# An XML Based Component Model for Generating Scientific Applications and Performing Large Scale Simulations in a Meta-computing Environment

Omer F. Rana, Maozhen Li, David W. Walker, and Matthew Shields

Department of Computer Science, Cardiff University, POBox 916, Cardiff CF24 3XF, UK

**Abstract.** The architecture of a component based environment for constructing scientific applications – generally referred to as a Problem Solving Environment (PSE), is described. Each component is a self-contained program, and may be a sequential code developed in C, Fortran or Java, or may contain internal parallelism using MPI or PVM libraries. A user visually constructs an application by combining components from a local or remote repository as a data flow graph. Components are self-documenting, with their interfaces defined in XML, which enables a user to search for components suitable to a particular application, enables a component to be configured when instantiated, enables each component to register with an event listener and facilitates the sharing of components between repositories. The data flow graph is also encoded in XML, and sent to a resource manager for executing the application on a workstation cluster, or a heterogeneous environment made of workstations and high performance parallel machines.

Components in the PSE can also wrap legacy codes. We also describe the architecture and implementation of a molecular dynamics application based on the Lennard-Jones code [18], containing MPI calls, executed on a cluster of workstations, and based on our generic component model. A user can submit simulation data to the application remotely using a Java based user interface. Users need not download any softwares for the simulation and do not need to know the exact implementation.

Keywords: Component-Based Development, Component Interface/Re-use, Metaprogramming Systems, Parallel Computing, Application Generators

K. Czarnecki and U.W. Eisenecker (Eds.): GCSE'99, LNCS 1799, pp. 210–224, 2000.
© Springer-Verlag Berlin Heidelberg 2000

## 1 Introduction

A Problem Solving Environment (PSE) [1] is a complete, integrated computing environment for composing, compiling, and running applications in a specific area. PSEs have been available for several years for certain domains, but most have supported different phases of application development, and cannot be used cooperatively to improve a programmer's productivity, primarily due to the lack of a framework for tool integration and ease-of-use considerations. Extensions to current scientific programs such as Matlab, Maple, and Mathematica are particular pertinent examples of this scenario. Developing extensions to such environments enables the reuse of existing code, but may also severely restrict the ability to integrate routines that are developed in other ways or using other applications. Multi-Matlab [2] is an example of one such extension for parallel computing platforms. Similarly, data flow approaches for constructing scientific applications range from tools such as Khoros, AVS, WebFlow to Iris Explorer.

The modern concept of a PSE for computational science [3] is based on the availability of high performance computing resources, coupled with advances in software tools and infrastructure which makes the creation of such PSEs for computational science a practical goal. PSEs have the potential to greatly improve the productivity of scientists and engineers, particularly with the advent of web-based technologies, such as CORBA and Java, for accessing remote computers and databases. At a 1995 NSF workshop on PSEs [4], the need to develop and evaluate PSE infrastructure and tools was stressed. Subsequently a number of prototype PSEs have been developed. Most focus on linear algebra computations [5] and the solution of partial differential equations, and as yet only a few prototype PSEs have been developed especially for science and engineering applications [7]. However, this is likely to change over the next few years. Tools for building specific types of PSEs include PDELab[11], a system for building PSEs for solving PDEs, and PSEWare[12], a toolkit for building PSEs focused on symbolic computations. More generic infrastructure for building PSEs is also under development, ranging from fairly simple RPC-based tools for controlling remote execution to more ambitious and sophisticated systems such as Legion[13] and Globus[14] for integrating geographically distributed computational and information resources. However, most of these PSEs lack the ability to build up scientific applications by connecting and plugging software components together.

Component-based software engineering (CBSE)[15] is receiving increasing amounts of interest. One goal of CBSE is to reduce development cost and improve software reuse. To apply these technologies to the construction of an effective

framework of PSEs is becoming vital and challenging. Similarly, free computational resources are increasingly becoming available on the Internet. A user who wants to utilise these resources must look for the appropriate library or set of libraries needed for his specific computational problems.

