A Java/CORBA based Visual Program Composition Environment for PSEs

Matthew S. Shields, Omer F. Rana, David W. Walker, Maozhen Li,
Department of Computer Science,
Cardiff University, POBox 916,
Cardiff CF24 3XF, UK

David Golby
Department of Mathematical Modelling,
British Aerospace Sowerby Research Center,
PO Box 5, Filton, Bristol, BS34 7QW, UK

Abstract

A Problem Solving Environment (PSE) is a complete, integrated computing environment for composing, compiling and running applications in a specific problem area or domain. Parts of the PSE are domain independent, such as the Visual Programming Composition Environment (VPCE), which may be used for constructing application in a number of different domains, however, other parts are domain specific, such as rules to support particular types of components. A domain independent VPCE is first described, which serves as a user interface for a PSE, and uses Java and CORBA to provide a framework of tools to enable the construction of scientific applications from components.

The VPCE consists of a component repository, from which the user can select off-the-shelf or in-house components, a graphical composition area on which components can be combined, various tools that facilitate the configuration of components, the integration of legacy codes into components and the design and building of new components.

The VPCE produces output using dataflow techniques in the form of a task graph, annotated with a performance model plus constraints for each component, expressed in XML. In addition, the VPCE supports a domain specific expert system based on JESS [9] to guide the user in component selection and to perform integrity checking.

Keywords: Problem Solving Environments, Visual Programming, Intelligent Interfaces, Scientific Computing, Java/CORBA

1 Introduction

A CORBA-based domain-independent problem solving environment for scientific computations and large-scale simulations is described. In this PSE, a user can visually construct domain specific applications by plugging together software components which are independent of location, programming language or platform. A neural network based data analysis application is constructed where a pre-existing MPI-based neural network is used for analysing data, without the user requiring to download any part of the neural network code. The complete application can be wrapped as a component, with the distinct tasks of data retrieval, sampling and storing being performed by other components. Wrapping applications is a method of encapsulation that provides clients with pre-defined interfaces for accessing server applications or components. 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.

Novel aspects of this work include an infrastructure for application development that allows for third party software components, an XML based component model for wrapping legacy codes, a rule base for supporting a user in selecting components, and support for computational steering. Previous work closely related to our efforts is outlined in section 2, and on which some of the techniques used in the PSE are based. An architecture for the PSE is presented in section 3, which illustrates the various sub-systems that constitute the PSE. A standard component model for the PSE is presented in section 3.1, which is defined as a collection of XML tags that all components must use. A prototype of a visual programming tool for building mathematical equations is described in section 3.2 and a neural network application is presented as an illustrative example in section 4.

2 Previous work

A Problem Solving Environment (PSE) is a complete, integrated computing environment for composing, compiling, and running applications in a specific area [10]. PSEs have been available for several years for certain specific domains, but most of these 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 severely restrict the ability to integrate routines that are developed in other ways or using other applications. Multi-Matlab [14] is an example of one such extension for parallel computing platforms.

The modern concept of a PSE for computational science [11] is based on the availability of high performance computing resources, coupled with advances in software tools and infrastructure which make 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 [15], the need to develop and evaluate PSE infrastructure and tools was stressed. Subsequently a number of prototype PSEs have been developed. Many early PSEs focus on linear algebra computations [4] and the solution of partial differential equations, and as yet only a few prototype PSEs have been developed especially for science and engineering applications [6, 16]. However, this is likely to change over the next few years. Tools for building specific types of PSEs have been developed, such as PDELab [18], a system for building PSEs for solving PDEs, and PSEWare [1], a toolkit for building PSEs for 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 Globus [7] and Legion [12] 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)[2] is receiving increasing interest in the software engineering community. The goal of CBSE is to reduce development costs and improve software reuse. The question of how to apply the technologies involved in CBSE to the construction of an effective framework for PSEs is becoming increasingly important.

The Gateway project [8] introduces a similar idea to the system being developed here, in that a component based system implemented using JavaBeans and utilising dataflow techniques to represent the meta-application as a directed graph is used. Unlike our system, which uses XML to define both the interface to all components within the system and the task graph that describes the constructed application, the Gateway system chooses to use the Abstract Task Descriptor (ATD) as its lowest level of granularity of instruction and to build up the instructions that define the application.

