A COLLABORATIVE CODE DEVELOPMENT ENVIRONMENT FOR COMPUTATIONAL ELECTRO-MAGNETICS

Matthew Shields, Omer F. Rana
Cardiff University
Cardiff, UK

David. W. Walker
Oak Ridge National Laboratory
Oak Ridge TN, USA

David Golby
BAe Systems
Filton, Bristol, 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. We describe a visual code development tool within a PSE, which enables computational scientists to construct applications by connecting components. The granularity of each component can vary from being a complete code, to a mathematical routine such as a matrix or PDE solver. We first outline the requirements of such an environment, illustrating these with our implementation. The implementation of a computational electro-magnetic solver is then described using this code development tool, based on a 2D boundary element code. We emphasise lessons learned, and the importance of using such an environment to support new application development.

Keywords: collaborative software, problem-solving environments, distributed computing, computational electromagnetics

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35407-1_22

1. INTRODUCTION

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 scientists' productivity, primarily due to the lack of a framework for tool integration and ease-of-use considerations.

The modern concept of a PSE for computational science [11] is based on the availability of high performance computing resources, coupled with specialised software tools and application specific knowledge. 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, enabling access to remote computers and databases.

The aim of our PSE is to provide the ability to build up scientific applications by connecting or plugging software components together, and to provide an intuitive way to construct scientific applications. Hence, a PSE must contain: (1) application development tools that enable an end user to construct new applications, or integrate libraries from existing applications, (2) development tools that enable the execution of the application on a set of resources. In this definition, a PSE must include resource management tools, in addition to application construction tools, albeit in an integrated way. Based on the types of tools supported within a PSE, we can identify two types of users: (1) application scientists/engineers interested primarily in using the PSE to solve a particular problem (or domain of problems), (2) programmers and software vendors who contribute components to achieve the objectives of the category (1) users. The PSE infrastructure must support both types of users, and enable integration of third party products, in addition to application specific libraries. In this paper we are concentrating on category (1) users.

Our application interface to the PSE is called the Visual Component Composition Environment (VCCE). The VCCE contains a Program Composition Tool (PCT), which enables a user to construct scientific applications by combining components obtained from local or remote component repositories. All components have interfaces defined in XML, based on a PSE-wide data model. A project investigating the use of XML for defining component properties and interfaces is OSD [22], which supports 'push'-based applications to 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 this description format is principally aimed at installing new versions of existing components, and does not facilitate the discovery or description of properties of a given component. Another component description scheme is IBM's BeanML [3], which enables a user to describe the properties of a component, using a specialised data model, and which is subsequently translated to Java code. This description scheme is primarily based on developing graphical components, and primarily aimed at Java. The Koala project [16] at INRIA is aimed at providing an object markup language, to enable serialization of a Java object. It has primarily focused on developing graphics applications, and has not been used for encoding properties of objects, such as execution constraints associated with a given object, or groups of objects. There is also work by the OMG in creating a CORBA Component Model (CCM) in XML, enabling an XML description to be automatically translated into CORBA IDL [5], and vice versa. A "component" is defined as a new basic meta-type in CCM, enabling components to be defined by extensions to standard IDL, and be either 'standard' or 'extended' component types. A CORBA component interacts with a 'Container', and provides support for call backs and a Portable Object Adapter (POA). Our component model is more generic, supporting both data types in a particular lanaguage (such as Java), and also execution specific details to be added to a component description. Hence, a component in our system can be a wrapped code, or compiled Java bytecode, with constraints defining what the code needs to run (such as whether it is an MPI code, and therefore requires MPI libraries to be available on the system), and constraints on the types of platforms on which the code can be run. Components are wrapped as binary codes, and the source code is not manipulated in any way – as in many instances, the source code is proprietary and not accessible. In the case of compound components, binary codes are integrated together, and the source code of the individual components is not involved. Component repositories must be statically connected to the PCT, prior to launching the VCCE. A user can also register new components with the local repositories which contain instances of local or remote components. The component model and a more detailed description of the VCCE architecture can be found in [14].

