# The Software Architecture of a Distributed Problem-Solving Environment

D. W. Walker*
Computer Science and Mathematics Division
Oak Ridge National Laboratory
PO Box 2008, Oak Ridge TN 37831-6367, USA

M. Li, O. F. Rana, M. S. Shields, and Y. Huang
Department of Computer Science
Cardiff University, PO Box 916, Cardiff CF24 3XF, UK

July 11, 2000

**Abstract**

This paper describes the functionality and software architecture of a generic problem-solving environment (PSE) for collaborative computational science and engineering. A PSE is designed to provide transparent access to heterogeneous distributed computing resources, and is intended to enhance research productivity by making it easier to construct, run, and analyze the results of computer simulations. Although implementation details are not discussed in depth, the rôle of software technologies such as CORBA, Java, and XML is outlined. An XML-based component model is presented. The main features of a Visual Component Composition Environment for software development, and an Intelligent Resource Management System for scheduling components, are described. Some prototype implementations of PSE applications are also presented.

## 1 Introduction

A problem-solving environment (PSE) is a complete, integrated computing environment for composing, compiling, and running applications in a specific area. A PSE may also incorporate many features of an expert system and can assist users in formulating problems, running the problem on an appropriate platform, and viewing and analyzing results. In addition, a PSE may have access to virtual libraries, knowledge repositories, sophisticated execution control systems, and visualization environments. The uses of PSEs include modeling and simulation, decision support, design optimization, and industrial process management.

*walker@msr.epm.ornl.gov

The main motivation for developing PSEs is that they provide software tools and expert assistance to the computational scientist in a user-friendly environment, allowing more rapid prototyping of ideas and higher research productivity. By relieving the scientist of the burdens associated with the inessential and often arcane details of specific hardware and software systems, the PSE leaves the scientist free to concentrate on the science.

This paper describes the general functionality and software architecture of a generic distributed PSE. The use of CORBA, Java, XML, and other software technologies is discussed, but this paper's emphasis is on describing the overall functionality of the PSE and on how the different sub-systems of the PSE interact, rather than on implementation details. Implementation details can be found in [29]. Therefore, some issues of importance in the design of PSEs are not fully addressed in this paper, although a brief reference to them is made in section 7.2, with respect to the current PSE infrastructure. These include security and authentication, fault tolerance, debugging, quality control and validation of software components.

The two major parts of the PSE are the Visual Component Composition Environment (VCCE) and the Intelligent Resource Management Sub-system (IRMS). These are described in Sections 2 and 3, respectively. In Section 4 other aspects of PSEs are discussed, such as the role of "intelligence" and mobile agents. Prototype implementations are described in Section 5. Related research into PSEs is presented in Section 6. A summary is given in Section 7.

# 2 The Visual Component Composition Environment

The Visual Component Composition Environment (VCCE) of the PSE is used primarily to construct applications from software components. In this context, an application is merely seen as a high-level component. The VCCE is used to construct components in the form of a dataflow graph. Once a complete application has been constructed, it is passed to the Intelligent Resource Management System (IRMS) to be scheduled on the distributed computing systems available on the network.

## 2.1 Components and Their Properties

In general, a component is a procedural or functional abstraction defined by its input and output interfaces and its semantics. Components have the following properties:

1. Components may be Java Beans or CORBA objects. They may be sequential codes written in Java, Fortran, or C; 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 [9]. Legacy codes, in Fortran for instance, can be wrapped as components.

2. 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 for solving a complex problem.

3. Each component is represented by a well-defined model specified in XML, as described in Section 2.5.

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 is licensed to 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. Ideally, this gives the runtime of the component as a function of its input, machine, communication, and network parameters. If no performance model is available, an upped bound on performance can be approximated based on criteria such as the number of inputs/outputs, the implementation language, the execution platform etc. Subsequent executions of the component can be logged into a database, and used to provide a performance model.

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

There are several ways in which a piece of executable code may become associated with a component. The binding of an executable with a component generally takes place after the VCCE has been used to construct the application. Thus, a component may be available only as an executable specific to a certain type of hardware, and in this case the binding process is trivial. Alternatively, the component may be available as source code, in which case the PSE must arrange for it to be compiled for a target architecture. Finally, if neither the source nor the executable code is available, the Software Information Service is used within the IRMS to locate an implementation of the component. This latter case is useful when using commonly available software, such as the BLAS, which may be available from a compute server. 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 composer of the application making the decision within the VCCE. To be executed, a component must be resolved to a given server implementation. Hence, although a component in the VCCE represents a place holder for executable code, it must be linked to a server which can implement the functionality, and this check is also made by the VCCE prior to invoking the resource management system. This topic is discussed further in Section 3.

There are four common types of component. The main task of *compute components* is to do computational work, such as performing a matrix multiplication or a fast Fourier transform. *Template components* require one or more components to be inserted into them in order to become fully functional. Template components can be used to embody commonly-used algorithmic design patterns. This allows a developer to experiment with different numerical methods or model physics. Thus, in a partial differential equation solver, a template might require a pre-conditioner to be inserted. Template components can also be used to introduce control constructs such as loops and conditionals into a higher-level component. For example, in a simple `for` loop, the initial and final values of the loop control variable, and its increment, must be specified, together with a component that represents the body of the loop. For an `if-then-else` construct, a Boolean expression and a component for each branch of the conditional must be given. A template component can be viewed as a compute component in which one or more of the inputs is a component. *Data management components* are used to read and convert between the different types of data format that are held in files and internet data repositories. The intention of this type of component is to be able to (1) sample data from various archives, especially if there is a large quantity of data to be handled; (2) convert between different data formats;(3) support specialized transports between components if large quantities of data need to be migrated across a network; and (4) undertake data normalization, and perhaps also generate SQL or similar queries to read data from structured databases. The fourth type of common component is the *user interface component*, which allows a user to control an application through a graphical user interface and plays a key role in computational steering.

## 2.2 Features of the VCCE

The central feature of the VCCE is a visual tool known as the Component Composition Tool (CCT) that enables a user to build and edit applications by plugging together components, by inserting components into pre-defined templates, or by replacing components in higher-level hierarchical components. A higher-level component may be constructed from multiple existing components by connecting an outport of one component to the inport of another component. The CCT allows an outport to be connected to an inport only if the two ports are compatible in the number and type of data objects associated with each port. The manipulation of components takes place on a scratch pad (or canvas), with existing components being dragged onto the scratch pad from a web-accessible Component Repository (CR), where they are then linked. Once a new hierarchical component has been built in this way, it may then be added to an appropriate CR for future use and for sharing with other application developers.

The basic functionality of the VCCE is similar to that of other modular visualization environments such as AVS [25], IBM Data Explorer [1], IRIS Explorer [15], and Khoros [39]. A recent article by Wright et al. [38] has reviewed these types of modular visualization environments. The VCCE differs from these environments through its use of an XML-based component model, and an event model that is able to support checkpointing and control constructs such as loops and conditionals. In addition, the VCCE incorporates a number of sophisticated support tools which are described below in this sub-section.

