Visual Programming Environments for Multi-Disciplinary Distributed Applications

Matthew S. Shields

May 27, 2004

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Declaration

This work has not previously been accepted in substance for any degree and is not concurrently submitted in candidature for any degree.

Signed ..... (candidate)

Date .....

STATEMENT 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ..... (candidate)

Date .....

STATEMENT 2

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date .....

Acknowledgements

The research undertaken toward this Ph.D. thesis during the period from September 1998 to September 2001 was jointly funded by grant number GR/M17327 from the Engineering and Physical Sciences Research Council and BAE SYSTEMS Ltd.

I'd like to thank David Golby, Carren Holden, David Rowse and Glen Gapper from BAE SYSTEMS, and David Walker and Omer Rana from the Department of Computer Science, Cardiff University, without whose help and encouragement this work would not have been possible.

My additional thanks go to Roger Philp from the Cardiff Centre for Computational Science and Engineering, Cardiff University for his help with the analysis of the Fortran 77 code used during the course of this research.

The prototype software was developed using the Java programming language [61], the Orbacus CORBA ORB [59] and the third party Java packages, JDOM [64], the Colt Distribution [54], SGT [27] and JChart [91]. The Java "arrow" drawing code in the prototype user interface is courtesy of Ian Wang.

This document was prepared with LaTeX 2_ε teTeX distribution [31] and input by hand using the "Vi Improved" (Vim) editor [80] and the TeXShop environment [69].

My unending gratitude and admiration to all those who contribute to the not-for-profit software movement.

Finally a thank you to those people who nagged, cajoled and encouraged me to finish this thesis: My wife Ruth, Ian Taylor, Daniel, Julie and Margaret Shields.

Abstract

A *Problem Solving Environment* is a complete, integrated computing environment for composing, compiling and running applications in a specific problem area or domain. A *Visual Programming Environment* is one possible front end to a problem solving environment. It applies the visual programming paradigms of "point and click" and "drag and drop", via a *Graphical User Interface*, to the various constituent components that are used to assemble an application. The aim of the problem solving environment presented here 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.

Problem solving environments promise a totally new user environment for computational scientists and engineers. In this new paradigm, individual programs combined to solve a problem in their given area of expertise, are wrapped as components within an integrated system that is both powerful and easy to use. This thesis aims to address: problems in code reuse; the combination of different codes in new ways; and problems with underlying system familiarity and distribution. This is achieved by abstracting application composition using visual programming techniques.

The work here focuses on a prototype environment using a number of demon-

stration problems from multi-disciplinary problem domains to illustrate some of the main difficulties in building problem solving environments and some possible solutions. A novel approach to code wrapping, component definition and application specification is shown, together with timing and usage comparisons that illustrate that this approach can be used successfully to help scientists and engineers in their daily work.

Contents

List of Figures

Listings

List of Algorithms

CHAPTER 1

Introduction

1.1 Introduction to a Problem Solving Environment

> *"A Problem Solving Environment is a computational system that provides a complete and convenient set of high level tools for solving problems from a specific domain. The PSE allows users to define and modify problems, choose solution strategies, interact with and manage appropriate hardware and software resources, visualise and analyse results, and record and co-ordinate extended problem solving tasks. A user communicates with a PSE in the language of the problem, not in the language of a particular operating system, programming language, or network protocol."*
> **Computer as Thinker/Doer, Gallopoulos, Houstis and Rice [44]**

The concept of the PSE promises to revolutionise the way in which computational scientists and engineers work with their computing tools and resources by abstracting away the underlying, often irrelevant, complexity of their computing environments, leaving them free to concentrate on using their domain expertise in solving problems. Details such as operating system specifics, data migration and

application execution are often a complicated, yet unrelated, part of a scientists working environment which PSE implementations aim to remove or at least alleviate. To realise a useful PSE, the user must have the ability to construct scientific applications by connecting, or plugging, software components together in an intuitive way and hide the underlying system complexities.

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 co-operatively to improve a programmer's productivity. The primary factors that prevent this are the lack of frameworks for tool integration and ease-of-use considerations. Extensions to current scientific programs such as Matlab, Maple, and Mathematica are particular pertinent examples of this scenario. Developing extensions to such environments enables the reuse of existing code, but may severely restrict the ability to integrate routines that are developed in other ways or using other applications. Multi-Matlab [79] is an example of one such extension for parallel computing platforms.

The modern concept of a PSE for computational science [45] is based on the availability of high performance computing resources, coupled with advances in software tools and infrastructure which make the creation of such PSEs a practical goal. PSEs have the potential to greatly improve the productivity of scientists and engineers, particularly with the advent of Internet-based technologies, such as CORBA and Java for accessing remote computers, databases and other resources. At a 1995 NSF workshop on PSEs [97], the need to develop and evaluate PSE infrastructure and tools was stressed. Subsequently, a number of prototype PSEs have been developed. Many early PSEs focus on linear algebra computations and the solution of Partial Differential Equations (PDEs), and as yet only a few prototype PSEs have been developed especially for science and engineering applications. However, this is likely to change over the next few years. Tools for building specific types of PSEs have been developed and more generic infrastructures for building PSEs are also under development, ranging from fairly simple Remote Procedure Call (RPC) based tools for controlling remote execution to more ambitious and sophisticated systems such as

Globus [34] and Legion [50] for integrating geographically distributed computational and information resources. However, most of these PSEs lack the ability to build up scientific applications by connecting and plugging software components together.

Component-Based Software Engineering (CBSE) [13] is receiving increasing interest in the software engineering community. The goal of CBSE is to reduce development costs and improve software reuse. The question of how to apply the technologies involved in CBSE to the construction of an effective framework for PSEs is becoming increasingly important and is considered in this work.

Web Services are a recent development in the distributed computing field, however they are only considered in this thesis in the recent related work, chapter 9 and future work, chapter 10.

In this thesis a CORBA-based domain-independent problem solving environment for scientific computations and large-scale simulations is considered. In this PSE, a user can visually construct domain specific applications by plugging together software components which are independent of location, programming language and platform.

1.2 Motivation

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 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. PSEs form a central component of both the Accelerated Strategic Computing Initiative (ASCI) currently underway in the United States, and recently formed e-Science program in the United Kingdom.

1.3 Hypothesis

To produce a general purpose Problem Solving Environment (PSE) for industry. The PSE can be used in various problem domains and with various "legacy" software components. It will provide an environment for executing software, that is easily configured, easily adapted for different users and user scenarios, and is not prohibitively expensive in terms of execution overhead. Techniques based around commodity hardware and software will be used to deal with the complexity of developing such a system.

The PSE will attempt to help BAE SYSTEMS, the industrial partner, in overcoming difficulties in areas such as the interaction between different scientists with differing specialities and multiple existing or envisioned software requirements. The need for software environments such as this is particularly pertinent in industry where the research scientist may not be a computer scientist, with expertise in systems and software other than those used on a daily basis, but a person trying to do a particular job. It is believed that in situations such as this the software environment produced will be of benefit to the scientists in their day-to-day work.

This research will concentrate on the prototype Visual Component Composition Environment (VCCE) and its use within a number of problem areas at BAE SYSTEMS, Cardiff University and Southampton University. The demonstration problems include a molecular-dynamics simulation, an electromagnetic wave scattering simulation and the use of the PSE within a Design Of Experiments (DOE) controlling the execution of a Computational Fluid Dynamics (CFD) code multiple times in an optimisation of aircraft wing design.

1.4 Questions Asked in this Thesis

- *How can a PSE be built?*
- *What is the overhead of using the PSE?*

- *What are the advantages to the scientist of using the PSE?*
- *Do the advantages of using the PSE outweigh the disadvantages?*
- *What is the cost of converting existing or "legacy" software to work within the PSE?*

1.5 Research Achievements

There are a number of novel aspects to this work. They are covered in the following chapters and where applicable are substantiated by published papers and compared to related work. The full list of papers is given in appendix A, page 161. The novel aspects of this work include:

An infrastructure for application development

The prototype provides an environment that allows a user to construct applications from "legacy" software wrapped as CORBA components. See chapters 5,7 and papers 3 and 4 in appendix A.

An extensible XML-based component model

Used to specify all of the components used within the PSE. It can be extended to represent the differing components used within the system and also the *work flow* graph relationship between the components in a constructed application. Pervasive throughout the whole thesis, including chapters 4, 5, 6, 7, first described in the paper 1 appendix A.

Loop and control components

Can be inserted into constructed work flows to allow the user to control application components in new combinations, for instance performing repeated execution or decision-based branching. Chapter 6.

Integration of optimisation and search algorithms

An extension to the control/loop mechanism that allows the user to insert an

algorithm as a loop component to give a more intelligent iterative control flow. Chapter 8

A working prototype

Important from a software engineering point of view. The software prototype was actually usable and is still being used in a limited fashion. Additionally many of the ideas prototyped in this work have been incorporated into a production PSE, used by many people in many different research specialities. This aspect of the work is not directly pertinent to this thesis, since it was carried out after the period of research but before this document was finished. However, it is discussed as part of the Conclusion and Future Work chapter, chapter 10.

1.6 Project History

The work considered in this thesis was carried out during the period from September 1998 to September 2001. This area of research is a particularly fast moving one with advances made in many areas including middleware infrastructure. Although the author is fully aware of work since the end of 2001, this document discusses the research hypothesis in the context of that period. Where available novel features are time stamped and compared to work of the same period. For completeness related work from 2002 to date is discussed, with reference to the research presented in this dissertation, in chapter 9, current and future work is discussed in the conclusion chapter.

1.7 Thesis Outline

The remainder of this thesis is as follows:

Chapter 2 - Related Work

In this chapter, the concept and architecture of the modern PSE is examined

together with the infrastructure needed to build one. Previous work closely related to these efforts is also outlined.

Chapter 3 - Software Engineering

A short overview of software engineering techniques and best practises and how they were used in this research to develop the prototype.

Chapter 4 - PSE Architecture

This chapter contains a description of the architecture, the design and building of the current prototype tool. A standard component model and definition language is introduced. Design decisions are analysed and previous prototypes examined.

Chapter 5 - Use Case 1, Simple Solver Component

The first of the industrial demonstrators, a two-dimensional boundary element code written in FORTRAN that performs an electromagnetic wave scattering simulation. Included are details of different wrapping techniques for the legacy code: simple executable calling; conversion from FORTRAN to Java; simple CORBA wrapper.

Chapter 6 - Control and Loop Components

Extensions to the prototype and definition language to include simple loop controls to allow users to perform parameter runs on solver codes.

Chapter 7 - BE3D, FE3D

In this chapter, two more industrial demonstrators that are *real* production codes and consequently substantially more complicated are discussed. The two large parallel solvers are wrapped using a more complicated and flexible CORBA wrapper; extensions to the component definition language for the CORBA components.

Chapter 8 - Design Of Experiments

This chapter examines a different use case for a PSE. The PSE is used to control an industry "Design of Experiments" process. First a generic domain space

search component is outlined; this is used to control the execution parameters of other components, in an example domain space search case.

Chapter 9 - Related Work post VCCE

The work detailed in this thesis finished at the end of 2001, however the field is fast moving. This chapter relates important work since that time to the work contained herein and provides a comparison.

Chapter 10 - Conclusion and Future Work

This chapter concludes the thesis. The hypothesis is evaluated, drawing conclusions from each of the use case examples. The achievements and contributions are discussed and the approach taken here is compared to those of similar projects. Areas where this work can be used in future projects and research is also discussed.

CHAPTER 2

Related Work

2.1 Introduction

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) [44, 45]. 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 goal is even more pertinent today with the advent of web-based technologies, such as CORBA and Java, for accessing remote computers and databases and the emerging promise of "computational grids" [35].

A second NSF-funded workshop on Scalable Scientific Software Libraries and Problem-Solving Environments was held in September 1995 [97]. 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.

PSEs come in many forms from the simplest, which could consist of sets of simple scripts for executing binary programs with data dependencies in the form of file based data. For example, the situation described in section 7.1, where a mathematical modeller wants to run a three dimensional boundary element solver. The input to the solver is a mesh file that itself is generated by a mesh generator program and an input file. The output from the solver is another data file which is passed to yet another piece of software for either post-processing or visualisation. The scientist may write a set of scripts that run each of the constituent programs in turn moving the output file to the location expected as input by the next program. At the other end of the scale there are large functionally complex PSEs with complicated graphical user interfaces and visualisation tools.

2.2 Related Work

There are a number of groups working on related and complementary work, in areas of PSEs such as middleware, seamless access to compute resources, user interfaces and visualisation techniques. Much of this work has been presented in a recent book by Houstis et al. [57]. In many ways, a PSE can be seen as a mechanism to integrate different software construction and management tools, and application specific libraries, within a particular problem domain. One can therefore have a PSE for financial markets [17], for gas turbine engines [33], etc. Focus on implementing PSEs is based on the observation that previously, scientists using computational methods wrote and managed all of their own computer programs. However now computational scientists must use libraries and packages from a variety of sources, those packages might be written in many different computer languages. Initially, many of the prototype PSEs that were developed focused on linear algebra computations, for instance NetSolve [20], or the solution of partial differential equations

(PDEs), for instance ELLPACK [56]. More recently prototype PSEs have been developed specifically for science and engineering applications Seismic Tomography [24], Application Engineering [26], Computational Chemistry [66], and Modelling and Simulation [108]. Tools for building specific types of PSEs, have been developed: PDELab [118], a system for building PSEs for solving PDEs; PSEWare [11], a toolkit for building PSEs focused on symbolic computations. More generic infrastructure for building PSEs is also under development. This infrastructure ranges from fairly simple Remote Procedure Call (RPC-based) tools for controlling remote execution, SCIDDLE [7] and Ninf [104], to more ambitious and sophisticated systems, such as Legion [50] and Globus [34], for integrating geographically distributed computing and information resources.

The Virtual Distributed Computing Environment (VDCE) [109] developed at Syracuse University is broadly similar to the PSE software architecture described in section 4.3. 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) [10] and the Network Weather Service [121] developed at the University of California, San Diego are scheduling systems, both make use of application performance models and dynamically gathered resource information.

2.3 Visual Programming

A Visual Programming Environment (VPE) is one possible front end to a PSE. Others range from simple command line interfaces to internet-based portal environments. Visual programming based on the specification of applications and algorithms with directed graphs is the basis of the Heterogeneous Network Computing Environment (HeNCE) [9] and the Computationally Oriented Display Environment (CODE) [86]. Browne et al. [14] have reviewed the use of visual programming in parallel computing and compared the approaches of HeNCE and CODE. Though a similar approach

is used by the Visual Component Composition Environment (VCCE) described in section 4.3 of this thesis, 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 [101] 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. SCIRun is a large PSE developed by the University of Utah mainly for medical problem solving, it has a large number of components for medical modelling and three dimensional visualisation and medical imaging. It is also part of the ambitious human body modelling project.

The basic functionality of the VCCE environment is similar to that of other modular visualisation environments such as AVS [78], IBM Data Explorer [1], IRIS Explorer [37], and Khoros [123]. An article by Wright et al. [122] has reviewed these types of modular visualisation environments. The VCCE differs from these environments through its use of an XML-based component model; an event model that is able to support check pointing; control constructs such as loops and conditionals. In addition the VCCE is open source and platform independent, written in the Java programming language, a major consideration for the industrial partner in this research.

The Gateway project [40] introduces a similar idea to the system being developed in this thesis. It is a component-based system implemented using JavaBeans [62] and utilising data flow techniques to represent the meta-application, the application comprised from components, as a directed graph. Unlike the prototype system in this thesis, which uses XML to define both the interface to all components within the system and the task graph that describes the constructed application, the Gateway system chooses to use the Abstract Task Descriptor (ATD) as its lowest level of granularity of instruction and to build up the instructions that define the application.

The Adaptive Distributed Virtual Computing Environment (ADViCE) [52] project 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 the work here, the ADViCE system has its own scheduler that allocates tasks to resources at run time.

The Arcade project [22] uses a slightly different approach in that the system has a three tier architecture. 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 computing environment.

The UNICORE project [111, 30] is a science and engineering grid making resources available over the Internet. UNICORE has a graphical user interface that allows a user to compose jobs consisting of multiple dependant tasks, submit those jobs and monitor them on the available resources. Dependencies between tasks, in the same manner as this work, indicates a temporal relationship or a data transfer. To create a seamless environment, tasks and resources are represented in abstract terms. A UNICORE server translates the abstract jobs and resource requests into platform-specific operations prior to execution, and schedules tasks according to dependencies. For each task, the input and output files are automatically imported/exported from/to the user's file space or transferred from earlier tasks in the same job. Explicit transfer tasks handle the high-speed data transfer between different sites. The UNICORE servers select the most efficient mechanism for each transfer.

Work similar to the component model presented in this work, see paper 1 in appendix A, was presented at the International Symposium on Generative and Component-Based Software Engineering [16]. The work presented, similar to the model in this thesis, addresses the shortcomings of traditional component models such as the JavaBeans framework. These component models have no capacity to define non-interface properties of components, such as license requirements, hardware/software requirements or performance models. The difference between the two, is that the work in this dissertation uses an XML-based model to define the non-functional requirements, whereas the other uses BCOOPL, a concurrent, object-

oriented language.

In the Gateway system, the Abstract Task Definition (ATD) forms the lowest level instruction, and all components must be defined in terms of ATDs. There is therefore no straightforward way of wrapping legacy codes, or providing for existing executables. The presented here provides an XML wrapper, which requires far less effort from a user than developing ATD definitions. However, both systems share the general design objective of constructing applications by linking sequential and parallel components. The ADViCE system is a visual composition tool above all, and provides little support for legacy applications. ADViCE does enable automatic selection of components from a library, based on particular parallel libraries being available to a user, i.e. PVM, MPI etc., and then sends this to a resource scheduler. The Arcade project requires the construction of specialised Applets, and can be restrictive due to security requirements of the Java sandbox. The approach for the VCCE is more generic, in that XML is adopted for specifying an interface to a component, and for encoding the connectivity graph. Also provided is an interface to a scheduling system.

2.4 Parallel Computing and CORBA

In chapter 7, CORBA is used to wrap two large parallel codes. CORBA was not originally intended to support parallelism within an object. However, some CORBA implementations provide support for multi-threading which enables a programmer to make more effective use of several simultaneous processors sharing a physical memory within a single computer, such as in the TAO ORB [99]. However, the sharing of a single physical memory does not support a large number of processors, since it could create memory contention. There are very few projects aimed at supporting coarse-grain parallelism in CORBA. The PARDIS [67] environment and Cobra [93, 28] being the most advanced projects in this direction, both of which extend the CORBA specification and add parallelism to CORBA objects based on a SPMD (Single Program Multiple Data) execution model. PARDIS designers propose

a new kind of object they call a SPMD object and Cobra designers provide parallel CORBA objects. SPMD objects represent parallel applications which roughly adhere to the SPMD style of computation. To support data distribution among different threads associated with a SPMD object, PARDIS provides a generalisation of the CORBA `sequence` called a `distributed sequence`. However, this new argument type requires a modification to the standard IDL compiler.

In the Cobra system, a parallel object belongs to a collection of objects and its interface must be defined in a particular system, for instance:

```
interface[*]
  Test{...}
```

where the object `Test` is a parallel object and belongs to a collection of objects. The * symbol signifies that the number of objects belonging to the collection is not specified in the interface, and defines a polymorphic type. The PARDIS system provides a mechanism to invoke operations on objects asynchronously, based on the concept of a `future` message. In the Cobra system, asynchronous invocations of services is handled by an extension of the Asynchronous Method Invocation (AMI) which is a core part of the CORBA messaging specification in CORBA 3.0.

The approach taken in this dissertation is quite different from the previous two. In wrapping the MPI-based legacy code as one or more components (CORBA objects), neither the CORBA specification nor any IDL compilers are modified. Hence, any CORBA system may be used with the techniques demonstrated here, whereas with PARDIS and Cobra additional software must be downloaded and installed. The MPI runtime is combined within the CORBA environment, using the MPI runtime to manage the *intra*-communications of components, and the CORBA ORB to manage *inter*-communications of components. Each component is composed of two parts: one is the wrapper which can be invoked by a client, the other is the execution unit of the component.

2.5 PSE Requirements and Aspects

During the initial phases of this work, due to the fact that part of the funding was from an industrial partner, looking for a solution to a problem, there were a number of requirements gathering discussion sessions. Out of these discussions came a number of functional requirements, "use case" scenarios and other aspects and ideas that could possibly be included in the design and implementation of a PSE. These ideas help in categorising common functionality and allow clarification of some of the design ideas for the PSE. Not all of these ideas would be implemented as part of this work, as this would in all probability require far too much time.

2.5.1 Use Case Scenarios

"Use Case" scenarios are a concept that has come out of the Software Engineering community. It is a way of specifying functionality in a system through the interaction of that system with a user. The user may or may not be an actual person. The technique is discussed further in section 3.2 in the software engineering chapter. The "Use Cases" that defined here are:

Running a legacy application as a wrapped component.

The granularity of the wrapping can vary, dependant upon a number of factors such as availability of source code. The code can be treated as a *black box* with input and output typically through files, or interfaces can be written to allow the interoperability of the code directly. The application may be a sequential code, or may contain internal parallelism using MPI or PVM.

Performing parameter runs.

Running existing or new applications multiple times with varying input parameters, to study the effect of parameter ranges on the result or to try and find an optimal solution for a problem.

Combining components.

Combine various wrapped third party or internal components to generate a

new application. The application can itself be stored as a separate component in a Component Repository, this is known as hierarchical composition.

Searching for suitable components.

Search for a component by name or functionality in various repositories maintained on the local machine or via the internet at a remote site, where each component is defined in some common component definition language.

Automatic application generation.

Developing a new application, using some form of automated mechanism to either select new components, or migrate an application to a different platform. In the latter case, the mechanism would be used to analyse effects of platform constraints on a given application code or set of components.

Visualising results.

The ability to visualise the results of an application either on a local machine or remotely using appropriate tools and techniques.

Computational Steering.

Enabling and supporting Computational Steering where the scientist can directly affect the running of an application by providing guidance in the form of modified data.

2.5.2 User Categories

In addition to the use cases, two categories of users could be identified for a PSE:

Application Users.

Such as physicists, chemists or biologists who are not interested in creating new components, other than compound components.

Application Developers.

Mainly computational scientists, who create new components both for their own use and for other Application Developers and Users.

2.5.3 Functional Modes

In the requirements specification, the PSE should provide two modes of execution:

Edit Mode.

The mode that enables components to be assembled together by selecting and connecting them visually into a work flow graph.

Execution Mode.

The constructed task graph is sent to the IRMS. The user has the option to visualise execution of components, and thereby perform computational steering.

2.5.4 Resource Stealing

With the explosive growth of the Internet and the emerging promise of computational grids, there are many unused or under-utilised computational resources. These can include different PSEs and libraries on the Internet that may be reused. A user who wants to utilise these resources must look for the appropriate library or set of libraries needed for his specific computational problems. Usually, such libraries can be found in established repositories. A well known repository for mathematical software, for example, is Netlib [15], which is maintained through the collaborative efforts of several institutions and universities. Once the appropriate library has been located, it must be downloaded and installed. This process depends on the nature of the software, as a lot of freely distributed software is not fully portable. Some software may need to be re-compiled and re-linked, which is a time-consuming task if the user does not have prior experience in software installation. A Distributed PSE (DPSE) should provide different interfaces to different computational resources. Based on the problem descriptions submitted by users, DPSEs can match good solutions for the problems through a performance model. Future PSEs may be less generic and more problem-specific in the software resources they provide. However, the user of a PSE may submit a task and wait for the results without worrying about how to

download and install components within the PSE. Web Services are beginning to provide functionality along these lines.

2.5.5 Aspects of PSEs

Most modern software applications are built in a modular fashion and PSEs are no different. A clear distinction can be made between those potential modules of a PSE that are generic and hence could be used in constructing other PSEs, and those that are specific to one application domain. The following parts of a PSE, as defined from the requirements, are specific:

The Components in a Component Repository. These are mostly specific, although some may have broad applicability, for example graphing components, generic filters, or data movement tools.

Expert Assistance. To aid a user in various aspects of application building and component selection.

Input Wizards. Tools that help with specific components.

Performance History Database. Records the performance of components under different circumstances.

The remaining modules listed here are more generic and could be used across multiple application domains:

Component Connection and Communication Framework. The mechanism by which data is passed between components and how components communicate with the environment itself.

Scheduling and Work Flow Engine. The mechanism by which components are allocated to compute resources and the control of execution of those components on their resources.

Data Management and Input/Output Some aspects of data input and output such as file reading and writing or data streaming can potentially be generalised into a number of re-usable components.

User Interface. The overall interface to a PSE, be it VPE, Portal or simple command line is abstract enough to be reused across problem domains.

It is not necessary to use all of the above modules in any one PSE, those used will depend on the level of functionality required and how sophisticated the system is going to be.

2.5.6 Wrapping Legacy Codes as CORBA Objects

Legacy codes are existing codes that often possess the following features:

- They are domain-specific, typically written for a specific purpose.
- They are not reusable in any format other than their original use.
- They are still useful. Unused or no longer required codes will fall into disuse and will eventually be lost or deleted.
- They are large, complex monoliths. Older, purpose built codes were rarely written with modularity in mind.

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. The ability to wrap legacy codes is often affected by the access to the source code. If no access to source code is available, for instance because the code is proprietary or even has been lost, then the wrapping can only really be done at application level. Whereas if full access to source is obtained then it is possible to wrap the code at many different levels and even re-engineer

the code to become more modular. An example of this explained in further detail in chapter 5. After being wrapped as CORBA objects, these legacy codes can be reused as components in the prototype PSE.

2.5.7 Summary

This chapter has provided a review of related work, including: other PSEs from application specific areas such as PDE solving, computational chemistry and financial markets; infrastructure upon which PSEs can be built; visual programming environments and techniques; component models. Also outlined are a series of "use case" scenarios and classifications which will be used throughout the dissertation. The following chapter is a brief discussion of software engineering techniques.

CHAPTER 3

Software Engineering and the Development Cycle

3.1 Introduction

For any modern software development project it is becoming increasingly important that best practises and good software engineering techniques are followed if the project is to fulfil its requirements and deadlines. This chapter briefly outlines relevant techniques, design principles and practises used during the course of this research. It is not intended to be a comprehensive review of the subject, more a list from various resources that the author finds useful.

Although the work undertaken in building the prototype for this research was not strictly a software engineering project there are enough similarities, that good software engineering techniques could and should be used. Like a lot of software projects it was not obvious from the outset of the project what the final software would look like. BAE SYSTEMS, the industrial sponsors, provided an idea of what the user interface should look and behave like based upon an existing software system, but the architecture was not clear.

3.2 Use Case Analysis

Use Case Analysis is a requirements gathering and software engineering technique, first discussed in [60], that specifies software in terms of the interactions between the software and its *users*. The user concerned, may or may not be human. A particular interaction may be between:

- A human operator and the software.
- Another piece of software and the specified software.
- A piece of equipment, such as a sensor, and the software.

In thinking about how the software would interact within its environment there is a completely different view from that specified by a functional view. Examining the interactions will highlight who or what the software needs to have contact with as opposed to the functional view of what actions it should perform. Both views are needed for a detailed requirements analysis.

3.3 Prototyping

Prototypes are used in many industries to try out ideas, as prototyping is much cheaper than full production. For example car manufacturers may make a clay model for wind tunnel testing or computer model for safety test simulation, building prototypes to examine specific aspects of a design or try out new ideas.

Software engineering is no different. Prototypes are built in the same fashion and for similar reasons - to analyse and expose risk, and to be able to make corrections before the process has gone to far. A prototype can be targeted to test one or more specific aspects of the design. Prototypes are designed to examine a small aspect of the overall design and so are quicker to develop than a full production application. The prototype can ignore details that are unimportant at the moment.

One thing that is important with prototyping is that the code written should be disposable. Prototypes are designed to be thrown away once their purpose has been fulfilled.

3.4 Tracer Code and Iterative Design

The term "tracer code" or "tracer bullets" was coined in the software engineering book, *The Pragmatic Programmer* [58]. It's an analogy to tracer bullets loaded on the ammunition belt of a machine gun. When fired, their phosphorous ignites and leaves a pyrotechnic trail from the gun to whatever they hit. This trail gives an instant feed back and because they operate in the same environment as the real bullets, external effects are minimised.

This analogy can be applied as a technique to the domain of software engineering, especially when attempting to build something that hasn't been built before. Requirements may be vague and languages, libraries and techniques may be unfamiliar. Rather than specifying the system in minute detail and producing large amounts of requirements, the tracer code method is to look for something that gets us from a requirement to some aspect of the final system quickly and in a repeatable manner.

Unlike prototypes, tracer code is not disposable. It contains all of the error checking, structure, documentation that any production code has, it is not fully functional. Once an end-to-end connection has been achieved among the components of the system, a check can be made to determine how close to the target the solution is, adjusting if necessary. Once on target adding functionality is easy.

Tracer development is consistent with the ideas of iterative design. A project is never finished, there will always be changes to make or functionality to add. Unlike the traditional approach where development is broken down into modules and sub-assemblies which are built in isolation and not assembled into the application until they are all ready.

Tracer code has a number of advantages:

- Users get to see something early and the developers have something to demonstrate.
- Developers build a structure to work in and have an integrated platform.
- All concerned parties have a better feeling for progress.

Prototype code is disposable. Tracer code is basic but complete and will form the framework for the final system. Prototyping is done before tracer coding and can be thought of as the basic requirements gathering.

Software engineers have for a long time tried to formalise design methodologies and processes. The classical approach was known as the *waterfall method.* In this method the developer starts with requirements gathering which leads into analysis and design, then coding, through to debugging and finally maintenance. At this stage the software is deemed stable and finished, software maintenance is the only stage left and this continues for the remaining useful life of the software. This model is very static and not workable in modern systems. Typically programmers will iterate around the design, code and debug stages many times until happy with the software.

Iterative Design is a modern methodology. It allows for a more flexible approach to designing software. The programmer designs and codes small pieces of functionality testing as the process goes. Functionality is added iteratively to the application through small design and code stages until the full functionality of the application is achieved.

Iterative design is very useful when the full extent of the functionality of the application is not known at the outset.

3.5 Design Patterns

Patterns in design processes have long been recognised in various disciplines such as architecture. Christopher Alexander and his colleagues proposed the idea of using a pattern language to architect buildings and cities. In recent years these ideas have migrated into the software engineering community, for a good discussion on how these ideas came into the domain of Computer Science see [68]. The seminal book cataloguing software design patterns is *Design Patterns* [46] by the "Gang of Four", known herewith as *GOF*. This book outlines and explains a series of commonly occurring patterns in object oriented software engineering. *Java in Practise* [117] is a Java-based pattern book that gives good examples of patterns in coding Java applications.

A brief list of some of the design patterns used in the VCCE prototype follow, their relevant page reference in the "Gang of Four" book is listed together with a brief description of their purpose. They will be referred back to in some of the following chapters where their specific use within the prototype will be explained.

Factory Method *GOF page 107*

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Proxy *GOF page 207*

Provide a surrogate or placeholder for another object to control access to it.

Visitor *GOF page 331*

Represent an operation to be performed on the elements of an object structure. Visitor defines a new operation without changing the classes of the elements on which it operates.

Singleton *GOF page 127*

Ensure a class only has one instance, and provide a global point of access to it.