Visual Programming is also an active research discipline, and various languages and tools have been developed within the community – a list of projects can be found at [28]. In the context of a PSE, various visual composition tools can be utilised, generally a user can construct an application by combining "program blocks" with a particular function, such as in AVS, Khoros and IRIS Explorer. In these systems the emphasis is on integrating blocks written in a particular programming language (such as C), or a particular scripting language (such as Scheme or Python). Visual programming tools to support parallel program construction have also been investigated, primarily as tools to combine language blocks from a particular parallel programming library, such as HeNCE [15] for PVM, to support specialiased data partition for array based computations [2], or to enable the management and description of specialised data structures [4]. The visualisation in these latter environments has primarily been aimed at facilitating program development and integration from different modules. These environments are very specialised, and involve blocks containing low level descriptions of program statments that need to be combined to generate a single program. Higher level component composition environments also exist, which involve modules which can range from complete applications to specialised language statements – albeit for a specific language, and generally in the context of a particular application domain, such as CLEMENTINE (for data mining). Each of these environments supports constructs to combine 'nodes' into a data flow graph, and enable the construction either of new functionality, or of a complete application. These tools provide support for managing conditionals, loops and compound components to varying extents. In our system we borrow concepts of data flow based composition from some of these environments, but also enable the description of specialised services, such as an 'events' service, which enables the execution of components either on a single machine, or a parallel computer. The event service interacts with a resource manager which can manage execution on a parallel machine. Our approach can therefore support binary components which support a specialised functionality (such as a mathematical library), or a complete application. We therefore borrow from existing work in visual programming languages, but provide support for checking component properties based on a system wide data model. Our approach is also more general, and can encapsulate approaches taken by systems specific to a given programming model, or to a specific application domain. We support the latter by providing specialised components that are specific to a given domain (such as components for reading from a structured database – for data mining [25]) and general purpose components for visualising the results of an experiment, and writing the results to a file, for instance.

1.1. RELATED WORK

We outline some existing PSE projects, which have become popular, and employ some aspects of the infrastructure described previously:

- The Gateway project [9] introduces a component based system implemented using JavaBeans and utilising dataflow techniques to represent the application as a directed graph. 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 [1] 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.
- SCIRun [17],[18] provides a programming environment to support interactive construction, de-bugging, and steering of large-scale scientific applications. The focus in SCIRun is on computational steering, supporting application, algorithm and performance steering.
- The Distributed Problem Solving Environment Component Architecture Toolkit (CAT) [19] is a component-based toolkit for integrating heterogeneous software components. Aimed specifically at science and engineering, a CAT component can be dynamically inserted into the system and be made to interact with other CAT components, regardless of differences between architecture, operating system, and programming language. The end-user interacts with this PSE through a graphical interface, which provides a visual workspace in which components can be created and connected. Before the user can decide which components and machines to

employ, she must have access to information about the hardware and software resources available on the system. This facility is provided by the CAT Resource Information Service (RIS). The RIS comprises an "Information Server" which maintains an LDAP database, and stores hardware and software meta-data, and an "Information Browser", a graphical tool packaged with the CAT that allows a user to search and browse the contents of the LDAP database.

- The Netsolve project [20] enables the user to define problems in a specialised language, not dissimilar to Matlab. Interfaces are also provided for Fortran, C and Java. Netsolve also supports access to both hardware and software based computational resources distributed across a network, supporting load-balancing and resource discovery using a collection of interacting agents.
- The Parallel ELLPACK project [21] is a PSE for PDE based applications. Implemented using the ELLPACK language and sequential solver libraries, it also contains finite element methods, third party solvers, and a graphical interface for problem specification. Support is also provided for running the generated application on parallel machines.

Other projects which share features of a PSE, but do not provide both a program integration/generation tool and a resource manager include PARDIS [23], PAWS [24], and various resource management systems. Based on existing projects, a PSE must therefore: (1) allow a user to construct domain applications by plugging together independent components. Components may be written in different languages, placed at different locations, or exist on different platforms. Components may be created from scratch, or wrapped from legacy codes; (2) provide a visual application construction environment; (3) support web-based task submission; (4) employ an Intelligent Resource Management System to schedule and efficiently run the constructed applications; (5) make good use of industry standards such as middleware (CORBA), document tagging (XML); (6) must be easy for users to extend within their domain.

2. BOUNDARY ELEMENT CODE

The boundary element code described in this paper is called *be2d.* The code is a two dimensional boundary element simulation code for the analysis of electro-magnetic wave scattering. The main inputs to the program are a closed two dimensional contour and a control file defining the characteristics of the incident wave. The contour file consists of a series of `x,y` coordinate pairs and is generated by a separate mesh generation program. The control file is a series of property values for the wave and consists of values for the wave frequency in Hertz, the wave direction in radians and a complex number representing the amplitude. For the computation of the matrix elements, the code uses a two dimensional formulation of Rau-Wilton-Glisson elements [26]. The outer integrations use one-point quadrature, while the inner integrations use two-point quadrature. A direct LU decomposition solver is used for computing the field.

The code is written in Fortran and to run the original version, the user first runs the mesh generator from the command line. This produces the data file that represents the two dimensional contour. The user then has to run the *be2d* solver from the command line, ensuring that the contour data file and the wave control file are in the same directory. The solver produces two output files, one containing the radar cross section data and the other the surface current.