The VCCE includes a tool for building inports and outports from a component's input and output interfaces. Although a component's interfaces cannot be changed, inports and outports can be constructed out of the data objects comprising the input and output interfaces, respectively. Each component has a set of default inports and outports defined by the author of the component. For example, suppose the input to a component consists of two floating point numbers, $a$ and $b$, and an integer, $j$. These could be configured into a single inport $(a, b, j)$, or as two inports $((a, b)$ and $(j)$, or $(a)$ and $(b, j)$, or $(b)$ and $(a, j))$, or as three inports $(a)$, $(b)$, and $(j)$. It is also permitted to have a data object associated with two or more inports. Thus, in the previous example, the inports could be configured as $(a, b)$ and $(b, j)$. In general, this results in a non-deterministic program with a race condition on $b$.

The VCCE also includes a text editor for creating a new component from scratch in some programming language. The Wrapper Tool can then be used to convert the new code into a CORBA-compliant component that can be placed in an CR, if required. The Wrapper Tool can also be used to convert legacy codes into components. A complete legacy application may be wrapped as a single component, or it may be divided and wrapped as a series of smaller components, provided the internal structure of the application is known.

The Annotation Tool allows a developer of a component to annotate it with information that may be of use to other users and parts of the PSE. This information might include an explanation of what the component does and of its input and output interfaces, together with a URL giving the web address of more detailed information. Other annotations include constraints on the component's use and information about its performance.

The Component Analysis Tool (CAT) of the VCCE can be used to display the hierarchical structure of a component. A component can be expanded to show the dataflow graph of the components comprising it. Alternatively, if it is not composed of lower-level components, the source code is displayed. This may be annotated to show the significance of important variables, data structures, and lines of code. A URL may be associated with each component giving the web address of documentation about the component explaining, for example, its purpose, the algorithms used, the meaning of the input and output arguments, etc. This documentation may also include links to sources of information related to the component external of the PSE.

The Expert Assistant (EA) helps users to locate and use components based on a decision tree and/or rule-based approaches. The EA is also responsible for matching user requirements to component types, based on component interfaces specified in XML (as discussed in Section 2.5). Therefore, the EA can be used to search component repositories that are registered for a particular PSE. The EA is a core part of the user interface. It encodes rules that are domain specific, and can help a novice user evaluate components that may be most useful for his or her particular problem. These domain specific rules can either be associated with a component for use within the EA, or they may be developed in-house by an organization using the PSE.

The Input Wizard checks that the input provided to an application does not violate any algorithmic constraints, and correctly represents the system being modelled.

Figure 1 shows the software architecture of the VCCE part of a PSE.

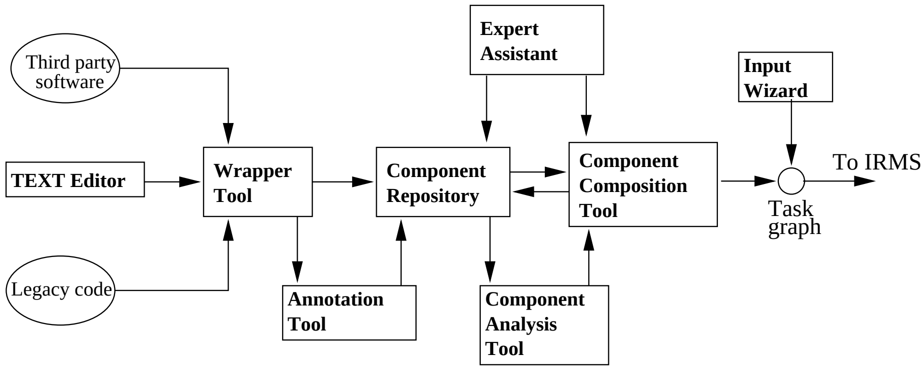


Figure 1: The software architecture of the VCCE

## 2.3 Computational Steering

In a common type of computational steering, a user interacts with a running application by changing the input to one or more components. Typically, the user would assess the current status of some data within an application by viewing it through a visualization interface, and then influence this visual representation by changing some component input value(s). The impact of this change would then be evident in the visualization, and perhaps lead the user to make further changes. An example would be the visualization of a flow, in which the user seeks to achieve some design goal by moving a boundary. Parker et al. [28] have discussed approaches to computational steering in the context of the SCIRun PSE and also review a number of other visual steering systems.

Computational steering can be provided for within the VCCE by inserting a loop that contains a user interface component into the task graph. This component receives data from one or more components, presents the data to the user, and relays any changes the user may make to the appropriate components.

Computational steering is often used in situations in which data are presented to the user from within a loop, for example, a time-stepping loop. Another use occurs when the user wishes to pause the application at some point in order to set a value based on the application results at that point.

These two types of computational steering can be provided for by having each component identified (through the component model described in Section 2.5) as either steerable or non-steerable. Steerable components may, or may not, be initiated upon entry in the paused state. If a steerable component is in the paused state, then input from a user interface component is required to make it continue. The user interface component steering the application can be used to switch a steerable component between the paused and non-paused states.

Figure 2 shows a portion of a task graph that is used for computational steering. The steerable component requires two inputs, $a$ and $b$. Input $a$ may come from either the user interface component doing the steering or from some preceding component in the task graph. A steerable component will always accept input preferentially from the user interface component, if any is available; otherwise, it will accept input from the preceding component. The output $c$ from the steerable component loops back into the user interface component, as well as being passed on to another component for consumption.

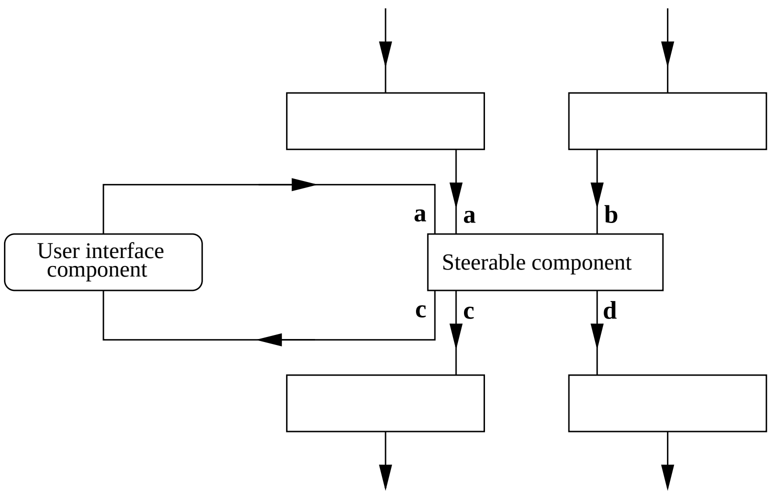


Figure 2: A portion of a task graph used for computational steering.

## 2.4 Collaborative Use of PSEs

The ability to use a PSE collaboratively is important for applications in research and education. The VCCE provides a collaborative software development environment since components created and placed in the Component Repository by one user can be used by other users. Another collaborative aspect of a PSE is that it provides multiple users with consistent representations of data from simulations and/or databases, often through a visualization interface. Within the VCCE, a PSE can be used collaboratively by "cloning" user interface components. This permits each user to view, or interact with, the data through their own clone of the master user interface component. Thus, all user interface components are clonable or non-clonable, and the clones may, or may not, inherit from the master the ability to steer a component in the task graph. The cloning facility is mediated through a web page associated with the master user interface. In a typical collaborative application, the master user would initiate an application in the paused state (see Section 2.3) and then wait for other users to access the web page through which they generate their user interface clones. Users may join the collaborative application at any time that the master user interface is active.

