided by the container.

3.1.6 Setting up the Execution Environment

Before a meta-program can be verified or executed, the user needs to ensure the existence of two types of server processes on remote host machines from which the meta-program accesses resources. Remote files are read or written by creating FTP connections to the host machines on which these files reside. To enable this, an FTP daemon process needs to be running on the remote host. This is not necessary for the local host because local files are accessed directly from the file system. Secondly, for executing a remote programs represented by a Program bean, the Symphony server needs to be running on the host machine on which the program resides. The Java class files for executing the Symphony server are included the Symphony_server.jar file that comes with the Symphony distribution.

3.2 Meta-program Operations

Symphony provides three operations that can be performed on a meta-program: verify, execute and stop. These operation are initiated from the menu that pops up by clicking on a Program bean. If there are multiple Program beans in the meta-program, any of them can be used to start an operation. Symphony provides visual feedback to the user during any meta-program operation so that the progress of the operation can be assessed. The following sub-sections describe the mechanism used by Symphony for visual feedback and details about each individual operation.

4 Application Example

This section describes a sample meta-program based on the Radio Frequency Pressing (RFP) simulation developed at the Department of Wood Science and Forest Products at Virginia Tech [2]. This system simulates heat and mass transfer in wood when subject to power input by an alternating electric field. The simulation, implemented in Fortran, takes 64 input parameters as a specially formatted file, of which 11 parameters (such as thickness of wood specimen, strength and frequency of electric field, initial temperature, etc.) must be specified by the user to run the simulation; the remaining parameters have default values which may also be changed. The simulation produces output files containing temperature, pressure and moisture data for the wood specimen with respect to time. This output data can be visualized as a 3-dimensional graph. For example, the temperature inside the wood specimen at any point in time during the simulated experiment can be visualized as a 3D graph of temperature vs. position vs. time. However, the output files produced by the simulation cannot be used directly for creating the required graphs. It must first be converted into a format that the 3D-graph plotting applet can understand. Thus, there are three major steps to running the simulation and visualizing the results:

- Obtain the input parameters from the user and create the input file
- Execute the simulation
- Convert simulation output to the required format and visualize the results

Figure 5 shows the meta-program constructed to execute this simulation. The screen-shot has been annotated by bean types, for reference.

Figure 6 shows a conceptual model of the meta-program configuration. As shown in the figure, the simulation program (denoted by the Program bean labeled “Fortran Simulation” in the meta-program) resides on host machine B. The input file needed by the simulation (File bean labeled “Input Data File” in meta-program) and the produced output files also reside on host B. Although not necessary, for purpose of illustration, the script for converting the input values to a formatted file (Program bean labeled “Script1”) and the script for converting the simulation output data to 3D graph data (Program bean labeled “Script2”) have been placed on different host machines than the simulation program. Both Script1 and Script2 read data from the standard input stream and write output to the standard output stream. Also, for simplicity, the pressure and moisture output from the simulation has been ignored in the meta-program. Only the temperature output is considered. The RFPInput bean shown in the meta-program, created by extending the abstract Producer bean, provides a dialog box for accepting input data values for the simulation. Similarly, the WireFrame bean, created by extending the abstract Consumer bean, accepts 3D-graph data in the specified format and displays a rotatable 3D wireframe graph in a separate window. The simulation program reads its parameters from the standard input stream which are obtained from the user by using the Parameters bean (created from the abstract Producer bean).

The following actions take place when the meta-program is executed from the composing environment, assuming that the execute operation is initiated from the Program bean labeled ’Script1’.

1. The Script1 Program bean asks the RFPInput bean to get ready, which in turn displays a dialog box for accepting the simulation parameters from the user. Figure 7 shows the dialog box. The user modifies the default parameter values and clicks the “Run Simulation” button. The RFPInput bean returns a ready status to the Script1 bean.