We describe a domain-independent PSE for scientific computations and large-scale simulations. In this environment, a user can visually construct domain-specific applications by plugging together software components written in various languages. We describe the interface of components within such as environment in detail, using an XML-based representation. We believe the universality of this approach will enhance component reuse, and enable sharing between numerical and tool repositories. We have created a molecular dynamics PSE (MDS-PSE) by wrapping a legacy code – a pre-existing MPI-based molecular dynamic simulation program written in C, into Java/CORBA objects. A user can submit input simulation data to the MDS-PSE and then receive the simulation results without downloading the MDS-PSE. The principal advantage is that behind the interface, the client need not know the exact implementation details. Wrapping can be accomplished at multiple levels: around data, individual modules, subsystems, or the entire system. This paper is organised as follows. Section 2 describes the architecture of our PSE, and section 3 describes the component model in detail. Section 4 shows the implementation of the molecular dynamics application, and we conclude the paper in section 5 with extensions and ideas for the future.

## 2 The Software Architecture of the PSE

The architecture of our generative environment consists of a Visual Program Composition Environment (VPCE), the Intelligent Resource Management System (IRMS), and CORBA Wrappers for legacy codes, as illustrated in Figure 1.

### 2.1 VPCE

The VPCE enables a user to develop an application by combining a diverse range of components that may come from different vendors. The components may be sequential codes written in C, Fortran, or Java, or they may be parallel implementations that make use of MPI or PVM libraries. Legacy codes, in Fortran for instance, can be wrapped as CORBA objects. The VPCE is itself a CORBA object, and can communicate with other applications using the ORB interface, as illustrated.

The main component of the VPCE is the Program Composition Tool (PCT), which is a visual tool that enables a user to build and edit applications by

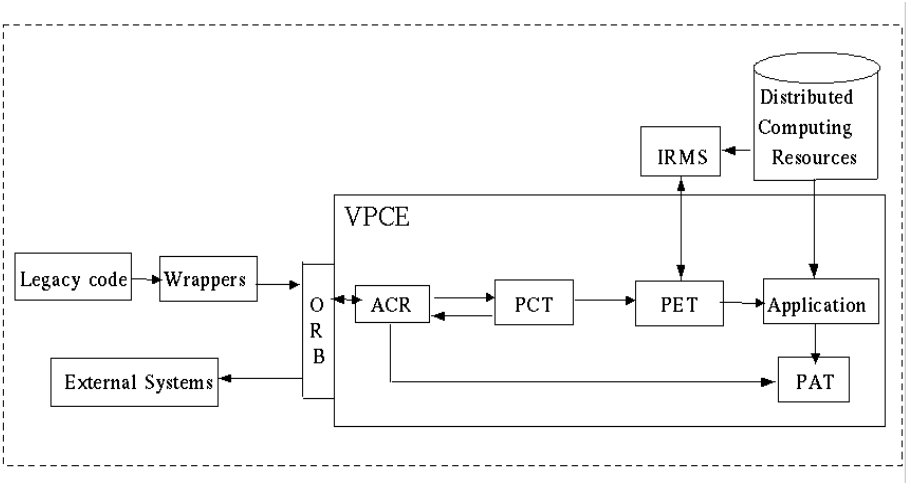



**Fig. 1.** *The CB-PSE Architecture*

plugging together components, by inserting application components into predefined templates, or by replacing application components in existing programs. The PCT allows a user to connect two components only if their interfaces are compatible. The PCT also enables a user to create new application components and place them in the Application Component Repository (ACR). The Program Execution Tool (PET) develops the task graph generated for each application, encodes this into XML (described in section 3), and passes the graph to the IRMS. The VPCE also includes a Program Analysis Tool (PAT) for displaying the hierarchical structure of an application component, describing the purpose and interface of each of its components, as well as of itself. Each component has a well defined interface defined in XML, and associated constraints on its I/O behaviour. When creating a new application component or placing it in the ACR, the user is prompted to optionally supply this information.

### 2.2 Wrapper

The Wrapper enables a program written in another programming language, to be converted into a CORBA object. The services provided by Wrapper are the same as those implemented in the legacy code. Either the complete legacy application may be wrapped as one object, or the application may be divided into smaller objects, provided the structure of the application is known. The granularity of a wrapped application is therefore dependent on how much is known about an application, whether the application can be successfully sub-divided, and

whether constraints on the interfaces of components of a sub-divided application can be derived. The use of hierarchy in the VPCE facilitates the sub-division and the use of wrappers.