## 2.5 The Component Model

XML (eXtensible Markup Language) is a subset of the document formatting language SGML (Standard General Markup Language), aimed at developing documents for the Web, and devised by the W3C. One objective of XML is to enable stored data intended for human use to also be manipulated by a machine, such as a search engine. XML defines standard tags used to express the structure of a document, in terms of a Document Type Definition (DTD). Hence, a DTD must be defined for every document that uses tags within a particular context, and the validity of a document is confirmed with reference to a particular DTD. Various DTDs have been defined for particular application domains, such as the BioInformatic Sequence Markup Language (BSML), Weather Observation Markup Language (OML), the Extensible Log Format (XLF) for logging information generated by Web servers, amongst others. The approach closest to the work described here is Open Software Description (OSD) from Microsoft and Marimba Technologies for defining component models that can facilitate the automatic updating of components. Using OSD, "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. XML does not define the semantics associated with a given tag, only the positioning of a tag with respect to other tags. However, one must associate and define semantic actions when parsing an XML document that cause particular actions to take place when particular tags are encountered. These actions can range from displaying the content of a document in a particular way, to running programs that are triggered as a result of reaching a particular tag.

Each component in the PSE architecture presented in this paper is represented by a well-defined component model specified in XML. XML is used because of its wide usage in component-based software development and support for it in browsers, such as Internet Explorer and Netscape. All components with interfaces in XML can be automatically catalogued as web pages and can be queried by a number of commercially available search engines. The query mechanism is particularly useful when PSEs share components or when the user wishes to evaluate other components that may be used as alternatives. XML has also become a useful integration mechanism with Java Beans, and numerous commercial tools are readily available to support this integration. These reasons suggest that XML is the most appropriate way of creating component interfaces within a PSE. The use of XML also enables the generation of context-sensitive help and leads to the development of self-cataloguing components.

XML tags are used to define a standard component model within the PSE that will be deployed within all subsystems. Components are stored in the Component Repository using this format, and any binary data associated with a component must also be tagged appropriately. The XML interface to a CORBA object or a Java Bean, for instance, does not make a distinction between an embedded component object or an object that is native to CORBA-IDL or Java-IDL. Therefore, a wrapped component can be a Java Bean, a COM object, C or C++ library calls, wrapped Fortran code called from C, calls to DLLs or shared-objects, and other run-time linking methods. In each case, the XML interface is used to define interface information, for accessing particular attributes and methods, for registering listeners, and for handling exceptions and events. Further, specialized support for expressing constraints on the run-time environment is also provided, along with mechanisms to express hierarchy and directory structures if components are stored within a structured file system.

Our XML based component model contains the following tags:

1. *Context and header tags.* These are 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 components, and works in a similar way to the Java package definition. A component can have one parent and multiple children.

2. *Port tags.* These are used to identify the number of input and output ports and their types. An input port can accept multiple data types and can be specified in a number of ways by the user. The DTD for the `ports` part of the component model can be specified as:

   ```
   <!ELEMENT ports      (inportnum outportnum
                         inporttype+ outporttype+)>
   ```

```
<!ELEMENT inportnum INTEGER>
<!ELEMENT outputnum INTEGER> ...
```

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

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

```
<ports>
<inport id=1 type=stream>
 <parameter value=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 general type:

```
<parameter type=X value=Y>
```

or

```
<parameter source=X target=Y>
```

are applicable within any tag, and are used to specify (name,value) pairs that occur frequently in interface definitions. The user may also define more complex input types, such as a `matrix`, `stream`, or an `array` in a similar way.

3. *Steerability tabs.* A component can be tagged to indicate that it can be steered via the user interface. In order to achieve this, the input/output to/from a component must be obtainable from a user interface. Hence, with a steerable component, the `inports` and `outports` of a component must have a specialized tag to identify this. Hence, for a conventional component without steerability, the ports definition is:

```
<ports>
 <inport id=1 type=stream>
 <parameter type=velocity value=10.2 />
</inport>
```

For a steerable component, the following definition would be used:

```
<ports>
 <steerable>
 <inport id=1 type=stream
 <parameter type=velocity value=10.2 />
</inport>
</steerable>
```

This will automatically produce an additional input port over which interactive inputs can be sent to the component, with the input being of the same type as in the non-steerable version of the component.

4. *Execution tags.* 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 also be specified by the user. 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.

5. *Help tags.* A user can specify an external file containing help for a particular component. The `help` tag contains a `context` option that 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 for 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 can be remote references specified by a URL. One or more help files may be invoked within a particular `context`, some of which may be local.

6. *Configuration tags.* As with 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 using `parameter,value`. This enables a component to be pre-configured within a given context (e.g., to perform a given action when a component is created or destroyed). 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.

7. *Performance model.* Each component has an associated performance model, which 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 parameterized model that can account for the range and types of data it deals with, to more complex models that are specified analytically.

8. *Event model.* Each component supports an event listener. Hence, if a source component can generate an event of type XEvent, then 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="output"
       name="overflow" filter="filter">
 <component id=XX> ... </component>
</event>
```

   The `target` identifies the component to be initiated 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 distinguish 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. This method must be used to receive the event 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 what method to invoke in another component, when the given event occurs.

9. *Additional tags.* These tags are not part of the component model and can be specified by the user in a block at the end of each section by using:

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

Variable tags will not be supported in the first version.

All applications constructed using a PSE based on the architecture discussed in this paper must adhere to this component model. A user may specify the component model using tags or have it encoded using a Component Model editor. The editor works in a similar manner to an HTML editor, where a user is presented with a menu based choice of available tags. Alternatively, a user of the component editor may define their own tags. Tags do not appear in the editor and are automatically inserted into the description once the user has entered the required text and wants to generate a component interface.

The Component Model in XML forms the interface between the VCCE and other parts of the PSE, as illustrated in Fig. 3. Therefore, the XML representation is pervasive throughout the PSE and links the VCCE to the Intelligent Resource Management System (IRMS) described in Section 3. Various representations can be obtained from the XML description in Scheme, Python, Perl, CORBA-IDL, or others, for connection to other systems that may be attached to the PSE. Similarly, XML is used to store components in the repository, and various query approaches may be employed within the VCCE to obtain components with given characteristics or components at a particular hierarchy.

The use of tags enables component definitions to be exchanged as web documents, with the structure available at either a single site 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 component interface is used.

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

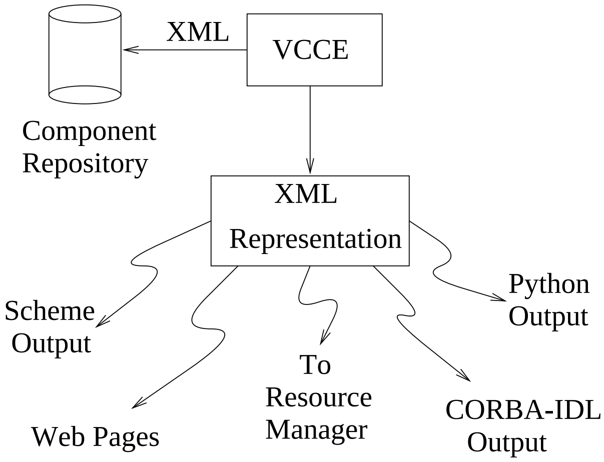


Figure 3: The XML-based component model.

instance, where multiple components may be stored in a single file for transfer over a network. 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 [36] and Microsoft's OSD [27] XML based frameworks.