The Adaptive Distributed Virtual Computing Environment (ADViCE) project [13] is another system that provides a graphical user interface that enables a user to develop distributed applications and specify the computing and communication requirements of each task within the task graph. Unlike the Gateway system, but similar to our own, the ADViCE system has its own scheduler that allocates tasks to resources at run time.

The Arcade project [5] uses a slightly different approach in that the system has a three tier architecture, with the first tier consisting of a number of Java Applets that are used individually to specify the tasks (either visually or through a scripting language), to specify resource needs, and to provide monitoring and steering. Each of these Applets then interacts with a CORBA interface which in turn interacts with the final execution user modules distributed over a heterogeneous environment.

In the Gateway system, the ATD forms the lowest level instruction, and all components must be defined in terms of ATDs. There is therefore no straight forward way of wrapping legacy codes, or providing provision for existing executables. We provide an XML wrapper, which requires far less effort from a user than developing ATD definitions. However, we share the general design objective of constructing applications by linking sequential and parallel components. The ADViCE system is a visual composition tool above all, and provides little support for legacy applications. ADViCE does enable automatic selection of

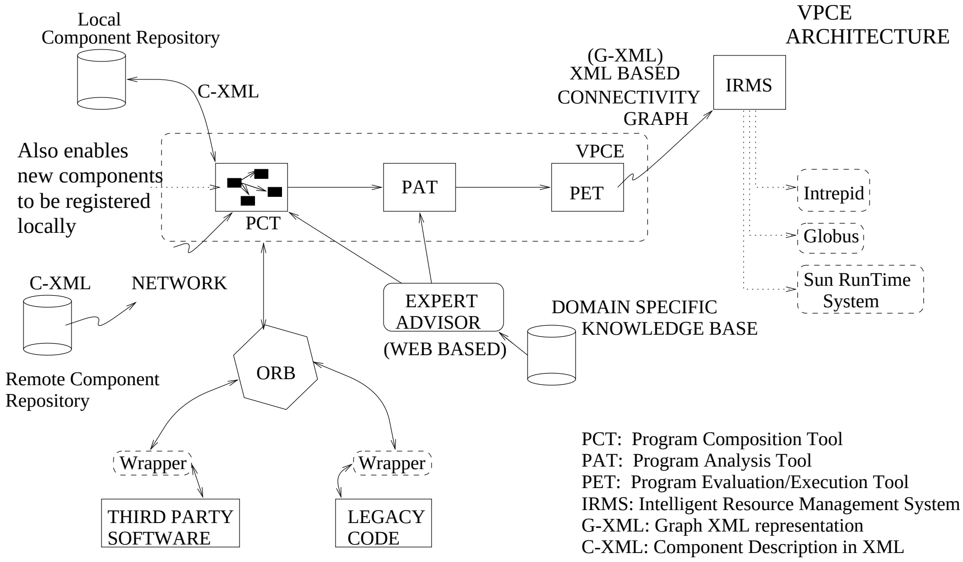

Figure 1: *Architecture of the PSE*

components from a library, based on particular parallel libraries being available to a user (such as PVM, MPI etc), and then sends this to a resource scheduler. The Arcade project requires the construction of specialised Applets, and can be restrictive due to security requirements of the Java sandbox. Our approach is therefore more generic, in that we adopt XML for specifying an interface to a component, and for encoding the connectivity graph. We also provide an interface to a scheduling and resource management system.

3 PSE Software Architecture

The PSE architecture is illustrated in figure 1. Within the PSE, the Visual Program Composition Environment (VPCE) contains a Program Composition Tool (PCT) to enable a user to construct scientific applications by combining components, which connects to local or remote component repositories. If a user is constructing a new application from existing components, then interaction is primarily through the PCT, which provides a graphical interface to the user. If new components are to be added, then the C-XML model must be used, and in this instance the PCT provides an XML parser. Hence, all components have interfaces defined in XML, based on a PSE-wide data model identified as C-XML. Component repositories must be statically connected to the PCT, prior to launching the VPCE. A user can also register new components with the local repositories which contain instances of local or remote components, provided the interface to these compound components is defined in C-XML.