In the version of the code that we use from within the PSE, various parts of the code and data generators are wrapped as CORBA objects. The components that make up the complete assembled code are:

- The mesh generator, for generating the ellipse curve which defines the example 'model' or geometry for the *be2d* program. This is the original mesh generator with a CORBA wrapper that generates a CORBA object representing the data set, instead of writing the data to a file.

- The wave control component that defines the characteristics of the incidence wave, frequency and angle. This is a simple object representation of the control file. This component has two outputs, the frequency of the wave and it's angle.

- The *be2d* boundary element solver. This is the original solver, modified to accept input data from CORBA objects and which outputs a radar cross section as a CORBA object.

- The database component, for storing the output from the solver. This component was not part of the original code, it takes multiple output objects from the solver and stores them in a database.

Each of these CORBA components is used through a CORBA server object, called the *ActionFactory*, based upon the Abstract Factory design pattern [12]. This pattern abstracts the object creation and execution details behind a factory interface, when the user selects a component for instantiation, the PSE connects to an instance of the *be2d ActionFactory* server. The factory object is then responsible for instantiating and executing that component. In this way the PSE does not need to know details about object instantiation or execution. The *Actionfactory* has a single method `exec()` which accepts a number of parameters, including the name of the process to execute, a set of input parameters and some details about the execution of the process, and returns a result set.

A typical call to the *ActionFactory* would be:

```
short[] result = factory_.exec(action_name_,
    numBddETag,
    parameter,
    fact_machine_name_,
    fact_obj_name_,
    fact_port_number_);
```

Where `action_name_` represents the name of the command to be executed, for instance `ActionReadMesh` is the action that generates the mesh object, the return value for which is a reference to a CORBA object representing the mesh. `ActionComputeRCS` is the action that executes the solver, it returns another reference to the CORBA object that represents the radar cross section result set. `numBddETag` and `parameter` are arrays representing the input data for this component, the value of these is either a simple data type as in the case of the wave frequency and angle, or a reference to a CORBA object in the case of the mesh. The action factory is responsible for deliverying input datasets to the appropriate components to be executed to complete a given action. The final parameters, `fact_machine_name_`, `fact_obj_name_` and `fact_port_number_` are values that the *ActionFactory* uses to decide what component to execute and where to run it. All of these values are stored in the XML component definitions, which are parsed by the PSE and represented by a proxy component. Hence, running a particular component is achieved by a single function call to the *ActionFactory* instance, found at component instantiation time, passing in the values stored in the proxy together with any output parameters from the previous component in the graph.

3. USING THE PSE

When the PSE is started, the VCCE first checks in the component directory or directories for all defined components. The directories that the application examines are defined in an application meta-data file. The XML component definitions are parsed, the proxy components created and added to the component tree ready to be selected by the user. Illustrated in Figure 1.