2. Script1 contacts the Symphony server on host A and starts execution of the Perl script for converting the

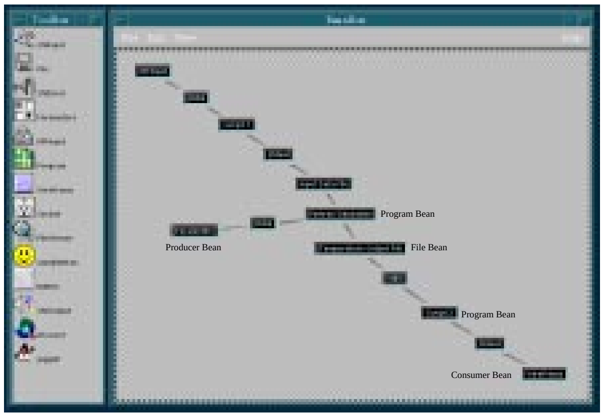


Figure 5. Meta-program for the Wood-based Composites RFP Simulation

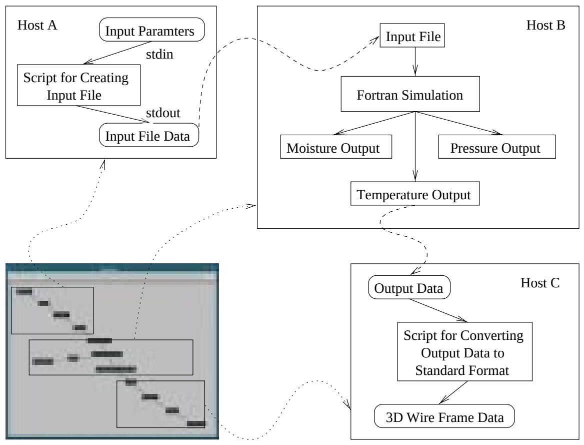


Figure 6. RFP Simulation Meta-Program Configuration

parameters obtained from the RFPInput bean to the required file format. It redirects data from the RFPInput bean to the standard input stream of the remote program, which produces the file data on its standard output. The output data from Script1 is directed to the File bean labeled ”Input Data File”, which results in the creation of a file on host B.

3. The ”Input Data File” bean sends an event to the “Fortran Simulation” Program bean to start execution, which in turn, asks the Parameters bean to get ready for execution. The Parameters bean reads the parameters description file identified by a URL and creates a user interface to accept the input parameters. The user modifies the default parameter values if needed and clicks the “Submit” button. The parameters bean returns a ready status to the Simulation bean.

4. The ”Fortran Simulation” bean contacts the Symphony server on host B to start execution of the Fortran simulation program, and redirects the data obtained from the Parameters bean to the standard input stream of the simulation program. The simulation program writes output to the standard output stream which is displayed in a dialog box that appears when the program starts execution. The remote program creates several output files, one of which is the file defined by the File bean labeled “Temperature Output File”.

5. After the simulation program is done and the output files have been completely created, an event is sent to the Program bean labeled “Script2” to begin execution. The Script2 bean contacts the Symphony server on host C and start execution of the Perl script for converting the temperature output data to 3D wireframe data. It redirects data from the “Temperature Output File” bean to the standard input of the remote script. After starting the execution of the remote script, the Script2 bean informs the WireFrame bean of the availability of data and gives it a handle to the standard output stream from the program.

6. The WireFrame bean reads the data stream obtained from the Script2 bean and displays the 3D WireFrame graph shown in Figure 8. This data transfer is a streaming transfer where the WireFrame bean reads data as it is being produced by the remote script. The graph can be rotated by dragging the mouse pointer in the window, with the left mouse button pressed.

5 Comparing Symphony to Other Approaches

This section provides a comparison of Symphony with other related work. Table 1 depicts the comparison. The first column of the table can be explained as follows. Symphony is a platform-independent system for visual composition of distributed legacy resources. The framework is extensible and allows programmers to easily add new components to the system, and is open because Symphony beans can be used in any standard JavaBeans container. Meta-programs can be composed for any application domain, and can integrate web-based resources with legacy resources. A meta-program can be stored in persistent storage and provides a username/password form of security. The table should not be interpreted to mean that the related systems do not have any other features. It only lists the features provided by Symphony.