The user can obtain advice about the suitability of a component in a given situation or advice as to the choices of components for a specific task, through the use of the VPCE's Expert Advisor (EA). The EA provides its own user interface, and can be invoked through the PCT or, as a stand-alone application using a Web browser. The EA is a Java based expert system with rules that constrain the types of data a component can handle, the internal data distribution that is supported, licensing requirements and other restrictions. A database of known facts is used to infer possible components that may be suitable within a particular application or library. The EA is also used by the Program Analysis Tool (PAT) to check the validity of the connectivity graph. The connectivity graph linking components together is defined in G-XML, as explained in section 3.1, and is checked for correctness by the PAT.

The checked connectivity graph is then passed to the Program Evaluation/Execution Tool (PET) for the run-time stage of the application. The PET passes the G-XML graph to the Intelligent Resource Management System (IRMS) to determine upon which computational resources to run the application. Based on the resources available and a performance model of components, it assigns different tasks to different computational resources. Using 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 the PET, and negotiates with local scheduling systems such as LSF and Codine, where available, to build a possible schedule. A pre-existing program can be wrapped as a CORBA object, but services offered are maintained as in the original legacy application. 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 applications can be derived. The use of hierarchy in the VPCE facilitates the sub-division and the use of wrappers.

The architecture of the PSE therefore consists of three tiers, where the first tier contains the PCT and the EA, along with component repositories, and associated with this tier is the C-XML model. The second tier consists of a Program Analysis and Execution tool, and legacy applications wrapped as Java or CORBA components. The interfaces to these applications are specified in tier one, using C-XML. The third tier consists of the intelligent resource management system, and interfaces to third party resource management systems, such as Globus. Hence, tiers one and two interact using C-XML, whereas tiers two and three interact using G-XML. The VPCE is thus primarily used to construct applications from software components. The VPCE has the following features:

1. A graphical user interface for the hierarchical construction of components by connecting an outport of one component to the inport of another component.

2. A facility for building new components from scratch in some appropriate

programming language, and wrapping them as Java beans. At present we support components written in C, Fortran and Java.

3. A facility for building inports and outports from a component's input and output interfaces. Although a component's interface cannot be changed, inports (outports) can be constructed out of the data objects comprising the input (output) interface. It should also be possible to replicate and merge channels. Each component has a set of default inports and outports defined by the author of the component.

4. A composition notation (or scripting language) providing control constructs such as loops and conditionals for managing the flow of execution of components. This is not supported within the present implementation, however the user has the capability to specify a pre-defined number of iterations through the code using XML tags, as described in section 3.1.

5. A facility for displaying the hierarchical structure of a component.

6. A facility for viewing documentation on a component giving, for example, its purpose, the algorithm used, the meaning of the input and output arguments, etc. This documentation may be linked to an HTML document, or other sources of information outside of the PSE.

The major sub-systems of the VPCE are as follows:

1. A Component Repository, containing a hierarchical set of folders for storing components that may be used in constructing other higher-level components and/or applications. The component access permissions determine which components a particular user is able to see in the repository. Specific and generic components are indicated by different colours. Template components, which are components which require missing subcomponents to be inserted by the composer, are indicated by a third colour.

2. A composition tool, identified as PCT above, that acts as a canvas, or "scratch pad" where components are joined together by channels connecting inports and outports. The composition tool will allow an outport to be connected to an inport only if they are compatible. The resulting higher-level components and applications may be inserted into the Component Repository, and at this stage access permissions are set, and optional performance model and explanatory information may be associated with the component.

Two categories of users can be identified for the proposed PSE: (1) application users, such as physicists, chemists or biologists who are not interested in creating new components (other than compound components) and therefore do not directly alter the rule base; (2) application developers, mainly computational scientists, who create new components and therefore could potentially require alteration to the rule base.

The VPCE provides two modes of execution, (1) an edit mode that enables components to be assembled together, and (2) an execution mode, where the constructed task graph is sent to the IRMS. In the second of these modes, the user has the option to visualise execution of components, and thereby perform computational steering.