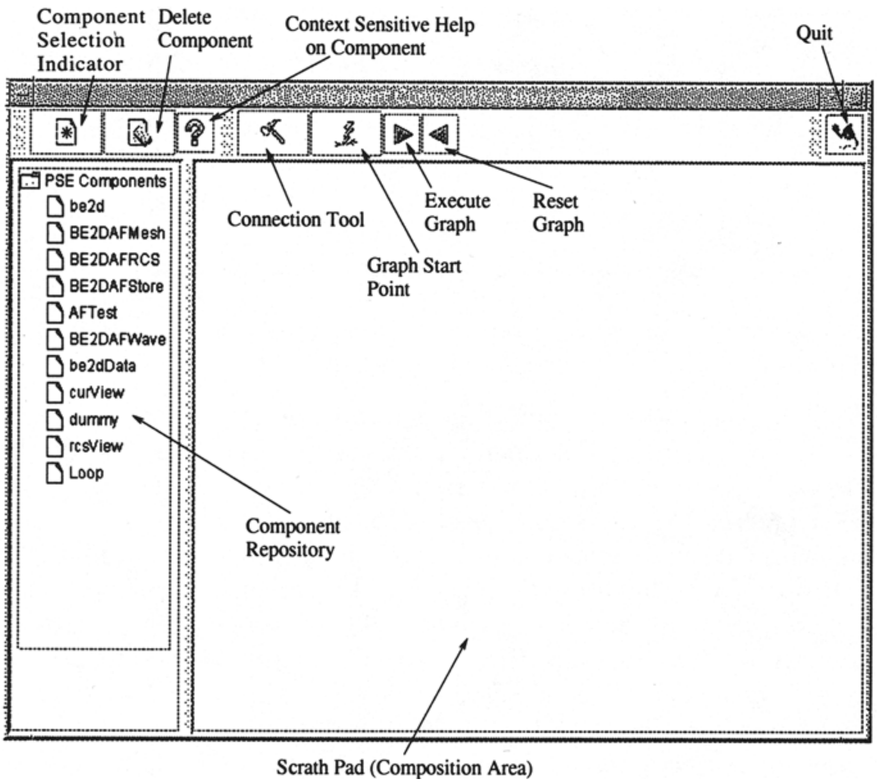


Figure 1 VCCE with loaded component directory

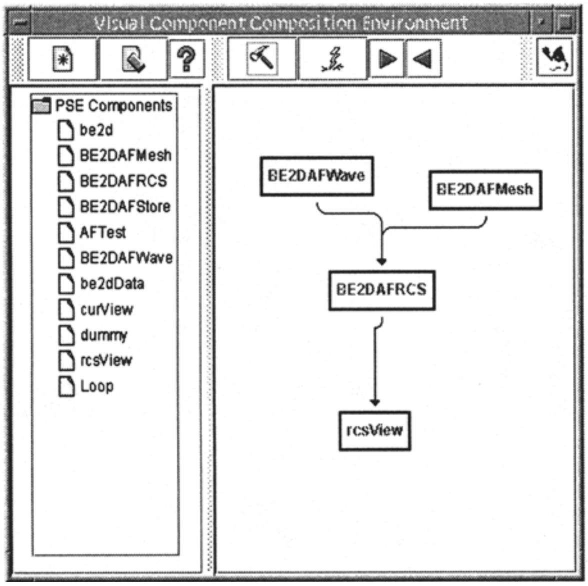


Figure 2 A constructed work flow graph

To assemble a set of components into an executable task graph, the user simply selects a component, from the tree with the mouse, and clicks on the scratch pad on the right of the screen. This intuitive selection process has the same features as many visual programming or windows based environments, such as mouse based selection and "drag and drop". Once the desired components have been selected and placed on the scratch pad, they can be connected together using the connection menu button. The user clicks this button then selects two components, parent first then child, the PSE then establishes a data flow connection from the parent to the child. Repeating this process allows the user to connect the components together into a task graph. The final task the user has to perform before executing the graph, is to assign a start node or nodes. A graph can have more than one start point, if there are two initial input generating components for instance. The assembled and connected *be2d* graph is illustrated in Figure 2. To execute the completed graph the user simply presses the start button to initiate the simulation.

The solver is combined with a graph generator (JChart [27]) as a third party component. The output generated from the code is illustrated in figure 3.

One of the features of the PSE is to provide the ability to perform iterations over components in a task graph. A more complex task graph

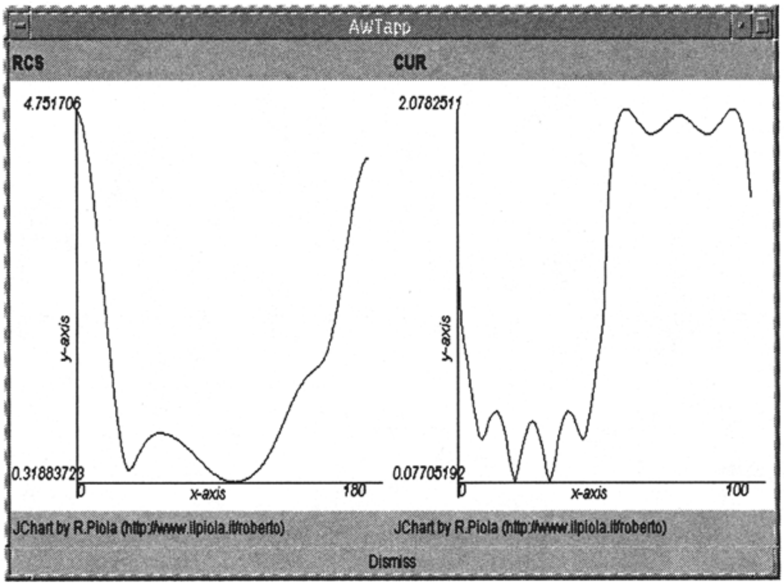
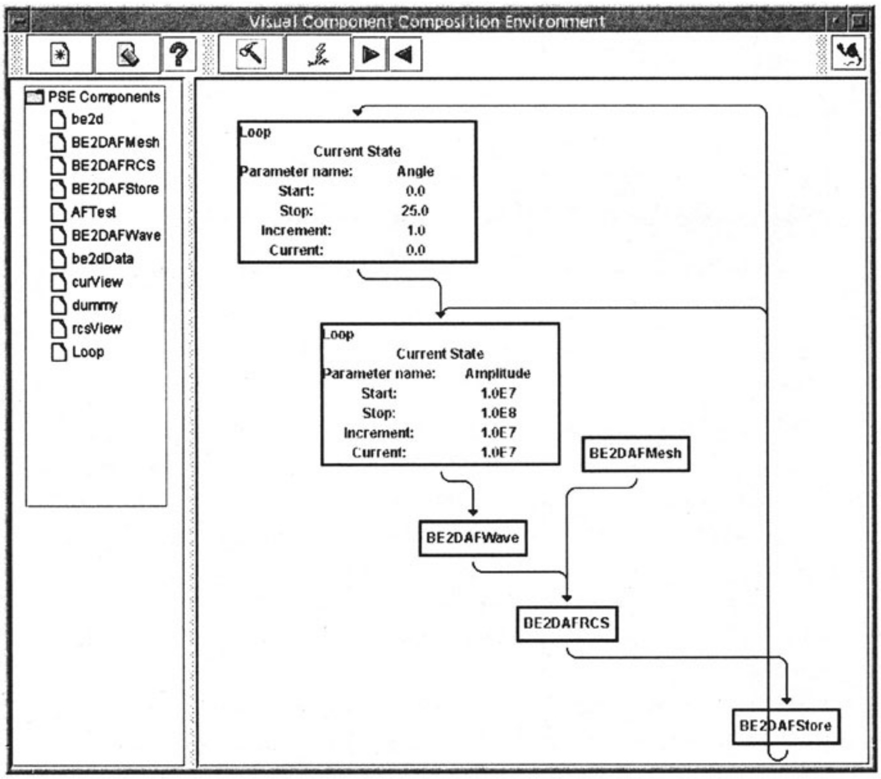