Observer *GOF page 293*

Define a one-to-many relationship between objects so that when one object changes state, all its dependants are notified and updated automatically.

3.6 Unified Modelling Language

The Unified Modelling Language, or UML for short, is a notation for describing the design of an object oriented software system. UML is a convergence of the three main notations that are used in the software engineering methodologies of Booch, OMT, and OOSE. UML is a general purpose notational language for the specification, documentation and aid to visualisation of the separate parts that make up an object oriented software system. The language is very extensive but the UML used in this document, as a concise means of describing relevant sections of the prototype software, is mainly limited to class diagrams and object interaction diagrams. For a description of the diagrams used see a good UML book, for example *UML Distilled* [38].

3.7 Summary

The software development that forms the basis of this research was approached with "best practises" in mind. As is the case with most research projects it was impossible to specify the software at the outset. The nature of research is such that mistakes will often be made and "dead ends" followed. Traditional software approaches are too static to be useful. Iterative design, prototyping and tracer bullet coding will provide a much more flexible environment in which to work.

CHAPTER 4

Problem Solving Environment Architecture

4.1 Introduction

This chapter covers the design, architecture and building of the prototype PSE. It takes the requirements specification and uses prototyping and "tracer" code techniques to iterate through designs. The chapter is broken up into different sections each covering aspects of the PSE including the component model, user interface, job scheduling and implementation. Not all aspects of the requirements were implemented and where this occurs in the chapter it is indicated, however these unfinished aspects are included for completeness.

Looking at a high level point of view of the prototype from the requirements, there are three important design considerations.

- For the front end of the system: an intuitive graphical user interface for visual programming.
- To represent the individual algorithmic parts used to compose an application: a component model.

- To choreograph the interaction between the components in the application: a data flow, work flow or task graph model.

4.1.1 Visual Programming and Visual Programming Environments

Graphical or visual programming environments have been in common use for many years, their use within parallel computing is reviewed in a technical report by Dongarra et. al. [14]. The form that these programming environments take is now fairly standardised at some level. The majority of systems are component-based with: a repository where the components are stored, browsed and selected; a composition area where the selected components are placed and joined together; and an underlying computation engine that performs the actual work of the system and generates the output that the user expects. The functionality of components within the system dictate the type of computation engine.

In an environment such as a programmer's Rapid Application Development (RAD) tool the aim is to generate computer code for an application by selecting the components that make up the user interface visually, rather than writing the application code by hand. The application programmer places components and then uses the environment to generate the code. The repository is a "widget" set comprising a selection of user interface components such as buttons and text boxes. The composition area is a blank user interface on which the user interface "widgets" are arranged and their interactions defined. The computation engine is the code generator and language compiler, with the end result being the finished compiled application executable.

In the PSE described in this dissertation, the aim is to build an application by selecting components from a repository and visually connecting them together, rather than taking individual codes and tools and either running them manually or through a script. The computation engine is an execution engine that is capable of scheduling and running the individual tools and choreographing the control and

data flow between them.

4.1.2 Components

In general, a component is a procedural or functional abstraction defined by its input and output interfaces and its semantics.

Components within the RAD tool example would be functional user interface components which would be defined their properties. For example a text field component would have some input text as a property, the semantics might include the other components that it can be placed on or attached to, and state change events that it can publish or subscribe to.

In the prototype PSE in this thesis, the input and output interfaces to components are the data types that are accepted as input and provided as output. A components semantics are the problem domain that it is used in and a description of the transformation process that it provides to turn input data into output. Each component is specified using the XML language, see section 4.4.2.

The Common Component Architecture (CCA) [21] is a community effort to standardise component programming for high performance computing. From its technical specification document the definition for a CCA is:

> A CCA consists of three types of entities: Ports, Components and Frameworks. Components are the basic units of software that are composed together to form applications. Instances of components are created and managed within a Framework which also provides the basic services that components use to operate and communicate with other components. Ports are the fully abstract interfaces of components which are managed by the framework in the composition process. A component is a piece of code, a port is an agreement between the authors, standard-compliant frameworks are the road to enable component exchange and reuse.

4.1.3 Task Graphs and Work Flow

The concept of work flow is not unique to the domain of computer science. It is used in areas as diverse as business processes and chemistry. Work flow and data flow are similar in terms of usage within computer science. They both allow the specification for the interaction between processes, either through the flow of control or flow of data from process to process.

There are multiple ways of representing the network of processes and the data or control connections between them, including *petri nets* and *graphs*. The network of components and the connections between them can be thought of in the mathematical model of a graph, specifically a Directed Cyclic Graph (DCG) or Directed Acyclic Graph (DAG). The components or processes are vertices and the connections between them arcs. As the data flows only one way through a component, each has an input and output, the graph is directed, i.e. data flows along arcs one way only. Also given a network of two components A and B, where output from component A is input to component B, if the output from component B is input to component A, the network is *cyclic*. If the output from B is not allowed back into the input to A then the graph is *acyclic*.

The computation engine of the Visual Component Composition Environment (VCCE), the prototype PSE discussed in this thesis, is a data flow execution model. The components are data driven with one or more input data types being processed by a component and the results being output as one or more output data types. So execution flows from component to component based on the connections between them. The data flow task graph is encoded in XML, see section 4.4.4, in a similar fashion to the components themselves. In fact the data flow task graph is really a compound component and can be represented as a component in the logical programming model. The XML representation consists of a series of component definitions for the components within the task graph and the connections between them represented as pairs of "parent/child" relationships. The parent in the pair is the start point in a data flow connection and the child is the end point. Thus given

a full set of these pairs it is possible represent a complete data flow task graph. The task graph can either be sent to a resource manager for executing the application on a workstation cluster, or a heterogeneous environment made of workstations and high performance parallel machines, or executed by the simple internal scheduler in the prototype.

4.2 Prototype Implementations

To explore the requirements for the visual programming front-end to the PSE and the component model several different prototypes were built. Each of these experiments helped to clarify the design process and lead to the final prototype explained in this work. Problems and solutions to different aspects of PSE design and execution were found using this prototyping process.

4.2.1 A Simple Arithmetic Equation Builder

The first prototype discussed here is a visual arithmetic equation builder. To simplify the process of building a PSE, a deliberately limited application domain was chosen. The equation builder is limited to the four simple operators: *addition*, *subtraction*, *multiplication* and *division*; and a simple integer or floating point operand. The prototype has a component repository from which components can be selected and a "scratch pad" that allows the components to be connected. The components consist of:

A display component

An instance of the display component has one function, to represent a value, it can be either a source or a destination component within an equation.

An operator component

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.

Using these components it is possible to build simple arithmetic equations of an arbitrary length. The user selects the components, connecting them together starting with at least two display components that provide the input into an operator component. The starting values of the top most components in the graph are set and when the graph is executed the data propagates down through the graph resulting in the final value being displayed in the display component at the base of the tree.

The equation and the interface are hard coded into a test class for this first prototype. This initial version is designed to examine the look of the user interface and test the potential use of JavaBeans as the component model. As can be seen

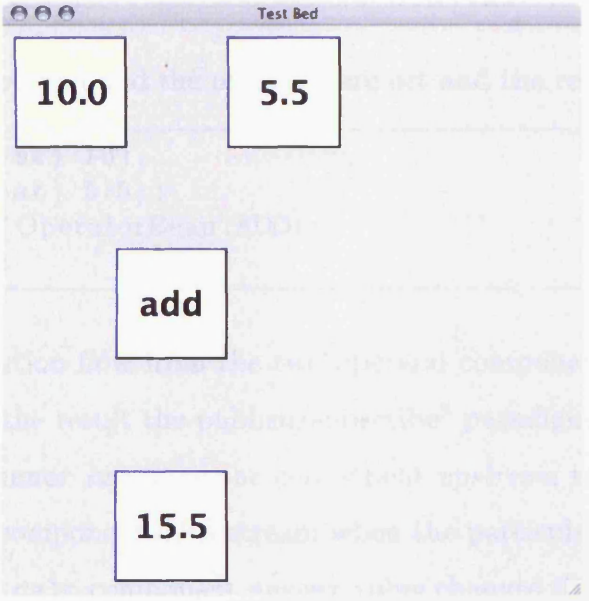

Figure 4.1: Simple Arithmetic Equation Builder User Interface

from figure 4.1, the user interface at this stage consists of the components on a pad. There is no component repository or a graphical way of connecting the components. The test class that builds the equation has a main set-up function that contains four variables representing the two operands, the operator and the result.

```
public NumberDisplayBean op1 = null;
public NumberDisplayBean op2 = null;
public OperatorBean calc = null;
```

```
4 public NumberDisplayBean ans = null;
```

The variables are all JavaBeans and are instantiated using the standard JavaBean constructor mechanism[1] of the `java.beans.Beans` class. The full listings for `NumberDisplayBean` and `OperatorBean` can be seen in appendix B sections B.1.1, B.1.2.

```
op1 = (NumberDisplayBean) Beans.instantiate(null, "
   NumberDisplayBean");
2 op2 = (NumberDisplayBean) Beans.instantiate(null, "
   NumberDisplayBean");
calc = (OperatorBean) Beans.instantiate(null, "OperatorBean");
4 ans = (NumberDisplayBean) Beans.instantiate(null, "
   NumberDisplayBean");
```

The values of each operand and the operator are set and the result calculated.

```
op1.setValue((float) 10);
2 op2.setValue((float) 5.5);
calc.setOperator(OperatorBean.ADD);
4 calc.calculate();
```

To perform the execution flow from the two operand components through the operator and finally to the result the publish/subscribe[2] paradigm in Java is used. A `PropertyChangeListener` *listens* to the component *upstream* in the work flow and fires an event to the component downstream when the particular property changes. In this way when the `calc` component answer value changes the value is communicated to the `ans` component. The code that perform this operation can be seen in the following code segment:

```
public class AnswerChangeListener implements
   PropertyChangeListener {
2    protected NumberDisplayBean ndBean = null;

4    public AnswerChangeListener(NumberDisplayBean inpBean) {
        ndBean = inpBean;
6    }
```

[1] A static *Factory Method*, see page 26

[2] Another name for the *Observer* design pattern, see page 27

```
8       public void propertyChange(PropertyChangeEvent pce) {
            if (ndBean != null) {
10              ndBean.setValue(((Float) pce.getNewValue()).
                    floatValue());
            }
12      }
}
```

An instance of the class is created and subscribed to the `calc` component using

```
calc.addPropertyChangeListener("answer", new AnswerChangeListener
    (ans));
```

and when the value changes the value is set on the `ans` component and displayed.

The location on the user interface for each "Bean" component is set using the standard AWT[3] `setLocation` method.

```
getContentPane().add(op1);
2 op1.setLocation((int) 10, (int) 10);
```

None of the standard Java AWT layout components are capable of simply setting components at a point location on a window so a custom layout is used, `BulletinLayout`[4].

This initial prototype helped to explore the use of JavaBeans as a visual component model. Although the functionality is totally static, it is extended in the next version.

4.2.2 Arithmetic Equation Builder - Version 2

The next iteration of the prototype is an extension to the functionality of the visual arithmetic equation builder from the previous section. As is good practise when prototyping the previous prototype was discarded, apart from the ideas and the lessons learnt with the new technology. This new prototype has some simple dynamic

[3] AWT: Java Abstract Window Toolkit is the standard Java user interface widget set

[4] *BulletinLayout* was written by David Geary, it lays out components as though they were pinned to a bulletin board

functionality that is capable of performing a specific set of functions. The prototype now has a component repository and a "Scratch Pad", seen here in figure 4.2, on to which components in the component repository can be instantiated by pressing the component "buttons" in the repository to add an instance of the chosen component.

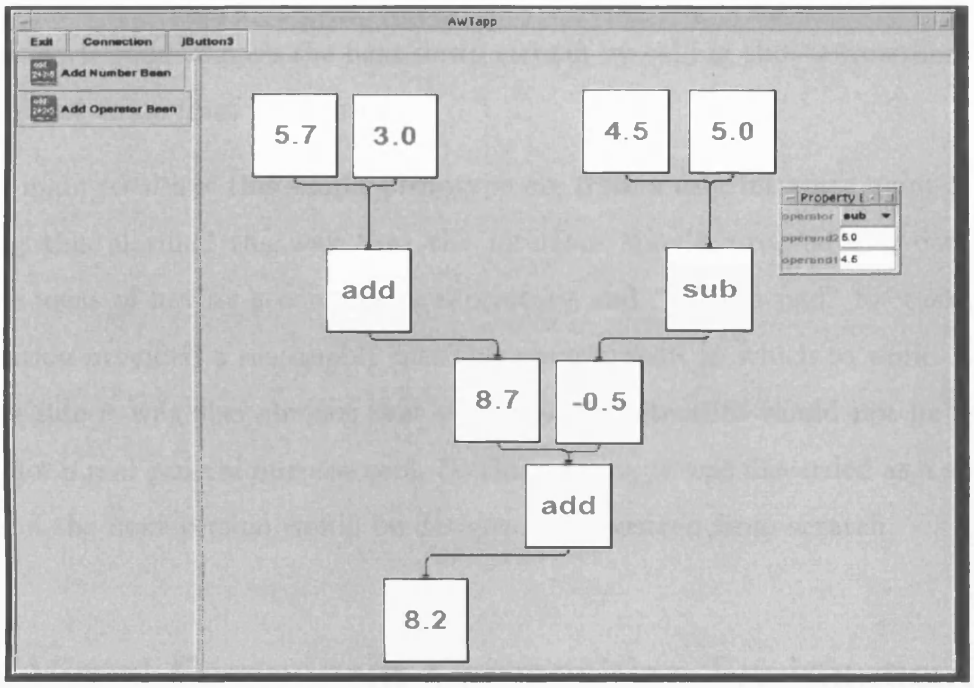

Figure 4.2: Visual Arithmetic Equation Builder Interface

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.

One solution using the Java programming language, provides the ability for the system to discover a component's properties at design time and display them via a simple "Object Inspector." This can be seen in figure 4.2 as a small floating window. 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. In this way the execution flow of the task graph cascades down the nodes of the graph as each node triggers the next down stream by calling the "set method" with the resultant value from its operation.

The main results of this simple prototype are from a user interface point of view. Building this clarified the way that the interface should proceed and confirmed that the ideas of having a component repository and "scratch pad" for task graph composition provided a reasonably intuitive environment in which to work. On the negative side it was also obvious that prototype architecture would not be flexible enough for a real general purpose tool. So this prototype was discarded as a starting point and the next version would be designed and written from scratch.

4.3 Visual Component Composition Environment

The main prototype developed during this research is called the Visual Component Composition Environment (VCCE). It is used primarily to construct applications from software components of a more complex nature than the previous simple example. 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 data flow graph. Once a complete application has been constructed, it can either be passed to the Intelligent Resource Management System (IRMS) to be scheduled on the distributed computing systems available on the network or executed using the built in scheduling system.

4.3.1 Features of the VCCE

The VCCE should have the following features, some of which are improved versions of functionality from the equation builder:

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

2. A facility for building new components from scratch in some appropriate programming language, and wrapping them as Java beans. Presently supported are components written in C, Fortran and Java.

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

4. A composition notation, or scripting language, providing control constructs such as loops and conditionals for managing the flow of execution of components. Conditional behaviour is not supported within the present implementation, however the user has the capability to specify a predefined number of iterations through the code using XML tags, as described in section 4.4.2, and an extension to the implementation to include looping is discussed in chapter 6.

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

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

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

1. A Component Repository, containing a hierarchical set of folders for storing components that may be used in constructing other higher-level components

and/or applications. The component access permissions determine which components a particular user is able to see in the repository. Specific and generic components are indicated by different colours.

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

3. An internal scheduling system. In cases where an external scheduler is available the VCCE can delegate execution to that system, otherwise it is necessary for the VCCE to be able to execute the composed task graph itself.

In the PSE prototype, a user can visually construct scientific applications by connecting together components, which can range in granularity from simple matrix manipulation routines, to complete application programs. Each component has an XML interface described using a common data model described in further detail in section 4.4.2.

The main module of the VCCE is the Program Composition Tool (PCT). This is a visual tool that enables a user to build and edit applications by plugging together components, by inserting application components into predefined templates, or by replacing application components in existing programs. The PCT allows a user to connect two components only if their interfaces are compatible. The PCT also enables a user to create new application components and place them in the Application Component Repository (ACR). The Program Execution Tool (PET) develops the task graph generated for each application, encodes this into XML (described in section 4.4.4), and passes the graph to the internal or external scheduler for execution.

4.4 VCCE Components

Components in the prototype PSE have the following properties:

- Components have a unique name or ID. This does not have to be globally unique, only locally, as the notion of "name space" can be used together with the local unique name to create a globally unique name.
- Components may be Java 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 [19]. Legacy codes, in Fortran for instance, can be wrapped as components.
- 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.
- Each component is represented by a well-defined model specified in XML, as described in section 4.4.2.
- Information is passed from one component to another via unidirectional 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 inports 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 outports together define its output interface.
- A set of constraints may be associated with each component, indicating on what platforms it can be run, and whether it requires generic software, such as MPI or the BLAS, or specific software versions such as JDK version or CORBA ORB, in order to run.

- Information on a component's purpose, the algorithms it uses, and other pertinent explanatory data is optionally associated with a component in the form of help files.

4.4.1 XML within the VCCE

XML (eXtensible Markup Language) [113] is a subset of the document formatting language SGML (Standard General Markup Language). It was devised, among other things, for developing documents for the Web, and is acknowledged by the W3C standards organisation. 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) scheme. 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, Chemical Markup Language (CML) among others. The approach closest to the work described here is Open Software Description (OSD) [114], submitted to WC3 August 1997, 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.

XML was chosen as the language of choice for two main reasons.

- XML is computing language and platform agnostic. This is very important if a PSE is to be platform independent and components are to be language independent.
- There are a number of high quality open source XML language parsers for many programming languages. For example, Xerces [6] from the Jakarta-Apache group, and Java API libraries for manipulating and generating XML documents, such as JDOM [64] and JAXP [63].

These reasons suggest that XML is an 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.

4.4.2 XML Component Model

Each component used within the VCCE must currently, although not necessarily, be either a Java or CORBA object. Components are self-documenting, with their interfaces defined in XML, which: enables a user to search for components suitable to a particular application; enables a component to be configured when instantiated; enables each component to register with an event listener; and facilitates the sharing of components between repositories. XML tags may be used to automatically derive help on particular components already present in the repository, or they may be used to query the availability of particular types of components. User supplied components must also have their interfaces defined in XML.

The XML-based component model ensures uniformity across all components, and helps to abstract component structure and implementation from component interface. The XML definition used within the prototype, enables the division of a component interface into a set of sections, where each section is enclosed within predefined tags. A parser capable of understanding the structure of such a document can identify and match components which meet this interface. The DTD identifying

valid tags does not need to be placed with each interface, as it can be obtained from a URL reference placed in the document header, and identified by the `href` tag. The XML definition can be used to perform information integrity, such as the total number of inports and outports, check the suitability of a component, the types of platforms that may support the component and internal component structure when available.

The component model was first described in a paper [95] in 1999 and was influenced by IBM's BeanML [119] and W3C's OSD [114] XML-based frameworks. BeanML is a component configuration and connection language. Unlike the language presented here or OSD, it is designed for use exclusively with the JavaBean component model. The BeanML script is executed by an interpreter and provides functionality for: the creation of new beans; accessing of existing beans; configuration of beans by setting/getting their properties and/or fields; binding of events from some beans to other beans; and calling of arbitrary methods in beans. Although comprehensive BeanML is not suitable for the data flow model presented in this thesis.

OSD is another component language with a specific purpose, in this case it is for describing software packages and their dependencies for use in automated software distribution environments. As with BeanML, OSD is not suitable for data flow modelling. The OSD idea of dependencies was used in this work to model the parent child relationship between nodes in a task graph and some of the implementation specification such as operating system version were used in specifying the executable part of the component. The idea from BeanML of registering events from one bean with another is adapted here and used.

The tags in the component language are divided into the following functional sections:

- `context` and `header` details
- input and output `ports`
- `execution` specific detail, such as whether the component contains MPI code

- a user specified `help` file for the component
- a `configuration` file for initialising a component
- a `performance model` identifying costs of executing the component, for the resource manager
- an `event` handler, which enables registering or recording of particular types of events.

A component may also contain specialised constraint or semantic tags in addition to the mandatory requirements identified above. Constraints can include security or licence constraints, where a component is required to run on a particular machine or cluster.

Some of the tags specified in the following sections are based on either BeanML or OSD. Component naming is common to both schemes but the work in this thesis has a more detailed description in the context and header tags, section 4.4.2.1. Neither scheme has the notion of data flow so the inport and outport tag specification, section 4.4.2.2, is unique to this work, at least at this date. The execution tags, section 4.4.2.3 are loosely based on OSD's `IMPLEMENTATION` tag set. The configuration tags, section 4.4.2.5, and the help file tags, section 4.4.2.4, are similar in style to OSD's `CODEBASE` tag although they are used for different purposes. The performance model tags, section 4.4.2.6, do not occur in either scheme, but the event mechanism tag, section 4.4.2.7, is similar to BeanML's event mechanism for registering components or JavaBeans as listeners to another component's events.

4.4.2.1 Context and Header Tags

Context and header tags are used to identify a component and the types of PSEs that a component may be usefully employed in. The component name must be uniquely identifiable within the scope it is being used, a "name space" defining the local scope can be used where the component is not locally unique. A component also has an alternative alphanumeric identifier and an "ID" which can be used to

differentiate between instances of the same component. Any number of PSEs may be specified. These details are grouped under the preface tag. The `hierarchy` tag is used to identify parent and child components, and works in a similar way to the Java package definition. A component can have one parent, and multiple children.

In the code segment 4.1 taken from example B.1.3, appendix B, 'DA01' has no children, indicating that it is at the bottom of the hierarchy. The component name, alternative name and ID can be seen clearly together with the PSE type which is *Generic*.

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

Listing 4.1: Example Component Header

4.4.2.2 Port Tags

Ports tags are used to identify the number of input and output ports, and their types. An input port can accept multiple data types and this can be specified in a number of ways by the user. For example

```
<inport id="3" parameter="Lambda" type="float" value="0.5">
</inport>
```

Listing 4.2: Simple Port XML

Input and output to a component can also come from and go to other types of sources, such as files or network streams. In this case, the inport and outport ports need to define an `href` tag, rather than a specific data type. The `href` definition is standardised to account for various scenarios where it may be employed, such as:

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

```
        <href name="http://www.cs.cf.ac.uk/PSE/" value="test.txt"
            >
    </inport>
</ports>
```

Listing 4.3: Example HREF Port XML

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

```
<ports>
    <inport id="1" parameter="regression" type="stream" value="
        NIL">
        <parameter="regression" value="NIL"/>
        <href name="file:/home/pse/test.txt" value="NIL">
    </inport>
</ports>
```

Listing 4.4: Example File Port XML

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

4.4.2.3 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 be specified by the user also. Such component specific details may be enclosed in any number of `type` tags. The execution tag is divided into a `software` part and a `platform` part. The former is used to identify the internal properties of the component, while the latter is used to identify a suitable execution platform or a performance model.

For example

```
<execution id="software" type="bytecode" value="extended">
    <type id="architecture" value="serial"/>
    <type id="class" value="com.baesystems.components.
        SimplexComponent"/>
    <type id="source" value="file:///home/compdata/cardiff/
        project/src/com/baesystems/components/SimplexComponent.
        java"/>
    <type id="classpath" value="/home/compdata/project/classes"/>
</execution>
```

Listing 4.5: Example Execution Tag XML

4.4.2.4 Help Tags

A user can specify an external file containing help on a particular component. The `help` tags contains `context` options which enables the association of a particular file with a particular option, to enable display of a specified help file at particular points in application construction. The contexts are predefined, and all component interfaces must use these. Alternatively, the user may leave the `context` field empty, suggesting that the same file is used every time help is requested on a particular component. If no help file is specified, the XML definition of the component is used to display component properties to a user. Help files can be kept locally, or they may be cross references using a URL. One or more help files may be invoked within a particular `context`, some of which may be local.

```
<help context="apidoc">
    <href name="file://f:\\cardiff\\project\\docs\\be2ddocs\\
        index.html" value="NIL"/>
</help>
```

Listing 4.6: Example Help Tag XML

4.4.2.5 Configuration Tags

Configuration tags are similar to the `help` tag. A user can specify a `configuration` tag, which enables a component to load predefined values from a file, from a network

address or by using a customiser or wizard program. This enables a component to be configured within a given context, to perform a given action when a component is created or destroyed, for instance. The `configuration` tag is particularly useful when the same component needs to be used in different applications, enabling a user to share parts of a hierarchy, while defining local variations within a given context.

4.4.2.6 Performance Tags

Each component can have an associated performance model, and this can be specified in a file, using a similar approach to component configuration defined above. A performance model is enclosed in the `performance` tag, and may range from being a numerical cost of running the component on a given architecture, to being a parameterised model that can account for the range and types of data it deals with to more complex models that are specified analytically.

4.4.2.7 Event Model Tags

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

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

Listing 4.7: Example Event XML

The `target` identifies the component to initiate when an event of a given `type` occurs on component with identity `id`, as defined in the `preface` tag of a component. The `name` tag is used to differentiate different events of the same type, and the `filter` tag is a place-holder for Java *property change* and *vetoable property change* events

support. Also, the filter attribute is used to indicate a specific method in the listener interface using which the event must be received for a particular method to be invoked.

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

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

<event type="initialise" name="start" filter="">
    <script>
        <call-method target="DA01" name="bayesian">
    </script>
</event>
```

Listing 4.8: Example Event Listener XML

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

4.4.2.8 Computational Steering Tags

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. With a steerable component, the `inports` and `outports` of a component must have a specialised tag to identify this. For a conventional

component without steer-ability, the ports definition is:

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

Listing 4.9: Example Non-Steering Port XML

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>
```

Listing 4.10: Example Steering Component XML

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.

Additional tags not part of the component model may be specified by the user toward the end of each section.

```
<add> ... <\add>
```

Variable tags are not supported in the first version of the prototype.

4.4.2.9 A Data Analysis Component Example

A data analysis component within the repository may be described by the XML code B.1.3, page 166.

Another example component interface B.1.4 in XML can be seen in Appendix B, page 167.

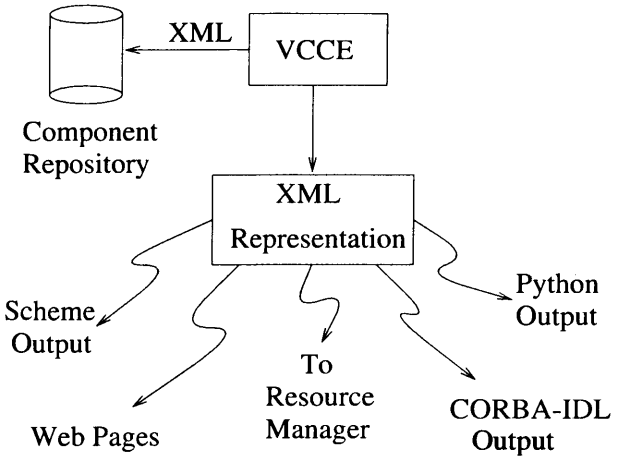

Figure 4.3: The XML-Based Component model.

4.4.3 Component Model Summary

All applications that employ the PSE prototype must adhere to the above component model. A user may specify the component model using tags, or may have it encoded using a component model editor, not implemented here, which also acts as a wizard and enables customisation. The editor would work in a similar manner to an HTML editor, where a user is presented with a menu-based choice of available tags, and can either choose one of these predefined tags or, different from an HTML editor, may define their own.

The Component Model in XML forms the interface between the VCCE and other parts of the PSE, and is used to store components in the repository. Various query approaches may be employed within the VCCE to obtain components with given characteristics or components at a particular hierarchy. The XML representation is therefore pervasive throughout the PSE, and links the VCCE to the IRMS or internal scheduler. Various representations could be obtained from the XML description in *Scheme*, *Python*, *Perl*, *CORBA-IDL* etc, for connection to other systems that may be attached to the PSE, see figure 4.3.

The use of tags enables component definitions to be exchanged as web documents, with the structure available at either a single or at particular certified sites. Hence, changes to the DTD can be made without requiring changes to component definitions

held by application developers, and will be propagated the next time a user utilises a component interface.

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.

4.4.4 XML Task Graph Model

As explained in section 4.1.3, the prototype PSE used in this work uses a data flow representation for the relationship between the processes in a constructed application. Connectivity between components in the VCCE, like the component interface definitions, is specified in XML in the form of a directed graph.

Component dependencies are enclosed in `connection` tags and consist of the constructs, `parent` and `child`. Distant relationships can be constructed from the recursive applications of these two basic types. Each connection between a pair of components is treated separately, however a single component can participate in more that one connection relationship allowing all pair wise combinations of: *One to One*; *One to Many*; and *Many to Many* relationships to be represented.

The input and output ports participating in the connection between two components are not specified explicitly in the XML representation. The Java implementation of the VCCE software attempts to match input port to output port by port *parameter* name in the first instance. If no match is found then the order of the ports in the component definition is used, i.e. the first connection to or from a component is connected to the first defined port. On reflection, this implicit rule is limiting to the taskgraph representation. It can cause a dependency on the order in which components are connected by the user in the graphical environment. In the case where input and output port names from the two participating components do not match, the order of their connection decides the particular ports to be connected. A solution to this problem is to include a full description of the participating ports

as part of the connection description. This is the case in more recent task graph representation languages such as that used by Triana, based on the work here.

In addition to the `connection` relationships, a representation of the start point or points for a task graph is needed. For this a `start` tag is used that labels one or more components as starting points for execution. While it is possible to simply examine all components in the task graph to find those with no parent connection, giving the start points, it makes the scheduling algorithms simpler and quicker to know in advance the start points specified by the user.

4.4.4.1 Example XML Task Graph

In the example XML task graph in B.1.5, page 168, there are four components: *be2dData*; *be2d*; *curView*; and *rcsView*. The start point is labelled in the graph as being *be2dData*, the connections are *be2dData* to *be2d*, *be2d* to *curView* and *be2d* to *rcsView*. The start point and one connection from the example can be seen below.

```
<start>
    <name alt="be2dData" id="be2dData01" inst="7065019">be2d data
       </name>
</start>
<connection>
    <parent>
        <name alt="be2dData" id="be2dData01" inst="7065019">be2d
           data</name>
    </parent>
    <child>
        <name alt="be2d" id="be2d01" inst="6670326">be2d</name>
    </child>
```

Listing 4.11: Example Task Graph Connection

So in this example there are examples of "One to One" and "One to Many" connections. This example is from a real application and is explained in context in chapter 5.

4.4.5 Compound Components

One useful mechanism specified in the requirements document for the VCCE is the ability for a user to create composite components from a number of existing components. These composite or compound components once created behave exactly like a simple component, in that it has a number of input and output ports and can be connected into a task graph like any other component.

The compound component is really a construct for the user of the system as opposed to something the system needs itself. It enables a user to logically group components into larger more complicated systems for re-use and sharing with other users, or to enable very large complicated systems to be viewed at a coarser level so that it can be seen in its entirety more easily. The grouping of components will not affect the execution of the completed application as the atomic components have not actually been changed in any way.