3.1 Component model and XML tags

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. Components have the following properties:

1. The components conform to the Java Beans standard. They may be sequential codes written in Java, Fortran, or C, or they may be parallel codes that make use of message passing libraries such as MPI, or they may exploit array-based parallelism through language extensions such as HPJava [3]. Legacy codes, in Fortran for instance, can be wrapped as Java Beans.

2. The components themselves may be hierarchical (i.e., constructed from other components) and be of arbitrary granularity. Thus, a component may perform a simple task, such as finding the average of a set of input values, or it may be a complete application.

3. Components may be specific implementations that are bound into a higher-level component within the VPCE, or they may be bound in later by the resource locator of the IRMS. This latter case is useful when using commonly available software such as the BLAS. For example, if it was necessary to perform a matrix-matrix multiplication it might be better to allow the PSE to find the BLAS routine to do this on some platform, rather than the program composer making the decision within the VPCE.

4. A component may have individual, group, or world access permissions to specify who may use it.

5. Information is passed from one component to another via uni-directional typed channels. A channel connects an *outport* of one component to an *inport* of another component. A component may have zero or more inports. The set of data objects referenced by the channels connected to a component's inport(s) together define its input interface. Similarly, a component may have zero or more outports, and the set of data objects referenced

by the channels connected to a component's outport(s) together define its output interface.

6. A set of constraints may be associated with each component indicating on what platforms it may be run, and whether it requires generic software such as MPI or the BLAS to be bound in later in order to run.

7. A performance model is optionally associated with each component. This gives the run time of the component as a function of its input and machine, communication, and network parameters.

8. Information on a component's purpose, the algorithms it uses, and other pertinent explanatory data is optionally associated with a component.

The XML-based component model ensures uniformity across components, and helps to abstract component structure and implementation from the 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 checking (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, where 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 permits 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, which requires a component 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 to (output 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. The definition for `href` is standardised 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 the 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 numeric 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.

  The performance model is also used by the IRMS to first derive a schedule and then dispatch tasks to particular machines. The use of a performance model can also facilitate dynamic scheduling, whereby the task dispatcher evaluates workload at each allocation instance to determine the best processor to run a task.

- 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. An `event` tag is used 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 through 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. A 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. Variable tags are not supported 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 as illustrated in figure 1. 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 [17], '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 [19] and Microsoft's OSD [17] XML-based frameworks.

We do not make direct use of a CORBA or JavaBean component model as these are more restrictive for our purpose. Both the CORBA and JavaBean models require the definition of a component interface in terms of its I/O types. We extend our component model to include additional parameters such as constraints on the execution platform and parallel libraries being employed. Such constraints are required by the intelligent resource manager during application scheduling. Our component model is therefore more generic, and the `ports` tags (as defined above) can be translated to CORBA or Java IDL. Although JavaBean component models allow for introspection and reflexion, and enable properties of a component to be modified dynamically, they do not support the extension of pre-defined types easily, or enable these to be enforced across all components. We provide support for specialised requirements of parallel or sequential components, which may also be used to automatically document and configure a component.

3.2 Prototype VPCE

A simple prototype has been built with a specific set of functions to demonstrate some problems and possible solutions in the design and implementation of a VPCE. The prototype is an elementary mathematical equation editor. It has two components in the component repository, a display component and an operator component. Using these components it is possible to build simple arithmetic equations of arbitrary length.

An instance of the display component has one function used to display a single value, the component can be either a source or a destination component within an equation. An instance of the operator component takes two input values, performs one of the four simple arithmetic operations on the inputs and calculates an output value.

The prototype illustrates three initial problems that arise in attempting to provide a dynamic environment. The first is to provide a mechanism by which the environment can discover at design time the properties, methods, inports and outports that a component provides. The second is to provide a mechanism that can be used to dynamically create links between components, and the third to provide dynamic method invocation on particular components within the environment.

The solutions to date, using the Java programming language provide the ability for the system to discover component properties at design time and display them via a simple "Object Inspector". i.e. for a display component to show the value that is set for the component as an editable string, for the operator component to show the two input values as editable strings and the operator as a selection from a "drop-down list". The system can also dynamically create links between two components and use these links to call "set methods" to update the properties of a given component instance.