### 2.3 IRMS

The IRMS parses every task graph obtained from the PET. Based on the resources available and a performance model of components, it assigns tasks to resources. Through the IRMS, an application built in the VPCE may be scheduled onto single or multiple processor machines. The IRMS provides local scheduling and allocation for tasks obtained from PET, and negotiates with local scheduling systems such as LSF and Codine, where available, to build a possible schedule.

## 3 The Component Model

Each component is either a Java or CORBA object, with its interface specified in XML, according to a standard data model applied to all components within the environment. Components are stored in the Component Repository using this format, and any binary data associated with a component must also be identified by tags. XML tags may be used to automatically derive help on particular components already present in the repository, or may be used to query the availability of particular types of components. User-supplied components must also have their interfaces defined in XML.

The XML-based component model ensures uniformity across components, and helps to abstract component structure and implementation from component interface. Our XML definition enables the division of a component interface into a set of sections, where each section is enclosed within predefined tags. A parser capable of understanding the structure of such a document can identify and match components which meet this interface. The Document Type Definition (DTD) identifying valid tags does not need to be placed with each interface, as it can be obtained from a URL reference placed in the document header, and identified by the `href` tag. The XML definition can be used to perform information integrity (such as the total number of inports and outports), check the suitability of a component for its intended use, the types of platforms that may support the component and internal component structure when available. The tags are divided into: `context` and `header`, `ports`, `execution` specific detail, such as whether the component contains MPI code, a user specified `help` file for the component, a `configuration` file for initialising a component, a `performance model` identifying costs of executing the component for use by the resource manager, and

an `event` handler, which enables registering or recording of particular types of events. A component may also contain specialised constraint tags in addition to the mandatory requirements identified above. Constraints can include security or license constraints, where a component is required to run on a particular machine or cluster. For instance, a data analysis component within the repository may be described as:

```
<?xml version="1.0" href=URL?>

<preface>
  <name alt=DA id=DA01>Data Analyser</name>
  <pse-type>Generic</pse-type>
  <hierarchy id=parent>Tools.Data.Data Analyser</hierarchy>
  <hierarchy id=child></hierarchy>
</preface>

<ports>
 <inportnum>2</inportnum>
 <outportnum>1</outputnum>
  <inportype id=1>float</inportype>
  <inport id=1 type=real>
    <parameter=regression value=NIL/>
  </inport>
  <inport id=2 type=float>
    <parameter=bayesian value=NIL/>
  </inport>
 <outportype> real </outportype>
</ports>

<execution id=software>
 <type>parallel</type>
 <type>MPI</type>
 <type>SPMD</type>
 <type>binary</type>
</execution>


<execution id=platform>
 <type> </type>
</execution>
```

```
<help context=instantiate>
<href name=file:/home/pse/help/data-analyser.txt value=NIL>
</help>
```

The XML-based DTD contains the following types of tags:

- Context and header tags: used to identify a component and the types of PSEs that a component may be usefully employed in. The component name must be unique, with an alternative alphanumeric identifier, however any number of PSEs may be specified. These details are grouped under the preface tag, hence:

  ```
  <!ELEMENT preface    (name pse-type+)>
  <!ELEMENT name       (name-list+)>
  <!ATTLIST name       alt %PCDATA
                       id  %PCDATA>
  <!ELEMENT pse-type   %PCDATA> ...
  ```

  The `hierarchy` tag is used to identify parent and child components, and works in a similar way to the Java package definition. A component can have one parent, and multiple children. In the example, 'DA01' has no children, indicating that it is at the bottom of the hierarchy.
- Ports: used to identify the number of input and output ports, and their types. An input port can accept multiple data types and this can be specified in a number of ways by the user. Input/output to/from a component can also come from/go to other types of sources, such as files or network streams. In this case, the inport and outport ports need to define an `href` tag, rather than a specific data type. We standardise our `href` definition to account for various scenarios where it may be employed, such as:

  ```
  <ports>
  <inport id=1 parameter=regression type=stream value=NIL>
   <parameter=regression value=NIL/>
   <href name=http://www.cs.cf.ac.uk/PSE/ value=test.txt>
  </inport>
  </ports>
  ```

  or when reading data from a file, the `href` tag is changed to:

  ```
  <ports>
  <inport id=1 parameter=regression type=stream value=NIL>
   <parameter=regression value=NIL/>
  ```

```
 <href name=file:/home/pse/test.txt value=NIL>
</inport>
</ports>
```

This gives a user much more flexibility in defining data sources, and using components in a distributed environment. The user may also define more complex input types, such as a `matrix`, `stream` or an `array` in a similar way.