Figure 3 Solver Output

containing two control components in addition to the components needed for the solver is illustrated in figure 4. When these control components are connected to another component, the user is prompted with a selection of control input parameters that are suitable for iteration. In the case of *be2d*, the two input parameters to the *wave* component are the frequency and angle of incidence, being floating point values, these are suitable for iteration and there is a separate control component for each. The user can set the start, finish and iteration values for each parameter. When the graph executes, the PSE will loop over the components iterating the input parameters according to the user defined values.

4. COMPONENT INSTANTIATION AND GRAPH EXECUTION

The *be2d* components used as the example in this paper are CORBA components. When the components are instantiated by the PSE upon selection by the user, the PSE has to establish a connection to the CORBA ORB that has a running instance of the *be2d* factory and get a reference to the factory from that ORB. The XML component definitions contain information about where to find the factory, the host machine, the port number and the name of the factory object as well as the CORBA version, in this case Orbacus.The PSE automatically



Loop
Current State

Parameter name:	Angle
Start:	0.0
Stop:	25.0
Increment:	1.0
Current:	0.0

Loop
Current State

Parameter name:	Amplitude
Start:	1.0E7
Stop:	1.0E8
Increment:	1.0E7
Current:	1.0E7

Figure 4 A work flow graph with control components

performs the connection to the ORB and retrieves a reference to the *be2d ActionFactory* when a proxy component is added to the scratch pad. When the proxy is told to execute by the PSE, it calls the `exec()` method on the instance of the factory it has a reference to, see the previous section on the be2d code 2 for more specific details.

In more general terms when a task graph is executed, each of the components in the graph is told to execute in turn, starting with the component or components that have been identified by the user as the starting point for the graph. After component has executed it sends back an event to the VCCE to say that it has finished executing. At this point the VCCE will transfer the output parameters from the completed component to the input parameters of the next component to be executed and then call that components `execute` method.

When the control components are introduced into a task graph, by connecting them to a suitable component, they provide the input to that component. In the case of the example in this paper, see figure 4, there are two control components. One to control the angle of the wave and the other to control the frequency. These control components work in a similar manner to the traditional "for...next" loop in most programming languages, stepping through a series of values from a start value until an end value is reached incrementing by a set value. At each iteration of the loop the PSE checks that the halting condition has not been reached and then passes the current loop value to the connected component. In the case of the wave component, this value is simply passed straight through to it's output parameter, where it becomes one of the input values for the solver. When, as in this example, there are more than one control components connected to a single component, the execution is equivalent to a nested loop and the execution will continue until the halting condition on the outer loop is reached. The inner loop value is reset to it's starting value every time the outer loop performs an iteration.

4.1. ERROR HANDLING

Error handling in the PSE is undertaken by a Program Analysis Tool, which performs a set of checks on the components. It checks that:

- The data types for input and output ports on components are consistent. The first check involves ensuring that the syntax for the data types match, based on the XML based component description provided. It then checks that the cardinality of the data types match, and finally, whether the input/output is streamed into/out of a component, or whether it needs to be read/written from/to a file.

- A log file exists for recording Execution errors, maintained local to the point where the component is being executed. Hence, regardless of where the *ActionFactory* undertakes component execution, the log file is maintained at the same place where the execution is undertaken.

- Component constraints have not been violated by the component execution tool. These constaints can relate to the availability of specialised programming libraries (such às MPI or PVM), the existence of specialised operating systems (such as Solaris) or the availability of specialised system requirements, such as memory.