# 3 The Intelligent Resource Management System

Once a complete application has been specified using the VCCE the resulting task graph, annotated with information about the components' performance and constraints on execution, is passed to the Intelligent Resource Management System (IRMS) to be scheduled for execution. The structure of the IRMS is shown in Fig. 4 and will now be described.

## 3.1 The Scheduler

The Scheduler is the most important part of the IRMS. It must analyze the task graph to determine an appropriate level of granularity, to resolve components (i.e., create or locate an executable corresponding to each component), and to associate each component with a hardware resource or with a local resource manager. After this, the task graph is said to be executable, and it is then passed to the Execution Controller.

The Scheduler must devise a schedule according to certain goals embodied in a scheduling policy and in accordance with a set of constraints. It makes use of

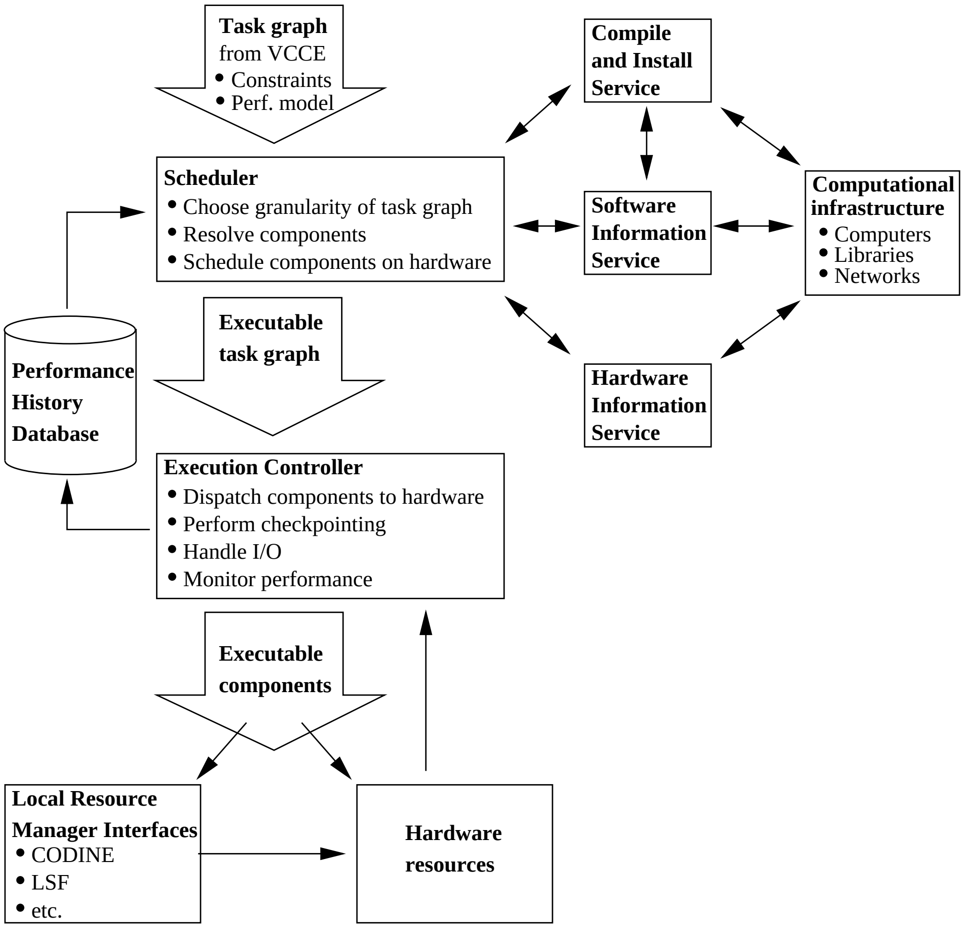


Figure 4: The IRMS architecture.

a number of information resources to devise an executable schedule. Scheduling involves the coupled activities of determining an appropriate level of granularity for scheduling the task graph obtained from the VCCE and deciding on which resources to run the components of the task graph.

Currently, a fairly simple Scheduler is envisioned that takes a top down approach, starting at the highest level of granularity. The task graph received from the VCCE is initially regarded as having a single node that represents the whole application. This node is then expanded (i.e., replaced by its sub-components), and an improved schedule is sought. If no such schedule can be found then the initial task graph and its associated schedule is accepted, and output to the Execution Controller. Otherwise, we expand again and repeat until we either generate a task graph that is worse then the previous one, or until we cannot expand any more. The metric used to decide if one task graph is better than its predecessor is important, and different metrics correspond to different scheduling policies. For example, the metric that always views TG(i) as better than TG(i+1) (provided the former obeys all the constraints) corresponds to a policy that attempts to always schedule at the highest granularity possible, regardless of cost or performance. The metric that regards TG(i) as better than TG(i+1) if it has a smaller expected time to completion will result in a schedule that attempts to minimize execution time. The expected time to completion is derived from the performance model associated with a component.

In the simple scheme described above, "expansion" means expanding each node of the current task graph, but they could be expanded one at a time (starting with the most resource hungry) to see which is best. The scheduling scheme is represented in Fig. 5.

### 3.2 Constraints

A component is said to be *compatible* with a piece of hardware if all its sub-components can be executed on that hardware. Clearly, the Scheduler must always expand components that are not compatible.

The Scheduler can run a component on a particular piece of hardware only if it can locate or create an executable for that component/hardware combination. If no executable is explicitly associated with a component, and if the location of the source code is known, then the Scheduler will use the Compile and Install Service to create an executable for the hardware platform of interest.

Resources are owned and operated by different individuals or organizations and exist in different administrative domains. Therefore, resource usage may have different security restrictions or be subject to particular policy constraints, restricting the scheduling of computational tasks to these devices (or groups of devices). Licensing constraints are a pertinent example of these restrictions, whereby mathematical or scientific software is restricted to a single machine or particular groups of machines.

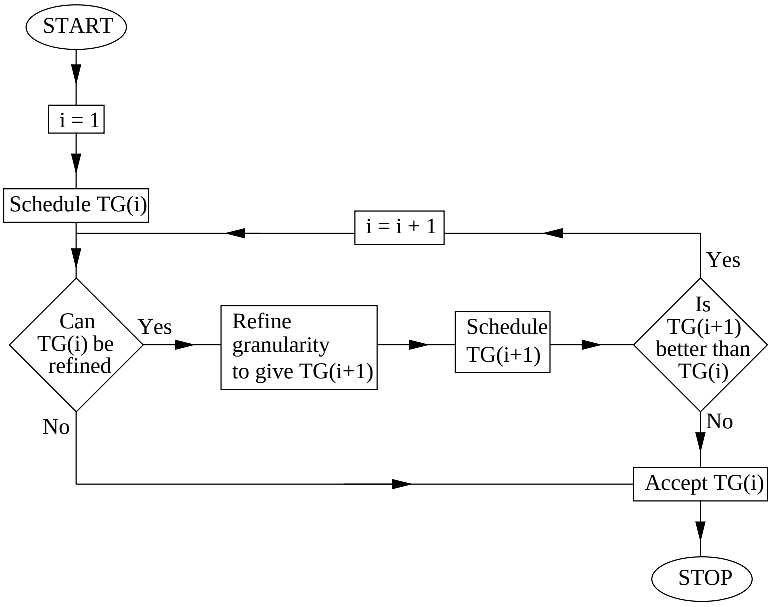


Figure 5: Scheduling a task graph. The input task graph TG(1) is the complete application received from the VCCE.