- Execution: a component may have execution specific details associated with it, such as whether it contains MPI code, if it contains internal parallelism etc. If only a binary version of a component is available, then this must be specified by the user also. Such component-specific details may be enclosed in any number of `type` tags. The execution tag is divided into a `software` part and a `platform` part. The former is used to identify the internal properties of the component, while the latter is used to identify a suitable execution platform or a performance model.
- Help: a user can specify an external file containing help on a particular component. The `help` tags contains `context` options which enables the association of a particular file with a particular option, to enable display of a pre-specified help file at particular points in application construction. The contexts are predefined, and all component interfaces must use these. Alternatively, the user may leave the `context` field empty, suggesting that the same file is used every time help is requested on a particular component. If no help file is specified, the XML definition of the component is used to display component properties to a user. Help files can be kept locally, or they may be cross references using a URL. One or more help files may be invoked within a particular `context`, some of which may be local.
- Configuration: similar to the `help` tag, a user can specify a `configuration` tag, which enables a component to load predefined values from a file, from a network address or by using a customiser or wizard program. This enables a component to be pre-configured within a given context, to perform a given action when a component is created or destroyed, for instance. The `configuration` tag is particularly useful when the same component needs to be used in different applications, enabling a user to share parts of a hierarchy, while defining local variations within a given context.
- Performance model: each component has an associated performance model, and this can be specified in a file, using a similar approach to component configuration defined above. A performance model is enclosed in the `performance` tag, and may range from being a numerical cost of running

the component on a given architecture, to being a parameterised model that can account for the range and types of data it deals with to more complex models that are specified analytically.

– Event model: each component supports an event listener. Hence, if a source component can generate an event of type XEvent, than any listener (target) must implement an Xlistener interface. Listeners can either be separate components that perform a well defined action – such as handling exceptions, or can be more general and support methods that are invoked when the given event occurs. We use an `event` tag to bind an event to a method identifier on a particular component.

```
<event target="ComponA" type="ouput" name="overflow" filter="filter">
 <component id=XX> ... </component>
</event>
```

The `target` identifies the component to initiate when an event of a given `type` occurs on component with identity `id`, as defined in the `preface` tag of a component. The `name` tag is used to differentiate different events of the same type, and the `filter` tag is a place-holder for JDK1.2 property change and vetoable property change events support. Also, the filter attribute is used to indicate a specific method in the listener interface using which the event must be received for a particular method to be invoked.

Event handling may either be performed internally within a component, where an event listener needs to be implemented for each component that is placed in the PSE. This is a useful addition to a component model for handling exceptions, and makes each component self-contained. Alternatively, for legacy codes wrapped as components, separate event listeners may be implemented as components, and may be shared between components within the same PSE. Components that contain internal structure, and support hierarchy, must be able to register their events at the highest level in the hierarchy, if separate event listeners are to be implemented. An simple example of an event listener is as follows:

```
<preface>
  <name alt=DA id=DA02>Data Extractor</name>
  <pse-type>Generic</pse-type>
  <hierarchy id=parent>Tools.Data.Data_Extractor</hierarchy>
  <hierarchy id=child></hierarchy>
</preface>

<event type="initialise" name="start" filter="">
```

```
 <script>
   <call-method target="DA01" name="bayesian">
 </script>
</event>
```

The `script` tags are used to specify the method to invoke in another component, when the given event occurs.

– Additional tags not part of the component model may be specified by the user in an

```
<add> ... </add>
```

section towards the end of each section. We will not support variable tags in the first version.

All applications that employ our PSE must adhere to this component model. A user may specify the component model using tags, or may have it encoded using a Component Model editor, which also acts as a wizard and enables customisation. The editor works in a similar manner to an HTML editor, where a user is presented with a menu based choice of available tags, and can either choose one of these predefined tags, or (different from an HTML editor) may define their own. The Component Model in XML forms the interface between the VPCE and other parts of the PSE, and is used to store components in the repository. The XML representation is therefore pervasive throughout the PSE, and links the VPCE to the IRMS. Various representations can be obtained from the XML description in *Scheme*, *Python*, *Perl*, *CORBA-IDL* etc, for connection to other systems that may be attached to the PSE.