Since the component source code is not modified, verification of component behaviour is not undertaken within the PSE. It is assumed that each component has been verified and operates accorded to specifications provided by the component developer. A user can however place specialised components, such as a 'loop' components to undertake parameter tests, prior to using the component in an application. Results of these runs can be recorded into a file, and analysed for discrepancies between the specified output (by the component developer) and the expected output (by the component user). The output can also be analysed by specialised statistical components to undertake correlations between different experimental runs.

5. SUMMARY AND CONCLUSIONS

A PSE is aimed at supporting an application scientist in solving a problem within a given application domain. The "problem solving" process involves a range of activities from both a user's point of view, and a systems point of view. The intention being to abstract details of software and hardware, which correspond to the system point of view, from the application scientist. The user's point of view relates to being able to specify the problem in a decompositional manner, whereby an application is composed of interacting components, each of which undertakes a particular function. The data flow approach of combining components to compose applications is perhaps the most intuitive way to construct applications, and has been used by a range of other tools (such as AVS Explorer and Khorous) – others being a "declarative specification", "script based specification", and a "high-level programming language based specification". We feel the graphical approach adopted within our PSE is more generic, and can be used to extract a script in XML. We provide support for handling "conditionals" and "iterators" within the data flow approach, in addition to hierarchy to compose "compound" components. Additional details of these can be found in [7]. Wrapping legacy codes as components within a PSE is a non-trivial undertaking, particular when making use of CORBA and Java based implementations. For instance, in order to support interoperability across programming languages, CORBA supports the minimal subset of data types across these. This could lead to data type incompatibility when wrapping Fortran or C codes, in terms of numerical precision and supported operations on particular data types such as complex numbers. The way in which legacy codes are wrapped can also affect the reusability of the resultant component. Wrapping the entire code as a single monolithic component is more straightforward, but smaller decomposed components may be more effectively reused in this way [8].

The VCCE simplifies the process of running a complex scientific code, using the intuitive visual programming paradigm. The application scientist does not need to configure software components, and can concentrate on undertaking parameter runs or visualising output from a solver. Various components are provided to achieve some of these functions, which may be local to the scientist or refer to components held at other sites.

Although there is an obvious overhead in having a legacy code wrapped as a CORBA object, the cost is not as great as it might appear. We have undertaken performance comparisons of wrapped legacy codes on both workstation clusters [8] and dedicated parallel machines [6]. The most time consuming part of using a CORBA object is the initial "handshaking" with the ORB. The VCCE performs the CORBA connection at component instantiation and not execution time. The user is still performing the process of building the graph at this time, so the cost in time is not really noticed. Once the graph comes to execution, the CORBA connections are already in place and the speed of execution is not affected by a discernible amount compared to the original code executed via the command line.

References

[1] 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/`.

[2] D. Banerjee and J. C. Browne, Complete Parallelization of Computations: Integration of Data Partitioning and Functional Parallelism for Dynamic Data Structures. In Proceedings of IPPS, 1996.

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

[4] D. R. Brown, B. V. Zander, The Whiteboard Environment: An Electronic Sketchpad for Data Structure Design and Algorithm Description. IEEE Symposium on Visual Languages, Nova Scotia, Canada, 1998.

[5] Object Management Group, The CORBA Component Model (CCM), OMG Documents: orbos/99-07-01, orbos/99-07-02, and orbos/99-07-03 See web site at: `http://www.omg.org`, 1999 Also, see web site at: `http://www.ditec.um.es/ dsevilla/ccm/`

[6] 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, November 2000 (to appear)

[7] D. W. Walker, M. Li, O. F. Rana, M. S. Shields, and Y. Huang, "The Software Architecture of a Distributed Problem-Solving Environment", Concurrency:Practise and Experience (to appear)

[8] M. Li, O. F. Rana and D. W. Walker, "Wrapping MPI-Based Legacy Codes as Java/CORBA Components", Journal of Future Generation Computer Systems, Special Issue on High Performance Java, (to appear)

[9] 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.

[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] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, Massachusetts, 1995.

[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] Matthew S. Shields, David W. Walker, Omer F. Rana, and Maozhen Li. An XML Based Component Model for Generating Scientific Applications and Performing Large Scale Simulations in a Meta-Computing Environment. *Procedings of the International Symposium on Generative and Component Based Software Engineering (GCSE)*, 1999.

[15] Heterogeneous Network Computing Environment, See web site at: http://netlib2.cs.utk.edu/hence/

[16] The Koala Project, See web site at: http://www-sop.inria.fr/koala/, 2000

[17] SCIRun: Scientific Computing and Imaging. See web site at: http://www.cs.utah.edu/~sci/.

[18] C. Hansen G. Kindlmann C. Johnson, S. Parker and Y. Livnat. Interactive Simulation and Visualization. *IEEE Computer*, December 1999.