### 3.3 Information Resources

If neither an executable nor the source code is available for a component, then the Software Information Service is used to attempt to find a suitable executable for the component.

The Hardware Information Service (HIS) provides the Scheduler with information about the hardware resources available and the network connecting them. At any given time, the HIS attempts to maintain an accurate picture of the capabilities, load, and accessibility of the resources within its local domain. Required information can include configuration details about resources such as CPU speed, disk and memory space, number of nodes in a parallel computer, or the number and type of network interfaces available; instantaneous performance information, such as point-to-point network latency, available network bandwidth, and CPU load; and application-specific information, such as memory requirements.

Performance information about components from previous runs is stored in the Performance History Database (PHD). This information can then be used by the Scheduler in conjunction with, or in place of, the component performance models to make scheduling decisions. Once a sufficiently large PHD has been built up, intelligent methods based on genetic and neural network algorithms (for example) can be used to estimate performance. This is an area for further research, and will be investigated in more detail in the future.

### 3.4 The Execution Controller

Components can be run on hardware under the direct control of the Execution Controller, or they may be passed to local resource managers that will handle their execution. The simplest case is where the entire application is handled by a single local resource manager, such as CODINE [11] or GRAM [18]. Coordinating applications whose sub-components are handled by multiple local resource managers, and also directly by the Execution Controller, may present significant difficulties.

## 4 Other Aspects of PSEs

### 4.1 Electronic Notebooks and Living Documents

Electronic notebooks are useful add-ons to PSEs. They provide a mechanism whereby the computational experiments conducted using a PSE can be recorded and annotated, thereby constituting a record of what has been done. Electronic notebooks are intended to be the digital equivalent of the lab books kept by experimental scientists. An advantage of an electronic notebook is that it allows experiments to be replayed, and serves as a safeguard against error and fraud. An electronic notebook can also be used to demonstrate adherence to principles of best practice.

A living document is the digital equivalent of a paper, thesis, or book, whereby the computational experiments conducted in a piece of research can be re-run, or run again with different input parameters. For example, suppose a figure in a living document was produced using a certain set of parameters. A reader of the document might want to find out what the figure would look like for different input parameters. Living documents represent the next phase in the electronic publication of research results, and it is expected that each PSE will have a set of living documents associated with it whose computational results were produced using the PSE.

Clearly, the concepts of a electronic notebook and a living document are quite similar, and the two could be merged, though it may be better to keep them distinct as they correspond to different phases of the scientific process.

### 4.2 Intelligence in PSEs

There are several ways in which a PSE can take advantage of "intelligence." The first is in the location and use of the components needed to solve a particular problem. This can range from a simple search interface to expert assistance in the form of a decision tree or rule-based system that guides the user in the selection of components. A second example of an intelligent method is the use of an *input wizard* that checks the consistency of the input data to an application, and looks for values out of expected acceptable ranges. This can be of particular importance when the input data is large and has complex interrelations. The third way in which intelligence can be used is in scheduling by the Intelligent Resource Management System (see Section 3). A database of previous performance history is built up to store the performance of components for different hardware platforms and inputs. This may then be used to help schedule components using genetic algorithms and neural networks.

### 4.3 Generic and Specific Aspects of PSEs

A clear distinction can be made between those parts of a PSE that are generic, and hence can be used in constructing PSEs in other application domains, and those that are specific to just one application domain. The following parts of a PSE are specific:

- The components in the Component Repository (mostly specific, although some may have broad applicability).
- Expert assistance (specific to an application domain).
- Input wizards.
- The performance history database.

Apart from these application-specific aspects of a PSE, most of the rest of the PSE infrastructure is generic and can be used across multiple application domains.

### 4.4 The Role of Mobile Agents in PSEs

Mobile agents can be used in PSEs in the areas of resource discovery, resource monitoring, and software propagation. When applied to resource discovery, they can be used to support the Software and Hardware Information Services to maintain an up-to-date picture of the resources available to the PSE and to seek out new resources. Thus, for example, mobile agents can be dispatched by the Hardware Information Service (HIS) to find computational resources whose access and usage policies permit them to be used by a PSE. Similarly, mobile agents dispatched by the Software Information Service can seek out generic components, such as the BLAS, and knowledge repositories on remote machines so they can be used subsequently in applications. Resource discovery using mobile agents can be performed continuously, and proceeds independently of the users of the PSE.

The computing and network resources used by a PSE must be monitored so that the IRMS can efficiently schedule applications. This could be done statically by having each resource send performance information (such as machine load or the bandwidth of a network link) to the HIS periodically or upon demand. Alternatively, this information could be gathered by mobile agents, thereby avoiding the need to keep a static agent or daemon continuously running on each remote resource.

The Compilation and Installation Service can use mobile agents to propagate software. If the source code is available for a component in the CR, then an agent can take this and attempt to compile and install the component on the machines known to the PSE through the HIS. The resulting executable can then be used in applications built within the VCCE. Again, this activity can take place continuously, independent of the users of the PSE.

### 4.5 Wrapping Legacy Codes as CORBA Objects

Legacy codes are existing codes that often possess the following features: 1) they are domain-specific; 2) they are not reusable; 3) they are still useful; and 4) they are large, complex monoliths. Wrapping legacy codes as CORBA objects is a method of encapsulation that provides clients with well-known interfaces for accessing these objects. The principal advantage is that behind the interface, the client need not know the exact implementation. Wrapping can be accomplished at multiple levels: around data, individual modules, subsystems, or the entire system. After being wrapped as CORBA objects, these legacy codes can be reused as components in a heterogeneous distributed computing environment.

## 5 Prototype Implementations

### 5.1 A Prototype Expert Assistant

A prototype Expert Assistant (EA) has been implemented using the Java Expert System Shell (JESS) which gives the ability to "plug-in" domain specific rules into the EA. To enable such a system to work, the prototype EA provides support for rule consistency checking (i.e., what happens if a new rule is added and how does it interfere with rules which already exist) and rule upgrade over a period of time (i.e., if rules are no longer required, and do not get recalled during PSE use, these rules can automatically "expire", so that the size of the rule base is manageable).

### 5.2 A Simple Mathematical Equation Builder

A small prototype with a specific set of functions has been built to demonstrate some of problems and possible solutions in the design and implementation of a VCCE. The prototype has just 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 just one function (i.e., to display a value) and 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 in which the scientist can work. The first is to provide a mechanism by which the environment can discover the properties, methods, inports and outports that a component provides at design time. The second is to provide a mechanism that can be used to dynamically create links between components. The third is 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 a component's properties at design time and display them via a simple "Object Inspector." Thus, for a display component the value that is set for the component is shown as an editable string, and for an operator component the two input values are shown 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.