The Java language and in particular the JavaBean Model provide some useful properties that are used in this prototype for the solutions to the above problems. A JavaBean exposes to the outside world:

- Properties: Internal states that can be set and queried externally by another program.
- Methods: Public methods that can be accessed by another program.
- Events: A bean may generate or receive events. A bean defines an event if it provides methods for adding and removing event listeners from a list of interested objects.

A VPCE must implement the following functions if it is to be able to manipulate JavaBeans:

- The ability to dynamically load an arbitrary class and instantiate an object from that class.
- Use the Java facilities of introspection and reflection to discover a components properties, methods, and events.
- Provide a mechanism for dynamically creating the connection between two components.

The value of an operator can be changed on one of the Operator Bean instances. A "drop down" list on the Object Inspector displays all possible values that can be selected. Figure 2 shows the property value of an Operand Bean being changed. A value is typed into the Edit Box by the user. A partial example of the G-XML description for the task graph described by the VPCE is as follows.

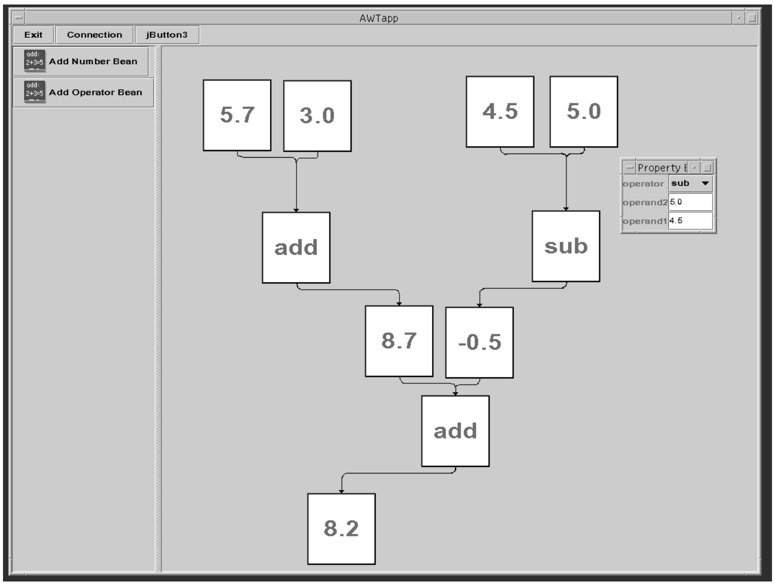

Figure 2: *Changing values on an Operator Bean Instance*

```
<preface>
  <name alt=OD id=OD01>Operand</name>
  <pse-type>Generic</pse-type>
  <hierarchy id=parent></hierarchy>
  <hierarchy id=child>OP01</hierarchy>
</preface>

<preface>
  <name alt=OP id=OP01>Operator</name>
  <pse-type>Generic</pse-type>
  <hierarchy id=parent>OD01</hierarchy>
  <hierarchy id=child>OD02</hierarchy>
  <hierarchy id=child>OD03</hierarchy>
</preface>
```

This brief extract from the XML task graph description, describes the relationships between four components. The first section describes an Operand component that has no parent – it is the root – and a single child. The second section describes an Operator component that has a parent – the component previously defined – and two children which are both Operand components. The following types of user scenarios can be identified that may be successfully attempted with the VPCE:

- Running a legacy application as a wrapped component. The interface to the legacy application is provided in C-XML. The application may be a sequential code, or may contain internal parallelism using MPI or PVM.

- Performing parameter runs on existing or new applications, to study the effect of parameter ranges. The same application code may be executed multiple times by either running the same code in parallel with different parameter values, or where strong dependencies exist within the code, running parts of the simulation in parallel. Partial results may be stored in a temporary file if dependencies are such that simulation needs to be suspended.

- Combine various third party components to generate a new application. The application can itself be stored as a separate component in the Component Repository.

- Searching for suitable components in various repositories maintained on the internet at a remote site, where each component adheres to the C-XML specification.