In terms of the XML representation of a compound component, it can be seen as a partial task graph. A compound component consists of a list of constituent components and a set of connection relationships, the same as a task graph. But, in addition the compound component will have the `header`, `port` and optionally `help` tag sections that a normal component would have. The `header` and `help` tags are specified by the user building the compound component. The `port` tags representing the input and output ports map to the corresponding ports on the outermost components in the group. For example, given a simple group with two components then the inputs to the group would be the input ports on the inner component with no parent connection and the output ports to the group would be the output ports on the inner component with no children. So given a compound component made up of a number of sub components, the number and type of the input ports would be the sum of all the input ports of all components which are not connected internally; the number and type of the output ports would be the sum of all output ports of all components which are not connected internally.

When a connection is made in a task graph between a component and a com-

pound component, the actual connection is mapped through to the corresponding internal component.

4.5 VCCE Component Implementation

Components in the prototype PSE do not perform any computation in their own right, they are merely representations or proxy components, for an underlying computational code, see page 26. A VCCE component provides a number of services for its computational code:

- A visual representation of the code within a GUI environment.
- An optional custom user interface for the code.
- Input ports that direct incoming input data in a form that the code can use and provide a visual representation of the port.
- Configuration for the code.
- A mechanism to execute the code.
- Output ports that take the code output and make it available in a form that can be passed to the next component in the application.

The VCCE component design is based on the ideas developed in the previous two prototypes examined in sections 4.2.1 and 4.2.2. Each component is defined by an XML definition file which contains the various values for name, executable, help files etc, defined in section 4.4.2. When the VCCE is started, each component in the repository is loaded from its XML definition and a proxy created.

4.5.1 Proxy Interface

Using sound software engineering principles everything is written to well defined interface classes, hence every proxy component implements a single main interface,

`ProxyInterface`. The interface is a full representation in Java of all of the values obtained from the XML definition. The full listing can be seen in appendix B.1.6. For example, the component `header` tag values from section 4.4.2.1 are represented by the methods,

```
// Returns the internal system name.
public String getInternalName();

// Returns the alternative name.
public String getAltName();

// Returns the external name representation.
public String getExternalName();

// Returns the instance ID (unique)
public String getInstanceID();

// Sets the instance ID (should be machine generated and unique).
public void setInstanceID(String InstIDStr);

// Returns the type of PSE the component can be used within.
public PseType getPseType();

// Returns the parent component, in a compound component.
public String getParent();

// Returns an iterator containing the child components in a
   compound component.
public ProxyIterator getChildren();
```

As can be seen above, there is a corresponding *getter* method for each value in the tag set. The responsibility for setting those values from the XML definition is delegated to the implementing classes, in this case `AbstractProxy`, an abstract class that contains a number of common *utility* methods and `SimpleProxy`, the actual proxy class.

There is an interface to represent input and output *port* objects, `PortInterface`. This has methods to *set* and *get*: the value of the data object coming in or going out of the proxy component; the name of the port; an optional URL `href`; and the *type* of the port which could be one of the following values - `Stream`, `File`, `String`, `Short`, `Float`, `Double` or `Object`. In the `ProxyInterface` there are methods, `getInports()`

and `getOutports()` which return sets of these `PortInterfaces`.

The visual representation of the proxy component is handled by another Java class called `SimpleVisualComponent`. In its simplest form this just displays a box representing the component with the component name displayed as text. Each visual component is constructed with a `ProxyInterface` so that there is a *one-to-one* mapping from a proxy to its graphical representation. It is the `SimpleVisualComponent`, a subclass of a Java `JPanel`, that gets displayed in the VCCE user interface explained in section 4.6.

4.5.2 Execution Interface

The proxy design up to this point has been relatively straightforward but in this section is a description of the part of the proxy that is responsible for the actual executable computation code, the `ExecutionInterface`. The underlying computational unit could take many forms from a simple Java code to a parallel Fortran code, as described in section 4.4. The execution proxy needs to be flexible and extensible to take this fact into account. As before a general interface is developed first and then code written to implement that. The `ExecutionInterface` has two important methods.

```
// Perform any necessary instantiation functions when a
// component is added to the container prior to execution.
public void instantiate();

// Execute the underlying subject process, and notify any
// listeners on completion.
public void execute();
```

All implementing *Execution* classes must included these methods. The first is a "set-up" method which is called by the VCCE when a proxy component is instantiated by the user. Implementing classes should use this method to perform pre-execution tasks. This might include contacting a *Name Server* in the case of a CORBA component implementation. The second method implementations perform the actual task of executing the underlying computational code.

In the simplest implementation of the `ExecutionInterface`, where the computational code is written in Java or is a simple command line executable, there are two implementing classes, the `JavaExecution` or `ExtendedJavaExecution` proxy. Later chapters in this dissertation examine more complicated implementations. These proxies are very similar, they can both be used to run Java-based components within the VCCE.

4.5.2.1 Java Execution Component

The first implementation, `JavaExecution`, executes a command line instruction to run the computational code. The `instantiate()` method in this simple case is a "no-op" empty method, there is no set up functionality to perform. The class has variables to represent the *classpath*, the *classname* and *argument* for the computational code. It makes use of the Java *Runtime.exec* functionality, line 10 below, to execute a command string as if it were a command line from a user.

```
public void execute() {
2     StringBuffer executeStrBfr = new StringBuffer("java␣-
         classpath␣");
      executeStrBfr.append(getClassPath());
4     executeStrBfr.append("␣");
      executeStrBfr.append(getClassName());
6     executeStrBfr.append("␣");
      executeStrBfr.append(getArguement());
8     try {
          System.out.println(executeStrBfr.toString());
10        Process process = Runtime.getRuntime().exec(executeStrBfr
             .toString());
          // once execution is finished publish the fact
12        try {
              process.waitFor();
14        } catch (InterruptedException e) {
              System.err.println("be2d␣execution␣failure");
16            e.printStackTrace(System.out);
          }
18        notifyExecution();
      } catch (IOException e) {
20        e.printStackTrace(System.out);
      }
22 }
```

Listing 4.12: Execution Method for `JavaExecution` Component

This simple case has no facility for handling input and output directly, this must be handled in other ways. For example in the case of large number of legacy codes their input and output is file-based and has to exist in specific places. Handling these files would be the job of other assistant components.

4.5.2.2 Extended Java Execution Component

The second implementation of the `ExecutionInterface`, `ExtendedJavaExecution`, instantiates and runs the Java computational code in the same Java Virtual Machine as the running VCCE. This implementation extends the simple `JavaExecution` case from the previous section and is more fully featured and also more complicated.

The `instantiate()` method implementation, called when the user instantiates the proxy component onto the VCCE GUI, checks the system classpath and appends the proxy classpath if necessary. It also attempts to instantiate an executable object from the class.

```
Class class_ = Class.forName(getClassName());
Object runnableApp_ = class_.newInstance();
```

If this fails then the VCCE is able to advise the user that the component cannot be instantiated. It is more useful for this failure to be apparent at the point of assembling the application than at runtime where nothing can be done.

The `execute()` method implementation is also more complicated as it performs three functions:

1. set input values for the computational code object from the values in the proxy
2. execute the instantiated code object
3. set the output values on the proxy from the values in the computational code object.

The proxy knows nothing about the computational code object apart from the details contained in the XML definition which have been used to generate the proxy component.

The main information about the classname and classpath which will be used to instantiate and execute the object is inherited from `JavaExecution`, section 4.5.2.1, and is set at proxy creation. The proxy has a set of input and output `PortInterface` objects that represent the input and output ports for the proxy and hence the computational code itself. Each of these `PortInterface` objects has a name, value and value data type. The *JavaBean* technique of using a strict naming convention for parameter names and the methods in the containing computational object that *set* and *get* the values of those parameters is used. The method names are the parameter names prefixed by either `set` or `get`. For example if in the XML input tag section for a given Java computational component there is the parameter definition, below

```
<inport id="3" parameter="Lambda" type="float" value="0.5">
</inport>
```

then in the computational class there must be corresponding methods to set and return the values of that parameter.

```
public void setLambda(float value);
public float getLambda();
```

Java reflection[5] is used to dynamically discover these methods on an object and call the set parameter methods on the computational object before execution, and the get parameter methods after execution. For example, in the code listing 4.13 below on line 6 there is an `Iterator` for the set of input ports, on line 8 the method name for the *set* method is constructed, line 10 uses reflection to return the `Method` object for the given name and parameter method *signature* and finally line 12 invokes that method with the input port value from the proxy on the instantiated computational object to set the input value.

The main method on the computational object that performs the work of the code is also discovered and called using reflection, lines 19 and 20. Finally in a reverse of the manner in which the input values are set on the code object, the return values after execution are passed onto the proxy object in the listing lines 25 to 30.

```
public void execute() {
    // set all inports in the instantiated object using method
    // invocation and reflection
    Class[] parameterType = new Class[]{Object.class};
    Object[] parameterValue = new Object[1];
    for (PortIterator it = getOwner().getInports(); it.hasNext()
        ;) {
        PortInterface port = it.next();
        String methodName = "set" + port.getParameter();
        try {
            Method setter = class_.getDeclaredMethod(methodName,
                parameterType);
            parameterValue[0] = port.getValue();
            setter.invoke(runnableApp_, parameterValue);
```

[5]A mechanism for dynamically discovering and executing an objects methods at runtime

```
        } catch (Exception e) {
14              e.printStackTrace(System.out);
        }
16  }
    // Invoke the execute method
18  try {
        Method execute = class_.getDeclaredMethod("execute", new
            Class[0]);
20      execute.invoke(runnableApp_, new Object[0]);
    } catch (Exception e) {
22      e.printStackTrace(System.out);
    }
24  // update all the outports
    for (PortIterator it = getOwner().getOutports(); it.hasNext()
        ;) {
26      PortInterface port = it.next();
        String methodName = "get" + port.getParameter();
28      try {
            Method getter = class_.getDeclaredMethod(methodName,
                new Class[0]);
30          port.setValue(getter.invoke(runnableApp_, new Object
                [0]));
        } catch (Exception e) {
32          e.printStackTrace(System.out);
        }
34  }
    notifyExecution();
36 }
```

Listing 4.13: Execution Method for `ExtendedJavaExecution` Component

This more advanced implementation of the `ExecutionInterface` can handle parameter values coming into and going out of the proxy object. The proxy is responsible for the interaction between components and passes information and control to the underlying computational code. In the next section, the VCCE framework is discussed in more detail and working examples of proxies can be seen in the following "use case" chapters, 5, 6, 7 and 8.

4.6 VCCE Implementation

The implementation of the third prototype, the Visual Component Composition Environment (VCCE) is discussed here. This prototype is a fully functional visual

programming environment that is used in the demonstration chapters to examine the behaviour of the PSE with regard to specific user cases. A full description of the implementation would be out of scope for this thesis. Instead this section focuses on certain aspects of the implementation, ignoring less interesting facets. Included here are the mechanisms by which proxy components, described in section 4.5, are created from their XML definitions, how connections between components are created to build an application and how that application is scheduled and executed by the local execution system. This section does not include a large amount of detail with regard to the VCCE GUI as this is really Java *Swing*[6] programming and not of particular interest here. The user interfaces discussed in the previous prototypes in sections 4.2.1 and 4.2.2 cover most of the concepts used here.

4.6.1 Component Factories and XML Parsing

Section 4.5 described the structure and implementation of the proxy components that provide a placeholder for the real underlying computational component within the VCCE framework. It described the fact that the proxy is created from an XML component definition. That process is described here.

The construction of similar objects with a common interface, i.e. the `ProxyInterface`, but different implementations, i.e. `JavaExecution` versus `ExtendedJavaExecution`, is a commonly found software engineering problem and is covered by the *Factory Method* design pattern, see page 26. A *factory* class called `ComponentFactory` is used, to which is delegated the parsing of the component XML definition files and creation of various objects of type `ProxyInterface`. "Polymorphism" is used to handle the different implementations of the proxy components in a common manner.

The `ComponentFactory` class is an object factory responsible for building the proxy components based on XML component definition files. These files are found

[6] Java Swing is the Java platform independent graphical component library used to build Java GUIs.

and loaded by the VCCE in a special component directory denoted by the `XML_DIR` property. This property comes from the `Settings` class, a *singleton* class, see page 26, that loads a user defined properties file. For example:

```
String compDir = Settings.getInstance().getProperty("WORKING_DIR"
    )
compDir += Settings.getInstance().getProperty("XML_DIR");
File componentDirectory_ = new File(compDir);
```

The `ComponentFactory` makes use of the open source JDOM API[7] which provides a much more "Java-centric" view of XML documents than the W3C Document Object Model (DOM). It provides, among other things, list-based access to nodes in a parsed XML tree. JDOM can use a number of XML parsers but the one used here is the open source Xerces[8] Java parser, part of the Apache XML Project.

An example of `ComponentFactory` use to return a set of components parsed from the XML files contained in the "component" directory can be seen in the code below.

```
Vector components_ = new Vector();
factory_ = new ComponentFactory();
while (!factory_.allComponentsLoaded()) {
    components_.add(factory_.getNextComponent());
}
```

Each XML definition complies with the standard document format, section 4.4.2, the `ComponentFactory` uses the XML parser and the JDOM API to retrieve values for parameters from the document and sets them in an internally created proxy object. Currently the factory handles one type of proxy, `SimpleProxy`, with multiple execution models. If the proxy model were to be extended, then the beauty of a factory model solution is that there is no need to change any code in the application apart from inside the factory itself.

The *preface*, *port* and *help* tag sections have a similar structure for all components. Preface values are assigned as direct attributes of the proxy object, port and

[7] `http://www.jdom.org`

[8] `http://xml.apache.org/xerces-j/index.html`

help values are represented by `AbstractPort` implementations `Inport`, `Outport` and `SimpleHelp` objects respectively. These objects are simply Java class representations of the XML component data together with "setter", "getter" and utility methods. The objects are created with the values from the XML interface and added as attributes or "sets" of attributes to the proxy object.

The final section of tags represent the `execution` definition. Depending on the values of these tags the `ComponentFactory` will create objects from different implementations of the `ExecutionInterface`. As more execution models are added to the system, this section of the factory object can be extended to include these new implementations. For instance in chapter 7 a new execution model that uses CORBA is added, the `ExecutionInterface` implementation for this model is in a class called `ActionFactoryExecution`. The `ComponentFactory` currently supports three execution `type` tag values.

```
<execution id="software" type="bytecode">
```

which causes the `JavaExecution` class to be used

```
<execution id="software" type="bytecode" value="extended">
```

which specifies `ExtendedJavaExecution` to be used and

```
<execution id="software" type="corba">
```

which tells the factory that the execution model should be an instance of `Action-FactoryExecution`.

4.6.2 Event Model

The execution of multiple components in the VCCE relies upon the framework knowing when the execution of any given component has completed. The framework starts components in a specific order but doesn't monitor their progress directly. Each component makes use of the *Observer* design pattern, page 27, to notify all "interested" parties once its execution is complete.

The main interested parties are:

- The scheduling system, so that it can complete dependencies in the task graph and execute the next component.
- The `SimpleVisualComponent` which provides the GUI representation of a proxy object, so that it can provide user feedback about the state of execution. For this prototype visual components turn from grey to red on completion of execution.

The VCCE project source contains a package called `vcce.event` which contains two interfaces and a class which provide the functionality for the event mechanism. The `ExecutionPublisherInterface` is the *producer* interface that all classes which wish to publish execution events must implement. It contains three methods to: add listeners; remove listeners; and notify listeners of events.

```
public interface ExecutionPublisherInterface {
    // Add an execution listener to the implementing object.
    public void addExecutionListener(ExecutionListenerInterface
        listener);

    // Remove an execution listener from the implementing object.
    public void removeExecutionListener(
        ExecutionListenerInterface listener);

    // Notify all interested listeners of an execution event.
    public void notifyExecution();
}
```

Listing 4.14: Execution Event Publishing Interface

The `ExecutionListenerInterface` is implemented by all classes that wish to register themselves to receive execution events, it contains a single method called by the event publisher.

```
public interface ExecutionListenerInterface extends EventListener
    {
    // Method called by the execution event publisher to notify
       the implementing object of an execution event action.
    public void executionPerformed(ExecutionEvent e);
}
```

Listing 4.15: Execution Event Listener Interface

The final class in the package is the `ExecutionEvent` class itself. This is a simple object that contains a reference to the sending source proxy from which the event came.

```
public class ExecutionEvent extends EventObject {
    // Constructor that takes the source of the event as its only
        parameter.
    public ExecutionEvent(Object source) {
        super(source);
    }
}
```

Listing 4.16: Execution Event Class

Both the listener interface and the event class inherit from the Java event mechanism classes.

Since the event mechanism is designed to report the completion of component execution, the implementation of the publishing interface is in the execution component. To be exact, the implementation is in an abstract class called `AbstractExecution` which implements `ExecutionInterface` which in turn extends `ExecutionPublisherInterface`. This abstract class implements some of the methods common to all execution components, including the event publishing mechanism. The class has, as an attribute, a list of all registered listeners. It implements: the `addExecutionListener` method by adding the listener to the list; the `removeExecutionListener` method by removing the listener from the list; and the `notifyExecution` method by calling the `executionPerformed` method on all the listeners in the list.

Each concrete implementation of the `ExecutionInterface` must call the inherited `notifyExecution` method from the abstract super class at the end of the `execute` method to "fire" the event mechanism. See the `JavaExecution` listing line 18 on page 58 or `ExtendedJavaExecution` listing line 35 on page 61.

The implementation of the listener interface is in the two places specified at

the beginning of this section. The internal scheduler implementation which needs to be notified on component execution is discussed in the next section, 4.6.3. The `SimpleVisualComponent` implementation of the `executionPerformed` method sets the background of the visual component to red to signify to the user that execution on that component has completed.

4.6.3 Internal Task Graph Scheduling and Execution

Task graph scheduling within the VCCE revolves around Java implementations of the XML task graph from section 4.4.4. As with the component model implementation, an effort has been made to keep the design both flexible and extensible. To that end the implementation has an *abstract* base class with the common functionality and concrete implementations that can use different algorithms to execute the connected components.

The representation of a task graph is encoded in the abstract base class called `AbstractExecutionGraph`. This class consists of a collection of networked "nodes" represented by the class `ExecutionGraphNode`. The task graph class has methods common to all task graph implementations such as: *add* nodes, *remove* nodes, *connect* nodes; and an abstract method, *execute* which starts the execution of the task graph. The implementation of the *execute* method is algorithm specific and is contained in concrete classes. The class also contains some utility methods to return a node by its content and to return typed node iterators that are used to traverse the nodes in the graph.

`ExecutionGraphNode` is a "wrapper" class that represents a proxy component, see section 4.5.1. It provides the functionality for the *parent/child* relationships between components in a task graph. Each `ExecutionGraphNode` node maintains a list of parent nodes and child nodes. Nodes are added when connections are made between nodes in the encompassing task graph. This is shown in the method `connectNodes` in `AbstractExecutionGraph` which is called when a connection is made between two components on the user interface.

```
public void connectNodes(ExecutionGraphNode parent,
    ExecutionGraphNode child) {
2     parent.addChild(child);
      child.addParent(parent);
4 }
```

As can be seen above, the *parent* node is added as a parent to the *child* and vice versa. In the `ExecutionGraphNode` implementation of these methods the nodes are added to the respective lists. The class also contains "delegation" methods that pass the method call straight through to the contained proxy component. These include methods to handle *ports* and *execution*, and utility methods that are used to check for the existence of *descendants* in the child list. The existence check is recursive, calling the check method in all children. This check is used in execution algorithms to check for the presence of loops in a graph.

4.6.3.1 Execution Graph: A Simple Scheduling Algorithm

A concrete implementation of the `AbstractExecutionGraph` is `ExecutionGraph`. This class contains a scheduling algorithm that takes a completed task graph and executes it. The algorithm can be seen in algorithm 4.1.

This algorithm uses a naive approach to scheduling the components for execution by analysing the connections between them and producing a linear execution order. This ordered list of components is then executed with the next node in the list being performed on notification of the previous node's completion. This first scheduling algorithm makes certain assumptions: a single start point in the task graph; no cyclic relationships; no parallelism in the execution. Although limited, this algorithm works for simple task graph cases and is enough to test the functionality of the VCCE. Later chapters will expand this area and add new algorithms, but because of the design based on abstract classes and concrete implementations, new implementations of algorithms can be added to the framework without changing any other code.

The full implementation of the algorithm in the class `ExecutionGraph` can be

start ⇐ starting node for *taskgraph*
nodecount ⇐ number of nodes in *taskgraph*
children ⇐ children of *start* node
for $i = 0$ to number of elements in *children* -1 **do**
  add *children* element at i to *executionlist*
end for
$i \Leftarrow 0$
current ⇐ element i in *executionlist*
while number of nodes in *executionlist* < *nodecount* **do**
  children ⇐ children of *current*
  for $j = 0$ to number elements in *children* **do**
    child ⇐ *children* element at j
    if *child* is not in *executionlist* **then**
      add *child* to *executionlist*
    end if
    $i \Leftarrow i + 1$
    current ⇐ element i in *executionlist*
  end for
end while
for $i = 0$ to number of elements in *executionlist* **do**
  execute element i in *executionlist*
end for

Algorithm 4.1: Simple Scheduling Algorithm

seen in appendix B.1.7. Once the execution order of the components in the graph has been decided using the `deriveOrderOfExecution` method the components in the list are executed in turn by calling the `execute` method on each proxy. This execution algorithm uses the event mechanism, section 4.6.2, to trigger the execution of the next component in the list. In the abstract super class, `AbstractExecutionGraph`, the implementation for the `addNode` method adds a node to the task graph and registers the task graph class as a listener to the proxy component. See listing 4.17.

```
public ExecutionGraphNode addNode(ProxyInterface node) {
    ExecutionGraphNode graphNode = null;
    graphNode = new ExecutionGraphNode(node);
    node.getExecution().addExecutionListener(this);
    nodes_.add(graphNode);
    return graphNode;
}
```

Listing 4.17: `addNode` Method from `AbstractExecutionGraph`

When a component has executed it calls the `notifyExecution` method which in turn calls the `executionPerformed` method on the object that is registered as a listener. The implementation of this method is *abstract* in the `AbstractExecutionGraph` class with the implementation delegated to concrete subclasses. In `ExecutionGraph` the method is used is used to trigger the execution of the next component in the ordered list. See listing 4.18.

```
public void executionPerformed(ExecutionEvent evt) {
    executionIndex_++;
    if (hasMoreToExecute()) {
        TransferPortData();
        executeNext();
    }
}
```

Listing 4.18: `executionPerformed` Method Implementation in `ExecutionGraph`

4.7 Summary

This chapter has outlined some of the important design and implementation ideas for the prototype PSE used throughout this thesis. It has explained the component model from the XML interface definition to the proxy component and execution model. Also shown was the mechanism by which components are loaded into the VCCE and how instantiated components are scheduled and executed using the internal scheduling mechanism with a simple scheduling example.

Two simple initial prototypes were discussed. The early prototypes were used to examine programming techniques and design options. Some ideas from the early prototypes have been carried through to the final design but prototype code, following sound software engineering principles, was discarded.

The remainder of the thesis is concerned with the use of a PSE as opposed to the design. Further design and architecture ideas are discussed in the post-related work and conclusion chapters (chapters 9 & 10) together with some implementations that have come about since the end of this work. The next chapter, takes the prototype

VCCE and starts to use it with a simple "use case".

CHAPTER 5

Simple Solver Component

This chapter covers the first use case for the prototype PSE, a simple solver component. As an initial demonstrator for the prototype VCCE, BAE SYSTEMS provided a relatively small two dimensional boundary element solver written in Fortran 77. Because of the relatively straightforward nature of this code it was possible to develop different versions of components that could provide the functionality of the code within the prototype PSE. Using different granularities of wrapping techniques and language conversion, enabled experimentation and provided comparisons between the native Fortran version and various other implementations. It would not be practical to perform experiments of this nature with the very much larger codes that are covered in chapter 7.

5.1 BE2D Solver

The boundary element code described in this chapter is called *BE2D*. The code is a two dimensional boundary element simulation code for the analysis of electromagnetic 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 and phase. For the computation of the matrix elements, the code uses a two dimensional formulation of *Rau-Wilton-Glisson* elements [96]. The outer integrations use one-point quadrature, while the inner integrations use two-point quadrature. A direct LU decomposition[1] solver is used for computing the field.

The output, once the code has executed, takes the form of two files, one that represents a Radar Cross Section (*rcs*) and the other a Current (*cur*). Generally in order for an engineer or scientist to be able to use these files they must be viewed in a graphing tool such as GNUplot [120] or similar.

To make the code easier to use within the PSE and also to provide some run-time speed experiments between component granularities and versions, the code was converted using several different techniques outlined in the remainder of the chapter.

To use the BE2D code within the VCCE would either require wrapping both the BE2D code itself and the simple mesh generation program as components, or assumptions need to be made about the presence of the required input files. Where these assumptions have been made they are indicated.

5.1.1 Simple Wrapped Fortran Version

The simplest way to execute Fortran or any other native code from within a Java program such as the VCCE is to use Java's built in *Runtime Execution* mechanism.

The runtime execution mechanism allows a Java program to run an executable program as if that program were executing from the command line, calling the

[1]LU decomposition (n.) a technique where a matrix A is represented as the product of a lower triangular matrix, L, and an upper triangular matrix U. This decomposition can be made unique either by stipulating that the diagonal elements of L be unity, or that the diagonal elements of L and U be correspondingly identical.[53]

executable with any required input arguments as parameters. The major advantage of this over other forms of wrapping techniques, is that the source code is not needed, the binary executable program. The runtime execution mechanism also gives the programmer the ability to trap the command line standard output and standard error message displays so that the calling program can display information to the user or recover from errors in the native code execution.

The execution of the *BE2D* code is very simple as it takes no parameters it looks for the model data file called "model" in the running directory. For example:

```
Process myProcess = Runtime.getRuntime().exec("be2d");
```

The `Process` object can be used in Java: to block the Java program execution until the native code has finished; to trap "standard out" and "standard error" messages from the operating system. In most programs, including these components, this function call is surrounded by code that blocks the Java program and traps the error and output messages and does something sensible with them. For example in the simple version:

```
Process process = Runtime.getRuntime().exec(execStr);

// Process output
BufferedReader stdout = new BufferedReader(new InputStreamReader(
    process.getInputStream()));
// Do something with the standard output
stdout.close();

// Process Errors
BufferedReader error = new BufferedReader(new InputStreamReader(
    process.getErrorStream()));
// Do something with the standard errors
error.close();

// Wait for the native process to finish
try {
    process.waitFor();
}
catch (InterruptedException e) {
    e.printStackTrace(System.out);
}
```

Using this simple direct execution mechanism both the *BE2D* solver binary and the corresponding mesh generation program are wrapped as separate components in the VCCE.

The execution of the binaries is handled by the component `CommandLineExecution`, listed in Appendix B, page 174. This component is a generalised component that takes a system specific command line string including optional parameters and executes it using the code in the example above. To create a specific instance of the `CommandLineExecution` component, that will run the *BE2D* solver and the mesh generator codes, a component definition is created in the XML-based language, as specified in section 4.4.2. The component definition can be seen in listing 5.1.

```
<?xml version="1.0"?>
<PSE>
    <preface>
        <name alt="nativeBe2D" id="be2d01">Native Be2D</name>
        <pse-type>Generic</pse-type>
        <hierarchy id="parent"></hierarchy>
        <hierarchy id="child"></hierarchy>
    </preface>
    <ports>
    <inportnum>1</inportnum>
    <outportnum>2</outportnum>
        <inport id="1" parameter="Command" type="string" value="/
            home/project/fortran/be2d/ellipse/be2d">
        </inport>
        <outport id="1" parameter="DataFile" type="file" value="/
            home/project/fortran/be2d/data/rcs">
        </outport>
        <outport id="2" parameter="DataFile" type="file" value="/
            home/project/fortran/be2d/data/cur">
        </outport>
    </ports>
    <execution id="software" type="bytecode" value="extended">
        <type id="architecture" value="serial"/>
        <type id="class" value="com.baesystems.components.
            CommandLineExecution"/>
        <type id="source" value="file:///home/compdata/cardiff/
            project/src/com/baesystems/components/
            CommandLineExecution.java"/>
        <type id="classpath" value="/home/compdata/project/
            classes"/>
    </execution>
    <execution id="platform">
```

```
26          <type id="java" value="jdk1.2"/>
        </execution>
28      <help context="apidoc">
            <href name="file:///home/scmmss/project/docs/be2ddocs/
               index.html"
30              value="NIL"/>
        </help>
32  </PSE>
```

Listing 5.1: The XML Component Definition for the *BE2D* Solver

The key sections of the component definition are:

- The command line string which specifies the path to the executable that the component will call. This is passed as a named input parameter to the `CommandLineExecution` component.

```
<inport id="1" parameter="Command" type="string" value="/
   home/project/fortran/be2d/ellipse/be2d">
2  </inport>
```

- The output data files specified as file paths. These are defined as named output parameters.

```
<outport id="1" parameter="DataFile" type="file" value="/
   home/project/fortran/be2d/data/rcs">
2  </outport>
<outport id="2" parameter="DataFile" type="file" value="/
   home/project/fortran/be2d/data/cur">
4  </outport>
```

- The class name of the component that the proxy will execute, together with some execution details. It can be seen from line 1 that the execution model is *software* with type *bytecode* and value *extended*. From section 4.6.1, this causes the `ComponentFactory` to instantiate the `ExtendedJavaExecution` model, section 4.5.2.2, which will execute the component in the same JVM and handle the input and output parameters.

```
<execution id="software" type="bytecode" value="extended">
2       <type id="architecture" value="serial"/>
```

```
    <type id="class" value="com.baesystems.components.
        CommandLineExecution"/>
</execution>
```

With codes wrapped as components in this fashion there is no configuration from within the VCCE. All the variables are set in the XML component files. The use of all of the components in this chapter is explained in section 5.3. In the next section a different version of the *BE2D* code converted from Fortran into Java is considered.

5.1.2 A Java Version

With any component-based system, whether computer-based or not, it is necessary for the components and the framework to have a common language. Where there are different languages, as for example with the *BE2D* code written in Fortran and the VCCE framework written in Java, mechanisms such as the simple wrapper in the previous section can be used to provide a common point of reference.

For new components it is generally advisable to write the component in the same language as the framework unless there is a good reason not to, i.e. speed of execution. For small or well understood components it is often quicker and easier to rewrite the algorithm in the language of the framework. The *BE2D* solver code, although not very small, was small enough that it would be worth converting for comparison cases, where larger codes would require too much effort.

To convert the Fortran code into Java, the use of automated language converters such as Fortran to Java converters were considered but abandoned fairly quickly at the time because of a number of factors. These included the fact that most of the converters had problems with the lack of support for complex numbers which are a major part of *BE2D* and a large number of scientific codes. Complex numbers caused a number of problems in converting the code, not least because of the fact that they are primitive types in Fortran, but are not even supported in the current standard Java API. A third party implementation of complex numbers for Java was found and used, Visual Numerics JNL [112].