The use of tags enables component definitions to be exchanged as web documents, with the structure available at either a single or at particular certified sites. Hence, changes to the DTD can be made without requiring changes to component definitions held by application developers, and will be propagated the next time a user utilises a component interface.

Component interconnectivity is also specified in XML, in the form of a directed graph. Component dependencies are enclosed in `dependency` tags and include constructs such as `parent-of`, `child-of` and `sibling-of`, enabling distant relationships to be constructed from recursive applications of these three basic types. Such dependencies can also be embedded within a JAR file, for instance, where multiple components may be stored in a single file for transfer over a network. Based on OSD [9], 'push'-based applications can automatically trigger the download of particular software components as new versions are developed. Hence, a component within a data flow may be automatically downloaded

and installed, when a new or enhanced version of the component is created. This approach is linked to event handlers, with specific events to identify when a new version of a particular component is available.

The use of the component model also requires that each component has a unique identifier across the PSE workspace, and is registered with a component repository. This is particularly significant when handling events, as event types will need to be based on component identities and their particular position in the data flow. The component models mentioned here have been influenced by IBM's BeanML [8] and Microsoft's OSD [9] XML-based frameworks.

## 4 The MD Application

The code used in the MDS-PSE is a three-dimensionsal molecular dynamics code for simulating a Lennard-Jones fluid. The code has been parallelised, and makes use of the MPI message passing library. The code models short-range atomic interactions by using a link-cell (geometric hashing) algorithm where all particles are hashed into three-dimensional $N_b \times N_b \times N_b$ cells. The cell size is at least the cut-off distance ($r_c$) used in the short-range force evaluation so that each particle interacts only with particles in the same cell or in the neighbouring cells. The symmetry of Newton's Third Law is exploited so that atoms in only 13 (instead of 26) neighbouring cells need to be examined. The code assumes an $N_c \times N_c \times N_c$ Face Centered Cubic (FCC) periodic lattice with a total of $N = 4N_c^3$ atoms. A "shifted-force" [17] Lennard-Jones 6-12 potential ensures that the potential and its first derivative are continuous at the cut-off distance. Particle positions are updated at each time step using a simple Verlet [18] leap-frog scheme. Details of the molecular dynamics algorithms used are described in the book *Computer Simulation of Liquids* [19].

A spatial decomposition [20] is used which distributes the cells in blocks over a three-dimensional mesh of processes, so that each process is responsible for the particles in a rectangular sub-domain. Point-to-point message passing is necessary to perform two tasks in the algorithm. First, particle information in cells lying at the boundaries of a process must be communicated to one or more neighbouring processes. This is necessary because these particles must interact with particles in neighbouring processes. The standard approach of creating "ghost" cells around the boundary of each process is used, and the communication can then be performed in a series of six shift operations (one for each face of the rectangular sub-domain). The second type of point-to-point communication arises when particles migrate from the sub-domain of one process to that of another.

Again this communication can be performed in a series of shift operations. In the message passing code the communication of boundary data and particle migration are combined to reduce the frequency (and hence the overhead) of message-passing.

### 4.1 The Architecture of MDS-PSE

Figure 2 illustrates the architecture of the MDS-PSE.

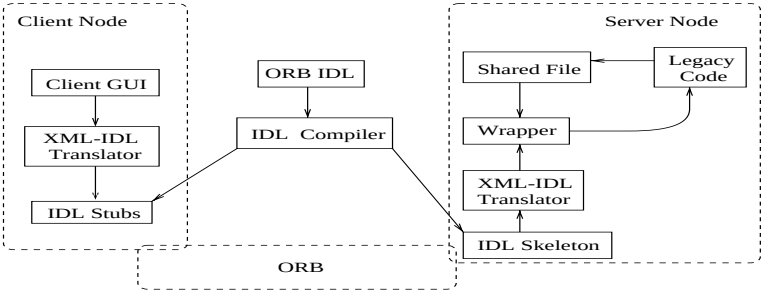
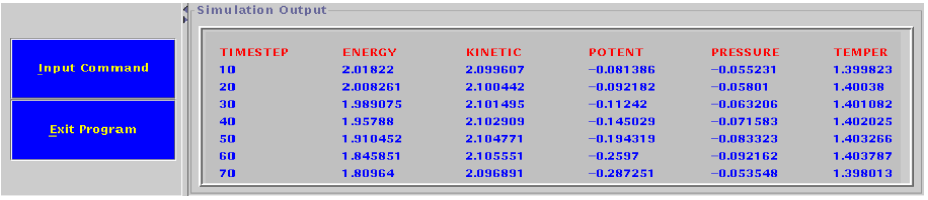