- Developing a new application using the EXPERT ADVISOR (EA) using JESS, to either select new components, or develop an application on a different platform. In the latter case, the EA is used to analyse the effects of platform constraints on a given application code.

- Visualising results of an application remotely. Enabling and supporting computational steering. In this case, a user may change a particular parameter in the simulation and expect a corresponding change in the visual output generated from the VPCE.

3.3 JESS based querying

The JESS based Expert Advisor (EA) is also provided as a component, with an interface defined in XML, and provides the following functions:

- A standard interface to upload rule files belonging to a particular application domain. The rule base supports a *frame* based representation scheme.

- A general user interface for enabling application users to interact with the rule base for selecting components. Such users may also be interested in why a particular type of component has been selected, and are provided with the inference mechanism used.

- A conflict resolution mechanism to maintain rule base integrity when new rules are added.

- Identification of constraints on data types and execution environment, obtained from the XML description. The conflict resolution mechanism mentioned above must ensure user defined constraints are not violated within a particular domain as new components are added.

- Component categorisation based on application domains or on performance.
- Support for representing uncertainty or imprecise information, especially when a given component may be suitable for more than one domain. In this case, a domain developer must be able to express component membership within single or multiple domains, and addition of new components or changes to a component must ensure that these membership functions are not violated – handled by the conflict resolution mechanism.

A typical session with the EA will involve the user launching JESS via a web browser, or as a stand-alone application, and going through a set of pre-defined questions for identifying component types. Based on selections made by the user, a pre-defined rule file is loaded, being restricted to rules for components within the particular application domain being considered by the user. Session 1 illustrates a user session with the EA Applet for the neural network application described in section 4. A user may add new rules if additional components are identified, or if a compound component is created. The terms used within the rule base correspond to XML tags listed in section 3.1. Although rule files may be dynamically loaded from various sites across the network, the inference engine is run at a single place on the network.

```
There are three approaches to data mining:

      Data Analysis;
      Data Management; and
      Visualisation.

Which of these do you wish to pursue?
Enter choice now (analysis, management or visualisation): analysis

Data Analysis selected ......

There are three categories of data type within the data analysis approach:

      Symbolic (aka Textual);
      Numeric; and
      Combination (with characteristics of both other categories.

Which of these options best describes the elements
that comprise your data set? (symbolic, numeric or combination):
symbolic
```

```
You have selected the numeric data type. More info at URL.

You have to enter the assumptions underpinning your data set

Enter assumption now: restrict number of records to less than 10000

       does your data file contain more 2000 records ? (yes or no):
no
```

Session 1: Expert Advisor User Session

4 A neural network based application

A neural network application has been developed to demonstrate the PSE architecture. Figure 3 illustrates the GUI, where output is plotted using a graph generator.

The neural network in this case also contains its own user interface. A typical session would involve the user identifying the data source, loading the complete or partial data set, running one or more analysis algorithms on the data set, and generating a graphical output from the system. The user is given the option of downloading an entire data set, or to send a filename of a data source to the remote host. Interactions between the GUI and the server implementing the neural network are via Java RMI, for improved performance. Alternatively, we may use the JDK ORB for communication between the GUI and the neural server. When using Java RMI, the GUI and the neural server must be registered individually, whereas when using the ORB, a name service must be supported. We can consider the GUI together with the neural server to be a single compound component in this case, with internal interaction using Java RMI. Data distribution at the remote site is managed by a local run time system, and sent as a parameter to the remote site by the client. The user can change parameters locally, such as the number of iterations or the state of the neural network at any part of the training phase. These parameters are immediately serialised and sent to the remote neural network for analysis.

The client GUI offers a data browser for inspecting the contents of a remote file, and performing simple checks for outliers on data sets. Data integrity is not performed at present, and left to the user or the underlying DBMS to handle. The data source may be a flat file or a structured database. The use of 'View Derivation' components enables a part of a database to be extracted reducing the size of data transfer to the analysis tool. All data does not need to be replicated at the input of each component, as data selection is based on component functionality and component position. Specialised components are used to perform tasks such as data extraction, data retrieval and sampling. Each of these components has interfaces defined in XML and can be connected together into a dataflow graph.