## 5.3 The Implementation of MDS-PSE

A legacy molecular dynamics simulation, written in C, has been wrapped as a distributed application using CORBA. The simulation code is parallel and uses MPI for interprocess communication. The wrapped verison of the code has been used as the basis for a prototype PSE for molecular dynamics simulations (MDS-PSE). When a user submits input data using MDS-PSE simulation results are returned to them. The user does not need to download the MDS-PSE application, and does not need to know where the simulation application runs or what programming language is used in the legacy code.

### 5.3.1 The MDS-PSE Architecture

Figure 6 illustrates the architecture of the MDS-PSE. Java IDL is used as the fundamental infrastructure for defining component interfaces in MDS-PSE. Java IDL is a CORBA compliant IDL shipped with JDK1.2. The Wrapper converts the legacy code in the server node to 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 performed in MDS-PSE are known at compile time, the client invokes the Wrapper through an IDL stub on the client side and an IDL skeleton on the server side. Using CORBA, the client can submit input simulation data and wait for simulation results without knowing the location of the wrapper and the exact implementation of the simulation.

### 5.3.2 The Implementation of MDS-PSE

The operations provided by the client and the wrapper in MDS-PSE are defined by the IDL definitions in Code Segment 1. The IDL interface hides the Client from the implementation of the wrapper and the programming language used to implement the application.

In Code Segment 1, the service provided by the Wrapper is `startSimulation()`, which has two input parameters. One parameter is the reference to the Client object, which will be used to invoke the Client in order to display simulation results to the user. The other parameter is the input simulation data received

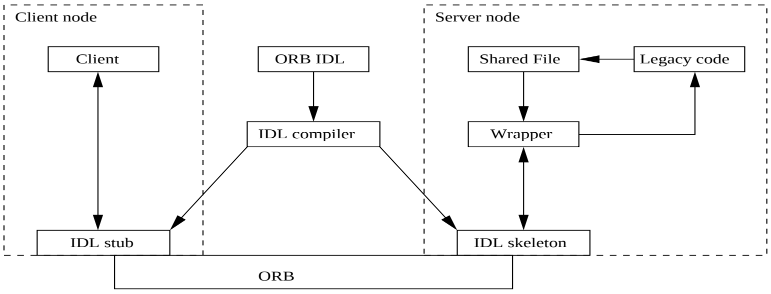


Figure 6: The MDS-PSE architecture.

from the user. The service provided by the Client is `displaySimulation()` which has six input parameters. The operation is used by the Client to display simulation results. It is the IDL definition that hides the exact implementation of the simulation from the user.

```
module Simulation
{
     interface Client{
           void displaySimulation(in unsigned long a,
                                  in float f1,in float f2,
                                  in float f3,in float f4,in float f5);
     };
     interface Wrapper{
           void startSimulation(in Client obj, in string SimulationParameters);
     };
};
```

**Code Segment 1. Operations defined through ORB IDL in MDS-PSE.**

- **Client**

The client provides a simple graphical user interface (GUI) to the user for input. Using the GUI, a user can input simulation parameters such as the number of processors to be used in each co-ordinate direction. After submitting simulation data, the user can see the simulation results such as the temperature, pressure, and energy associated with the current configuration of particles.

After the user submits input simulation data to the MDS-PSE, the Client initializes the Java IDL and obtains the reference `WrapperRef` to the Wrapper through the Java IDL Naming Service and requests a service provided by the Wrapper (`WrapperRef.startSimulation()`), as shown in Code Segment 2.

```
class Client{
     ...
     ORB orb = ORB.init(args, null);
     org.omg.CORBA.Object objRef =orb.resolve_initial_references("NameService");
     NamingContext ncRef = NamingContextHelper.narrow(objRef);
     NameComponent nc = new NameComponent("Wrapper", "");
     NameComponent path[] = {nc};
     Wrapper WrapperRef =WrapperHelper.narrow(ncRef.resolve(path));
```

```
    WrapperRef.startSimulation(ClientRef, simulationParameters);
    ...
}
```

**Code Segment 2. The Client code used to request a service provided by the Wrapper object.**

There are many steps needed to output the simulation results. Each time data are output on the server side, the Wrapper calls back to the Client and invokes the Client to display the simulation results to the user. The Client connects the Client object implementation reference (`ClientRef`) into the ORB and waits for requests from the Wrapper. In such a situation, the Client becomes an object and provides services (`displaySimulation()`) to the Wrapper, as shown in Code Segment 3.

```
class Client{
    ...
    ClientImplementation ClientRef = new ClientImplementation();
    orb.connect(ClientRef);
    ...
    void displaySimulation(unsigned long a,
                           float f1, float f2,
                           float f3, float f4, float f5)
    {
      ...
    }
}
```

**Code Segment 3. The Client code used to provide a service to the Wrapper object.**

- **Wrapper**

The Wrapper performs two sets of functions. First, it initializes the ORB, identifies itself in the ORB(`Wrapper`) through the Java IDL naming service, connects its implementation (`WrapperRef`) to the ORB, waits for the Client request, and then invokes the MDS legacy code, as shown in Code Segment 4. The Wrapper uses the Java Native Interface (JNI) to communicate with a C function (`NativeC()`) to invoke the MDS code.

```
class Wrapper{
    ORB orb = ORB.init(args, null);
    WrapperImplementation WrapperRef = new WrapperImplementation();
    orb.connect(WrapperRef);
    omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
    NamingContext ncRef = NamingContextHelper.narrow(objRef);
    NameComponent nc = new NameComponent("Wrapper","");
    NameComponent path[] = {nc};
    ncRef.rebind(path, invokeServerRef);
    java.lang.Object sync = new java.lang.Object();
    synchronized (sync) {
          sync.wait();
    }
    ...
```

```
    void startSimulation( Client obj, String SimulationParameter)
    {
        NativeC(String SimulationParameter);
    }
    ...
}
```

**Code Segment 4. The Wrapper code used to invoke the MDS legacy code.**

Second, the Wrapper reads the simulation results from a shared file and requests the Client

(`obj.displaySimulation()`) to display the results to the user, as shown in Code Segment 5.

```
class Wrapper{
    ...
    readData();
    obj.displaySimulation( a,f1,f2,f3,f4,f5);
    ...
}
```

**Code Segment 5. The Wrapper code used to request a service from the Client.**

# 6 Related Work

The current concept of a PSE for computational science has its origins in an April 1991 workshop funded by the U.S. National Science Foundation (NSF) [16, 17]. The workshop found that the availability of high performance computing resources, coupled with advances in software tools and infrastructure, made the creation of PSEs for computational science a practical goal, and that these PSEs would greatly improve the productivity of scientists and engineers. This is even more true today with the advent of web-based technologies, such as CORBA and Java, for accessing remote computers and databases.

A second NSF-funded workshop on Scalable Scientific Software Libraries and Problem-Solving Environments was held in September 1995 [30]. This workshop assessed the status of PSE research and made a number of recommendations for future development. One particular recommendation was the need to develop PSE infrastructure and tools and to evaluate these in complete scientific PSEs.