Complex number arithmetic proved to be more than a straight translation from the Fortran code to its Java equivalent. As complex numbers in Java are objects and Java does not yet support operator overloading, the complex number objects have to use their operator methods, *add*, *multiply*, *subtract* and *divide*. These methods are either *unary* or *binary*, taking a single argument which is used to modify the object calling the method or taking two arguments on a static class method that creates a new result without any modification side effects. For example the binary divide method:

```
Complex a = new Complex(0.0, 1.0);
Complex b = new Complex(0.0, 2.0);
Complex result = Complex.divide(a, b);
```

returns a new `Complex` object `result`, as opposed to the unary divide:

```
Complex a = new Complex(0.0, 1.0);
Complex b = new Complex(0.0, 2.0);
Complex result = a.divide(a,b);
```

where the method modifies the object `a` with the result of the division and assigns `a` to `result`. In other words in the first case there are three complex number objects the two original numbers and a new one which contains the result of the calculation. In the second two complex number values result from the calculation, discarding the original value of `a`. Thus calculations involving more than an arbitrary number of parameters become far more complicated in Java than their Fortran equivalents.

For example the Fortran code snippet

```
integer iap, iam, ian
complex caa, cax, cay
dimension caa(iam,iam), cax(iam), cay(iam), iap(iam)
integer i
complex clsum
clsum = czero
cax(i) = (cay(iap(i)) - clsum) / caa(iap(i),i)
```

becomes the Java code

```
Complex[][] caa, Complex[] cax, Complex[] cay;
int[] iap, int iam, int ian)
Complex clsum;
clsum = new Complex(czero);
cax[i] = Complex.divide(Complex.subtract(cay[iap[i]], clsum), caa
    [i][iap[i]]);
```

As can be seen, the Fortran code is much easier for an engineer to understand when compared to the much more obscure, for this type of programming, language of the Java code snippet. The Fortran version of the calculation uses standard mathematical notation and can chain arbitrary numbers of operator/operand pairs together. The Java version is limited by the fact that the operator methods operate on pairs of operands only, chaining of arbitrary length calculations involves bracketing each of the methods separately.

The translation of the Fortran code to Java, although time consuming, followed a pattern. The source for the Fortran code was broken into a number of functional file sections that were mirrored in an Object Oriented fashion in the Java objects.

The Fortran code consists of the following source files:

be2d.f: The main control program.

bessy0.f, bessj0.f: The respective routines for the y0 Bessel and the j0 Bessel functions used in the program. These were taken from Numerical Recipes [92].

inputm.f, inputc.f: The routines for reading the model and control data files.

outcur.f, outrcs.f: The output routines for the *rcs* and *cur* result data.

fill.f: Routine to populate the matrix from the input data.

lu.f: The matrix decomposition and solve routines.

dutil.f, hank.f, sing.f, tutil.f: Various utility and diagnostic functions.

The Java version has a similar structure but transposed to the object world. For the standalone version there is a class called `Be2DTest` that coordinates the solver function. Two classes `Inputm` and `Inputc` read the model and control files respectively

and return data objects that implement two interfaces, as is Java "best practise". The model interface can be seen in B.2.2, page 175 and the control interface in B.2.3, page 176.

`Be2DTest` uses the data from the model and control objects, referenced through their interfaces to fill the matrix using a static method of the class `MatrixFill`, `MatrixFill.fill()`. The LU decomposition and LU solve functions are again static functions, this time of the class `LUSolver`, `LUSolver.lud()` and `LUSolver.lus()` respectively. Finally the output is generated and written to two files in the current directory, *cur* and *rcs*, using the methods to output `OutCur.write()` and `OutRcs.write()`. The Bessel functions and other utility functions are coded as static functions of the `MatrixFill` class.

This particular modular design for the Java version of the code was chosen so that as well as giving us a standalone Java version, it would be easy to incorporate the functionality into a component in the VCCE. In addition, separating the I/O functionality into their own classes allows for the possibility of incorporating a component that has Java I/O for ease of use within the VCCE and and native execution for speed of calculation. To include the native code directly would require two levels of wrapping, first providing C functions that use the original Fortran solver code, and then via Java Native Interface (JNI) calls that call the C functions.

The initial standalone version of the Java code took 24.3 seconds but the execution time was improved to 15.1 seconds, by profiling the calculations the code was performing. By replacing the static binary complex number operators by the unary method wherever there was no side effect problem with modifying the original number, 9 seconds were taken off the execution time. This is almost entirely down to the fact that there are substantially less "new", object creation operations in the code. Further optimisation could almost certainly be achieved through further investigation, but that is for future work.

Within the VCCE the *BE2D* components are executed using one of the execution proxy classes outlined in chapter 4. As with previous examples, specific instances of components are created using a generalised proxy interface and an XML properties

file. In this case, either the `JavaExecution` or `ExtendedJavaExecution` proxy can be used. These proxies are used to run Java-based components within the VCCE. The decision on which one to use, since they both perform the same functionality, is based on whether to run the component in the same JVM or not. See section 4.5.

The component definition file for the Java-based *BE2D* has a similar structure to the one for the "native" wrapper version in code segment 5.1. The important differences, apart from the obvious naming section which is ignored here, are:

- The inputs for this component are file-based, hence the definitions:

```
<inport id="1" parameter="control" type="stream"
    value="NIL">
  <href name="file:///home/scmmss/project/src/be2d/
     be2dComponents/control" value="NIL"/>
</inport>
<inport id="2" parameter="model" type="stream" value="NIL">
  <href name="file:///home/scmmss/project/src/be2d/
     be2dComponents/model" value="NIL"/>
</inport>
```

- The outputs are are also file-based:

```
<outport id="1" parameter="cur" type="stream" value="NIL">
  <href name="file:///home/scmmss/project/src/be2d/
     be2dComponents/cur" value="NIL"/>
</outport>
<outport id="2" parameter="rcs" type="stream" value="NIL">
  <href name="file:///home/scmmss/project/src/be2d/
     be2dComponents/rcs"  value="NIL"/>
</outport>
```

- The execution section lists the class name for the component to run and also some housekeeping details such as JDK version, source code location and class-path. In this case the `JavaExecution` is being used model to execute the components.

```
<execution id="software" type="bytecode">
  <type id="architecture" value="serial"/>
  <type id="class" value="be2d.be2dComponents.Be2DTest"/>
  <type id="source" value="file:///home/scmmss/project/src/
     be2d/be2dComponents/*.java"/>
```

```
  <type id="classpath" value="/home/scmmss/project/classes;/
     home/scmmss/local/3rdPartyJava/JNL/Classes"/>
</execution>
<execution id="platform">
  <type id="java" value="jdk1.2"/>
</execution>
```

From these details the proxy component is able to create an instance of the `Be2DTest` class and execute it. Taking input from the specified files and writing the output to the output files. A comparison between this mechanism and the others in this chapter follows at the end of the chapter.

5.1.3 CORBA Wrapped Fortran Version

The two main industrial wrapped codes described in chapter 7 use CORBA as a communication mechanism. As a prototype and test case before using the large codes, which due to access restrictions and hardware requirements can only be run on machines inside BAE SYSTEMS offices, the *BE2D* native Fortran executable was also wrapped as a CORBA component.

The Common Object Request Broker Architecture (CORBA)[106], is an object oriented, distributed computing framework. It is language independent with distributed object interfaces specified in a common Interface Definition Language (IDL). Systems built with this technology are split into client and server side objects with object "stubs" on both sides. Methods are invoked by the client side object discovering a reference to the server side object through a registry and then calling the method through the use of "helper" code.

To wrap the *BE2D* code there has to be a server side CORBA object that executes the solver and a client side object that is represented as a proxy within the VCCE. The proxy calls the method on the remote server object when it is executed within a running task graph. The server object is within a running CORBA server and waits for a message from the component client on the VCCE to start the execution of the solver. Once the solver has finished execution, the server runs some

post processing on the *rcs* and *cur* files using the Awk text processing language to format the files. The *URL* location of the files is then returned so the client can view the data using a graph viewer that can handle *HTTP* streams.

The CORBA component interface is specified in IDL by the code below

```
module Be2dComponent {
        interface Run {
                string runBe2d();
                };
};
```

This simple interface has a single method that runs the solver code and returns the *URL*. From this interface, the client and server side "stub" code skeletons are generated and then have to be implemented with code to actually execute the solver.

The client component obtains a CORBA reference to the remote server object. A named port is used to discover the `NamingService` registry and using that a generic CORBA object is created. Then using the helper code the generic object is resolved to an instance of the remote *BE2D* object. Executing the remote object method is a call on the resolved local object which returns a *URL* to the solver data. The VCCE client component then displays the data, accessed via the returned *URL*. This can be seen in code snippet below.

```
Properties props = new Properties();
props.put("org.omg.CORBA.ORBInitialPort", "1050");
ORB orb = ORB.init(args, props);
org.omg.CORBA.Object objRef = orb.resolve_initial_references("
    NameService");
NamingContext ncRef = NamingContextHelper.narrow(objRef);
NameComponent nc = new NameComponent("Run", "");
NameComponent path[] = {nc};
Run runRef = RunHelper.narrow(ncRef.resolve(path));
urlString = runRef.runBe2d();
Be2dClient be2d = new Be2dClient(urlString);
be2d.show();
```

The server side object has to be started before the client component can call it. In the server class, the `main` method that gets called when the class is executed, first uses the named port to obtain a reference to the *NamingService* registry. The

class that implements the IDL interface, `Be2dComponentServant` is then bound to a `NamingContext` object within the registry so that it can be discovered by the client code.

```
Properties props = new Properties();
props.put("org.omg.CORBA.ORBInitialPort", "1050");
ORB orb = ORB.init(args, props);
Be2dComponentServant be2dComponentRef = new Be2dComponentServant
    ();
orb.connect(be2dComponentRef);
org.omg.CORBA.Object objRef = orb.resolve_initial_references("
    NameService");
NamingContext ncRef = NamingContextHelper.narrow(objRef);
NameComponent nc = new NameComponent("Run", "");
NameComponent path[] = {nc};
ncRef.rebind(path, be2dComponentRef);
java.lang.Object sync = new java.lang.Object();
synchronized(sync) {
    sync.wait();
}
```

The `Be2dComponentServant` class that the server code uses to delegate the execution of the *BE2D* solver implements the method.

```
String runBe2d();
```

The code inside this method is the same as the simple native wrapper code in section 5.1.1. It uses Java's `Runtime.getRuntime().exec()` mechanism to execute first the solver code and then two post processes that format the two output files, suitable to be returned via *HTTP URLs*. The directory that contains the files is returned as a string representation of the *URL*.

As the CORBA implementation is hidden behind the component interface, the VCCE uses the generic execution component, `ExtendedJavaExecution`, to run the component. The XML component definition is the same as the definition for the Java-based *BE2D* solver, in section 5.1.2, apart from the classname of the component to run, so is not reproduced here.

This initial experiment with CORBA wrappings for components within the VCCE allowed the testing of ORB implementations, enabled experiments with block-

ing calls and return values, and generally provided a learning curve for using CORBA with a relatively small and lightweight code.

5.2 The Graph Viewer Component

Visualisation of data is a core requirement for a PSE, as specified in use case 2.5.1 on page 17. The VCCE is no different and for the visualisation of the data from the *BE2D* component, a graph plotting component is needed. Rather than spending too much time writing a graph plotting library from scratch, a third party library JChart [91] was used. A component which uses the library was written which can be used through the *proxy* mechanism and an XML component definition. A screen shot of the graph plotter in use can be seen in figure 5.2

5.3 Using the PSE

When the PSE is started, the VCCE first checks in the component directory or directories for all defined components, in XML files. The directories that the application examines are defined in an application properties file. The XML component definitions are parsed, the proxy components created and added to the component tree ready to be selected by the user, as illustrated in figure 5.1.

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 to 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

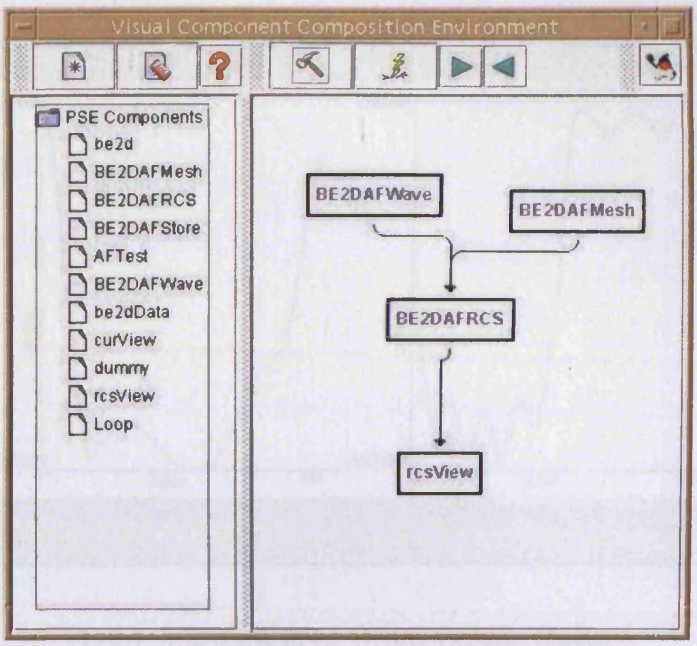

Figure 5.1: VCCE with BE2D Task Graph

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 task graph can have more than one start point, if there are two initial input generating components for instance. The assembled and connected *BE2D* task graph is illustrated in figure 5.1. To execute the completed task graph the user simply presses the start button to initiate the simulation. The solver is combined with the graph viewer from section 5.2 and the output generated from the code can be seen in figure 5.2.

5.4 Component Instantiation and Graph Execution

Upon being dropped on the "scratch pad" the component is instantiated and the `instantiate()` method is called. What this method actually does is dependant on the component programmer. For simple Java components this method will normally be a no operation method. In the case of CORBA components, such as the simple CORBA BE2D component described in 5.1.3, the method is used to resolve the references to the CORBA ORB and the remote CORBA component. Because this

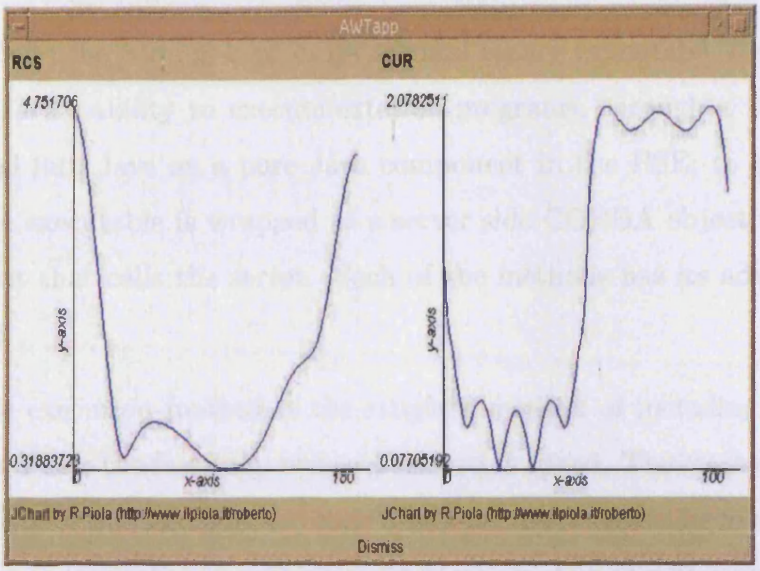

Figure 5.2: Graph of BE2D Solver Output

method runs in a separate thread within the VCCE program, this CORBA "hand-shaking" can be carried out behind the scenes while the user continues to build the task graph. In this way the possibly time consuming initialising code is done before the user actually executes the task graph and the whole process appears, to the user, to be faster.

When a task graph is executed, each of the components in the graph is told to execute in turn by the scheduler calling the `execute()` method, starting with the component or components that have been identified by the user as the starting point for the graph. After a 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 component's `execute()` method.

5.5 Summary and Conclusions

This chapter has demonstrated three different ways in which a code written in Fortran can be used as a component within a PSE. From the most straightforward

mechanism where the code is kept in its original binary executable format and executed using Java's ability to execute external programs; through a version of the code converted into Java as a pure Java component in the PSE; to the final version where the executable is wrapped as a server side CORBA object with a client side component that calls the server. Each of the methods has its advantages and disadvantages.

The simple execution method is the simplest method of including a code as a component. It is also the fastest in terms of execution speed. The main disadvantage is the inflexibility of this solution, the code needs the input files to be in a set location and the output is generated as a file in a set location. In order to make use of this component, file readers and writers will need to be built to act as the input and output ports for the code.

The native Java version of the code is the most flexible component, it is easy to provide different input and output mechanisms such as *Streams* or *Sockets*. However it relies on access to the original source code. It is time consuming to convert even a small code such as this from one language to another, and Java's lack of primitive *complex number* support makes the code hard to write and relatively slow to execute.

The CORBA wrapped code is useful because it provides a distributed client server architecture which is useful for times when a code may be sensitive and only allowed to be run on certain computers.

Experimenting with the relatively simple *BE2D* code has highlighted some useful ideas for the far more complicated 3D codes to come in chapter 7.

CHAPTER 6

Control and Loop Components

This chapter discusses another "use case" for the VCCE. A different way of using the included codes, known as "parameter runs" is shown. Here the user performs a number of consecutive executions of a solver while perturbing the input parameters to examine the effect on the solution. Typically this technique is used as a manual domain space search or optimisation. This chapter describes extensions to the prototype and definition language that include simple loop controls to allow users to perform parameter runs automatically on component based solver codes. A new scheduling algorithm is also needed to handle the extensions. Finally the PSE mechanism is compared to the traditional scripted methods for performing parameter runs, illustrating the flexibility and ease of use that the VCCE provides.

6.1 The Use Case

The second "use case", section 2.5.1, defines:

> Performing parameter runs on existing or new applications, to study the effect of parameter ranges on the result. Here an application code is run

multiple times in succession with perturbed input parameters in order to try and find a particular solution for a problem.

Parameter runs are a very common task in scientific and engineering computing. In the case of an engineering design and manufacturing company, such as the industrial partner in this work BAE SYSTEMS, an example of this problem could be seen with the simple *BE2D* code from chapter 5. The solver is an electro-magnetic wave scattering simulation and can be used to simulate RADAR waves reflecting off the surface of an aircraft. One of the input parameters is the angle of incidence of the wave, which could be used to simulate the difference in wave scattering as an aircraft flies over the RADAR system. As the aircraft approaches at a distance, the angle of incidence approaches 90^o, overhead, from 0^o, the horizon, and then moves off toward 180^o, the opposite horizon. A parameter run would be used to automatically perturb the angle of incidence input parameter from 0^o to 180^o in defined increments for each solver execution and store or display the results for examination.

6.2 Control Components

Until this point the work in this thesis has only been concerned with simple work flows in the prototype PSE. Execution has flowed from single or multiple start points down to single or multiple end points. Execution branches have been included where either multiple inputs enter a component, *flow joins*, or a component has multiple outputs, *flow splits*. Here the concept of *control* components is introduced, these allow us to perform flow control operations. These components can be thought of in programming construct terms, such as *for...next* loops or *if...then* branches. A control construct is a user defined section of work flow that allows the user a certain degree of control over how that work flow executes. These can allow selective execution of sub-sections of work flow through conditional branch constructs, or repeated execution of sub-sections through looping constructs.

The control components discussed here, work in a similar manner to the tradi-

tional "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 inport of the next component.

Conditional branching constructs, although important are not considered here as they are out of the scope of the current user requirements. They are mentioned briefly in the future work, chapter 10. The control construct discussed here is the loop component.

6.3 Loop Constructs

As in a traditional, non-visual, program languages, there are different types of loop in VCCE. One loop is discussed here, a simple iterative loop where the loop executes itself and the sub-work flow a fixed number of times over a set value with a set increment value until a fixed halting condition is reached. In chapter 8 a more complicated constraint-based loop construct, capable of non-linear iteration is discussed.

When the control components are introduced into a task graph, by connecting them to a suitable component, they generate one of the input parameters to that component. To illustrate the functionality of the control components the *BE2D* code parameter run will be used as an example. For this case there are two control components. One to control the angle of the incidence wave and the other to control the wave frequency. Using these two controls the *BE2D* solver is executed a fixed number of times for defined parameter ranges. The connected task graph can be seen in figure 6.1. As can be seen in the figure, the simple example from chapter 5 has been used. There are three components, a wave generator and mesh generator as inputs to the solver code, and the output from the solver going to a graph component. Now there have been added two control components that provide the inputs to the wave component. When these control components are connected to another component,

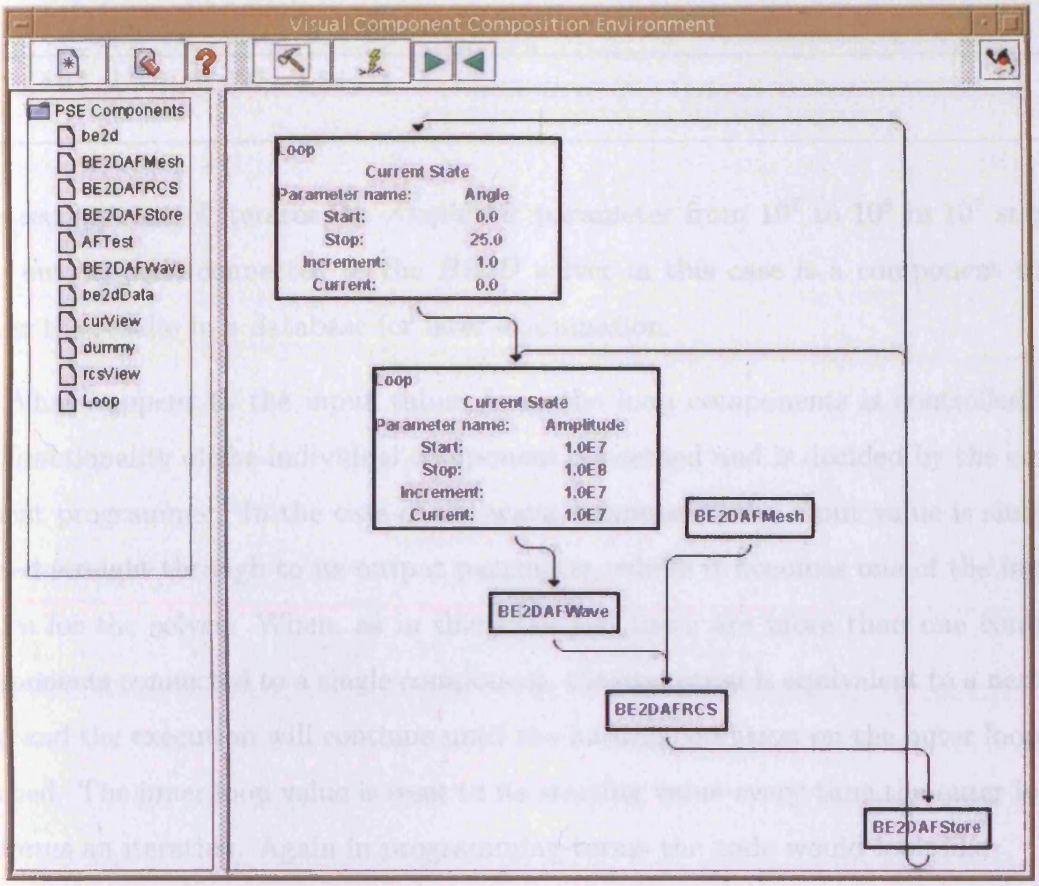

Figure 6.1: A Work Flow Graph with Loop Components

the user is prompted with a selection of control input parameters for iteration. In the case of *BE2D*, the two input parameters to the *wave* component are the frequency and angle of incidence. Since these are floating point values, they 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.

The top control in the figure changes the named parameter *Angle* in the wave component, from the interface it can be seen that the user has entered some values into the control: *start* value is 0.0, *stop* is 25.0 and the *increment* is 1.0; this means that this loop will iterate from 0 to 25 in single integer increments. In programming terms it would be equivalent to

```
for (int i=0; i<25; i++) {
}
```

The second control iterates the *Amplitude* parameter from 10^7 to 10^8 in 10^7 steps. The output node connected to the *BE2D* solver in this case is a component that stores the results in a database for later examination.

What happens to the input values from the loop components is controlled by the functionality of the individual component concerned and is decided by the component programmer. In the case of the wave component, the input value is simply passed straight through to its 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 its starting value every time the outer loop performs an iteration. Again in programming terms the code would look like

```
for (int i=0; i<25; i++)
    for (int j=1.0e+07; j<1.0e+08; j+=1.0e+07) {
    }
```

Finally, from the figure, there are two connections coming from the storage component back up to the two control components completing the loops. This is necessary for the implementation to know what components are contained within the loop and which are not. It should be clear that the mesh generation component is outside the loops and therefore is only executed once. Its output value is stored within the solver component and reused.

6.4 Control Implementation

The implementation of the loop constructs in the VCCE model involved a number of extensions to the framework as well as the additional components which would perform the iteration functionality. The task graph XML representation language

has to be extended to allow the inclusion of loops, this in turn impacts the Java language representation and the scheduling algorithm. Finally the component model needs to be extended to create the new type of control component and provide a user interface representation within the VCCE.

The implementation here illustrates why good software engineering practises are important in projects such as this. Programming to *interfaces* enables the creation of new implementations of ideas such as components or scheduling algorithms without impacting the main framework. This also allows the VCCE prototype to be backwardly compatible with older components and task graphs.

6.4.1 XML Task Graph Language Extensions

There are two extensions to the XML language representation needed. The first is the major issue of specifying the loop connection to a specific parameter input port on a component. The other is a secondary issue that appeared as a side effect of having two loop components in the task graph.

The language extension to include the loop component is only necessary for the connection. The loop component itself does not have an XML component definition as it is specific and internal to the VCCE and consequently does not need an external representation. It is possible to construct a generalised component that can perform a looping function that needs an external XML interface representation but that is left for future work. The *connection* definition in the task graph is extended so that if the *parent* in the connection is a loop then there is an extra tag that specifies the loop parameter and the values. For example, in appendix B.3.1 the full XML task graph definition for our *BE2D* parameter run example can be seen. The extension to the connection tags can be seen in listing 6.1. In this extract the representation for the outer connection from figure 6.1 where the loop controlling the *Angle* parameter is connected to the loop controlling the *Amplitude* parameter. The second loop does nothing with this parameter value except pass it straight through as another output. The extension in line 6 specifies the named parameter together with the

start, increment and halting values and the *type* of these values. The type must be either a valid number type for the increment of the loop to make sense.

```
<connection>
  <parent>
      <name alt="Loop" id="loop01" inst="4081527">
          Control Loop
      </name>
      <loop parameter="Angle" port_type="Float" start="0.0" halt=
         "25.0" increment="1.0" current="3.0" />
  </parent>
  <child>
      <name alt="Loop" id="loop01" inst="1947116">
          Control Loop
      </name>
  </child>
</connection>
```

Listing 6.1: XML Task Graph Extension for Control Structures

This code listing together with the full listing in appendix B.3.1 shows that there is no direct connection between the outer loop control component and the *wave* component that it is providing one of the input parameters for. The connection is implied by the fact that the loop is connected to a second loop that is in turn connected to the actual component. This will become clear in the implementation explanation in section 6.4.2 but in effect if a loop receives an input then it is just passed straight though as an output. Hence, the need for the parameter name and values to be passed within the connection definition of the task graph.

The cause of the side effect extension to the language can also be seen here. In the specification of the language, the "name" information for a component consists of a real name, an alternative name and an ID. In listing 6.1 there are two instances of the same loop component in a single task graph. The original naming system is not enough to differentiate between them so the naming system had to be extended to include an *instance* ID which is only used within a task graph to differentiate between to instances of the same component. The instance ID is a guaranteed unique generated value. In our example here the first loop has been assigned an instance ID `inst="4081527"`.

6.4.2 Loop Component Implementation

The loop component like any other component relies on a proxy and implements the proxy interface. The main proxy class is `ControlProxy`, and like `SimpleProxy` this extends the `AbstractProxy` abstract class and implements the `ProxyInterface`, section 4.5.1. All this means that the `ControlProxy` can be used in the VCCE framework as the framework relies on interfaces and not implementations.

The control behaviour in `ControlProxy` is defined in an interface called, not surprisingly, `ControlInterface`, the full listing of this can be seen in appendix B.2.3. The interface has "setter" and "getter" methods for the loop parameters: *start* value; *halt* value; and *increment* value. The "getter" method returns the `PortType`, a constant that represents the data type for the loop. The loop can iterate over any numerical data type, which could be an actual input parameter or just an integer for the number of loop steps. To set up the loop connection to control a specific parameter value on an input port, there are methods to set or add ports to the control component and to return the parameter name that the loop is controlling.

```
// Set the input port that the loop will iterate over.
public void setLoopedPort(PortInterface aPort);

// Return the parameter name for the selected port.
public String getParameterName();

// Adds an output port to the control component, used in cases of
    loop nesting where the value from the outer loop needs to be
    propagated through to the inner loops.
public void addOutport(PortInterface aPort);
```

To perform the iteration functionality of the loop, checking the halting condition and then if appropriate incrementing the current value, there are the methods

```
// Returns the state of the halting condition. true if current
    value is less than the halting value, false otherwise.
public boolean haltingCondition();

// Increment the loop.
public void performStep();
```

```
6
  // Reset the loop to its start value;
8 public void resetLoop();
```

The loop proxy, `ControlProxy`, implements the `ControlInterface` but delegates the majority of the work to the execution object within the proxy. The implementation of the `ExecutionInterface`, section 4.5.2, in the loop component is provided by the class `ControlExecution`. It is this execution object that also implements the `ControlInterface` that the `ControlProxy` delegates the functionality to. For example, the implementation of the check halting condition method in `ControlProxy` is

```
public boolean haltingCondition() {
2     return ((ControlInterface) execution_).haltingCondition();
  }
```

where the object `execution_` is an instance of `ControlExecution`. The implementation in the `ControlExecution` class is

```
public final boolean haltingCondition() {
2     if (getPortType() == PortType.SHORT) {
          return (((Short) getCurrentValue()).compareTo((Short)
              haltValue_) >= 0);
4     } else if (getPortType() == PortType.FLOAT) {
          return (((Float) getCurrentValue()).compareTo((Float)
              haltValue_) >= 0);
6     } else {
          return true;
8     }
  }
```

Listing 6.2: Loop Control Halting Condition

In this implementation the current value of the loop is compared to the halting value, returning the appropriate true or false result. If the *haltingCondition* is true then the loop has finished. Note here that the data type of the parameter iterating is checked using the `getPortType` method. The comparison needs to be aware of whether the data is integer or floating point. The `performStep` method has a similar implementation, checking the data type and then incrementing the current value by the appropriate increment value.

The user interface to the loop component within the VCCE which can be seen in figure 6.1 is provided by an extension to the `SimpleVisualComponent`, section 4.5.1. The new subclass, `ControlVisualComponent` users the "getters" and "setters" in the `ControlInterface` to provide the interaction between the user and the loop component for the *start*, *halt* and *iteration* values.

The new execution model is more complicated than our previous `AbstractExecution` implementations which just had two methods to call, `instantiate` and `execute`. The `ControlExecution` implementation of `execute` is still called, the method checks the halting condition and performs the loop iteration. However, the scheduling algorithm has to be aware that there is a control component in the task graph and react accordingly. The new scheduling algorithm and the implementation is explained in the next section.

6.4.3 `ControlExecutionGraph`: An Improved Scheduling Algorithm

The simple scheduling algorithm implementation, `ExecutionGraph`, introduced in section 4.6.3.1 was developed before the control components and cannot take advantage of the new methods provided by the control interface. The addition of these new components requires an extended scheduling implementation. Both the task graph/scheduler class and the node class that wraps the proxy component within the graph are extended.