[19] Dennis Gannon and Randy Bramley. Component Architecture Toolkit. See web site at: http://www.extreme.indiana.edu/cat/.

[20] Henri Casanova and Jack Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.

[21] E. N. Houstis, J. R. Rice, S. Weerawarana, A. C. Catlin, P. Papachiou, K.-Y. Wang, and M. Gaitatzes. Parallel ELL-PACK: A Problem Solving Environment for PDE Based Applications on Multicomputer Platforms. See web site at: http://www.cs.purdue.edu/research/cse/pellpack/paper/pellpack-paper-1.html.

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

[23] Katarzyna Keahey and Dennis Gannon. PARDIS: CORBA-based Architecture for Application-Level PARallel DIStributed Computation. *Proceedings of Supercomputing97*, November 1998.

[24] Peter Beckman, Patricia K. Fasel, William F. Humphrey, and Susan M. Mniszewski. Efficient Coupling of Parallel Applications Using PAWS. *Proceedings of High Performance Distributed Computing (HPDC) 7 Conference, Chicago*, 1998.

[25] Omer Rana, David Walker, Maozhen Li, Steven Lynden, and Mike Ward. PaDDMAS: Parallel and Distributed Data Mining Application Suite. Proceedings of IPDPS/SPDP, Cancun 2000.

[26] S.M. Rao, D. R. Wilton and A. W. Glisson. Electromagnetic Scattering by Surfaces of Arbitrary Shape. IEEE Trans. Antennas Propagat., Vol. AP-30, pp. 409-418, May 1982.

[27] The JChart Package, See web site at: `http://www.ilpiola.it/roberto/jchart/index_e.html`.

[28] Visual Programming and Languages, See Web site at: `http://cui.unige.ch/Visual/`

DISCUSSION

Speaker: David Walker

Masaaki Shimasaki : Does VCCE provide the functionality in composition of software (i.e., network programming) similar to that of the AVS system for visualization?

Richard Fateman : Can you comment on the extent to which the PSE/visual programming environment design has drawn on the decades of experience in the programming language design community, in particular, solutions to issues of functional composition, information hiding, name spaces, scope, exception handling?

David Walker : Visual Programming is an active research discipline, and various languages and tools have been developed within the community – a list of projects can be found at `http://cui.unige.ch/Visual/`. In the context of a PSE, various visual composition tools can be utilised, generally a user can construct an application by combining "program blocks" with a particular function, such as in AVS, Khoros and IRIS Explorer. In these systems the emphasis is on integrating blocks written in a particular programming language (such as C), or a particular scripting language (such as Scheme or Python). Visual programming tools to support parallel program construction have also been investigated, primarily as tools to combine language blocks from a particular parallel programming library, such as HeNCE for PVM, to support specialiased data partition for array based computations, or to enable the management and description of specialised data structures. The visualisation in these latter environments has primarily been aimed at facilitating program development and integration from different modules. These environments are very specialised, and involve blocks containing low level descriptions of program statments that need to be combined to generate a single program. Higher level component composition environments also exist, which involve modules which can range from complete applications to specialised language statements – albeit for a specific language, and generally in the context of a particular application domain, such as CLEMENTINE (for data mining). Each of these environments supports constructs to combine 'nodes' into a data flow graph, and enable the construction either of new functionality, or of a complete application. These tools provide support for managing conditionals, loops and compound components to varying extents. In our system we borrow concepts of data flow based composition from some of these environments, but also enable the description of specialised services, such as an 'events' service, which enables the execution of components either on a single machine, or a parallel computer. The event service interacts with a resource manager which can manage execution on a parallel machine. Our approach can therefore support binary components which support a specialised functionality (such as a mathematical library), or a complete application. We therefore borrow from existing work in visual programming languages, but provide support for checking component properties based on a system wide data model. Our approach is also more general, and can encapsulate approaches taken by systems specific to a given programming model, or to a specific application domain. We support the latter by providing specialised components that are specific to a given domain (such as components for reading from a structured database – for data mining) and general purpose components for visualising the results of an experiment, and writing the results to a file, for instance.

Anne Trefethen : When combining a set of modules into a single hierarchical module are you creating a single interface to those modules or are you combining the modules by manipulating their source code in some way?

Bruce Char : Are there any major efforts in describing components (through XML or other means) outside of the scientific computation community that problem-solving environment builders should be aware of? How close are we to having component description standards that would allow components such as commercial document processing or databases to be incorporated in this kind of PSE architecture?