Since the 1991 workshop, PSE research has been mainly directed at implementing prototype PSEs and at developing the software infrastructure – or "middleware" – for constructing PSEs. Much of this work has been presented in a recent book by Houstis et al. [19]. Initially, many of the prototype PSEs that were developed focused on linear algebra computations [10] and the solution of partial differential equations [21]. More recently prototype PSEs have been developed specifically for science and engineering applications [12, 13, 22, 33]. Tools for building specific types of PSEs, such as PDELab [35] (a system for building PSEs for solving PDEs), and PSEWare [6] (a toolkit for building PSEs focused on symbolic computations) have been developed. More generic infrastructure for building PSEs is also under development. This infrastructure ranges from fairly simple RPC-based tools for controlling remote execution [2, 32], to more ambitious and sophisticated systems, such as Legion [20] and Globus [14], for integrating geographically distributed computing and information resources.

The Virtual Distributed Computing Environment (VDCE) developed at Syracuse University [34] is broadly similar to the PSE software architecture described in Sections 2 and 3. However, components in the VDCE are not hierarchical, which simplifies the scheduling of components. Also, the transfer of data between components in VDCE is not handled using CORBA, but instead is the responsibility of a Data Manager that uses sockets. The Application-Level Scheduler (AppLeS) [5] and the Network Weather Service [37] developed at the University of California, San Diego [5] are similar to the IRMS and HIS described in Section 3, since both make use of application performance models and dynamically gathered resource information.

Visual programming based on the specification of applications and algorithms with directed graphs is the basis of the Heterogeneous Network Computing Environment (HeNCE) [4] and the Computationally Oriented Display Environment (CODE) [26]. Browne et al. have reviewed the use of visual programming in parallel computing and compared the approaches of HeNCE and CODE [7]. Though a similar approach is used by the VCCE described in Section 2, HeNCE and CODE were designed for use at a finer level of algorithm design; thus, they require a greater degree of sophistication in their design. SCIRun [28] is a PSE for parallel scientific computing that also uses directed graphs to visually construct applications and has been designed to support visual steering of large-scale applications.

# 7 Discussion and Conclusions

This paper has described the software architecture of a problem-solving environment for collaborative computational science and engineering. The PSE is designed to provide transparent access to heterogeneous distributed computing resources and is intended to enhance research productivity by making it easier to construct, run, and analyze the results of computer simulations. A PSE may be used for tasks such as rapid prototyping, design optimization, and detailed analysis.

## 7.1 Summary

Much of the PSE infrastructure described in this paper is generic and can be used to build PSEs for a range of applications. Thus, a PSE for a particular application domain has a tiered structure, with the lowest tier consisting of the generic PSE infrastructure. The middle tier contains the domain-specific infrastructure (components, expert assistance, input wizards, performance history database) that is needed to create a PSE for an application domain. The upper tier contains the models (high-level components and associated input and output data) that will be used from within the PSE.

Three key aspects of a PSE are collaborative tools, visualization, and intelligence. It is these features that distinguish a PSE as a powerful environment for research and knowledge discovery, rather than being merely a sophisticated interface.

Collaborative working is supported in several ways within a PSE. First, the VCCE provides a collaborative software development environment through the concept of a web-accessible Component Repository. Second, collaborative data analysis and exploration is supported through the cloning of user interface components. Third, electronic notebooks and living documents are a mechanism for sharing the results of previous simulations, and provide a research record.

Visualization is becoming increasingly important in computational science and engineering, both as a mechanism for runtime monitoring and steering of applications, and for post-mortem data analysis and navigation. Therefore, it is essential that a PSE tightly integrates visualization, computation, and analysis. This can be done using the component-based software engineering approach upon which the VCCE is based. Using the VCCE, computation and analysis components can be linked to user interface components that display the data visually. It is important to make the mode of visualization "resource aware." Thus, if the visualization platform is a CAVE a fully immersive representation of the data should be provided. If a semi-immersive flat-panel display is used, the visualization should be appropriate for that platform. Finally, if a PC is used as the visualization platform the data could be displayed using VRML, or some other mechanism for depicting virtual reality on a PC.

"Intelligence" is important for ensuring the PSE is easy to use and efficient, and is incorporated into the PSE in a number of ways. The Expert Assistant provided in the VCCE helps users to locate and use components using decision tree and/or rule-based approaches. The Input Wizard checks that the input provided to an application does not violate any physical or algorithmic constraints. Lastly, the IRMS can use information from previous runs of an application or component to make scheduling decisions. This approach can make use of intelligent methods, such as genetic algorithms and neural networks.

PSEs have the potential to fundamentally change how computational resources, instrumentation, and people interact in doing research. Although some interesting PSEs are now beginning to emerge, research into PSEs of the type described in this paper is still at an early stage in many areas. It is hoped that PSE research will lead to the adoption of software standards that in turn will provide the basis on which PSE infrastructure can be based. The Common Component Architecture is one such standardization effort [3].

### 7.2 Other Issues and Future Work

Additional issues that will be addressed in future versions of our PSE infrastructure include support for fault tolerance, debugging and state management of components, user authentication, and component integrity checking. This section briefly outlines our approach to tackling some of these issues in the future, and is not meant to be an exhaustive coverage of possible ways to tackle these concerns. Issues such as quality of service (QoS) are currently beyond the scope of this work, and we rely on third party tools from low level infrastructure providers to guarantee this. Support for resource reservation is also not provided, and real time applications which require such services must negotiate resources with providers before commencing a simulation.

Fault tolerance can be provided in the PSE in two ways, (1) replicating common and more frequently used components to ensure availability, (2) updating the component model to support state checkpointing. Replication can be easily integrated into our current PSE infrastructure, as every component within the VCCE is primarily a place holder or reference to an executable code. Such a reference can contain both a primary reference, which is always given preference, and multiple secondary references. Hence, subject to licensing, the executable code for a component may exist in multiple places, and when adding a new reference to the component repository, the developer is encouraged to provided multiple locations where the executable code for the component may be found. If a new executable code is being added into the PSE, the developer is again encouraged to place the code at multiple locations. To achieve this, the component integration tool [24] will prompt the user to provide multiple sites for an executable code.

State checkpointing can be achieved by updating the component model to include a `checkpoint` flag, which suggests that certain input variables, or state variables within a component can be saved periodically. A checkpoint service is then provided by a given component, which monitors the state of these variables at regular intervals, and maintains their values in a database. All components which need to be checkpointed must subscribe to this service first. If a component crashes during execution, it can be restarted from its last stored state. The validity of this approach depends on the type of component, and the type of state variables being recorded. It is left to the developer of a component to ensure that enough state information within a component is tagged with the `checkpoint` flag to ensure that a component can be re-started.

Security is an important concern for many users of scientific applications, and this can relate to a number of issues, (1) authenticating users requiring access to a component (perhaps due to licensing), (2) ownership and rights of generated data from a simulation, (3) re-play of the access process to subvert password protection, and (4) component modification. Some of these issues can be dealt with by providing security certificates for every component through a central issuing authority. Each component executable can also be signed prior to adding it to a repository, to ensure that it cannot be subsequently modified. Additional concerns with respect to migratable code, in the use of mobile agents for instance, is still an open research issue [31], and being investigated actively in the agents community. Low level security provision through a Secure Sockets Layer (SSL) may be too limiting, as our emphasis is at a high level of component manipulation in the VCCE, and not at the communications layer. Individual component developers may include SSL as a way to communicate with their components; however, the PSE infrastructure cannot impose such a requirement.

## References


[1] G. Abram and L. Treinish, "An Extended Data-Flow Architecture for Data Analysis and Visualization," *Computer Graphics*, Vol. 29, No. 2, pages 17–21, 1995.