**Fig. 2.** *The MDS-PSE Architecture*

We use an XML to Java IDL translator and a CORBA-compliant ORB shipped with JDK1.2, as the fundamental infrastructure for defining component interfaces in MDS-PSE. The Wrapper packages the legacy code in the server node as a CORBA object. It is responsible for communication with the legacy code through a shared file and provides services to the Client. Since the operations are pre-defined in the MDS-PSE, a Client invokes the Wrapper with the IDL stub on the client side and uses the corresponding IDL skeleton on the server side.

**Client** The Client provides a graphical user interface (GUI) to the user for input, as illustrated in Fig. 3. Using the GUI, a user can input simulation parameters such as the number of processors in each co-ordinate direction. After submitting the input simulation data, the user can view the simulation results, as in Fig. 3, such as the `temperature`, `pressure` and `energy` during the simulation.

**Wrapper** The Wrapper performs two functions. First it initialises the ORB, and then registers itself as `WrapperRef` with the ORB Naming Service, and waits for the Client request to invoke the MDS legacy code. The Wrapper uses the Java




| TIMESTEP | ENERGY | KINETIC | POTENT | PRESSURE | TEMPER |
|---|---|---|---|---|---|
| 10 | 2.01822 | 2.099607 | −0.081386 | −0.055231 | 1.399823 |
| 20 | 2.008261 | 2.100442 | −0.092182 | −0.05801 | 1.40038 |
| 30 | 1.989075 | 2.101495 | −0.11242 | −0.063206 | 1.401082 |
| 40 | 1.95788 | 2.102909 | −0.145029 | −0.071583 | 1.402025 |
| 50 | 1.910452 | 2.104771 | −0.194319 | −0.083323 | 1.403266 |
| 60 | 1.845851 | 2.105551 | −0.2597 | −0.092162 | 1.403787 |
| 70 | 1.80964 | 2.096891 | −0.287251 | −0.053548 | 1.398013 |

**Fig. 3.** *The simulation output in MDS-PSE*

Native Interface (JNI) to communicate with a C function (`NativeC()`) to invoke the MDS-PSE. All Client and Wrapper objects must be registered with the ORB Naming Service to be used in the MDS-PSE.

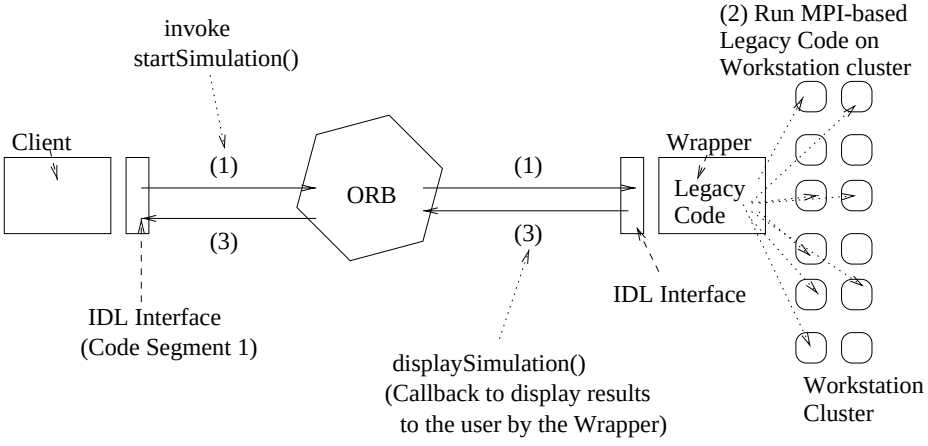



**Fig. 4.** *The invocation sequences of the MDS-PSE*

Figure 4 illustrates the MDS-PSE, and the sequence of steps involved in running an application program. The legacy code is run on multiple machines and has internal parallelism using MPI calls.

We have also divided the MD code into computational sub-tasks, and wrapped these as separate CORBA objects. Each object maintains its local data distribution, and objects need to be run in a pre-defined sequence, with data sharing taking place through intermediate files.