`ControlGraphNode` is the class that holds a `ControlProxy` object within the task graph implementation. It extends `ExecutionGraphNode`, section 4.6.3, so it inherits all the functionality providing the parent and child relationships with other nodes in the graph. The extensions provide access to the internal loop component's `haltingCondition` method, methods to set and reset the loop values, and methods to add output ports that represent the parameters being iterated over.

In the extension to the task graph representation there are now two types of node, the original node representing computational components, `ExecutionGraphNode`

and the new implementation representing loop components, `ControlGraphNode`. The new algorithm can be seen in algorithm 6.1, it works by partitioning the task graph into sub-graphs and then scheduling each sub-graph individually in a "divide and conquer" manner.

```
for all loop components in task graph do
  create node lists for each loop component
  recursively add any contained loop component children to loop list
end for
for all nodes not in existing list do
  create node lists for any blocks of components outside of loops
end for
for all lists in order do
  order components in list
  repeat
    for all components in list do
      execute component
    end for
  until list does not contain loop or halt = true
end for
```

Algorithm 6.1: Extended Scheduling Algorithm

It can be seen from the algorithm, that at various points a check will need to be made on components to see if they are loop components and if they are, check the halting condition. This, together with the task graph partitioning, make up the major differences between the previous scheduling algorithm.

The implementation is sub-classed from the same abstract base class `AbstractExecutionGraph` as the *naive* algorithm and so implements the two abstract methods

```
1 public abstract void executionPerformed(ExecutionEvent evt);
2 public abstract void execute();
```

`execute` runs the composed application and `executionPerformed` is the "call back" function from each component upon completion of its execution. The execute method

```
1 public void execute() {
2     scheduleNodes();
```

```
    Collections.sort(graphPartitions_, new PartitionComparitorRun
        ());
    partitionIterator_ = graphPartitions_.listIterator();
    currentPartition_ = (Vector) partitionIterator_.next();
    nodeIterator_ = currentPartition_.listIterator();
    currentNode_ = (ExecutionGraphNode) nodeIterator_.next();
    executeNext();
}
```

partitions the nodes, sorts the partitions and then sets variables for the current partition, current node and iterators before calling the **executeNext** method that executes the next component in the current partition.

```
private void executeNext() {
    if (currentNode_.isControl()) {
        if (!((ControlGraphNode) currentNode_).haltingCondition())
            {
            transferPortData();
            if (nodeIterator_.hasNext()) {
                currentNode_ = (ExecutionGraphNode) nodeIterator_
                    .next();
                executeNext();
            } else if (partitionIterator_.hasNext()) {
                currentPartition_ = (Vector) partitionIterator_.
                    next();
                nodeIterator_ = currentPartition_.listIterator();
                currentNode_ = (ExecutionGraphNode) nodeIterator_
                    .next();
                executeNext();
            }
        } else {
            currentNode_.execute();
        }
    } else {
        transferPortData();
        currentNode_.execute();
    }
}
```

The **executeNext** method checks to see if the current node is a control node, if it is and the halting condition is false then output data from any parent nodes is transfered using the **transferPortData** method and the next node in the current partition or the first node in the next partition is executed. If the current node is not a control node then the output data from the parent components is transfered to

the current node's inports and the component is executed. The `transferPortData` method calls the `sendOutputToChild` method on all the parent components of the current component.

```
private void transferPortData() {
    for (EGNIterator it = currentNode_.getParents(); it.hasNext()
        ;) {
        it.next().sendOutputToChild(currentNode_);
    }
}
```

The `sendOutputToChild` method is in the `ExecutionGraphNode` class. The behaviour of the method had to be amended to include control components, and is dependant on whether the child node that the data is to be transfered to is a normal component or a control component.

```
public void sendOutputToChild(ExecutionGraphNode child) {
    if (!child.isControl()) {
        for (PortIterator it = component_.getOutports(); it.
            hasNext();) {
            PortInterface port = it.next();
            child.setComponentInportValue(port.getParameter(),
                port.getValue());
        }
    } else {
        for (PortIterator it = component_.getOutports(); it.
            hasNext();) {
            PortInterface port = it.next();
            ((ControlGraphNode) child).setComponentOutport((
                PortInterface) port.clone());
        }
    }
}
```

If the child component is a standard component then the outport values are set as inport values on the child component. If the child component is a loop component then a copy of the outport is added as an outport of the loop component. This functionality is needed only in the case of a nested loop. The child loop component does nothing with this input so it is just passed straight through as an output.

6.5 Summary and Comparison

One of the features of the PSE is to provide the ability to perform iterations over components in a task graph, for example to perform parameter runs on a solver code. This chapter has examined extensions to our components and framework that support this ability. The component model has been extended to include a new *control* component, and the task graph representation and scheduling algorithm have been extended to support that new component.

There are many scripting languages that can be used to perform the type of parameter run loop operation described here, from *shell* scripting through to separate languages such as *Perl* or *Python*. There are advantages to using scripts if the script is for personal use or is not going to be used by non-programmers. Scripts do not rely on a large framework such as the VCCE and do not require individual computational codes to be wrapped as components. They also put less of an overhead on the execution time for the work flow as they are a lot closer to the execution environment, i.e. the operating system, of the individual codes.

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. The work in this chapter has shown a simple user interface and component framework that makes constructing iterative loops over components in an application a much more intuitive task. The user does not need to know any details about scripting languages, or similar, and is free to concentrate on the more important details of what they want to do as opposed to how to do it.

The next chapter examines a "real" industrial code and a different method of code wrapping using CORBA. The control constructs introduced in this chapter become more important when using real codes and will be revisited again.

CHAPTER 7

BE3D, FE3D - Parallel Components

In this chapter, two new solver codes from the client BAE SYSTEMS are discussed, together with a different mechanism for wrapping and including them as components within the PSE framework. The two codes here are real "production" solvers that are commercially sensitive and considerably larger and more complicated than the *BE2D* code examined in chapter 5. These codes will provide a more realistic examination of the VCCE and show that this technology can be used by scientists for real work. The new wrapper mechanisms are compared to the existing methods. Some performance comparisons with the standalone code and an analysis of the benefits to the user conclude the chapter.

7.1 The Parallel Solver Codes

The two new solver codes used in this chapter are very similar in usage and form although the algorithms are very different, hence they are covered as a single chapter. Both codes are written in Fortran and use the Message Passing Interface (MPI) [81] to facilitate parallelisation across multiple processors. The codes are complicated, both in terms of lines of code and in usage, and are also commercially sensitive. The

commercial sensitivity has two direct impacts on the use of the codes as components within a PSE.

1. Source code is not available. This means that the only possible use within a PSE is by wrapping the binary executable. Tighter integration will be impossible, and input and output mechanisms are reliant on the original implementation.

2. Execution environment is limited. The commercial sensitivity of the code means that installation and execution of the binary is only allowed on a specified machine or machines. In the case of these particular codes that means a single machine at a BAE SYSTEMS site behind a firewall.

The two codes are called *BE3D* and *FE3D*, both codes are simulations for the analysis of electro-magnetic wave scattering in three dimensions. The first of the two codes is a three dimensional production version of the *BE2D* code examined in chapter 5. The second code uses a *finite element* algorithm in three dimensions. It is the addition of the third spatial dimension to the codes that make these examples so much more complicated than the simple *BE2D* example used previously.

As with the more straight forward two dimensional code, and in common with the vast majority of simulation codes of this broad type, execution can be broken down into three distinct stages:

1. Mesh or geometry generation.

2. Execute solver code.

3. Visualise or analyse results.

In addition to these three stages there may be additional intermediary stages to perform data format translation and data storage tasks. To use both of these codes in their original form, the user first runs the appropriate mesh generator from the command line. This produces the file that represents the three dimensional data.

The user then runs the solver from the command line, ensuring that the data file and the wave control file are in the correct directory. The solver produces two output files, one containing the radar cross section data and the other the surface current.

Computational solver codes generally take input and provide output in very strict formats. The formats used to represent data are varied and often specific for a particular code within a particular organisation. Often the mesh generator that provides the major input to a solver is written specifically for that solver and its data format. This specificity and the tight coupling between solver and mesh generator hinders their reuse as individual components within a PSE. Although there may be many formats for the data, translation may be possible from one to another. An intermediate component may be able to perform translation between the format the mesh generator produces and the solver consumes and the format the solver produces and a data visualiser consumes. These intermediary components will allow us to connect a mesh generator component to a different solver component within the PSE.

Data storage at both the end of the solver process and at intermediary stages, especially in the situation where the user is performing multiple executions in a parameter run, section 6.1, is an important aspect of the day-to-day work of a scientist. Specialised components will provide access to storage repositories which could be databases or file storage. The decision to store data may depend on the cost of the individual execution time of a component, some solvers may take weeks to produce results, or whether the results are likely to be re-used. Intermediary results may be stored to save some execution time in the computational stage if the application is re-run. For example, a scientist may run the mesh generation once, store the generated mesh and then re-use that multiple times within a parameter run.

7.2 CORBA Wrappers and the Action Factory

The method used to turn the two solvers and their mesh generation and storage functions into components that can be used within the PSE is to wrap the codes as CORBA components. Functionally the mechanism is similar to that used in section 5.1.3 to wrap the simple *BE2D* code. Here rather than writing separate CORBA wrappers unique to each component, a third party framework originally provided by Dassault is used, an industrial partner of BAE SYSTEMS, but now an "open source" project called Action Factory [25]. The Action Factory system is designed to help integrators to fill the gap between developers and users, by providing a clean data flow to object oriented mapping, and enabling a quick and mostly automated way to build actions and computational paths on top of existing developer's components. The Action Factory provides a single point of access to wrapped codes through the use of a Remote Procedure Call (RPC) type interface implemented in CORBA behind which the codes sit.

In practise 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 appropriate Action 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 performs the connection to the ORB and retrieves a reference to the Action Factory 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 5.1 for more specific details.

7.2.1 Action Factory Component Implementation

With the example codes used from within the PSE, various parts of the codes and data generators are wrapped as CORBA objects. The components that make up

the complete assembled application code are:

- The mesh generator, for generating the data which defines the example 'mesh' or geometry for the solver code. This is the original mesh generator with a CORBA wrapper that returns a reference to a CORBA object representing the data set, instead of writing the data to a file.
- Control components that define the characteristics of the incidence wave, frequency and angle.
- The *BE3D* and *FE3D* solvers. These are the original solvers, modified to accept input data from CORBA objects and provide output as CORBA objects.
- 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. In this case a specialist scientific database suitable for storing the large amounts of data generated by the solver codes.

The Action Factory object is responsible for instantiating and executing each of these CORBA wrapped components. The PSE does not need to know details about object instantiation or execution. The Action Factory 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 Action Factory from within the `execute` method of a proxy component would be:

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

Listing 7.1: Example Action Factory Function Call

The parameters in this method have the following meanings

`action_name_`

Represents the name of the command to be executed. For instance `ActionReadMesh` is an *action* that generates a 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, represented by `parameter`, or a reference to a CORBA object in the case of the mesh, represented by `numBddETag`. The Action Factory is responsible for delivering input datasets to the appropriate components to be executed to complete a given action.

`fact_machine_name_`, `fact_obj_name_` and `fact_port_number_`

Are values that the Action Factory uses to decide what component to execute and where to run it. They represent the name of the machine the Action Factory is running on, the object name of the Action Factory to use and the port number needed to contact the Action Factory.

`result`

The return value from the function call which will be a reference to a CORBA object represented as a *short* value. The reference is internal to the Action Factory which provides methods to retrieve a CORBA object given the reference.

All of these parameter 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 Action Factory 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.

The *execution* tags of an example XML component definition which are used to create a proxy that can call the Action Factory can be seen in listing 7.2. In this definition, lines 9 to 14 contain the definition for the Action Factory server, including: the CORBA ORB implementation, "orbacus" in this case; the host machine name, *sg20*; the port number; and the name of the CORBA service, *emmaAF*. This Action Factory service will provide a number of Action factory instances which in turn provide a number of *actions*. This component uses one single *action* which is defined in lines 1 to 8. The definition contains: the name of the specific Action Factory instance, *FE3D* which provides the wrappers for the *FE3D* application; the action name, `ActionextractRCSSlice` which computes the RCS output for the solver; the machine name hosting this Factory, which can be different or in this case the same as the service host machine; the Factory object reference name; and the port number for this Factory instance.

```
<execution id="software" type="corba">
    <type id="architecture" value="parallel"/>
    <type id="actionFactory" value="FE3D"/>
    <type id="action_name" value="ActionextractRCSSlice"/>
    <type id="fact_machine_name" value="sg20"/>
    <type id="fact_obj_name" value="emmaOF2"/>
    <type id="fact_port_number" value="1324"/>
</execution>
<execution id="platform">
    <type id="corba" value="orbacus"/>
    <type id="host" value="sg20"/>
    <type id="port" value="1555"/>
    <type id="server" value="emmaAF"/>
</execution>
```

Listing 7.2: Example Action Factory XML Definition

These parameter values provide enough information to be able to contact the CORBA Action Factory, get a specific instance of the Action Factory that provides the *FE3D* functionality and call a specific method or *action* on that factory.

To actually execute a function call on the Action Factory, the proxy component model needs to be extended again. The execution model of the components is extended to handle this new component wrapping technique. `ActionFactoryExecution` extends `AbstractExecution` and therefore implements the `ExecutionInterface`, see section 4.5.2. The `ComponentFactory` is extended to instantiate an `ActionFactoryExecution` object if the software type is "corba", listing 7.2 line 1. The `ActionFactoryExecution` object has attribute values for each of the parameter properties defined in the XML component definition and implements the `instantiate` and `execute` methods of the `ExecutionInterface`.

The `instantiate` method is called when the user drops the component onto the "scratch pad" of the VCCE. For these CORBA wrapped components, the method performs the pre-execution CORBA initialisation, "hand shaking" and component reference resolving. The simple CORBA wrapper example described in chapter 4 proved to be useful from a development point of view, shortening substantially the learning curve when it came to implementing the more complicated Action Factory examples. The instantiate method can be seen in listing 7.3.

```
public void instantiate() {
    System.setProperty("EMMAF.host", host_);
    System.setProperty("EMMAF.port", String.valueOf(port_));
    System.setProperty("EMMAF.server", server_);
    java.util.Properties props = System.getProperties();
    props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
    props.put("org.omg.CORBA.ORBSingletonClass", "com.ooc.CORBA.
       ORBSingleton");
    System.setProperties(props);

    String internalArgs[] = {host_, getPort(), server_};
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(internalArgs,
       props);
    org.omg.CORBA.Object obj = ((com.ooc.CORBA.ORB) orb).
       get_inet_object(host_, port_, server_);

    try {
        Class[] parameterTypes = new Class[]{Class.forName("org.
           omg.CORBA.Object")};
        Object[] parameters = new Object[]{obj};
        Method[] methods = factoryActionHelperClass_.
           getDeclaredMethods();
```

```
18          String methodName = factoryActionHelperClass_.getName()
                + ".narrow";
            Method narrowMethod = factoryActionHelperClass_.
               getDeclaredMethod(methodName, parameterTypes);
20          factory_ = (com.SixS.FactoryAction)narrowMethod.invoke(
               factoryActionHelperClass_, parameters);
        } catch (Exception e) {
22          e.printStackTrace(System.out);
        }
24  }
```

Listing 7.3: `ActionFactoryExecution instantiate` Method

lines 1 **to** 12

Instantiate the CORBA ORB on the named host machine and port, first setting some default system properties that the CORBA implementation looks for.

lines 14 **to** 20

Attempts to return the named Action Factory server object reference. It uses a generated "helper" class to try and resolve the factory name described in the XML definition to a reference that the CORBA name server recognises.

The value of separating the instantiation functionality from the execution functionality within the execution model can be seen clearly here. The "housekeeping" details performed in the `instantiate` method take a discernible amount of time from a user perspective, usually in the order of five seconds but dependant on network bandwidth and load. Performing this functionality when the user instantiates the component, in a separate thread, hides the time it takes as the user can continue creating the application from the components while the instantiation code executes in the background. Doing this does not actually make the system any faster but it will appear to the user as if it is when compared to the other case where both initialisation and execution are performed in the same method. One other benefit is that if the initialisation fails at application creation time as opposed to run time the user can select a replacement component.

The `execute` method can be seen in listing 7.4.

```
public void execute() {
    Vector parameterVect = new Vector();
    Vector numBddETagVect = new Vector();
    // getting inport values
    for (PortIterator pi = getOwner().getInports(); pi.hasNext()
        ;) {
        PortInterface port = pi.next();
        if (port.getType() == PortType.OBJECT) {
            numBddETagVect.add(port.getValue());
        } else {
            parameterVect.add(port.getValue().toString());
        }
    }

    // converting input numBddETag to short[]
    int i = 0;
    short[] numBddETag = new short[numBddETagVect.size()];
    for (Iterator it = numBddETagVect.iterator();
        it.hasNext();) {
        numBddETag[i] = Short.parseShort((String) it.next());
        i++;
    }

    i = 0;
    // converting parameterVect to String[]
    String[] parameter = new String[parameterVect.size()];
    for (Iterator it = parameterVect.iterator();
        it.hasNext();) {
        parameter[i] = (String) it.next();
        i++;
    }

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

    // setting outport values
    for (PortIterator pi = getOwner().getOutports(); pi.hasNext()
        ;) {
        String value = new Short(result[0]).toString();
        PortInterface port = pi.next();
        port.setValue(value);
    }
    getOwner().notifyObservers();
    notifyExecution();
}
```

Listing 7.4: `ActionFactoryExecution` execute Method

lines 2 **to** 10

Set up the input data to the component *action* to be called. If the input port type is of type "object" then assume this is an Action Factory object reference and it goes in the `numBddETagVect` object reference array. Otherwise assume it is one of the standard number or string data types and it goes in the parameter array.

lines 16 **to** 20

Convert any object reference values from string form to *short*.

line 32

Calls the Action Factory `exec` method to run the specified *action* with the input parameters, object references and host machine details.

lines 35 **to** 38

Take the result set from executing the *action* and place the values on the output ports of the proxy component.

lines 40 **and** 41

Notify that execution has finished on this component to all interested parties.

To include the facility to execute a new method of component wrapping, using a third party CORBA framework, it has only been necessary to change and extend the framework in two places. A new execution model was created, that implemented the core `instantiate` and `execute` methods and extended the component factory so that it can create instances of the new execution model where appropriate. The remainder of the framework including the scheduling algorithm and user interface was not touched. This again ratifies the decision to spend time creating an extensible framework in the first place.

7.3 Cost Benefit Analysis

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. Using the Action Factory model with CORBA, data transfer between components is kept on the server side as much as possible. When the mesh generator or the solver components are executed the only value returned to the client proxy component and VCCE is an object reference to the data set. It is the object reference and not the actual data set that is passed to the next component in the task graph. This technique of using CORBA as a control flow rather than a data flow was illustrated with another form of CORBA component wrapping in the paper at SC2000 [77].

Performance comparisons of wrapped legacy codes on both workstation clusters and dedicated parallel machines have been tried. 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 task graph at this time, so the cost is not noticeable. 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.

Execution time is not the only cost involved in using CORBA wrapped codes as components within the VCCE. Although frameworks such as the Action Factory and automated "Wrapper Generators" help, developing components from large codes such as the those illustrated in this chapter is a time consuming and skilled job. The cost of developing the component-based codes will only be justified if the codes are to be re-used by other, potentially non-programmer or casual, users. The component wrapping allows the code and the associated components such as mesh generators, data visualisation tools and data storage tools to be treated as "black box" components by new users. Components connected visually, help files found and displayed easily and previously saved task graphs, all help to make the job of re-using large codes by new or unfamiliar personnel much more straightforward.

The next chapter examines the last use case scenario. The notion of parameter runs and loop control components is extended to perform a non-linear optimisation of a problem using a wrapped solver code. A simple case demonstrates how this technique can be used to perform a "Design of Experiments".

CHAPTER 8

Design Of Experiments

"The central problem in optimization for engineers is the formulation and execution of problems, rather than the mathematical techniques themselves."

Optimal Engineering Design, Siddall [105]

A common activity in scientific computing and engineering design is that of optimisation. Searching a domain space using non-linear programming techniques to minimise or maximise the value of a function over a set of domain variables is a well researched area. It is not the aim of this work to cover in detail the problem of optimisation, merely to demonstrate how one well known search algorithm can be incorporated within the prototype VCCE as a component, and used to control the flow of execution.

The simple looping component, described in chapter 6, is sufficient for simple cases where the user needs to iterate over a fixed range of values for a variable or variables, so called "parameter runs". If the user needs to have more control over when an iteration stops, or how the step changes in relation to some state as is the case for non-linear minimisation or maximisation, then a more complex method of

control is needed.

The VCCE has been used as a front end to a PSE at Southampton University to look at design optimisation problems [102]. The work at Southampton was based around their "Options" suite of search/optimisation programs and various Computational Fluid Dynamics (CFD) codes with the VCCE providing the user interface and a CORBA implementation as its middleware.

The intention here is not to repeat that work, but demonstrate how optimisation algorithms can be encoded as control components within the prototype PSE to steer execution runs for a particular code to a desired optimisation. This is illustrated with respect to a particular minimisation algorithm, "The Simplex Method".

8.1 The Simplex Method

The "Downhill Simplex Method in Multi-dimensions" is an algorithm from "Numerical Recipes" [92], due to Nelder and Mead [85], used for multi-dimensional minimisation, that is, finding the minimum of a function of one or more independent variables.

The main concept used by the algorithm is the simplex. A simplex is a geometrical figure consisting, in n dimensions, of $n + 1$ points (or vertices) and all their interconnecting line segments, polygonal faces etc. Only non-degenerate figures, i.e. ones that enclose a finite inner N-dimensional volume, are considered. At each iteration of the method the objective function, the function being minimised, will be evaluated at $n + 1$ points in the N dimensional parameter space. For example, in two dimensions, where there are two parameters to be estimated, it will evaluate the function at three points around the current optimum. These three points would define a triangle; in more than two dimensions, the "figure" produced by these points is called a simplex. Intuitively, in two dimensions, three points will allow us to determine "which way to go", that is, in which direction in the two dimensional space to proceed in order to minimise the function. The same principle can be applied to

the multi-dimensional parameter space, that is, the simplex will "move" downhill; when the current step sizes become too "crude" to detect a clear downhill direction, the simplex is too large, and in this case the simplex will "contract" and try again.

An additional strength of this method is that when a minimum appears to have been found, the simplex will again be expanded to a larger size to see whether the respective minimum is a local minimum. Thus, in a way, the simplex moves like a smooth single cell organism down the objective function, contracting and expanding as local minima or significant ridges are encountered.

Termination criteria can be delicate in any multi-dimensional minimisation, and with the simplex, one "cycle" of the algorithm is identified and the vector distance covered in that step is measured. When that distance is less than a specified tolerance then the algorithm halts. It is worth noting that the halting criteria for this algorithm can be fooled by a single anomaly, therefore it is usually good practise to restart the minimisation at the point where it claims to have found the minimum.

A common aspect of all estimation procedures is that they require the user to specify some start values, initial step sizes, and a criterion for convergence. All methods will begin with a particular set of initial estimates, start values, which will be changed in some systematic manner from iteration to iteration; in the first iteration, the step size determines by how much the parameters will be moved. The simplex algorithm is initiated with an initial set of $n + 1$ parameter sets , the $n + 1$ vertices of the simplex. All these vertices must lie in the allowed parameter space constrained by the parameter bounds provided by the user.

A detailed analysis of the particular merits of the algorithm is outside the scope of this work. The algorithm may not be the *best* method in terms of efficiency but it is often used as a "quick and dirty" method when a solution is needed quickly as the algorithm is relatively straightforward to implement. Although simple it is still a realistic illustration of how a search mechanism would work within a PSE. The framework in which it is contained is generic enough for other more efficient algorithms to be used in its place at a later stage.

The original algorithm has been implemented in both the C and Fortran programming languages, and was re-implemented here, for use in the VCCE, in Java using object oriented programming techniques. The benefit of this method as opposed to using one of the original implementations wrapped as a component, was to enable it to be more tightly integrated within the VCCE and to make a pluggable framework so that the evaluation function used by the algorithm can be easily replaced. The algorithm itself within the component can also be replaced to make a completely new control component.

The Java re-implementation of the algorithm, a full listing of which can be seen in appendix B.4.1, makes use of *The Colt Distribution* [54], an open source scientific library for scientific and technical computing in Java. The library has highly optimised and efficient representations for one dimensional and two dimensional matrix types and also implementations of general purpose function types that can be applied automatically to the elements in a matrix. This is the "visitor" design pattern, page 26.

Any function can be used as the evaluation function for the simplex method. The only criteria are to be able to accept as an input parameter a set of values that represent a point in the N-dimensional domain space, and return a single value as the result of the evaluation. The function can be anything from a simple mathematical penalty function through to an entire complex external solver wrapped to appear as a penalty function.

In the prototype implementation this is achieved through the use of a Java interface that all evaluation functions must implement. The interface is very simple with a single method shown in listing 8.1, that returns a `double` as the function evaluation and takes as its input parameter an `Object`. This interface comes from the *Colt* library and is used so that any function implementation based on it can be applied to the elements of a *Colt* matrix object. The input parameter is actually a vector as described in the simplex algorithm, and the *Colt* type `DoubleMatrix1D`, a one dimensional matrix type, is used to represent this.

```
public interface DoubleObjectFunction {
    // Applies the function to an argument.
    public double apply(Object argument);
}
```

Listing 8.1: Evaluation Function Interface

8.2 Penalty Functions

Estimation procedures are generally unconstrained in nature. Therefore, parameters will move around without any regard for whether or not permissible values result. When these non-permissible values occur, a penalty can be assigned to the objective function, in the form of a very large value, to "discourage" the minimisation. As a result, the various estimation procedures usually move away from the regions that produce those functions. However, in some circumstances, the estimation will get stuck, and a very large value of the objective function will result. This could happen, if for example, the regression equation involves taking the *logarithm* of an independent variable which has a value of *zero* for some cases, in which case the *logarithm* cannot be computed.

To constrain a procedure to a specified range, the constraint must be specified in the objective function as a penalty function. This allows control over the permissible values of the parameters. For example, taking the simple function

$$y = -{x_1}^2 - {x_2}^2 \tag{8.1}$$

where the two parameters (x_1 and x_2) are to be constrained by the following bounding conditions

$$\begin{aligned} g_1 &= {x_1}^2 - 17.0x_1 + 66.0 + {x_2}^2 - 5x_2 \\ g_2 &= {x_1}^2 - 10.0x_1 + 41.0 + {x_2}^2 - 10x_2 \\ g_3 &= {x_1}^2 - 4.0x_1 + 45.0 + {x_2}^2 - 14x_2 \\ & \quad \textit{where} \\ g_1 &> 0 \text{ and } g_2 > 0 \text{ and } g_3 > 0 \end{aligned} \tag{8.2}$$

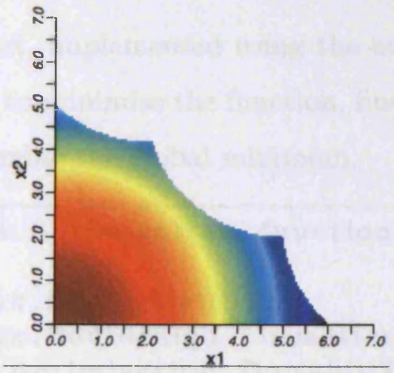

Figure 8.1: $-{x_1}^2 - {x_2}^2$ Plot with Constraints

A large penalty can be assigned to these parameters if this condition is not met. In graphical form the constraints can be seen as the hemispherical "bites" taken out of the plot in figure 8.1. This simple example function is used as a demonstration for the simplex algorithm in section 8.3.

One final aspect to note is that most penalty functions are independent of the search algorithm which allows us to experiment with these different types of penalty with the simplex method component.

8.3 Penalty Function Implementation

This section evaluates the implementation of two penalty functions.

8.3.1 Evaluation Function with Simple Penalty

Consider an implementation of the simple case for the function, $y = -{x_1}^2 - {x_2}^2$ and the boundary conditions g_1, g_2 and g_3, equations 8.1 and 8.3 defined in section 8.2. As can be seen in figure 8.1, there is a function where the evaluation is highest for $(x_1 = 0, x_2 = 0)$ as indicated by the dark red colour and three local minima at approximately, by eye only, $(5.0, 2.0), (3, 2.5), (2.0, 4.0)$ where the lighter blue and yellow colours occur. The simplex component can be used with the evaluation

function and its penalty cost, implemented using the evaluation function interface from listing 8.1, to attempt to minimise the function, finding the values for the local observed minima and determine the global minimum.

```
1  package com.baesystems.optimisation.function;
2
3  import cern.colt.matrix.DoubleMatrix1D;
4  import cern.colt.matrix.doublealgo.Formatter;
5  import com.baesystems.optimisation.DownhillSimplex;
6  import com.baesystems.optimisation.function.DoubleObjectFunction;
7
8  public final class ComplexEvaluationFunk implements
       DoubleObjectFunction {
9      public double apply(Object argument) {
10         DoubleMatrix1D ptry = (DoubleMatrix1D)argument;
11         double x1 = ptry.get(0);
12         double x2 = ptry.get(1);
13         double g1 = x1*x1 - 17.0*x1 + 66.0 + x2*x2 - 5*x2;
14         double g2 = x1*x1 - 10.0*x1 + 41.0 + x2*x2 - 10*x2;
15         double g3 = x1*x1 - 4.0*x1 + 45.0 + x2*x2 - 14*x2;
16         double y;
17
18         if (g1 <= 0) {
19             y = 10000;
20         }
21         else if (g2 <= 0) {
22             y = 10000;
23         }
24         else if (g3 <= 0) {
25             y = 10000;
26         }
27         else {
28             y = -x1*x1 - x2*x2;
29         }
30         return y;
31     }
32 }
```

Listing 8.2: Implementation of Evaluation Function with Penalty

The evaluation function and its penalty can be seen in listing 8.2. In lines 13 to 15, the values for g_1, g_2 and g_3 are first calculated. Lines 18, 21 and 24 compare the calculated values to *zero*, if any of the three values are less than or equal to *zero* then a penalty value is assigned as the return value, in this case 10000, a suitably high value given the bounds on the function, otherwise the evaluated value of the

function itself is returned.

The code illustrates that given equations for the evaluation function, equation 8.1, and the penalty function, equation 8.3, the Java function implementation required for use in the simplex method is straightforward to write and would only require a simple understanding of the Java programming language. Both the evaluation function and the penalty functions can clearly be recognised in the code. In the next section 8.3.2 a more complex penalty function is examined.

8.3.2 Function with FMC Penalty

The function with the simple penalty described in section 8.3.1 has one major drawback. It does not represent any relationships between the three penalty conditions g_1, g_2 and g_3 – they are all evaluated independently. If any of the conditions fail then a single penalty value is returned with no indication which of the three caused the failure and how that affected the evaluation function. A more complicated penalty function might involve interaction between the penalty and evaluation functions. Such a function is considered in this section.