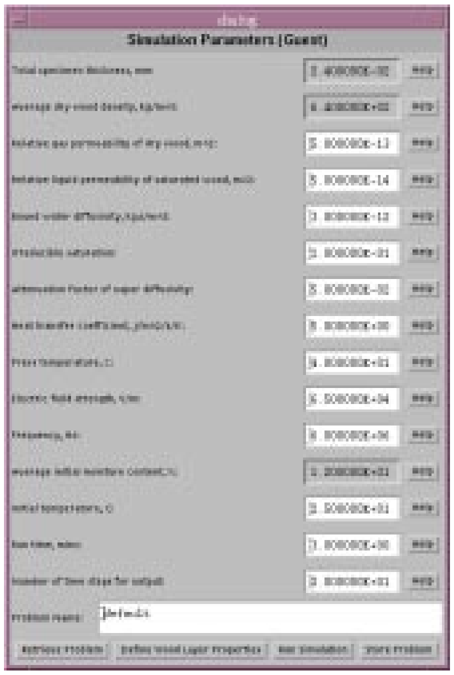


Figure 7. User Interface Created by the RFPInput Bean

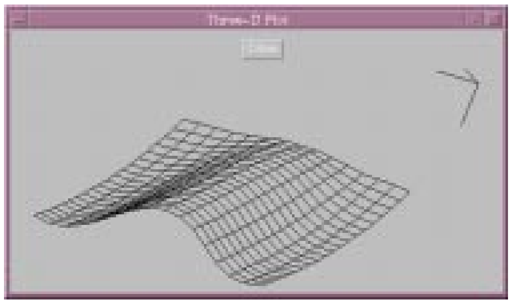


Figure 8. 3D Wire Frame Graph of Simulation Results

6 Conclusions

Symphony attempts to address the problems outlined in the Introduction for scientific and engineering problem-solving tasks that involve the use of distributed legacy resources. It allows users to visually compose legacy programs and data distributed on different machines by specifying the data-flow relationships among them. These programs and data can be used without any modifications. It also allows transparent execution of the composed application in a manner that respects the data-flow requirements of executable components. Execution transparency means that all system-level operations of program execution and of moving data across geographically distributed locations are largely transparent to the user.

Table 1. Comparison of Symphony to Related Work

	Symphony	AVS	Javamatic	Web //ELLPACK	NetSolve	WebFlow	Infospheres
Platform Independent	√		√			√	√
Visual	√	√		√	√	√	
Compositional	√	√			√	√	
Distributed	√	√			√	√	√
Legacy Support	√		√				
Extensible	√	√			√	√	√
Open	√		√			√	√
Generic	√		√				√
Web-aware	√		√	√		√	
Persistent	√	√	√	√	√		√
Secure	√	√	√	√	√	√	√
Graphical	√		√	√		√	

Symphony has been implemented as a set of Java beans which can be customized and composed in a beans builder tool. It provides six core beans for representing legacy resources: Program, File, Socket, Stdin, Stdout and Stderr. These beans can be used to build meta-programs based on the data-flow patterns between executable components. The Program bean communicates with the Symphony server for executing remote applications. The server has been implemented by using the Java Remote Method Invocation (RMI) mechanism.

Other goals for the system were for it to be: extensible, open, generic, web-aware, persistent, secure and graphical.

Symphony enables extensibility by providing abstract beans which can be extended to add new bean types to the environment. Three abstract beans have been provided: the Producer bean, the Consumer bean, and the Filter bean. These beans can be used to implement components that act as data producers, consumers, or filters respectively. The abstract beans do not define the manner in which data is obtained, consumed or transformed, thus providing considerable flexibility to programmers who wish to implement new bean types.

Use of standard JavaBeans mechanisms makes the Symphony framework open in the sense that the set of beans can be used in any standard JavaBeans container. This has enabled the application of Symphony components in Sieve, a JavaBeans-based collaborative workspace, where multiple users can collaborate on composing a Symphony application in real-time.

Although, work on Symphony was initiated from the perspective of science and engineering applications, the system is sufficiently generic to be used for visually composing and executing any set of distributed resources outside of this context.

The Program and File beans allow the user to specify web-accessible resources. Thus, web-based resources can be effectively composed with legacy resources. The composed application can also be saved to persistent storage and reloaded later for user or modification. Symphony also provides basic security in terms of username/password authentication for protected resources.

The Producer abstract bean can be used to implement new bean types that provide graphical interfaces to legacy applications as illustrated by the Parameters bean. Similarly, the Consumer abstract beans can be used to create visualization components for viewing results. This has been illustrated by implementing the FileViewer and WireFrame beans.

7 Limitations and Future Work

This sub-section outlines limitations of the Symphony framework. Some of these limitations stem from the limitations of the current JavaBeans architecture and some from the time constraints faced during the design and development of Symphony.

- **Aggregation:** There is currently no mechanism where a meta-program can be saved as an aggregate bean which can be loaded back into the bean container as an individual bean. An aggregate should have its own representation on the workspace and it should be expandable to see and manipulate the constituent beans of the meta-program. It should be possible to connect an aggregate bean to other beans or other aggregates to form more complex, modularized meta-programs. Multiple levels of aggregation should be supported. An aggregate bean should also have other properties of its own which represent the properties of the meta-program as a whole. More research is needed to determine what properties and operations a meta-program should have as a whole (e.g., how would an aggregate be executed).