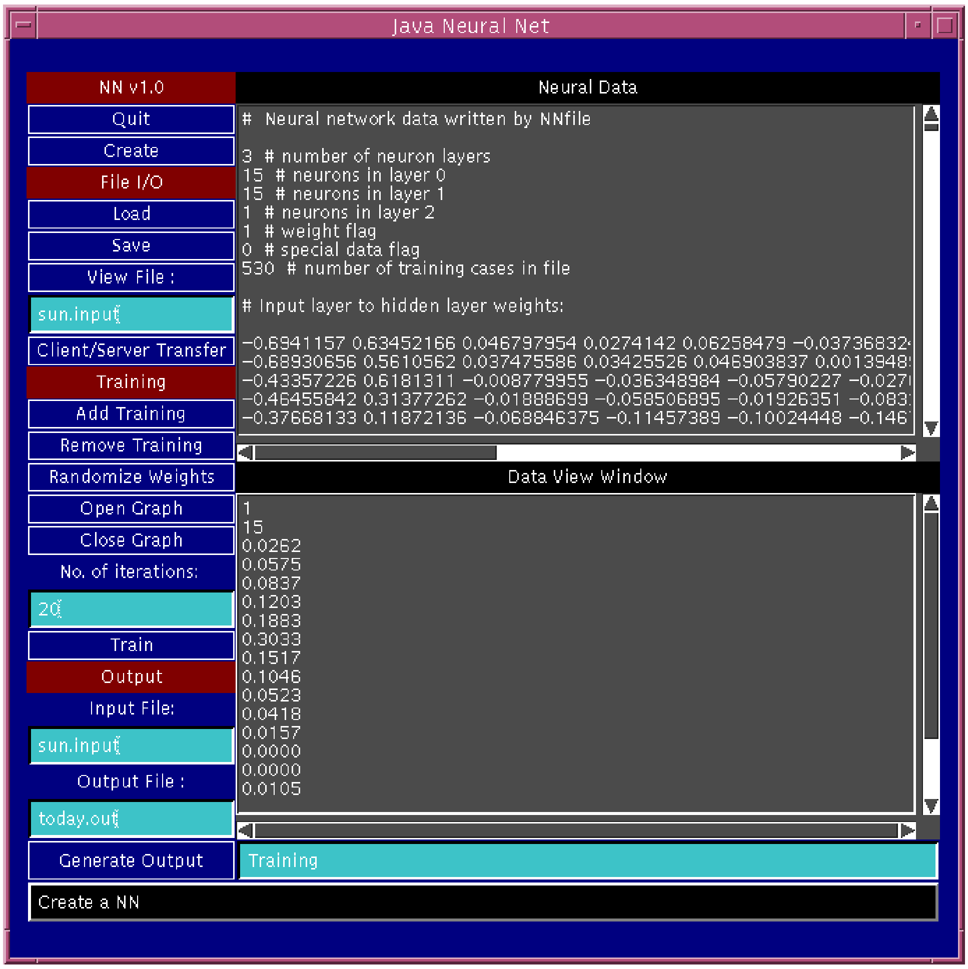

Figure 3: GUI for neural network application

5 Conclusion

Problem Solving Environments based on the emergence of software tools to support heterogeneous meta-computing, are currently an active area of research. Current advances in networking and distributed object technologies provide the infrastructure to make a general purpose distributed PSE a reality.

Our system is similar in some ways to other projects in this area. The majority use the dataflow paradigm together with a graphical component based composition tool with which to build meta-applications. Many are now using third party systems for the underlying communication, security and local resource management and scheduling. For example, many of the systems mentioned in section 2 use Globus [7] for their communications and security infrastructure, and many provide the means to connect to resource managers such as Codine, LSF or the Sun Run-Time system. The difference in our approach is the pervasive use of XML throughout the system, from the definition of component interfaces, through to the self documentation aspect of components, to the task graph output of the VPCE which is passed to the IRMS for scheduling and execution. The benefits of this approach are the ability to enforce a single interface model that all components must adhere to, which enable component properties to be searched based on rules, enable checks on data types and data distributions between connected components, and can facilitate the development of a component catalogue. Defining an interface in XML provides a richer model than CORBA IDL, consequently also supporting integration of components in different programming languages. The dis-advantage is the additional requirement of defining an XML interface, a necessity for all users submitting new components to the Component Repository. Particular user categories for a PSE are identified, and the use of a rule based tool for advising users on the suitability of components within a particular problem domain is suggested.