[2] P. Arbenz, C. Sprenger, H. P. Lüthi, and S. Vogel, "SCIDDLE: A Tool for Large-Scale Distributed Computing," Technical Report 213, Institute for Scientific Computing, ETH Zürich, March 1994.

[3] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinksi, "Toward a Common Component Architecture for High-Performance Scientific Computing," in Proceedings of High Performance Distributed Computing (HPDC99), 1999.

[4] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam, "Graphical Development Tools for Network-Based Concurrent Supercomputing," in Proceedings of Supercomputing 91, pages 435–444, 1991.

[5] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao, "Application Level Scheduling on Distributed Heterogeneous Networks," in Proceeding of Supercomputing 1996, Pittsburgh, Pennsylvania, November, 1996.

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

[7] J. C. Browne, S. I. Hyder, J. J. Dongarra, K. Moore, P. Newton, "Visual Programming and Debugging for Parallel Computing," Technical Report TR94-229, Department of Computer Sciences, University of Texas at Austin, 1994.

[8] S. Browne, "The Netlib Mathematical Software Repository," D-lib Magazine, Sep. 1995.

[9] B. Carpenter, Y.-J. Chang, G. Fox, D. Leskiw, and X. Li, "Experiments with HP Java," *Concurrency: Practice and Experience*, Vol. 9, no. 6, pp. 633-648, June 1997.

[10] 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, Fall 1997.

[11] CODINE. See `http://www.geniasoft.com/products/codine/description.html` for further information.

[12] J. E. Cuny, R. A. Dunn, S. T. Hackstadt, C. W. Harrop, H. H. Hersey, A. D. Malony, and D. R. Toomey, "Building Domain-Specific Environments for Computational Science: A Case Study in Seismic Tomography," *Int. J. Supercomputing Appl.* Vol. 11, No. 3, pp. 179-196, Fall 1997.

[13] 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, Fall 1997.

[14] I. Foster and C. Kesselman, "GLOBUS: A Metacomputing Infrastructure Toolkit," *Int. J. Supercomputing Appl.* Vol. 11, No. 2, pp. 115-128, Summer 1997. See also web site at `http://www.globus.org/`.

[15] D. Foulser, "IRIS Explorer: A Framework for Investigation," *Computer Graphics*, Vol. 29, No. 2, pages 13–16, 1995.

[16] E. Gallopoulos, E. N. Houstis, and J. R. Rice, "Workshop on Problem-Solving Environments: Findings and Recommendations," *ACM Computing Surveys*, Vol. 27, No. 2, pp. 277-279, June 1995.

[17] 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.

[18] GRAM. See `http://www.globus.org/gram/` for further information.

[19] E. N. Houstis, J. R. Rice, E. Gallopoulos, and R. Bramley, *Enabling Technologies for Computational Science: Frameworks, Middleware, and Environments*, published by Kluwer Academic Publishers, ISBN 0-7923-7809-1, 2000.

[20] A. S. Grimshaw, A. Nguyen-Tuong, M. J. Lewis, and M. Hyett, "Campus-Wide Computing: Early Results Using Legion at the University of Virginia," *Int. J. Supercomputing Appl.* Vol. 11, No. 2, pp. 129-143, Summer 1997.

[21] E. N. Houstis, S. B. Kim, S. Markus, P. Wu, N. E. Houstis, A. C. Catlin, S. Weerawarana, and T. S. Papatheodorou, "Parallel ELLPACK Elliptic PDE Solvers," Intel Supercomputer Users Group Conference, Albuquerque, New Mexico, 1995.

[22] D. R. Jones, D. K. Gracio, H. Taylor, T. L. Keller, and K. L. Schuchardt, "Extensible Computational Chemistry Environment (ECCE) Data-Centered Framework for Scientific Research," in Domain-Specific Application Frameworks: Manufacturing, Networking, Distributed Systems, and Software Development, Chapter 24. Published by Wiley, 1999.

[23] M. Li, O. F. Rana, M. S. Shields and D. W. Walker, "A Wrapper Generator for Wrapping High Performance Legacy Codes as Java/CORBA Components," SuperComputing 2000, Dallas, USA, November 2000.

[24] M. Li, O. F. Rana, M. S. Shields, D. W. Walker and D. Golby, "Automating Component Integration in a Distributed Problem Solving Environment," Active Middleware Services Workshop at HPDC9, Pittsburgh, USA, August 2000.

[25] H. D. Lord, "Improving the Application Environment with Modular Visualization Environments," *Computer Graphics*, Vol. 29, No. 2, pages 10–12, 1995.

[26] P. Newton and J. C. Browne, "The CODE 2.0 Graphical Parallel Programming Language", in Proceedings of the ACM International Conference on Supercomputing, July, 1992.

[27] The Open Software Description Format, `http://www.w3.org/TR/NOTE-OSD`.

[28] S. G. Parker, M. Miller, C. D. Hansen, and C. R. Johnson, "An Integrated Problem Solving Environment: The SCIRun Computational lSteering System," in Proceedings of the 31st. Hawaii International Conference on System Sciences (HICSS-31), pages 147–156, January 1998.

[29] O. F. Rana, M. Li, M. S. Shields, and D. W. Walker, "Implementing Problem-Solving Environments for Computational Science." Proceedings of the European Conference on Parallel Computing (EuroPar 2000), held in Munich, Germany, 29 August - 1 September, 2000.

[30] 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, Fall 1996.

[31] Security in Mobile Agent Systems, See web site at: `http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/security.html`

[32] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima, "Ninf: A Network-Based Information Library for Globally High Performance Computing," in *Proceedings of the 1996 Parallel Object-Oriented Methods and Applications Conference*, Santa Fe, New Mexico, February 1996.

[33] G. Spezzano, D. Talia, S. Di Gregorio, R. Rongo, and W. Spataro, "A Parallel Cellular Tool for Interactive Modeling and Simulation," *IEEE Computational Science and Engineering*, Vol. 3, No. 3, pp. 33-43, 1996.

[34] H. Topcuoglu, S. Hariri, W. Furmanski, J. Valente, I. Ra, D. Kim, Y. Kim, X. Bing, and B. Ye, "The Software Architecture of a Virtual Distributed Computing Environment," in Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC-6), Portland, Orgeon, August 1997.

[35] S. Weerawarana et al., Proceedings of the Second Annual Object-Oriented Numerics Conference, Sunriver, Oregon, April 1994.

[36] S. Weerawarana, J. Kesselman and M. J. Duftler, "Bean Markup Language (BeanML)," Technical Report, IBM TJ Watson Research Center, Hawthorne, NY 10532, 1999.

[37] R. Wolski, N. T. Spring, and J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," *Future Generation Computer Systems*, Vol. 15, pp. 757–768, 1999.

[38] H. Wright, K. Brodlie, J. Wood, and J. Procter, "Problem Solving Environements: Extending the Role of Visualization Systems," Proceedings of the European Conference on Parallel Computing (EuroPar 2000), held in Munich, Germany, 29 August - 1 September, 2000.

[39] M. Young, D. Argiro, and S. Kubica, "Cantata: Visual Programming Environment for the Khoros System," *Computer Graphics*, Vol. 29, No. 2, pages 22–24, 1995.