## 5 Conclusion and Future Work

We describe an environment for generating scientific computing applications by combining components. The environment is based on the Java programming language, with components being Java or CORBA objects defined with XML interfaces, and with support for parallelism provided, for example, via MPI libraries. The framework is currently under development as part of a larger multidisciplinary Problem Solving Environment (M-PSE) project.

The current MDS-PSE may be used to explore possible methods for wrapping legacy codes for use in a CORBA environment. The MDS-PSE may be extended to: 1) provide different level wrappers for the legacy code, and divide the MPI-based legacy code into distributed objects, with embedded message passing calls; 2) explore approaches to wrapping different computational resources and providing different wrappers as components for constructing applications. At present we do not handle any errors or exceptions that may occur during this client-server interaction, delegating this task to the underlying ORB. We will be considering specialised error messages in the future.

## References

1. E. Gallopoulos, E. N. Houstis and J. R. Rice, "Computer as Thinker/Doer :Problem-Solving Environments for Computational Science", IEEE Computational Science and Engineering, Vol. 1, No. 2, pp. 11-23, 1994
2. Vijay Menon and Anne E. Trefethen, "MultiMATLAB: Integrating MATLAB with High-Performance Parallel Computing", SuperComputing97, 1997.
3. E. Gallopoulos, E. N. Houstis and J. R. Rice, "Workshop on Problem-Solving Environment:Findings and Recommendations", ACM Computing Surveys, Vol. 27, No. 2, pp 277-279, June 1995
4. J. R. Rice and R. F. Boisvert, "From Scientific Software Libraries to Problem-Solving Environments", IEEE Computational Science and Engineering, Vol. 3, No. 3, pp. 44-53, 1996.
5. H. Casanova and J. J. Dongarra, "NetSolve: A Network-Enabled Server for Solving Computational Science Problems", Int. J. Supercomputing Appl. Vol. 11, No. 3, pp. 212-223, 1997.
6. Alan W. Brown, Kurt C. Wallnau, " The Current State of CBSE", IEEE Software, September 1998.
7. K. M. Decker and B. J. N. Wylie, "Software Tools for Scalable Multilevel Application Engineering", Int. J. Supercomputing Appl. Vol. 11, No. 3, pp. 236-250, 1997.
8. Sanjiva Weerawarana, Joseph Kesselman and Matthew J. Duftler "Bean Markup Language (BeanML)", IBM TJ Watson Research Center, Hawthorne, NY 10532, 1999

9. The Open Software Description Format, See Web page at http://www.w3.org/TR/NOTE-OSD.
10. G. Spezzano, D. Talia, S. Di Gregorio, "A Parallel Cellular Tool for Interactive Modeling and Simulation", IEEE Computational Science and Engineering, Vol. 3, No. 3, pp. 33-43, 1996.
11. S. Weerawarana. Proceedings of the Second Annual Object-Oriented Numberics Conference, Sunriver, Oregon, April 1994.
12. R. Bramley and D. Gannon. See Web page on PSEWare at http://www.extreme.indiana.edu/pseware.
13. A. S. Grimshaw, "Campus-Wide Computing: Early Results Using Legion at the University of Virginia", Int. J. Supercomputing Appl. Vol. 11, No. 2, pp. 129-143, 1997.
14. I. Foster and C. Kesselman, "GLOBUS: A Metacomputing Infrastructure Toolkit", Int. J. Supercomputing Appl. Vol. 11, No. 2, pp. 115-128, 1997.
15. Alan W. Brown, Kurt C. Wallnau, " The Current State of CBSE", IEEE Software Sep. 1998.
16. S. Browne, The Netlib Mathematical Software Repository. D-lib Magazine, Sep. 1995.
17. J. G. Powles, W. A. B. Evans, and N. Quirke, "Non-destructive Molecular Dynamics Simulation of the Chemical Potential of a Fluid", Mol. Phys, Vol. 46, pp. 13476–1370, 1982.
18. L. Verlet, "Computer Experiments on Classical Fluids I. Thermodynamical Properties of Lennard-Jones Molecules", Phys. Rev., Vol. 159, pp. 98–103, 1967.
19. M. P. Allen and D. Tildesley, "Computer Simulation of Liquids", Claredon Press, Oxford, 1987.
20. S. Plimpton", "Fast Parallel Algorithms for Short-Range Molecular Dynamics", J. Comput. Phys., Vol. 117, pp. 1–19, March 1995.