In summary, this paper provides a synopsis of features necessary for a domain independent PSE. Various sub-systems within the PSE are identified, and prototype versions of some of these components are illustrated.

References

[1] R. Bramley and D. Gannon. PSEWare. See web site at: `http://www.extreme.indiana.edu/pseware`.

[2] Alan W. Brown and Kurt C. Wallnau. The Current State of CBSE. *IEEE Software*, September 1998.

[3] B. Carpenter, G. Fox, D. Leskiw, X. Li, and Y. Wen. Language Bindings for a Data-Parallel Runtime. *NPAC - Syracuse University, Syracuse, New York 13244*, 1997.

[4] H. Casanova and J. J. Dongarra. NetSolve: A Network-Enabled Server for Solving Computational Science Problems. *Int. Journal of Supercomputing Applications*, 11(3), 1997.

[5] Zhikai Chen, Kurt Maly, Piyush Mehrotra, and Mohammad Zubair. Arcade: A Web-Java Based Framework for Distributed Computing. See web site at: http://www.icase.edu:8080/.

[6] K. M. Decker and B. J. N. Wylie. Software Tools for Scalable Multilevel Application Engineering. *IEEE Computational Science and Engineering*, 11(3), 1997.

[7] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Int. Journal of Supercomputing Applications*, 11(2), 1997.

[8] Geoffrey Fox, Tomasz Haupt, Erol Akarsu, Alexey Kalinichenko, Kang-Seok Kim, Praveen Sheethalnath, and Choon-Han Youn. The Gateway System: Uniform Web Based Access to Remote Resources. *Proceedings of JavaGrande Conference*, 1999.

[9] Ernest Friedman-Hill. Jess: The Java Expert System Shell. See web site at: http://herzberg.ca.sandia.gov/jess/, 1999.

[10] E. Gallopoulos, E. N. Houstis, and J. R. Rice. Computer as Thinker/Doer :Problem-Solving Environments for Computational Science. *IEEE Computational Science and Engineering*, 1(2), 1994.

[11] E. Gallopoulos, E. N. Houstis, and J. R. Rice. Workshop on Problem-Solving Environment: Findings and Recommendations. *ACM Computing Surveys*, 27(2), 1994.

[12] A. S. Grimshaw. Campus-Wide Computing: Early Results Using Legion at the University of Virginia. *Int. Journal of Supercomputing Applications*, 11(2), 1997.

[13] Salim Hariri, Haluk Topcuoglu, Wojtek Furmanski, Dongmin Kim, Yoonhee Kim, Ilkyeun Ra, Xue Bing, Bouqing Ye, and Jon Valente. *Problem Solving Environments*, chapter A Problem Solving Environment for Network Computing. IEEE Computer Society, 1998. See web site at: http://www.ece.arizona.edu/~hpdc/projects/ADViCE/papers/bkch.html.

[14] Vijay Menon and Anne E. Trefethen. MultiMATLAB: Integrating MATLAB with High-Performance Parallel Computing. *Proceedings of SuperComputing97*, 1997.

[15] J. R. Rice and R. F. Boisvert. From Scientific Software Libraries to Problem-Solving Environments. *IEEE Computational Science and Engineering*, 3(3), 1996.

[16] G. Spezzano, D. Talia, and S. Di Gregorio. A Parallel Cellular Tool for Interactive Modeling and Simulation. *IEEE Computational Science and Engineering*, 3(3), 1996.

[17] W3C. The Open Software Description Format. See web site at: http://www.w3.org/TR/NOTE-OSD.

[18] S. Weerawarana. PDELab. *Proceedings of the Second Annual Object-Oriented Numerics Conference*, 1994.

[19] Sanjiva Weerawarana, Joseph Kesselman, and Matthew J. Duftler. Bean Markup Language (BeanML), 1999. IBM TJ Watson Research Center, Hawthorne, NY 10532.