It is useful at this point to introduce some formal notation for the general case *Design Optimisation Problem.* The following expressions are from Optimal Engineering Design [105]. A general form for a simple penalty function to throw up a barrier or distortion of the original optimisation function and thus forces the search toward feasibility, can be written as

$$U_p = U(\bar{x}) + r \sum_i |\langle \phi_i \rangle| + r \sum_j |\psi_j| \tag{8.3}$$

where $U(\bar{x})$ is an optimisation or objective function set up defining the total value in terms of the independent variables.

$$U(x_1, x_2, \ldots, x_n) = maximum \tag{8.4}$$

ψ_i and ϕ_j are respectively equality and inequality constraints which define feasibility

with respect to all possible causes of failure.

$$\psi_i(x_1, x_2, \ldots, x_n) = 0 \qquad i = 1, m \tag{8.5}$$

$$\phi_j(x_1, x_2, \ldots, x_n) \geq 0 \qquad j = 1, p \tag{8.6}$$

r is some fixed large number, and $\langle\alpha\rangle$ is an unsatisfied constraint.

$$\langle\alpha\rangle = \begin{cases} \alpha & \text{if } \alpha \leq 0 \\ 0 & \text{if } \alpha > 0 \end{cases} \tag{8.7}$$

In this expression of the penalty function, unsatisfied constraints are weighted heavily by multiplying the unsatisfied inequality constraints by a large number and adding them to the objective function value in order to discourage these situations. Satisfied inequality constraints are ignored by giving them a *zero* weight. In the example simple penalty function, equation 8.3, a simpler form of this general case equation of the penalty function was formed by ignoring the equality constraints in the equation. The constraints are tested as inequalities greater than zero

This type of general penalty function is severe and it can sometimes cause the search algorithm to stall especially on a boundary of a constraint. The equation 8.3 is only active when the constraints are violated or the search enters an infeasible solution region. This class of penalty functions are called *exterior* penalty functions. *Interior* penalty functions are only active within the feasible region, so that the search is warned when the constraint line is approached.

Another widely used method examined here and implemented as another example of a penalty function for the simplex component is the Fiacco-McCormick penalty function [32]. It can be represented by the expression

$$U_p = U(x_1, x_2, \ldots, x_n) + r^2 \sum \frac{1}{\phi_k^s} + \frac{1}{r} \sum \langle\phi_i\rangle^2 + \frac{1}{r} \sum (\psi_j)^2 \tag{8.8}$$

The symbol $\langle\alpha\rangle$ has the same meaning in Equation 8.7 and indicates an unsatisfied constraint, while s indicates a satisfied one. The value of r is different to the value in the simple penalty function in equation 8.3 where it is simply assigned some fixed large value. Here the value of r is set to a value between 1 and 0. The usual use

of the function is to solve a sequence of optimisation problems where r is reduced rapidly each time toward 0.

This function is a mixed penalty function with both interior and exterior constraints. Consider the *inequality* terms of equation 8.8 in turn:

r^2/ϕ_k^s

The first interior term with r equal to *one* will push the surface up asymptotically[1], moving closer and closer but not touching the constraint line. As the constraint line is approached ϕ_k^s will become smaller, if a false constrained optimum point is found, as the value of r is reduced the effect of $1/\phi_k^s$ is reduced and the false constrained optimum point will approach the true one.

ϕ_i^2/r

The exterior term has a similar effect for the exterior region, as r is reduced a false unconstrained optimum will be shifted toward the true optimum. Typically this reduction of r and the re-evaluation of the function is not expensive as it rarely takes more than three runs with the starting point of the next run being the optimised point found by the previous.

The example objective function from equation 8.1 and the penalty functions from equation 8.3 can be used within the Fiacco-McCormick method. In addition to the original constraints g_1, g_2 and g_3 some additional boundaries are added that will constrain the function to positive values of x_1 and x_2 between 0 and 7 which can be seen on the coloured area of the graph in figure 8.1 and are represented by g_4, g_5, g_6 and g_7 in equation 8.9. As before the constraints are tested as inequalities greater than zero. Only the *inequality* elements of the function will be used ignoring the

[1] *asymptote* A line related to a given curve such that the distance from the line to a point on the surface approaches *zero* as the distance of the point from an origin increases without bound.

last term in the equation 8.8.

$$
\begin{aligned}
g_4 &= 7 - x_1 \\
g_5 &= 7 - x_2 \\
g_6 &= x_1 \\
g_7 &= x_2 \\
&\quad where \\
g_4 &> 0 \text{ and } g_5 > 0 \text{ and } g_6 > 0 \text{ and } g_7 > 0
\end{aligned}
\tag{8.9}
$$

The Java implementation of the Fiacco-McCormick penalty function with the constraints is shown in listing 8.3. The original constraints, equation 8.3, can be seen in lines 10 to 12 and the new constraints, equation 8.9, in lines 13 to 16. The satisfied and unsatisfied inequalities are summed in lines 20 to 27, the penalty function is evaluated in line 28, the objective function is evaluated in line 29 and finally, a value of the objective function plus the penalty is returned.

```
public final class FMCMethodEvaluationFunk implements
   DoubleObjectFunction {
2   private static final double PARAMR = 0.5; // constant r

4   public double apply(Object argument) {
        DoubleMatrix1D ptry = (DoubleMatrix1D)argument;
6       double x1 = ptry.get(0);
        double x2 = ptry.get(1);
8       double penalty;
        double[] constraints = new double[7];
10      constraints[0] = x1*x1 - 17.0*x1 + 66.0 + x2*x2 - 5*x2;
        constraints[1] = x1*x1 - 10.0*x1 + 41.0 + x2*x2 - 10*x2;
12      constraints[2] = x1*x1 - 4.0*x1 + 45.0 + x2*x2 - 14*x2;
        constraints[3] = (7 - x1);
14      constraints[4] = (7 - x2);
        constraints[5] = x1;
16      constraints[6] = x2;
        double violated = 0;
18      double satisfied = 0;

20      for (int i = 0; i < constraints.length; i++) {
            if (constraints[i] <= 0) {
22              violated += constraints[i] * constraints[i];
            }
24          else {
```

```
                satisfied += 1 / constraints[i];
            }
        }
        penalty =  1 / PARAMR * violated + PARAMR * PARAMR /
            satisfied;
        double objective = -x1*x1 - x2*x2;
        return objective + penalty;
    }
}
```

Listing 8.3: Implementation of the Objective with a Fiacco-McCormick Penalty

Like the simple function implementation it can be seen from the code that even with the more complicated expression of the Fiacco-McCormick function, the Java implementation is not overly hard to write. The same format could be used with any arbitrary sets of objective and penalty functions and, given the code in the listing, even someone not fully conversant with Java would be able to modify the expression parts of the code to implement a new function without too much difficulty.

8.4 Simplex Component

8.4.1 Simplex Algorithm Implementation

Within the VCCE the simplex component is functionally similar to the loop component in chapter 6. Its inport values are the parameters to be fed into the evaluation function and the iteration stops when a halting condition is true. In this case this happens when the new value returned from the evaluation is within a predetermined δ value of the previous evaluation, the algorithm tolerance. The main algorithm is implemented as a set of static functions within a class called `DownhillSimplex` which can be seen in appendix B.4.1. The class has three main functions.

8.4.1.1 Simplex `amoeba` function

```
public static final int amoeba(DoubleMatrix2D simplexP,
    DoubleMatrix1D vectorY, double FTOL, DoubleObjectFunction funk
    )
```

Description

This function is the main function in the multi-dimensional minimisation of the function $func(x)$ where x is a vector in N dimensions, by the downhill simplex method of Nelder and Mead. Note: On output `simplexP` and `vectorY` will have been reset to $N+1$ new points all within `FTOL`, the defined fault tolerance, of a minimum function value. The function works by first determining which point is the highest (worse), next-highest, and lowest (best) by looping over points in the simplex. It then computes the fractional range between the highest and the lowest, $(high - low)/(high + low) * 2$ and returns the number of iterations if this is lower than the tolerance. If the function is returning, then the best point and value are put at the front of the vector. If the function doesn't finish then a new iteration is begun and repeated until the tolerance has been met or the number of iterations exceeded. In the iteration, first an extrapolation by a factor of -1 through the face of the simplex across from the high point is performed, i.e. reflect the simplex from the high point, using the function `amoeba` in section 8.4.1.2. The same function is used to extrapolate other contractions and reflections until the halting condition is met.

Parameters

`simplexP`: a matrix, of dimension $(N + 1, N)$ whose rows are vectors which are the vertices of the starting simplex.

`vectorY`: a vector of length $N + 1$, whose components must be preinitialised to the values of the objective function evaluated at the $N + 1$ vertices (rows) of `simplexP`. This initialisation is done by the function in section 8.4.1.3.

`FTOL`: the fractional convergence tolerance to be achieved in the function value.

`funk`: the function to be minimised.

Return value

The number of iterations taken to halt the algorithm either by finding a solution within tolerance or exceeding the maximum number of iterations.

8.4.1.2 Simplex `amoebaTry` function

```
private static double amoebaTry(DoubleMatrix2D simplexP,
   DoubleMatrix1D vectorY, DoubleMatrix1D pSum,
   DoubleObjectFunction funk, int indexHi, double fac)
```

Description

This function extrapolates by a factor through the face of the simplex across from the highest point, evaluates the new point with the objective function, and replaces the highest point if the new point is better.

Parameters

`simplexP`: a matrix representing the simplex.

`vectorY`: the start vector

`pSum`: the simplex column sum vector

`funk`: the function to be evaluated

`index`: the index in vectorY of the highest value

`fac`: factor of extrapolation

Return value

The computed value of the objective function

8.4.1.3 Simplex `initialise` function

```
public static final DoubleMatrix1D initVectorY(DoubleMatrix2D
   simplexP, DoubleObjectFunction funk)
```

Description

This function initialises and returns a vector of dimension $N + 1$ where its contents are calculated by evaluating the function to be minimising at the $N + 1$ vertices (rows) of the simplex. To evaluate the function the code calls the `DoubleObjectFunction` interface function `apply` on `funk`.

Parameters

`simplexP`: the starting simplex

`funk`: the objective function to be evaluated

Return value

An initialised vector Y of size $N + 1$

8.4.2 Simplex Component Implementation

The static functions from the class `DownhillSimplex` described in the last section provide the functionality for executing the simplex algorithm. Embedding that functionality as a component within the VCCE framework is done by a wrapper class that uses the static methods. The wrapper class is called `SimplexComponent`. `SimplexComponent` is instantiated and executed from the standard proxy component, `SimpleProxy`, and the Java execution model, `ExtendedJavaExecution` section 4.5.2.2. The XML component definition can be seen in appendix B.4.2, and the execution tags that load the `SimplexComponent` can be seen below

```
<execution id="software" type="bytecode" value="extended">
  <type id="architecture" value="serial"/>
  <type id="class" value="com.baesystems.components.
     SimplexComponent"/>
  <type id="source" value="file:///home/compdata/cardiff/project/
     src/com/baesystems/components/SimplexComponent.java"/>
  <type id="classpath" value="/home/compdata/project/classes"/>
</execution>
<execution id="platform">
  <type id="java" value="jdk1.2"/>
</execution>
```

Line 3 specifies the class name for this new class that will get instantiated when the proxy is created. The `execute` method implementation within `SimplexComponent` which will run the simplex algorithm can be seen in listing 8.4. Lines 5 to 8 create a *unit* matrix e that is then used to initialise the simplex P using $P_i = P0 + \lambda e_i$ where $P0$ is the start point for the minimisation and λ is a "guess" at the problem's characteristic length scale, generally a suitable small value such as 5×10^{-3}. Finally in line 23, the `DownhillSimplex` method `initVectorY` is called, section 8.4.1.3, to set up the algorithm and in line 24 call the main `amoeba` method, section 8.4.1.1, to perform the minimisation.

```
DoubleMatrix2D simplexP = factory2D_.make(3, 2);
DoubleMatrix1D vectorY;

// initialise the unit vector e
DoubleMatrix2D unit = factory2D_.make(3, 2, 0.0);
unit.set(1, 0, 1.0);
unit.set(2, 0, 0.5);
unit.set(2, 1, Math.sqrt(0.75));

// Pi = P0 + λe_i
// create each row and store the start points in two temp arrays
double[] startX = new double[simplexP.rows()];
double[] startY = new double[simplexP.rows()];
for (int i = 0; i < simplexP.rows(); i++) {
    double xPoint = p0_.get(0) + unit.get(i, 0) * lambda_;
    double yPoint = p0_.get(1) + unit.get(i, 1) * lambda_;
    simplexP.set(i, 0, xPoint);
    simplexP.set(i, 1, yPoint);
    startX[i] = xPoint;
    startY[i] = yPoint;
}

vectorY = DownhillSimplex.initVectorY(simplexP, solveFunk_);
int iterations = DownhillSimplex.amoeba(simplexP, vectorY
    , 0.0001, solveFunk_, maxIter_);
```

Listing 8.4: Evaluating the Simplex

The parameter values defining the `lambda` value, coordinates of the start point `p0_` and the maximum number of iterations `maxIter_` are set as input parameters from the XML component definition using the reflection mechanism for parameters, described in section 4.5.2.2, and corresponding "setter" methods in the `SimplexComponent`. For example the parameter `lambda` in the XML definition can be seen in the listing below

```
<inport id="3" parameter="Lambda" type="float" value="0.5">
</inport>
```

the corresponding "setter" function from `SimplexComponent` is

```
public void setLambda(Object lambda) {
    lambda_ = ((Float)lambda).doubleValue();
}
```

The final input parameter which represents the evaluation function `solveFunk_`, is also set in the XML definition but in this case, another feature of the input tag mechanism which allows us to specify alternatives for the values of the input rather than a free format value is used. The evaluation function specifies a Java class which implements the function equation and the `DoubleObjectFunction` interface. For the example objective function $y = -{x_1}^2 - {x_2}^2$ and the penalty function implementations the XML can be seen below.

```
<inport id="1" parameter="SolveFunction" type="option" value="com
    .baesystems.optimisation.function.SimpleEvaluationFunk">
2   <option name="simple" value="com.baesystems.optimisation.
      function.SimpleEvaluationFunk">
    </option>
4   <option name="complex" value="com.baesystems.optimisation.
      function.ComplexEvaluationFunk">
    </option>
6   <option name="Fiacco-McCormick" value="com.baesystems.
      optimisation.function.FMCMethodEvaluationFunk">
    </option>
8 </inport>
```

This specifies three Java function classes:

`SimpleEvaluationFunk`

: This is a objective function with no constraints, it implements $y = -{x_1}^2 - {x_2}^2$.

`ComplexEvaluationFunk`

: This is the objective function plus the simple constraints, specified in section 8.3.1.

`FMCMethodEvaluationFunk`

: This function implements the Fiacco-McCormick function, specified in section 8.3.2.

The default value for the options is specified in the inport tag definition, line 1 in the listing above, and in this case is `SimpleEvaluationFunk`. The corresponding "setter" function in the `SimplexComponent` class to set the function can be seen

below. It uses the Java class loader to instantiate the function object from its string name representation.

```
public void setSolveFunction(Object funk) {
    if (String.class.isInstance(funk)) {
        solveFunk_ = (DoubleObjectFunction)(Class.forName((String
            )funk)).newInstance();
}
```

8.4.3 Simplex Component Usage

In normal use the simplex algorithm is run multiple times with different values of $P0$, the starting point. A single execution may find a minimum given a large enough number of iterations but better results are found if less steps and multiple executions are used. For this example the loop component is used to control the multiple executions of the simplex by looping over either the x_1 or x_2 coordinate of $P0$. The task graph can be seen in figure 8.2

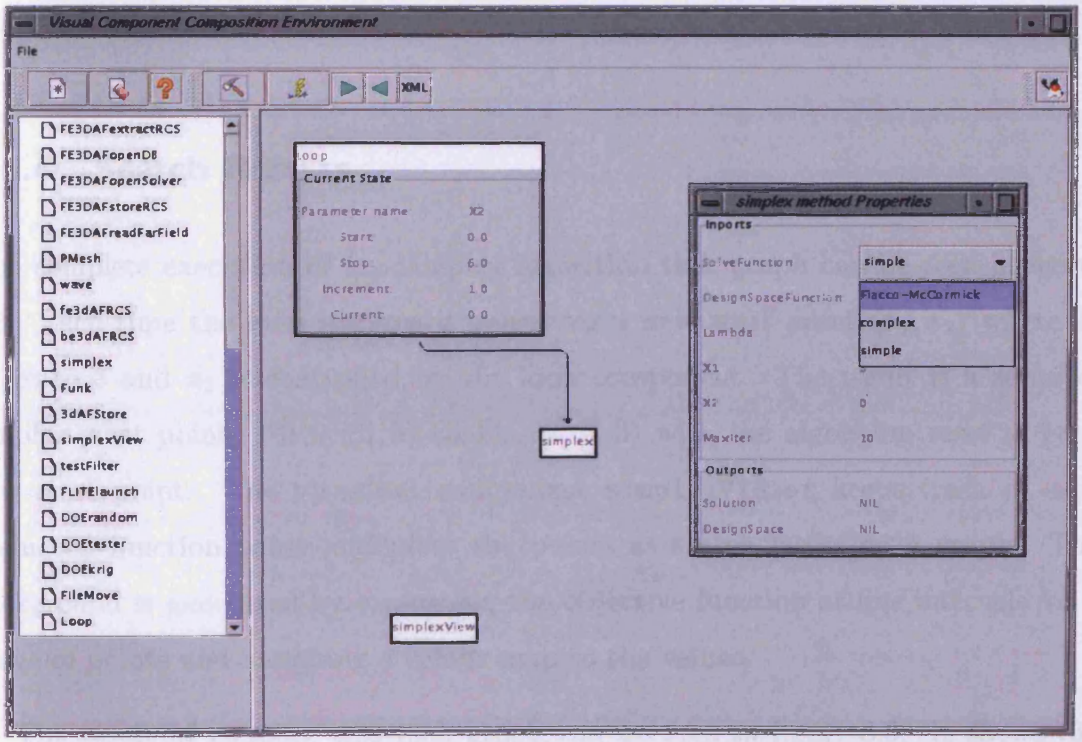

Figure 8.2: Choosing the Function on the Simplex Component

The figure shows the simplex component, a loop component set to iterate over the x_2 parameter from 0 to 6 in steps of 1, and a component called *simplexView* which is responsible for displaying graphically a two dimensional domain space and the search path taken by each iteration of the simplex algorithm. The other panel displayed in the figure is an editor for the parameters in a proxy component, in this case the `SimplexComponent`.

`ExtendedVisualComponent` is an extension to the `SimpleVisualComponent`, section 4.5.1, along the same lines as the `ControlVisualComponent`, section 6.4.2. Whereas the simple component was only capable of displaying the name of a component and its execution state, the extended version provides a simple user interface for setting and displaying parameter values. Parameters are edited in text fields apart from parameters with preset options like the evaluation function, these are represented by "drop down" list components which can be seen in the previous figure. The `ExtendedVisualComponent` implementation uses the inports and outports from the proxy component to dynamically build the user interface for the parameter values.

8.4.4 Search Results

The complete execution of the simplex algorithm task graph can be seen in figure 8.3. Each time the loop iterates it generates a new start point (x_1, x_2) where x_1 is set to 3 and x_2 is controlled by the loop component. The result is a series of simplex start points $P0 = (3,0), (3,1), \ldots, (3,5)$ with the algorithm reset at each new start point. The visualiser component `simplexViewer` keeps track of each evaluated function value and plots the points as search paths on a graph. The background is generated by evaluating the objective function at fine intervals for a range of points and assigning a colour map to the values.

The result of running this task graph can be seen in figure 8.3. It shows the domain space plot as coloured bands ranging from dark red for high values to lighter greens and blues for the minimum values. The 6 simplex minimisations can be seen

as a series of trace paths starting from each of the start points. Each simplex can be seen converging on local minima. To keep the display clear the number of iteration steps per simplex run has been set very low at 10 steps.

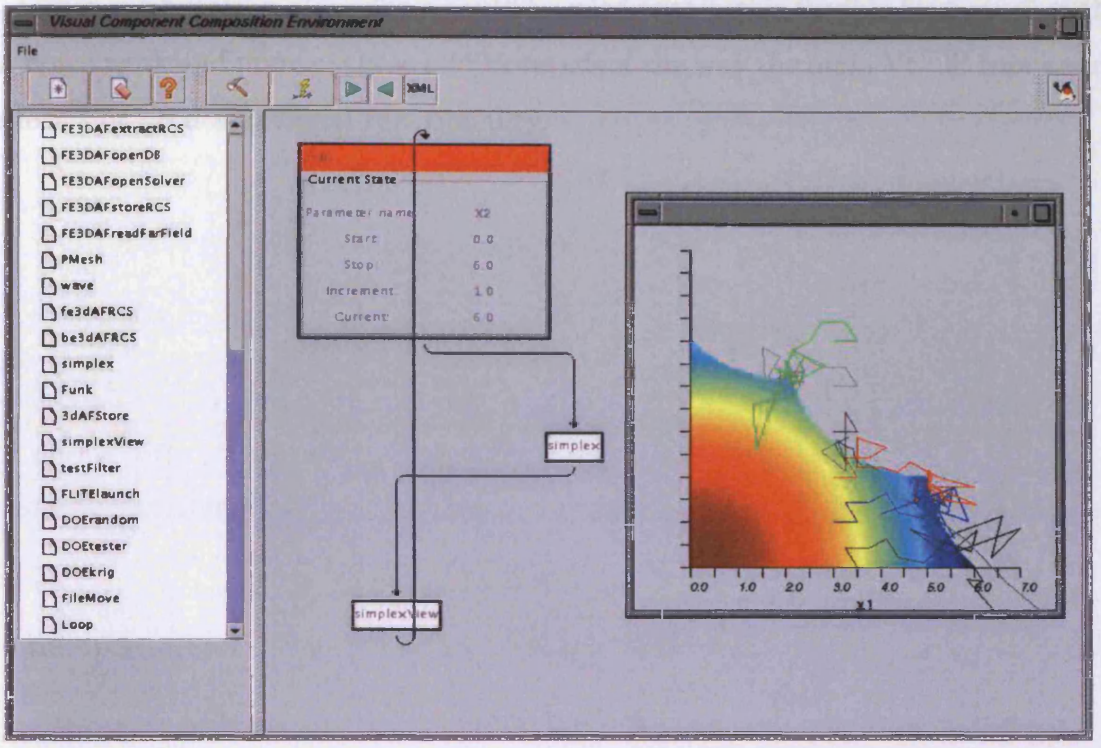

Figure 8.3: Completed Simplex with Visualisation

8.5 Conclusion

This simple function evaluation with various penalty functions has illustrated how the simplex algorithm can work as a component within the VCCE. Writing objective and penalty functions can be straightforward given the function interface and framework. A scientist could use the mechanism to add their own functions to be evaluated and this could even be used to wrap a large solver for a more complex parameter evaluation or design of experiments. For instance, the *BE2D* solver could be executed within a function object using the same execution mechanism as the *BE2D* component in chapter 5. The input to the function would be a simplex in three dimensions because there are two inputs, wave incident angle and frequency.

The function would execute the solver and return a fitness value for the objective function based on some evaluation of solver output.

Other optimisation algorithms could be used within this flexible framework without much work and none of these additions affect the way the main VCCE framework performs or need additional functionality.

CHAPTER 9

Recent Related Work

Although the work presented in this thesis was finished at the end of 2001 the completion of this dissertation has taken considerably longer, due in part to work constraints. Since then research in the area of PSEs has moved at a very rapid pace not least because of the influence of the grid computing community. The research discussed in relation to this thesis in chapter 2 only covers the time frame of the original work. This chapter relates advances in research since that date to the work here and puts into the context of current thinking some of the decisions made. The chapter is broken down into four sections: middleware advances; computing environments; component models; and PSE usage.

9.1 Advances in Middleware: The Advent of Grid Computing

The distributed computing middleware field has changed rapidly in the period since this work was finished. CORBA, the middleware chosen in implementing a distributed communication framework in this work, is now a mature platform with

many implementations by different vendors. It is still used widely among industrial projects but has fallen out of favour with the scientific research community due in part to its unsuitability for handling large data sets.

Since the publication of Foster and Kesselman's "Grid Blueprint" [35], grid technologies and in particular Globus [34] have become the middleware of choice for distributed computing research. The Globus Alliance [47] is developing some of the technologies needed to build computational grids. The result of this work is the Globus Toolkit, a set of services and software libraries to support grids and grid applications. The Toolkit includes software for: security; information infrastructure; resource management; data management; communication; fault detection; and portability.

Globus is not the only grid computing platform, UNICORE [111], discussed in section 2.3, also has matured and is now widely used across Europe. UNICORE v3.0 is a Java based grid computing infrastructure for accessing resources over the Internet. UNICORE Plus is a research project and a commercial product, UNICOREpro, that intends to extend UNICORE to provide: systems administration; modelling of resources; application specific extensions; advanced data management and computational steering.

There are many grid infrastructure projects around the world making use of the new grid technologies. The EUROGRID Project (Application Testbed for European GRID Computing) [89] is a project to build a European domain specific grid based on the UNICORE system and encompassing UNICORE Plus extensions as an alternative to the US based grid projects which mainly use Globus. The main aims are to develop tools for easy access to HPC resources in the application areas of: biomolecular simulations; weather forecasting; and mechanical engineering.

The DataGrid project [43] aims to enable new scientific exploration through the intensive computation and analysis of very large shared databases across distributed scientific communities. It concentrates on storage facilities for very large data sets in fields such as: high energy physics; biology and medical imaging; earth observations. DataGrid makes use of the Globus toolkit and infrastructure.

EGEE: Enabling Grids for E-science in Europe [29] is a new project that will build on the experiences and software developed for the DataGrid project to develop a robust and reliable grid infrastructure in Europe.

In America, prototypes for the National Technology Grid are being built by the National Partnership for Advanced Computational Infrastructure (NPACI) [84] and the National Computational Science Alliance (NCSA) [83]. NASA is using Globus to build the NASA Information Power Grid [82]. Lawrence Livermore, Los Alamos and Sandia National Laboratories are using Globus to build a testbed for resource management under the Accelerated Strategic Computing Initiative (ASCI). As with the European infrastructure projects outlined here these projects are all attempting to build computational grids for collaborative scientific computing.

All of these grid infrastructure projects could, and in many cases are, being used to host component based PSEs such as the one described in this thesis. Replacing the CORBA middleware layer with a grid solution is discussed in the future work section of the next chapter.

9.2 Grid Computing Environments and Portals

Engineers and scientists now have a wide choice of computational modules and systems available, enough so that navigating this large design space has become its own challenge. A survey of twenty eight different PSEs by Fox, Gannon and Thomas [39], as part of the Global Grid Forum's Grid Computing Environments working group, indicates that such environments generally provide "some back-end computational resources, and convenient access to their capabilities". Furthermore, work flow features significantly in both of these descriptions. In many cases, access to data resources is also provided in a similar way to computational ones. Often PSE and Grid Computing Environment (GCE) is used interchangeably, as PSE research predates the existence of grid infrastructure.

A number of the PSEs first discussed in chapter 2 are still actively being devel-

oped and have now been extended to work within a grid computing infrastructure. These include SCIRun [65] and Netsolve, (which has become GridSolve [2]), both of which make use of the Globus Toolkit. Interestingly grid-enabled PSEs are often of a more hierarchical or layered nature than the PSE built in this work. For example GridSolve is built upon Globus and provides services which can be used by higher level environments such as SCIRun.

In addition, there are emerging intermediary level systems that provide access to low level grid resources and additional functionality on top of those resources. The Commodity Grid Kit (CoG) [72], is a project which provides different language implementations of reusable commodity frameworks including Java, Python and CORBA. These frameworks provide high level programmatic access to low level grid resources and services, insulating the PSE and portal programmer from a lot of the low level code.

GridLab [3] is another middleware project that is attempting to provide an abstraction layer to grid application programmers, including PSE developers, above the level of the Globus Toolkit or UNICORE services. GridLab is developing the Grid Application Toolkit (GAT), an API and services that provide a uniform interface to grid resources and services.

TENT [100] is a distributed work flow management system for engineering applications, and like the PSE in this thesis it has a graphical user interface for composing work flows from components. The components are formed by wrapping codes, and the wrapper, like the proxy components in the work here, is responsible for the transfer of data and control between components using channels. The wrapper allows the integration of any code without modification so that any type of legacy code can be handled; components are stored in a repository; a user can configure work flow by moving components from the repository to the editor in a drag-and-drop manner; work flow can be executed, modified, stored, or restored; the user can set the parameters of each code, can modify them during the run and thus perform numerical experiments; results can be visualised by putting appropriate tools into the work flow. Like GridSolve, TENT is making use of the CoG kit to provide access to a

grid environment.

Li and Baker [76] provide an extensive review of various grid portals currently available. Based on their definition, a grid portal provides "end users with a customised view of software and hardware resources specific to their particular problem domains". In some ways, this definition shares common themes with that of a GCE. The focus in their work is primarily on web-based portals which differ from the focus in the work presented in this thesis, where the focus is on a more graphical environment. However, they emphasise three generations of portal technologies:

- Generation 1 focuses on a graphical interface and the use of the Globus toolkit, primarily tightly coupled with Globus-based grid middleware tools.
- Generation 2 portals are aimed at specifying "portlets", essentially user customisable services which can run on top of a web server. Grid Portlets are intended to be independent components that can utilise a number of different grid middleware toolkits. This is the current state of affairs in portals, with GridSphere [87] being a commonly used toolkit to support the construction of such Portlets.
- Generation 3 involves the extension of the Portlet idea with semantic annotations. Our component model supports a concept similar to Portlets, in that each individual component present within a tool repository has an XML-based interface. The XML description can also be extended with semantic properties if required.

The GridLab Portal is implemented using the GridSphere portal framework which in turn is built using other toolkits and frameworks such as the CoG kit, Globus toolkit and MyProxy [88], a repository framework for handling security credentials.

Other portals include the Astrophysics Simulation Collaboratory [98] in America and the AstroGrid [74] portal in the UK. The first is a portal for large scale simulations of relativistic astrophysics and the second is a portal that aims to provide a

data-grid to all UK astronomical observatory data.

Based on the surveys above, the VCCE may be classified as a graphical Grid Computing Environment, both a problem solving and a programming environment. It provides a user portal to enable the composition of scientific applications from components and an execution environment built using CORBA as the middleware layer.

9.3 Component and Work Flow Models

Since this work first identified and published a description of the XML based component model [95] in 1999 many other projects have used similar ideas. It is now almost standard practise for component based systems to define specifications and interfaces in XML syntax. There are many other prominent projects that use similar component definitions, not least of which are Web Services and Globus OGSA [36] grid services which both use an interface definition language called WSDL [115] and a message protocol called SOAP [116] written in XML.

A recent development in the area of work flows is the addition of a new research group, Work Flow Management (WFM-RG), in the Global Grid Forum to explore the application work flows and their execution in a grid environment. An ongoing survey of scientific work flows [107] has identified the current active work flow research projects and the models and languages they use. One of the goals of the WFM-RG is to get a consensus on work flow standards.

Current proposed languages for work flow include the Business Process Language for Web Services (BPEL) [5] and Web Services Flow Language (WSFL) [75] both of which are work flow languages specifically designed for use with web services technology. Syntactically they are very similar to the work flow language presented in this thesis providing specification for services, which can be thought of in terms of this work as components, and the connections between them.

DAGMan [23] is a work flow meta-scheduling system for the Condor [41] schedul-

ing system that can schedule dependant jobs, specified in directed acyclic graph onto a grid infrastructure. The syntax is not XML based and the dependencies have to be *acyclic* unlike the model in this thesis which allows cyclic dependencies.