- **Control-Flow and Loop Control:** The connection mechanism in Symphony only allows data-flow connections. There is currently, no way to depict a control connection between beans instead of a data connection. It would be useful to be able to specify control connections between program beans even if there is no data dependency between programs, such that when one program finishes, it automatically triggers execution of the next one. Such a capability would be useful in several scenarios such as for sequencing the execution of concurrent programs executed on the same machine to avoid overload, or to force error-prone programs to execute first in order to avoid unnecessary computation if they fail, or cause independent programs to execute concurrently on different machines to speed-up program execution. Also, currently Symphony beans cannot handle loops in the meta-program. Currently, any operation on a meta-program with loops could result in a deadlock or an infinite loop. This can be avoided by introducing a special loop control bean and by giving sequence identifiers to meta-program operations.

- **Data Routing Efficiency:** During meta-program execution, all data is currently routed through the BeanBox for ease of development and testing. In some cases this is not efficient as it involves two or more copy operations to transfer the data from the source to destination instead of a single copy. This can be very inefficient if large amounts of data need to be transferred. One solution to this problem is to decouple the data copying operations from the data control operations, transferring the copy responsibility to a server-side entity. In this case the beans would instruct server-side entities to perform the required data transfer at the right time and provide them with enough information about the source and destination of the data to be routed.

- **Security:** Symphony currently provides a minimal level of security which involves username and password authentication for remote resources. Explicit security features could be included as part of the Symphony execution environment. Currently it is not possible to package a Symphony meta-program as an applet because of the restrictions of the applet security model. If, however, it were possible to package the beans as an applet and sign the applet with a digital signature, the applet could be run from within a browser with enhanced privileges. For some applications, the simplistic username/password security can become cumbersome, especially if more than one person needs to use a meta-program. There must be a mechanism which allows execution of certain programs on a server machine without requiring login authentication. An example of such a mechanism would be to use a dictionary approach where every execution request is authenticated against a program dictionary. Execution is allowed if an only if the request matches an entry in the dictionary. Communication security can be provided by using a secure sockets-based RMI transport instead of the default socket-baesd transport provided by the RMI mechanism.

References

[1] Craig Upson, Thomas Faulhaber, David Kamins, Davin Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, and Andries van Dam. The Application Visualization System: a Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, pages 30–42, July 1989.

[2] Department of Computer Science, Virginia Tech. Research in Problem Solving Environments at Virginia Tech. URL: http://www.cs.vt.edu/ pse/, 1998.

[3] Department of Computer Sciences, Purdue University. Problem Solving Environments. URL: http://www.cs.purdue.edu/research/cse/pses/, 1998.

[4] E. Houstis, J. Rice, S. Weerawarana, A. Catlin, P. Papachiou, K.-W. Wang and M. Gaitatzes. Parallel ELLPACK: A Problem-Solving Environment for PDE Based Applications on Multicomputer Platforms. *ACM Transactions on Mathematical Software*, (To Appear) 1998.

[5] Efstratois Gallopoulos, Elias Houstis and John Rice. Computer as a Thinker/Doer: Problem Solving Environments for Computational Sciences. *IEEE Computational Science and Engineering*, pages 11–23, 1994.

[6] Philip Isenhour. Sieve: A Java-Based Framework for Collaborative Component Composition. Master's thesis, Virginia Tech, Blacksburg, VA, 1998.

[7] John Ambrosiano, Steve Fines and Mladen Vouk. Problem-Solving Environments in the Year 2000 and Beyond, 1995.

[8] Khoral Research, Inc. What is khoros? URL: http://-www.khoral.com/khoros/whatis.html, 1998.

[9] John Rice and Ronald Boisvert. From Scientific Software Libraries to Problem-Solving Environments. *IEEE Computational Science and Engineering*, pages 44–53, 1996.

[10] Sun Microsystems, Inc. Java Object Serialization Specification. URL: http://www.javasoft.com/-products/jdk/1.1/docs/guide/serialization/spec/-serialTOC.doc.html, 1997.

[11] Sun Microsystems, Inc. The JavaBeans (tm) Tutorial. URL: http://www.javasoft.com/beans/docs/Tutorial-Nov97.pdf, 1998.