David Walker : A project investigating the use of XML for defining component properties and interfaces is OSD, which supports 'push'-based applications to 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 this description format is principally aimed at installing new versions of existing components, and does not facilitate the discovery or description of properties of a given component. Another component description scheme is IBM's BeanML, which enables a user to describe the properties of a component, using a specialised data model, and which is subsequently translated to Java code. This description scheme is primarily based on developing graphical components, and primarily aimed at Java. The Koala project at INRIA is aimed at providing an object markup language, to enable serialization of a Java object. It has primarily focused on developing graphics applications, and has not been used for encoding properties of objects, such as execution constraints associated with

a given object, or groups of objects. There is also work by the OMG in creating a CORBA Component Model (CCM) in XML, enabling an XML description to be automatically translated into CORBA IDL, and vice versa. A "component" is defined as a new basic meta-type in CCM, enabling components to be defined by extensions to standard IDL, and be either 'standard' or 'extended' component types. A CORBA component interacts with a 'Container', and provides support for call backs and a Portable Object Adapter (POA). Our component model is more generic, supporting both data types in a particular lanaguage (such as Java), and also execution specific details to be added to a component description. Hence, a component in our system can be a wrapped code, or compiled Java bytecode, with constraints defining what the code needs to run (such as whether it is an MPI code, and therefore requires MPI libraries to be available on the system), and constraints on the types of platforms on which the code can be run. Components are wrapped as binary codes, and the source code is not manipulated in any way – as in many instances, the source code is proprietary and not accessible. In the case of compound components, binary codes are integrated together, and the source code of the individual components is not involved. Component repositories must be statically connected to the PCT, prior to launching the VCCE. A user can also register new components with the local repositories which contain instances of local or remote components.
Richard Fateman : It would be useful for designers to be conversant with issues of functional programming, functions as first-class objects, lexical scope. One reference would be the text "Structure and Interpretation of Computer Programs" by H. Abelson and G. Sussman, McGraw Hill/MIT Press.
David Walker : We agree that functional programming languages provide an elegant way of describing composition, and in fact have been the basis of other work, such as "algorithmic skeletons". The functional programming paradigm is however difficult to grasp for many non-computer scientists, and there is little suggestion that functional programming languages, such as Haskell, Miranda and Hope, have achieved the adoption compared to C or Java. Our emphasis is on visual programming, whereby applications can be visually constructed from blocks. There is a possibility, however, of translating the visual representation into a functional language, and a translator can be written to achieve this.
Mladen Vouk : Errors committed by users during specification and solution engineering using problem-solving environments can be very costly. Standard software engineering teaches us that verification and validation (V&V) should be an integral part of the specification process (this includes both domain-specific, math-specific, and environment-

predicated V&V). Your system does not appear to have any explicit V&V hooks, tools, or process points. Why? Do you plan to incorporate them into the system in the future?
David Walker : Error handling in the PSE is undertaken by a Program Analysis Tool, which performs a set of checks on the components. It checks that:

- The data types for input and output ports on components are consistent. The first check involves ensuring that the syntax for the data types match, based on the XML based component description provided. It then checks that the cardinality of the data types match, and finally, whether the input/output is streamed into/out of a component, or whether it needs to be read/written from/to a file.

- A log file exists for recording Execution errors, maintained local to the point where the component is being executed. Hence, regardless of where the *ActionFactory* undertakes component execution, the log file is maintained at the same place where the execution is undertaken.

- Component constraints have not been violated by the component execution tool. These constaints can relate to the availability of specialised programming libraries (such as MPI or PVM), the existence of specialised operating systems (such as Solaris) or the availability of specialised system requirements, such as memory.

Since the component source code is not modified, verification of component behaviour is not undertaken within the PSE. It is assumed that each component has been verified and operates accorded to specifications provided by the component developer. A user can however place specialised components, such as a 'loop' components to undertake parameter tests, prior to using the component in an application. Results of these runs can be recorded into a file, and analysed for discrepancies between the specified output (by the component developer) and the expected output (by the component user). The output can also be analysed by specialised statistical components to undertake correlations between different experimental runs.
Masaaki Shimasaki : In your development of problem-solving environments, are there any specific design features that relate directly to electromagnetics computation or to the specific needs of end users?
David Walker : Our PSE contains specialised components used for supporting electromagnetic applications. These include:

- The mesh generator, for generating the ellipse curve which defines the example 'model' or geometry for the *be2d* program. This is the original mesh generator with a CORBA wrapper that generates a CORBA object representing the data set, instead of writing the data to a file.

- The wave control component that defines the characteristics of the incidence wave, frequency and angle. This is a simple object representation of the control file. This component has two outputs, the frequency of the wave and it's angle.

- The *be2d* boundary element solver. This is the original solver, modified to accept input data from CORBA objects and which outputs a radar cross section as a CORBA object.

- The database component, for storing the output from the solver. This component was not part of the original code, it takes multiple output objects from the solver and stores them in a database.

Each of these CORBA components is used through a CORBA server object, called the *ActionFactory*, based upon the Abstract Factory design pattern.