GSFL [71] is a proposed work flow framework for grid services that will attempt to leverage advances in web services work flow and adapt them form use in the Open Grid Services Infrastructure (OGSI).

Triana [110] is a distributed PSE which uses a component and work flow model based directly on the language definition in this thesis. Unlike the BPEL and WSFL languages, the model presented here and used in Triana is independent of the underlying middleware technology and in addition allows cyclic dependencies between tasks. Loops and halting conditions in both BPEL and WSFL are represented in the language constructs themselves whereas, with the model presented in this work, these are handled by specific components within the work flow. Triana is discussed further in the future work chapter as many of the ideas put forward here are implemented in that project.

ICENI [42] is another PSE or work flow environment that uses XML pervasively throughout its work flow specification and component model. The language is similar to the work presented here and that used by many of the other projects in that it defines components, their properties and the connections between them.

XCAT [70] is a project at Indiana University to implement the Common Component Architecture (CCA), first discussed in chapter 4, onto a grid infrastructure. The CCA specification describes the construction of portable software components that may be re-used in any CCA compliant runtime framework. There are many different frameworks envisioned from simple standalone applications to parallel systems. The XCAT framework is designed to support applications built from components that are distributed over a computational grid of resources and distributed services. It is based on Globus and the CoG kit for its core security and remote task creation, and it uses RMI over XSOAP for its communication.

9.4 PSEs in Science and Engineering

Since the work in this thesis was finished PSEs have advanced rapidly and are now being used for real scientific research. Numerical optimisation techniques within PSEs such as those described and prototyped in chapter 8 can now be done on a large scale with PSEs and grid infrastructure.

An example is a grid-based approach to the validation and testing of lubrication models [49]. This work uses gViz [51], an extension to the Iris Explorer workflow tool and existing visualisation systems to enable visualisation and computational steering in grid computing environments. The gViz project use XML in many areas [12]: data representation; visualisation presentation; visualisation application description; and audit trail for project history. The experiment performs a parameter space search, in a similar manner to the design of experiments example in chapter 8 of this thesis, to optimise a complex lubrication model solver, wrapped as a penalty function within a simplex algorithm over 36 parameters. The whole experiment is run across multiple parallel nodes and can be interactively visualised and remotely steered. This experiment is a realisation of many of the ideas presented here.

The Fraunhofer Research Grid [55] are developing a PSE based on the Fraunhofer computing grid. Like the PSE in this work the system is component based with a set of XML-based definition languages collectively called the Grid Application Definition Language (GADL). There are a subset of four languages for: resource description; job description; software component interfaces; data and data flow description. Unlike many work flow models, the Fraunhofer work use Petri nets instead of DAGs to represent the dependencies between components. DAGs model the behaviour of a system but not the state, so it is not possible to model a loop directly. Petri nets allow state to be represented so loops and controls can be modelled directly. The work in this thesis bypasses this need by delegating control a specific control components within the work flow.

Taverna [90] is a PSE developed as part of the UK e-Science MyGrid project [48] to enable *in silico* experiments in biology on the grid. Taverna is based on the web

services standards and provides XML-based web services definitions and a work flow execution environment.

Kepler [4] is a general purpose PSE with a graphical user interface and data flow model from Ptolemy II [94], a set of Java packages that support concurrent modelling and design. Kepler provides a work flow environment, representing work flow in specific XML syntax called Model Markup Language (MOML). Like the syntax outlined in this dissertation, work flows can be hierarchical in definition. The focus of Kepler is *actors*, where an actor is a re-usable component that communicates with other actors through channels. Kepler works in two domains: either a process network where actors model a series of processes, communicating by messages through channels; or in a synchronous data flow domain where the communication along the channels is a data flow.

The Chimera Virtual Data System (VDS) [8] is part of the GriPhyN project. Chimera has a very data centric approach to work flow, choosing to focus on the result of a computation and working backward generating intermediate virtual data stages until a point where the actual data is available. At this point the work flow can be executed to generate the required result. The Chimera system consists of four primary components: a Virtual Data Language, used to describe virtual data results; a Virtual Data Catalogue, used to store virtual data entries; an Abstract Planner, which resolves all dependencies for a virtual data product and forms a location and existence independent plan; a Concrete Planner, which maps an abstract, logical plan onto concrete, physical grid resources. The concrete work flow is expressed in the DAGMan format and scheduled by the Condor scheduling system.

Cactus [18] is a modular, parallel, command line framework for solving systems of partial differential equations from many disciplines of science and engineering.

As this chapter illustrates the field of PSE research, and in particular distributed PSEs and computational grids, has moved at a rapid pace over the two and a half years since the work in this dissertation was completed. Many of the ideas discussed and prototyped here are now a reality in production systems. The next chapter concludes this dissertation and focuses on the successes and failures of the work.

Some of the descriptions in this related work chapter will be discussed with reference to future work and directions.

CHAPTER 10

Conclusion and Future Work

10.1 Introduction

This thesis has examined the use of Problem Solving Environments (PSEs) for creating scientific work flows for use by scientists and engineers in their day-to-day working lives. It has emphasised, through the building and use of several prototype environments, how visual programming and software component-based techniques can be used to build PSEs. The work has concentrated on XML-based definitions for both components and a work flow within the environment. This is now accepted standard practise within the PSE and grid computing communities. Techniques for wrapping "legacy" codes as components for reuse within work flow and the use of PSEs for non-linear optimisation were discussed. The use of CORBA as the chosen middleware layer has proved to be unfashionable in the longer term, with PSE designers now choosing to use grid computing technologies. The component-based design and visual programming interface of the VCCE can still be reused by building on top of the new grid computing infrastructure.

The remainder of this final chapter, evaluates each of the major sections discussed in this work examining the successes and failures. Where areas are lacking possible

solutions are discussed. This chapter also provides the opportunity to assess future directions, including research groups that are still using, or have built on, the work described in this dissertation.

10.2 Critical Evaluation

10.2.1 VCCE Prototype

The main result of the work undertaken in this thesis is the Visual Component Composition Environment (VCCE) prototype PSE and its component based framework. When this project first started there were a number of different visual programming tools that potentially could have been used such as AVS, Khoros and Iris Explorer. None of these were deemed suitable for a number of reasons:

- Although most of these tools run on multiple platforms they are not platform independent. A different version of each tool would be needed for each supported platform. The VCCE is written in Java and is platform independent.
- The VCCE is open source and freely available. The other tools are commercial products and so may have licence restrictions.
- The VCCE has an XML-based component model which can be converted into other XML model formats through XSLT translations. The component models of the other tools are proprietary. In addition, moving components from one operating system platform to another within the same application framework can be problematic.

The VCCE interface, although simple, is complete enough to be able to perform most of the functionality needed in composing work flow from components. The user is able to: select the component from a repository, which is dynamically loaded at runtime; instantiate the component onto a workspace; connect the component to other components; and execute the completed work flow via an internal scheduler.

Both simple loops and more complex non-linear loops are supported. A simple loop allows the user to iterate over a sub-section of the task graph a predetermined number of times. This loop can be used to perform "parameter runs", executing a solver for instance, a number of times so that the effect of varying input parameters can be observed. See chapter 6. A non-linear loop is one that does not iterate in a linear step but rather modifies the step value dependant upon some condition. It is used in chapter 8 to minimise a mathematical function.

The process of developing the prototype was iterative with new versions containing progressively more functionality. Each of the previous prototypes was discarded. The project adhered to software engineering best practises, which throughout the length of the development cycle proved worthwhile again and again. Due to the fact that the framework was written to well defined interfaces, extending the functionality of components and the framework did not require much extra work. For instance, in chapter 6 the component proxy, `SimpleProxy`, needed to be replaced with a new proxy that would provide control functionality for the new loop component. Creating the new component was simply a matter of implementing the `ProxyInterface` in a new class called `ControlProxy`. Because both component proxies implement the same interface they can be used in the same framework without any changes. The internal scheduler had to be changed to handle the new control component functionality but because that also is written to an interface replacing the scheduler did not change the framework at all. In fact, the future work, section 10.4, will show how this extensibility could be leveraged to build components for the VCCE, using different middleware technologies such as grid computing or web services.

10.2.2 XML-Based Component Model

The XML-based component model is perhaps the single most important idea to come out of this work. At the time this work started, XML was gaining momentum in the development and research community, as a platform and language independent data representation format. There were a number of emerging XML schema for different purposes, such as mathematical markup, chemical markup, etc. Two

schema for component definition existed, IBM's BeanML and W3C's OSD. Both of these provided definitions for components, but neither could be used as a data flow language to represent the connecting data "channels" between components in a generic manner. BeanML can be used to represent connections but is JavaBean specific. The "channel" would have to be a method on the child JavaBean in the connection. OSD can represent dependencies between software components but is used to specify software installation and version dependencies.

The component model presented in this thesis took ideas from both languages to specify a data or work flow language that can specify component information such as: component name and name space scope, alternative name and ID - all of which can be used to uniquely identify a component and instances of that component; component input and output interfaces in the form of typed data inports and outports; a component execution model which helps the PSE to run the component - this may include the executable name, programming language, dependencies, or in the case of CORBA host name and name server; component help files.

Using XML as the language for defining the component interfaces has proved to be a popular decision. XML component models are now widely used across a large range of software projects. From the smallest software "plugins" to an editor, through the many PSE projects, to service-based infrastructure projects such as web services and grid services.

The XML component model presented here, is closely related to the XML-based formats for specifying web and grid services, W3C's WSDL. The data flow model is very close to the flow languages that are used to connect the services such as BPEL4J, WSFL or GSFL. In fact due to the nature of XML and its roots in SGML it is relatively straightforward to translate between similar formats. This translation can be used to connect work and data flows between systems. The future work, section 10.4, has an example of using this translation.

Development of the component model has not stopped with the end of this project. It has now been extended and provides the basis for the component definition and connection language in another PSE project, called Triana.

10.2.3 Distributed Middleware

With the benefit of hindsight and examining recent advances in middleware technology, see chapter 9, it would appear that the choice of CORBA as the middleware layer was a mistake. These advances in middleware technology inevitably affect the way the PSE described in this dissertation is viewed, however these should not be thought of as a conflict. At the start of this project it was not at all obvious that grid computing, then just at the start of its research, would become the de facto standard for distributed scientific middleware. Writing the middleware layer from scratch, in hindsight, was unnecessary but the solution did work. CORBA wrapped codes such as BE3D and FE3D, chapter 7, ran successfully from within the VCCE on remote machines. The problems with large data transfer and CORBA were alleviated through the use of the Action Factory that kept the data on the server side, only returning a reference to the data to the VCCE client.

The proxy components used to represent the distributed components within the VCCE were a good solution. Separating the *instantiation* code from the *execution* code made the system appear, to the user at least, to be faster. The extensibility of the proxy framework allowed the CORBA execution proxy, `ActionFactoryExcution` section 7.2.1, to be created easily and used within the VCCE. The instantiation of the component included the CORBA name service handshaking, and reference resolving. The execution was a remote method call on the resolved distributed component. This example could be extended to use both web services through a web service invocation solution such as Axis or grid services through the OGSA framework. The VCCE framework and component model would not need to be extended to incorporate the new grid-based components, apart from the issue of security which is discussed in the future work, section 10.4. A discussion of grid-based PSEs can be found here [73].

10.2.4 Non-Linear Optimisation Techniques

Chapter 8 provided an interesting use of the VCCE to enable the user to perform non-linear optimisation within a visual programming environment. A new loop component was designed into which optimising algorithms such as the simplex algorithm can be inserted. The optimising loop evaluates a *cost* or *objective* function trying to minimise that function.

As with the rest of the system, the loop and function code was designed to be extensible. The simplex algorithm could be replaced by another suitable, more complex algorithm. The function is an interface with a single method. Any implementation of that interface can be used in the loop component. The XML component definition can be used to specify the Java executable class name for the function so it is dynamically loaded at run time. Several simple examples were shown with constraint-based cost functions and the more complex objective function with Fiacco-McCormick penalty. An illustration of the loop in use, minimising the function $y = -{x_1}^2 - {x_2}^2$, was also shown.

This simple use case can be extended to provide a real solution by wrapping a complex solver code such as FE3D in a cost function. At each iteration of the loop, the solver would be executed and a fitness function evaluated over the parameters the scientist is interested in. This is a common task for engineers in particular. The industrial partner in this work, BAE SYSTEMS, uses techniques such as this in "design of experiments" to optimise among other things, airflow over wing calculations for aircraft models.

10.3 Other users of the VCCE

The VCCE has been used by Southampton University as a visual programming front end to their engineering PSE in the GEODISE project, see [103, 102].

BAE SYSTEMS continue to use the VCCE with the solver components implemented here and a component that was capable of executing command line opera-

tions specified in the XML component definition. This is used as a visual scripting language.

The XML component model has become the basis of the model used in the GridOneD Triana PSE.

The experience gained in developing the VCCE is being used in a number of e-Science projects such as GridLab and GridOneD.

10.4 Other Issues and Future Work

This section of the dissertation could easily be as long as the rest of the chapters put together. The VCCE was the result approximately three "man years" of research and development. In programming terms this is not a large amount of time so this work concentrated on certain areas such as component design and PSE use cases. The areas covered by PSEs today is enormous. A huge amount of effort is going into PSE research and the closely related subject of grid computing. The current UK e-Science funding for projects in these areas is evidence of this.

Some of the many issues not addressed in the VCCE include, in no particular order: support for fault tolerance; control branching components; debugging and state management of components; user authentication and security; component editors; intelligent advisers; component integrity and version checking; support for multiple middleware layers and protocols; implementation of hierarchical components; external scheduling systems; resource reservation; experiment reproducibility and data provenance; process monitoring; process migration; scalability; complex component user interfaces; components as computational services; and complex visualisation tools.

Fault tolerance is a very important aspect of any production system that is to be used in the real world. The VCCE was only designed to be a prototype and so fault tolerance was not a design consideration. Fault tolerance can be built into a PSE at many different levels: the availability of components or services - a component

cannot be used if it is not available - where replication of components in multiple repositories can help; reliability of connections in data channels - error checking on received data and successfully received messages might be needed; graceful failure - an error in a single component should not crash the entire work flow.

The VCCE has a control construct for looping but no control construct component implementing an *if...then* logical branch. A branching component is fairly simple to implement but a decision needs to be made whether the control logic should be at component level or within the flow connection language itself. Languages such as WSFL for choreographing the connections between web services have control such as looping and logical branching at the language level. The Triana PSE implements the logic in purpose built components. The trade off is that support at the language level does not require special purpose components, but cannot implement all possible constructs, such as *while...do*, *repeat..until* or complex *if...then* statements, without bloating the language representation. Including the ability for all control at the language level would turn a simple flow language into an almost complete programming language. This would invalidate one of the main goals of PSEs, abstraction of complexity.

Debugging of distributed systems is a notoriously complex task. The VCCE has no support for this but a commonly used method in distributed Java systems is to use logging. Log messages can have priorities such as debug, warning or error and can be sent from remote processes to a logging server. State management of components is also not considered in the VCCE. If a work flow is stopped mid-way through its execution, either through error or user intervention, then the state of all the components in the flow could be used to restart the work flow later. Internal component state requires the component builder to implement checkpointing. This will only be worth while for components that take a long time to run as it may be quicker to re-run the component than write the checkpointing code. Where there is no checkpointing implemented, given a work flow where the current point of execution is known, i.e. which components have been executed already and which have not, and a data object for the current position in the workflow, a course grained

level of checkpointing is available. The work flow can be restarted at the point it was stopped by passing the data object into the first unexecuted component. This system has been implemented in Triana.

Security is important in PSEs for a number of reasons: if the PSE is able to access remote resources then user authentication will be needed; some components may be of restricted use or contain licence restrictions. Some web services and all grid services are secured using Grid Security Infrastructure (GSI). This implements X.509 certification where a user has to generate a credential that is used as a token and passed to the service as a verification of identity. It is possible to implement security at individual component level through the use of mechanisms such as SSL. This is not a scalable solution and any production PSE should build security into the infrastructure.

Support tools such as component editors and wizards for creating components are a necessary part of a full PSE. In the VCCE components were custom created by hand using a text editor for the XML definitions. This would not be attractive in a system that is to be used by someone not familiar with the intricate details, one of the goals of PSEs. A component model editor would allow a user to specify the component model using tags, and also act as a wizard and enable customisation. An editor would work in a similar manner to an HTML editor, where a user is presented with a menu based choice of available tags, and can either choose one of these predefined tags or, different from an HTML editor, may define their own. Component wizards would generate skeleton "boilerplate" code into which the component writer can insert an algorithm without having to write component input and output or control code. A system like this is implemented in Triana.

Many potential PSE users would like the system to have the ability to suggest appropriate components from the repository. This would be helpful for several reasons: new or novice users would be able to paraphrase their problem and have potential similar work flow solutions previously used suggested to them; partial composed work flows would have suitable components suggested - i.e. if a mesh generator is connected to a non-compatible solver then the system would insert a translation

component; domain expert knowledge could be saved in the system and not lost when the expert is no longer available. An expert adviser could be implemented using the Java Expert System Shell (JESS) which gives the ability to "plug-in" domain specific rules into the adviser.

Component integrity and version checking is important for scientific validity and experiment reproducibility. If an in silico experiment run on a PSE produces an important result then the validity of the components that make up the work flow must be proved. To be able to reproduce the experiment it is not enough to have the work flow, the correct versions of the valid components must be known.

A flexible PSE should not be reliant on a single middleware implementation. The PSE should be able to sit above the middleware layer and utilise whatever distribution mechanisms and services are available. The VCCE implemented an extensible framework that could be used with multiple middleware such as web services and grid services. The GridLab GAT, provides a middleware independent layer that a PSE could be built on top of. Dynamically loaded adaptors provide the middleware capability. The PSE would make a GAT function call which would be dynamically mapped to an appropriate resource through middleware implementation chosen by the GAT engine. Triana makes use of this technique to distribute components via middleware such as web services and peer-to-peer networks.

Hierarchical components are an important part of a PSE user interface. In large task graphs it may be impossible to see every component given a finite screen size. A user may group components into compound components to make the algorithm easier to understand. The component model used in the VCCE has support for compound components as they can be thought of as a component containing sub-work flows. There was not enough time to implement the user interface aspect of hierarchical components in the VCCE but it has been implemented in Triana using the same component model.

The VCCE has several internal scheduling algorithms implemented. In many cases it may not be possible for a PSE to schedule components itself. Remote resources often have job submission queues to which the PSE must submit component

processes for execution. Some of these resources are controlled by schedulers that have the ability to handle dependencies for jobs, for example Condor. In this case the PSE would hand the completed work flow to the external scheduler for execution. This has been implemented in a collaboration between Triana and the DataGrid project. The internal task graph format must be translated into the format of the external scheduler prior to execution.

Resource reservation is another interesting aspect of future work. Some real time applications, such as large detector data signal processing, may require the PSE to reserve a certain amount of computing resources before the application can be executed. Whether the PSE implements the reservation request or passes it onto to a resource management application will be dependant upon the management of the resources the PSE has available to it.

Experiment reproducibility and data provenance have already been covered by state management and component integrity and version checking. To reproduce an experiment, three things are needed: the original input data set; the task graph specifying the algorithm; and the correct versions of all of the components involved.

Distributed PSEs will have component processes running on multiple resources on a network or computational grid. The user of the PSE will want the ability to monitor the various remote processes to discover their state. A heavily loaded machine may not be able to allocate enough processing to the remote job and so the user may require that particular process be moved to a more suitable location. Process migration is a complicated research area and may involve checkpointing if the run time for the process is suitably large. Monitoring information can be relatively simple to provide but migration is an area of research that is currently not well implemented, especially for heterogeneous networks where binary compatibility of components is not supported.

The VCCE was a prototype PSE and consequently cannot be thought of as being scalable. Large PSEs must be able to handle hundreds if not thousands of concurrent processes. Many common tasks in scientific simulation are data intensive and easily implemented as parallel processes, in a SPMD fashion. A PSE should

be able to discover available resources and start multiple copies of task graphs and components on those resources to increase the speed of the data processing. Many physics and astronomy applications, such as gravitational wave data processing or the SETI@home radio signal processing, and engineering applications, such as design of experiments domain searches, can use as many resources as are available. The data and processes are discrete with no communication needed between processing resources.

Components in a PSE such as Triana have user interfaces of a much more complex nature than the ones in the VCCE. The VCCE components have the ability to display the name of the component and an indication of its execution state, the component turned red once execution was completed. The major reason for PSEs existence is usability. Components must be able to provide the user with a rich interface that provides feedback about the component's state and allows complex interactions such as computational steering to take place. Implicit in the rich interface is the existence of fully featured visualisation tools. A PSE without visualisation is almost like a waggon without wheels. A scientist will want quick ways to visualise intermediate results that provide an indication of the state of a simulation. Visualisation tools should also enable remote access from mobile devices such as PDAs.

Components within the VCCE are internal to the framework. Increasingly in research and business, components are being thought of as computational services. The idea of a component as a computational service that can reside anywhere on the network has now been generalised into the concept of web and grid services that can be described by an open standard description language, WSDL. The idea of component and service being interchangeable relies on the fact that both are basically "black box" processing units that provide well defined input and output interfaces. Service implies a network whereas component is normally thought of as local. The ultimate realisation of this is service based work flows where the service can be implemented using any technology but discovered and communicated with, using standards-based descriptions, advertisements and protocols.

10.5 Summary

The goal of a PSE is to *support* an application scientist in solving a problem within a given application domain. A PSE should make the task of "problem solving" simpler by abstracting the details of hardware and software. It is natural for a user to decompose a large problem into smaller problems and a PSE should support this. The data flow approach taken in this dissertation, where an application is composed from available computational components, each with a specific purpose, is perhaps the most intuitive method of accomplishing this. Visual programming has been used in other tools to great success and is now the most common way of composing data or work flow applications. It is shown here to be a very effective way of representing applications composed from components. The graphical approach to programming used here is generic and the generated XML task graph can be converted into many different formats. The VCCE provides support for looping components. These are used to perform "parameter runs" and non-linear optimisation loops utilising algorithms such as the simplex to minimise functions and provide domain space search capabilities. Large solvers such as the parallel FE3D are wrapped as monolithic components within the VCCE. In cases such as this where source code is not available this may be the only solution. Smaller granularity components such as the broken down BE2D code can provide more code re-use.

With the benefit of hindsight the work involved implementing the CORBA middleware that makes the large solvers available to the VCCE was unnecessary. Grid computing now enables PSEs to be built on top of a feature rich middleware platform. The modular design of the VCCE allows components to be built on top of the new technologies.

The XML-based component model described here is also now a very common feature of most modern PSEs and has been generalised into the notion of components as services using the standards-based WSDL specification. The component model is the one thing that will live on after this project.

APPENDIX A

Publications

Chronological list of conference papers and journal publications associated with this research.

1. *An XML Based Component Model For Generating Scientific Applications and Performing Large Scale Simulations in a Meta-Computing Environment*, Matthew S. Shields, David Walker, Omer Rana, Maozhen Li, paper presented at the International Symposium on Generative and Component Based Software Engineering (GCSE), Erfurt, Germany, 27-30 October 1999. Proceedings only available on CD-ROM.

2. *Implementing Problem Solving Environments for Computational Science*, Omer F. Rana, Maozhen Li, Matthew Shields, David Walker and David Golby, European Conference on Parallel Processing (EuroPar), Munich, Germany, August 2000. Springer Verlag.

3. *A Java/CORBA based Visual Program Composition Environment for PSEs*, M. S. Shields, O. F. Rana, David W. Walker, Li Maozhen, David Golby, paper published Concurrency: Practice and Experience. Volume 12, Issue 8, 2000, pp687-704. (Special Issue: ACM 1999 Java Grande Conference (Part 3) Issue

Edited by Geoffrey Fox.) John Wiley & Sons, Ltd.

4. *A Collaborative Code Development Environment for Computational Electro-Magnetics*, Matthew Shields, Omer F. Rana, David W. Walker and David Golby, Software Architectures for Scientific Computing Applications, 8th working conference organised by the IFIP Working Group on Numerical Software (WG 2.5) on behalf of the IFIP Technical Committee on Software: Theory and Practice, Ottawa, October 2000. Proceedings published The Architecture of Scientific Software, eds. RF Boisvert and PTP Tang, pub. Kluwer Academic Publishers, Massachusetts, USA, pp. 119-141, 2001. ISBN 0-7923-7339-1.

5. *The Software Architecture of a Distributed Problem-Solving Environment*, David. W. Walker, M. Li, O.F.Rana, M. S. Shields, and Y. Huang, paper published Concurrency: Practice and Experience. Volume 12, Issue 15, 2001, pp1455-1480.

6. *Component-based Problem Solving Environments for Computational Science*, Maozhen Li, Omer F. Rana, David W. Walker, Matthew Shields, Yan Huang, book chapter in "Component-based Software Development" (Ed: Kung-Kiu Lau), World Scientific Publishing, 2003.

APPENDIX B

Code Listings

B.1 Code from Chapter 4: Problem Solving Environment Architecture

B.1.1 Number Display Bean

Code from section 4.2

```
public class NumberDisplayBean extends JLabel {
    protected float value = 0;
    protected Float oValue = new Float(value);
    protected PropertyChangeSupport listeners = new
       PropertyChangeSupport(this);

    // Obligatory no argument constructor
    public NumberDisplayBean() {
        setPreferredSize(new Dimension(100, 100));
        setVisible(true);
        setFont(this.getFont().deriveFont(Font.BOLD, (float)30));
        setBorder(BorderFactory.createBevelBorder(BevelBorder.
           LOWERED, Color.black, Color.blue));
        setBackground(Color.white);
        setOpaque(true);
        this.setHorizontalAlignment(CENTER);
```

```
    }

    // Get the value of value.
    public float getValue() {return value;}

    // Set the value of value.
    public synchronized void setValue(float  v) {
        if (value != v) {
        value = v;
        setText(new Float(value).toString());
        repaint();
        fireValueChange();
        }
    }

    public void addPropertyChangeListener(String propertyName,
       PropertyChangeListener l) {
        listeners.addPropertyChangeListener(propertyName, l);
    }

    public void removePropertyChangeListener(String propertyName
       , PropertyChangeListener l) {
        listeners.removePropertyChangeListener (propertyName, l);
    }

    protected void fireValueChange() {
        listeners.firePropertyChange("value", oValue, oValue =
           new Float(value));
    }
} // NumberDisplayBean
```

B.1.2 Operator Bean

Code from section 4.2

```
public class OperatorBean extends JLabel {
    public static final String ADD = "add", SUB = "sub", MLT = "
       mlt", DIV = "div";
    protected float operand1 = 0, operand2 = 0;
    protected String operator = ADD;
    protected float answer;
    protected String oOperator = new String(operator);
    protected Float oAnswer = new Float(answer);
    protected PropertyChangeSupport listeners = new
       PropertyChangeSupport (this);
```

```
10      // Obligatory no argument constructor
        public OperatorBean() {
12          setPreferredSize(new Dimension(100, 100));
            setVisible(true);
14          setFont(this.getFont().deriveFont(Font.BOLD, (float)30));
            setBorder(BorderFactory.createBevelBorder(BevelBorder.
               LOWERED, Color.black, Color.blue));
16          setBackground(Color.white);
            setOpaque(true);
18          this.setHorizontalAlignment(CENTER);
        }
20
        // Get the value of operator.
22      public String getOperator() {return operator;}

24      // Set the value of operator.
        public void setOperator(String  v) {
26          operator = v;
            setText(operator);
28          repaint();
            fireOperatorChange();
30      }

32      public float getOperand1() {return operand1;}

34      public float getOperand2() {return operand2;}

36      public void setOperand1(float v) {
            if (operand1 != v) {
38              operand1 = v;
                calculate();
40          }
        }
42
        public void setOperand2(float v) {
44          if (operand2 != v) {
                operand2 = v;
46              calculate();
            }
48      }

50      public void addPropertyChangeListener(String propertyName,
           PropertyChangeListener l) {
            listeners.addPropertyChangeListener(propertyName, l);
52      }

54      public void removePropertyChangeListener(String propertyName
           , PropertyChangeListener l) {
```

```
        listeners.removePropertyChangeListener(propertyName, 1);
    }

    protected void fireOperatorChange() {
        listeners.firePropertyChange("operator", oOperator,
            oOperator = new String(operator));
    }

    protected void fireAnswerChange() {
        listeners.firePropertyChange("answer", oAnswer, oAnswer
            = new Float(answer));
    }

    protected synchronized void calculate() {
        try {
            if (operator == ADD) {
                answer = operand1 + operand2;
            } else if (operator == SUB) {
                answer = operand1 - operand2;
            } else if (operator == MLT) {
                answer = operand1 * operand2;
            } else if (operator == DIV) {
                answer = operand1 / operand2;
            }
            fireAnswerChange();
        } catch(Exception e) {
            System.out.println(e.toString() + ' ' + new Float(
                operand1).toString() + ' ' + new Float(operand2).
                toString() + ' ' + operator);
        }
    }

    public String toString() {
        return "OperatorBean";
    }
} // OperatorBean
```

B.1.3 Example XML Data Analysis Definition

Code from section 4.4.2.9, page 50

```
<?xml version="1.0"?>
<preface>
    <name alt="DA" id="DA01">DataAnalyser</name>
    <pse-type>Generic</pse-type>
```

```
    <hierarchy id="parent">Tools.Data.Data Analyser</hierarchy>
    <hierarchy id="child"></hierarchy>
</preface>

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

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

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

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

B.1.4 Example Component Interface Described in XML

Code from section 4.4.2.9, page 50

```
<?xml version="1.0"?>
<PSE>
<preface>
    <name alt="be2d" id="be2d01">be2d</name>
    <pse-type>Generic</pse-type>
    <hierarchy id="parent">be2d.be2dComponents.Be2DTest</
        hierarchy>
    <hierarchy id="child"></hierarchy>
```

```
</preface>
<ports>
    <inportnum>2</inportnum>
    <outportnum>2</outportnum>
    <inport id="1" parameter="control" type="stream" value="NIL">
        <href name="file:///home/scmmss/project/src/be2d/
            be2dComponents/control" value="NIL"/>
    </inport>
    <inport id="2" parameter="model" type="stream" value="NIL">
        <href name="file:///home/scmmss/project/src/be2d/
            be2dComponents/model" value="NIL"/>
    </inport>
    <outport id="1" parameter="cur" type="stream" value="NIL">
        <href name="file:///home/scmmss/project/src/be2d/
            be2dComponents/cur" value="NIL"/>
    </outport>
    <outport id="2" parameter="rcs" type="stream" value="NIL">
        <href name="file:///home/scmmss/project/src/be2d/
            be2dComponents/rcs" value="NIL"/>
    </outport>
</ports>
<execution id="software" type="bytecode">
    <type id="architecture" value="serial"/>
    <type id="class" value="be2d.be2dComponents.Be2DTest"/>
    <type id="source" value="file:///home/scmmss/project/src/be2d
        /be2dComponents/*.java"/>
    <type id="classpath" value="~/project/classes;/~/local/3
        rdPartyJava/JNL/Classes"/>
</execution>
<execution id="platform">
    <type id="java" value="jdk1.2"/>
</execution>
<help context="apidoc">
    <href name="file:///home/scmmss/project/docs/be2ddocs/index.
        html" value="NIL"/>
</help>
</PSE>
```

B.1.5 Example XML Task Graph

Code from section 4.4.4, page 52

```
<?xml version="1.0" encoding="UTF-8"?>
<PSEtaskgraph type="serialized">
    <component>
```

```
        <name alt="be2dData" id="be2dData01" inst="7065019">be2d
            data</name>
        <location xcoord="207.0" ycoord="97.0" />
    </component>
    <component>
        <name alt="be2d" id="be2d01" inst="6670326">be2d</name>
        <location xcoord="242.0" ycoord="212.0" />
    </component>
    <component>
        <name alt="curView" id="curView01" inst="6326139">cur
            viewer</name>
        <location xcoord="137.0" ycoord="366.0" />
    </component>
    <component>
        <name alt="rcsView" id="rcsView01" inst="2433702">rcs
            viewer</name>
        <location xcoord="293.0" ycoord="358.0" />
    </component>
    <start>
        <name alt="be2dData" id="be2dData01" inst="7065019">be2d
            data</name>
    </start>
    <connection>
        <parent>
            <name alt="be2dData" id="be2dData01" inst="7065019">
                be2d data</name>
        </parent>
        <child>
            <name alt="be2d" id="be2d01" inst="6670326">be2d</
                name>
        </child>
    </connection>
    <connection>
        <parent>
            <name alt="be2d" id="be2d01" inst="6670326">be2d</
                name>
        </parent>
        <child>
            <name alt="curView" id="curView01" inst="6326139">cur
                viewer</name>
        </child>
    </connection>
    <connection>
        <parent>
            <name alt="be2d" id="be2d01" inst="6670326">be2d</
                name>
        </parent>
        <child>
            <name alt="rcsView" id="rcsView01" inst="2433702">rcs
```

```
                viewer</name>
44          </child>
        </connection>
46  </PSEtaskgraph>
```

B.1.6 VCCE Proxy Component Interface

Code from 4.5, page 55

```
/**
2  * Copyright &copy 2000 Cardiff University & BAE SYSTEMS Ltd, All
   * rights reserved.
4  * @author Matthew S. Shields
   */
6  package vcce.visualproxy;

8  import java.awt.*;
   import vcce.observer.*;
10
   /**
12  * The public interface to all proxy components within the VCCE.
   * Implements the Proxy pattern, pp207-217 Design Patterns,
14 * Gamma et al.
   */
16 public interface ProxyInterface extends SubjectInterface {
       // Returns the internal system name.
18     public String getInternalName();

20     // Returns the alternative name.
       public String getAltName();
22
       // Returns the external name representation.
24     public String getExternalName();

26     // Returns the instance ID (unique)
       public String getInstanceID();
28
       // Sets the instance ID (should be machine generated and
          unique).
30     public void setInstanceID(String InstIDStr);

32     // Returns the type of PSE the component can be used within.
       public PseType getPseType();
34
       // Returns the parent component, in a compound component.
```

```
36      public String getParent();

38      // Returns an iterator containing the child components in a
            compound component.
        public ProxyIterator getChildren();
40
        // Returns an iterator of input ports.
42      public PortIterator getInports();

44      // Returns a string representation of this components input
            values.
        public String inportsToString();
46
        // Returns the number of input ports
48      public int inportCount();

50      // Returns the number of output ports
        public int outportCount();
52
        // Returns an iterator of output ports.
54      public PortIterator getOutports();

56      // Returns the executable component of the proxy.
        public ExecutionInterface getExecution();
58
        // Returns an iterator of the help components.
60      public HelpIterator getHelps();

62      // Returns deep clone of the proxy, excluding help components
        // which are shared between proxy instances.
64      public Object clone();

66      // Display a particular help component, based on context.
        public void displayHelp(HelpContext context);
68
        // Returns the value of a given output port, identified by
70      // parameter string.
        public Object getOutportValue(String parameter);
72
        // Sets the value of an input port identified by the
            parameter string.
74      public void setInportValue(String parameter, Object value);

76      // Sets the value of the inport based on a user selected
            value.
        public void setInportOption(String parameter, String option);
78
        // Reset the state of the component.
80      public void resetComponent();
```

```
} // End of ProxyInterface Interface
```

B.1.7 ExecutionGraph

Code from 4.6.3.1, page 69

```
public class ExecutionGraph extends AbstractExecutionGraph {

    // An array of integers corresponding to the order in which
       the nodes in the graph must be executed.
    private int[] executionOrder_;

    // Index to the current node in the execution order.
    private int executionIndex_;

    // Implementation of abstract method, called by a node on
       completion of it's execution. If there are more nodes to
       execute in the graph this function calls the executeNext
       method.
    public void executionPerformed(ExecutionEvent evt) {
        executionIndex_++;
        if (hasMoreToExecute()) {
            TransferPortData();
            executeNext();
    }

    // Implementation of abstract, responsible for determining
       the order of execution and starting the first node in the
       graph.
    public void execute() {
        deriveOrderOfExecution();
        executionIndex_ = 0;
        executeNext();
    }

    // Private utility method, returns True if there are more
       nodes left to execute, False otherwise.
    private boolean hasMoreToExecute() {
        return (executionIndex_ < executionOrder_.length);
    }

    // Private utility method to execute the next node in the
       order.
    private void executeNext() {
        ExecutionGraphNode aNode = getNodeFromOrder(
```

```
            executionIndex_);
32          aNode.execute();
        }
34
        private void TransferPortData() {
36          ExecutionGraphNode current = getNodeFromOrder(
                executionIndex_);
            for (EGNIterator it = current.getParents(); it.hasNext()
                ;) {
38              it.next().sendOutputToChild(current);
            }
40      }

42      private ExecutionGraphNode getNodeFromOrder(int index) {
            return ((ExecutionGraphNode) nodes_.elementAt(
                executionOrder_[index]));
44      }

46      // Private utility method that analyses the nodes in the
            graph and derives a straight forward list in order of
            execution.
        private void deriveOrderOfExecution() {
48          int i = 0;
            int current = 0;
50          executionOrder_ = new int[nodes_.size()];
            for (EGNIterator it = start_.getChildren(); it.hasNext()
                ;) {
52              executionOrder_[current] = nodes_.indexOf(it.next());
                current++;
54          }
            while (current < executionOrder_.length) {
56              ExecutionGraphNode tempNode = (ExecutionGraphNode)
                    nodes_.elementAt(
                        executionOrder_[i]);
58              for (EGNIterator it = tempNode.getChildren();
                     it.hasNext();) {
60                  int childIndex = nodes_.indexOf(it.next());
                    boolean found = false;
62                  for (int x = 0; x < current; x++) {
                        if (executionOrder_[x] == childIndex) {
64                          found = true;
                        }
66                  }
                    if (!found) {
68                      executionOrder_[current] = childIndex;
                        current++;
70                  }
                    i++;
72              }
```

```
        }
74 }
```

B.2 Code from Chapter 5:

BE2D - Industrial Demonstrator

B.2.1 Command Line Execution Component

Code from section 5.1.1, page 74

```
// Concrete implementation of an AbstractExecution. This class
2 // encapsulates a CommandLine executable component.
public class CommandLineExecution {
4     // Command Line
    private String command_ = "";
6
    // Command line argument, if required.
8   private String argument_ = "";

10  private String data_ = "";

12   // Setters
    public void setArgument(Object argumentStr) {
14      argument_ = (String)argumentStr;
    }
16
    public void setCommand(Object command) {
18      command_ = (String)command;
    }
20
    public void setDataFile(Object output) {
22      data_ = (String)output;
    }
24
    // Getters
26  public String getDataFile() {
        return data_;
28  }

30  public void execute() {
        String execStr = command_ + "␣" + argument_;
32      try {
```

```
            Process process = Runtime.getRuntime().exec(execStr);
34          // Process output
            BufferedReader stdout = new BufferedReader(new
               InputStreamReader(process.getInputStream()));
36          String outStr = "";
            while (outStr != null) {
38              outStr = stdout.readLine();
                if (outStr != null) {
40                  System.out.println("process message: " +
                        outStr);
                }
42          }
            stdout.close();
44          // Process Errors
            BufferedReader error = new BufferedReader( new
                InputStreamReader(process.getErrorStream()));
46          String errStr = "";
            while (errStr != null) {
48              errStr = error.readLine();
                if (errStr != null) {
50                  System.out.println("process error:" + errStr);
                }
52          }
            error.close();
54
            try {
56              process.waitFor();
            }
58          catch (InterruptedException e) {
                e.printStackTrace(System.out);
60          }
        }
62      catch (IOException e) {
            e.printStackTrace(System.out);
64      }
    }
66
    public void reset() {
68      //no-op
    }
70} // End of CommandLineExecution Class
```

B.2.2 BE2D Model Data Java Interface

Code from section 5.1.2, page 78

```
// Interface to the model object, provides read-only access to
// the object representation of the model file which defines the
// 2D boundary for the simulation.
public interface Model {
    // Returns number of co-ordinate pairs in the model.
    public int n();

    // Returns array of integers containing the x co-ordinates.
    public double[] x();

    // Returns array of integers containing the y co-ordinates.
    public double[] y();
} // End of Model Interface
```

B.2.3 BE2D Control Data Java Interface

Code from section 5.1.2, page 78

```
// Interface to the control object, provides read-only access to
// the object representation of the control parameters file.
public interface Control {
    // Returns a string defining the name of the current problem
    public String getTitle();

    // Returns the frequency of the incident wave in Hz
    public double getF();

    // Returns direction the wave is travelling from in radians
    public double getBeta();

    // Returns the complex amplitude of the wave
    public Complex getEcomp();

    // Returns boolean flag lcur. If true cur output written to
    // the default file "cur" in the current directory
    public boolean getLcur();

    // Returns boolean flag lrcs. If true rcs output written to
    // the default file "rcs" in the current directory
    public boolean getLrcs();

    // Returns the rcs output file object, or null if not valid
    public File getRcsFile();

    // Returns the cur output file object, or null if not valid
```

```
28      public File getCurFile();

30      // Returns rcs output stream URL object, null if not valid
        public URL getRcsURL();
32
        // Returns cur output stream URL object, null if not valid
34      public URL getCurURL();
    } // End of Control Interface
```

B.3 Code from Chapter 6: Control and Loop Components

B.3.1 XML Task Graph with Loop Constructs

Code from section 6.4.1, page 95

```
    <?xml version="1.0" encoding="UTF-8"?>
2   <PSEtaskgraph type="serialized">
        <component>
4           <name alt="Loop" id="loop01" inst="4081527">
                Control Loop
6           </name>
            <location xcoord="36.0" ycoord="40.0" />
8       </component>
        <component>
10          <name alt="Loop" id="loop01" inst="1947116">
                Control Loop
12          </name>
            <location xcoord="103.0" ycoord="201.0" />
14      </component>
        <component>
16          <name alt="BE2DAFMesh" id="be2dAF03" inst="6044039">
                be2d action factory mesh
18          </name>
            <location xcoord="434.0" ycoord="225.0" />
20      </component>
        <component>
22          <name alt="BE2DAFWave" id="be2dAF02" inst="3325285">
                be2d action factory wave
24          </name>
            <location xcoord="234.0" ycoord="377.0" />
26      </component>
```

```
    <component>
28      <name alt="BE2DAFRCS" id="be2dAF04" inst="2102960">
            be2d action factory rcs
30      </name>
        <location xcoord="352.0" ycoord="453.0" />
32  </component>
    <component>
34      <name alt="BE2DAFStore" id="be2dAF05" inst="4686789">
            be2d action factory store
36      </name>
        <location xcoord="379.0" ycoord="548.0" />
38  </component>
    <start>
40      <name alt="Loop" id="loop01" inst="4081527">
            Control Loop
42      </name>
    </start>
44  <start>
        <name alt="BE2DAFMesh" id="be2dAF03" inst="6044039">
46          be2d action factory mesh
        </name>
48  </start>
    <connection>
50      <parent>
            <name alt="Loop" id="loop01" inst="4081527">
52              Control Loop
            </name>
54          <loop parameter="Angle" port_type="Float" start="0.0"
                halt="25.0" increment="1.0" current="3.0" />
        </parent>
56      <child>
            <name alt="Loop" id="loop01" inst="1947116">
58              Control Loop
            </name>
60      </child>
    </connection>
62  <connection>
        <parent>
64          <name alt="Loop" id="loop01" inst="1947116">
                Control Loop
66          </name>
            <loop parameter="Amplitude" port_type="Float"
68              start="1.0e+07" halt="1.0e+08" increment="1.0e+07
                    "
                current="1.0e+08" />
70      </parent>
        <child>
72          <name alt="BE2DAFWave" id="be2dAF02" inst="3325285">
                be2d action factory wave
```

```
74              </name>
            </child>
76      </connection>
        <connection>
78          <parent>
                <name alt="BE2DAFMesh" id="be2dAF03" inst="6044039">
80                  be2d action factory mesh
                </name>
82          </parent>
            <child>
84              <name alt="BE2DAFRCS" id="be2dAF04" inst="2102960">
                    be2d action factory rcs
86              </name>
            </child>
88      </connection>
        <connection>
90          <parent>
                <name alt="BE2DAFWave" id="be2dAF02" inst="3325285">
92                  be2d action factory wave
                </name>
94          </parent>
            <child>
96              <name alt="BE2DAFRCS" id="be2dAF04" inst="2102960">
                    be2d action factory rcs
98              </name>
            </child>
100     </connection>
        <connection>
102         <parent>
                <name alt="BE2DAFRCS" id="be2dAF04" inst="2102960">
104                 be2d action factory rcs
                </name>
106         </parent>
            <child>
108             <name alt="BE2DAFStore" id="be2dAF05" inst="4686789">
                    be2d action factory store
110             </name>
            </child>
112     </connection>
        <connection>
114         <parent>
                <name alt="BE2DAFStore" id="be2dAF05" inst="4686789">
116                 be2d action factory store
                </name>
118         </parent>
            <child>
120             <name alt="Loop" id="loop01" inst="1947116">
                    Control Loop
122             </name>
```

```
        </child>
    </connection>
    <connection>
        <parent>
            <name alt="BE2DAFStore" id="be2dAF05" inst="4686789">
                be2d action factory store
            </name>
        </parent>
        <child>
            <name alt="Loop" id="loop01" inst="4081527">
                Control Loop
            </name>
        </child>
    </connection>
</PSEtaskgraph>
```

B.3.2 Loop Control Interface

Code from 6.4.2, page 97

```
public interface ControlInterface {
    // Returns a constant data type for this loop.
    public PortType getPortType();

    // Returns the initial loop value.
    public Object getInitValue();

    // Sets the initial loop value.
    public void setInitValue(Object value);

    // Returns the halting value for the loop.
    public Object getHaltValue();

    // Sets the halting value for the loop.
    public void setHaltValue(Object value);

    // Returns the current loop value.
    public Object getCurrentValue();

    // Returns the loop increment value.
    public Object getIncrement();

    // Sets the loop increment value.
    public void setIncrement(Object value);
```

```
26      // Returns true if current value is less than the halting
            value, false otherwise.
        public boolean haltingCondition();
28
        // Increment the loop.
30      public void performStep();

32      // Set the input port that the loop will iterate over.
        public void setLoopedPort(PortInterface aPort);
34
        // Return the parameter name for the selected port.
36      public String getParameterName();

38      // Reset the loop to it's start value;
        public void resetLoop();
40
         // Adds an output port to the control component, used in
            loop nesting where the value from the outer loop needs to
             be propagated through to inner loops.
42      public void addOutport(PortInterface aPort);
    }
```

B.4 Code from Chapter 8: Design of Experiments

B.4.1 Downhill Simplex

Code from section 8.4, page 128

```
    /**
2    Copyright &copy 2001 Cardiff University & BAE SYSTEMS Ltd,
     All rights reserved.
4
     Algorithm: Downhill Simplex Method in Multidimensions, adapted
6    from Numerical Recipes (FORTRAN77 version), Press, Flannery,
     Teukolsky, Vetterling, section 10.4, pp289-293.
8
     @author Matthew S. Shields
10   */
    package com.baesystems.optimisation;
12
    import cern.colt.function.DoubleComparator;
```

```
14 import cern.colt.matrix.DoubleFactory1D;
   import cern.colt.matrix.DoubleMatrix1D;
16 import cern.colt.matrix.DoubleMatrix2D;
   import cern.colt.matrix.doublealgo.Formatter;
18 import cern.jet.math.Functions;
   import com.baesystems.optimisation.function.DoubleObjectFunction;
20 import java.util.Vector;

22 /**
    * Static library class for optimisation functions
24  */
   public final class DownhillSimplex {
26
       private static int IT_MAX = 100; // Max iterations
28
       private static final DoubleFactory1D factory1D_ =
          DoubleFactory1D.dense;
30
       // helper function sets the max iterations and calls the real
32        amoeba.
       public static final int amoeba(DoubleMatrix2D simplexP,
          DoubleMatrix1D vectorY, double FTOL, DoubleObjectFunction
          funk, int maxIter) {
34         IT_MAX = maxIter;
           return amoeba(simplexP, vectorY, FTOL, funk);
36     }

38     /**
        Multidimensional minimisation of the function func(x) where
40      x is a vector in N dimensions, by the downhill simplex
        method of Nelder and Mead.
42      Note: On output simplexP and vectorY will have been reset to
        N+1 new points all within FTOL of a minimum function value.
44
         @param simplexP A matrix, of dimension (N+1, N) whose rows
            are vectors which are the vertices of the starting
            simplex.
46       @param vectorY A vector of length N+1, whose components
            musts be pre-initialised to the values of "funk"
            evaluated at the N+1 vertices (rows) of simplexP.
         @param FTOL The fractional convergence tolerance to be
            achieved in the function value.
48       @param funk The function to be minimised.
         @return The number of iterations taken.
50      */
       public static final int amoeba(DoubleMatrix2D simplexP,
          DoubleMatrix1D vectorY, double FTOL, DoubleObjectFunction
          funk) {
52         DoubleMatrix1D pSum;  // column sum values
```

```
        int indexHi; // index of the highest point in vectorY
54      int indexNext; // index of the next highest point
        int indexLo; // index of the lowest point
56      double RTOL; // the fractional range
        double yTry; // the value returned from funk
58      double ySave; // the value returned from funk
        int iterations = 0;
60
        pSum = getColumnSumVector(simplexP);
62      for (;;) {
            // First we must determine highest (worse), next-
               highest, and lowest (best) points in the simplex.
64          indexLo = 0;
            if (vectorY.get(0) > vectorY.get(1)) {
66              indexHi = 0;
                indexNext = 1;
68          }
            else {
70              indexHi = 1;
                indexNext = 0;
72          }
            for (int i = 0; i < vectorY.size(); i++) {
74              if (vectorY.get(i) <= vectorY.get(indexLo)) {
                    indexLo = i;
76              }
                if (vectorY.get(i) > vectorY.get(indexHi)) {
78                  indexNext = indexHi;
                    indexHi = i;
80              }
                else if ((vectorY.get(i) > vectorY.get(indexNext)
                   ) && (i != indexHi)) {
82                  indexNext = i;
                }
84          }

86          // Compute the fractional range between the highest
               and the lowest and return if satisfactory. If
               returning, put best point and value in slot 0.
            RTOL = 2.0 * Math.abs(vectorY.get(indexHi) - vectorY.
               get(indexLo)) / (Math.abs(vectorY.get(indexHi)) +
               Math.abs(vectorY.get(indexLo)));
88          if (RTOL < FTOL) {
                double swap = vectorY.get(0);
90              vectorY.set(0, vectorY.get(indexLo));
                vectorY.set(indexLo, swap);
92              for (int i = 0; i < simplexP.columns(); i++) {
                    swap = simplexP.get(0, i);
94                  simplexP.set(0, i, simplexP.get(indexLo, i));
                    simplexP.set(indexLo, i, swap);
```

```
96              }
                // amoeba found solution within tolerance
98              return iterations;
            }
100
            if (iterations >= IT_MAX) {
                // amoeba exceeded max iterations
102             return iterations;
            }
104         iterations += 2;
106
            // Begin a new iteration. First extrapolate by a
               factor of -1 through the face of the simplex
               across from the high point, i.e. reflect the
               simplex from the high point.
108         yTry = amoebaTry(simplexP, vectorY, pSum, funk,
               indexHi, -1.0);

110         if (yTry <= vectorY.get(indexLo)) {
                // gives a result better than the best point, so
                   try an additional extrapolation by a factor of
                    2.
112             yTry = amoebaTry(simplexP, vectorY, pSum, funk,
                   indexHi, 2.0);
            }
114         else if (yTry >= vectorY.get(indexNext)) {
                // The reflected point is worse than the second-
                   highest, so look for an intermediate lower
                   point, i.e. do a one-dimensional contraction.
116             ySave = vectorY.get(indexHi);
                yTry = amoebaTry(simplexP,vectorY, pSum, funk,
                   indexHi, 0.5);
118             if (yTry >= ySave) {
                    // Can't seem to get rid of the high point.
                       Better contract around the lowest(best)
                       point.
120                 for (int i = 0; i < simplexP.rows(); i++) {
                        if (i != indexLo) {
122                         for (int j = 0; j < simplexP.columns
                               (); j++) {
                                double value = 0.5 *
124                                 (simplexP.get(i, j) +
                                       simplexP.get(indexLo, j));
                                pSum.set(j, value);
126                             simplexP.set(i, j, value);
                            }
128                         vectorY.set(i, funk.apply(pSum));
                        }
130                 }
```

```
                    // Keep track of function evaluations and
                        recompute PSum
132                 iterations += simplexP.columns();
                    pSum = getColumnSumVector(simplexP);
134             }
                else {
136                 // correct the evaluation count
                    iterations--;
138             }
            }
140     }
    }
142
    /**
144  Initialise and return a vector Y of dimension N+1 where it's
     components are calculated by evaluating the function funk at
146  the N+1 vertices (rows) of the simplex.

148  @param simplexP the starting simplex
     @param the objective function to be evaluated
150  @return an initialised vector Y of size N+1
     */
152 public static final DoubleMatrix1D initVectorY(DoubleMatrix2D
        simplexP, DoubleObjectFunction funk) {
        DoubleMatrix1D vectorY = factory1D_.make(simplexP.rows())
            ;
154     for (int i = 0; i < simplexP.rows(); i++) {
            double yTry = funk.apply(simplexP.viewRow(i));
156         vectorY.set(i, yTry);
        }
158     return vectorY;
    }
160
    /**
162  Extrapolates by a factor fac through the face of the simplex
     across from the high point, tries it, and replaces the high
164  point if the new point is better.

166  @param simplexP A matrix representing the simplex.
     @param vectorY the start vector
168  @param pSum the simplex column sum vector
     @param funk The function to be minimised.
170  @param index the index in vectorY of the highest value
     @param fac factor of extrapolation
172  @return the calculated value of funk
     */
174 private static double amoebaTry(DoubleMatrix2D simplexP,
       DoubleMatrix1D vectorY, DoubleMatrix1D pSum,
       DoubleObjectFunction funk, int indexHi, double fac) {
```

```
        double yTry;
176     double fac1 = (1.0-fac)/simplexP.columns();
        double fac2 = fac1-fac;
178
        DoubleMatrix1D pTry = factory1D_.make(simplexP.columns())
            ;
180     for (int i = 0; i < simplexP.columns(); i++) {
            double pTryElem = pSum.get(i)*fac1 - simplexP.get(
                indexHi, i)*fac2;
182         pTry.set(i, pTryElem);
        }
184
        // Evaluate the function at the trial point
186     yTry = funk.apply(pTry);

188     // If it's better than the highest, then replace the
            highest
        if (yTry < vectorY.get(indexHi)) {
190         vectorY.set(indexHi, yTry);
            for (int i = 0; i < simplexP.columns(); i++) {
192             double pSumElem = pSum.get(i) - simplexP.get(
                    indexHi, i) + pTry.get(i);
                pSum.set(i, pSumElem);
194             simplexP.set(indexHi, i, pTry.get(i));
            }
196     }
        return yTry;
198 }

200 /**
     Calculate the sums of the columns in the simplex and return
202  them as a vector.

204  @param simplexP the simplex whose column sums we are
         calculating
     @return A vector containing the column sums
206  */
    private static DoubleMatrix1D getColumnSumVector(
        DoubleMatrix2D simplexP) {
208     DoubleMatrix1D pSum = factory1D_.make(simplexP.columns())
            ;
        for (int i = 0; i < simplexP.columns(); i++) {
210         pSum.set(i, (simplexP.viewColumn(i)).zSum());
        }
212     return pSum;
    }
214 } // End of DownhillSimplex Class
```

B.4.2 Simplex Component XML Definition

Code from 8.4, page 128

```
<?xml version="1.0"?>
<PSE>
    <preface>
        <name alt="simplex" id="simplex">simplex method</name>
        <pse-type>Generic</pse-type>
        <hierarchy id="parent"></hierarchy>
        <hierarchy id="child"></hierarchy>
    </preface>
    <ports>
        <inportnum>6</inportnum>
        <outportnum>2</outportnum>
        <inport id="1" parameter="SolveFunction" type="option"
            value="com.baesystems.optimisation.function.
            SimpleEvaluationFunk">
             <option name="simple" value="com.baesystems.
                optimisation.function.SimpleEvaluationFunk">
             </option>
             <option name="complex" value="com.baesystems.
                optimisation.function.ComplexEvaluationFunk">
             </option>
             <option name="Fiacco-McCormick" value="com.baesystems
                .optimisation.function.FMCMethodEvaluationFunk">
             </option>
        </inport>
        <inport id="2" parameter="DesignSpaceFunction" type="
            option" value="com.baesystems.optimisation.function.
            SimpleEvaluationFunk">
             <option name="simple" value="com.baesystems.
                optimisation.function.SimpleEvaluationFunk">
             </option>
             <option name="complex" value="com.baesystems.
                optimisation.function.ComplexEvaluationFunk">
             </option>
             <option name="Fiacco-McCormick" value="com.baesystems
                .optimisation.function.FMCMethodEvaluationFunk">
             </option>
        </inport>
        <inport id="3" parameter="Lambda" type="float" value="0.5
            ">
        </inport>
        <inport id="4" parameter="X1" type="float" value="3">
        </inport>
        <inport id="5" parameter="X2" type="float" value="0">
        </inport>
```

```
        <inport id="6" parameter="MaxIter" type="short" value="10
            ">
        </inport>
        <outport id="1" parameter="Solution" type="object" value=
            "NIL">
        </outport>
        <outport id="1" parameter="DesignSpace" type="object"
            value="NIL">
        </outport>
    </ports>
    <execution id="software" type="bytecode" value="extended">
        <type id="architecture" value="serial"/>
        <type id="class" value="com.baesystems.components.
            SimplexComponent"/>
        <type id="source" value="file:///home/compdata/cardiff/
            project/src/com/baesystems/components/SimplexComponent
            .java"/>
        <type id="classpath" value="/home/compdata/project/
            classes"/>
    </execution>
    <execution id="platform">
        <type id="java" value="jdk1.2"/>
    </execution>
    <help context="apidoc">
        <href name="file://f:\\cardiff\\project\\docs\\be2ddocs\\
            index.html" value="NIL"/>
    </help>
</PSE>
```

Bibliography

[1] G. Abram and L. Treinish. An Extended Data-Flow Architecture for Data Analysis and Visualization. *Computer Graphics*, 29(2):17–21, 1995.

[2] S. Agrawal, J. Dongarra, K. Seymour, and S. Vadhiyar. NetSolve: Past, Present, and Future - A Look at a Grid Enabled Server. In F. Berman, G. Fox, and A. Hey, editors, *Making the Global Infrastructure a Reality*. Wiley Publishing, 2003.

[3] Gabrielle Allen, Kelly Davis, Konstantinos N. Dolkas, Nikolaos D. Doulamis, Tom Goodale, Thilo Kielmann, André Merzky, Jarek Nabrzyski, Juliusz Pukacki, Thomas Radke, Michael Russell, Ed Seidel, John Shalf, and Ian Taylor. Enabling Applications on the Grid: A Gridlab Overview. *International Journal of High Performance Computing Applications: Special issue on Grid Computing: Infrastructure and Applications*, 2003.

[4] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *16th Intl. Conference on Scientific and Statistical Database Management (SSDBM)*, 2004.

[5] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services Version 1.1.

[6] Apache XML Project. Xerces XML Parser.
See web site at: `http://xml.apache.org/`.

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

[8] A. Arbree, P. Avery, D. Bourilkov, R. Cavanaugh, S. Katageri, J. Rodriguez, G. Graham, J. Vöckler, and M. Wilde. Virtual Data in CMS Productions. Technical report, GriPhyN, 2003.

[9] 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.

[10] 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*, 1996.

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

[12] K.W. Brodlie, J. Wood, D.A. Duce, J.R. Gallop, D. Gavaghan, M. Giles, S. Hague, J. Walton, M. Rudgyard, B. Collins, J. Ibbotson, and A. Knox. XML for Visualization. In *EuroWeb 2002*, 2002.

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

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

[15] S. Browne. The Netlib Mathematical Software Repository. *D-lib Magazine*, September 1995.

[16] H. de Bruin. A Grey–Box Approach to Component Composition. In *Proceedings of the International Symposium on Generative and Component Based Software Engineering (GCSE)*, 1999.

[17] O. Bunin, Y. Guo, and J. Darlington. Design of Problem Solving Environment for Contingent Claim Valuation. In *EuroPar*. Springer Verlag, 2001.

[18] The Cactus Maintainers. The Cactus Computational Toolkit.
See web site at: `http://www.cactuscode.org`.

[19] B. Carpenter, Y-J. Chang, G. Fox, D. Leskiw, and X. Li. Experiments with 'HP Java'. *Concurrency: Practice and Experience*, 9(6):633–648, June 1997.

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

[21] CCA Forum. The Common Component Architecture Technical Specification - version 0.5. Technical report, Common Component Architecture Forum, 2001.

[22] 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/`.

[23] Condor Team. DAGMan: A Directed Acyclic Graph Manager.
See website at `http://www.cs.wisc.edu/condor/dagman/`.

[24] 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.*, 11(3):179–196, 1997.

[25] Dassault Aviation. Action Factory.
See website at `http://actionfactory.sourceforge.net`.

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

[27] Donald W. Denbo. SGT: The Scientific Graphics Toolkit.
See web site at: `http://www.epic.noaa.gov/java/sgt/index.html`.

[28] A. Denis, C. Perez, and T. Priol. Portable Parallel CORBA Objects: an Approach to Combine Parallel and Distributed Programming for Grid Computing. In *Proceedings Euro-Par2001*, pages 835–844. Springer, 2001.

[29] EGEE: Enabling Grids for E-science in Europe.
See website at `http://public.eu-egee.org/`.

[30] D. Erwin and D. Snelling. UNICORE: A Grid Computing Environment. In *Euro-Par 2001*, pages pp 825–834, 2001.

[31] Thomas Esser. *teTeX: a complete TeX distribution for UNIX compatible systems.* See website at `http://www.tug.org/teTeX/`.

[32] A. V. Fiacco and G. P. McCormick. *Non-Linear Programming: Sequential and Unconstrained Minimisation Techniques.* Wiley, 1968.

[33] S. Fleeter, E. Houstis, J. Rice, C. Zhou, and A. Catlin. GasTurbnLab: A Problem Solving Environment for Simulating Gas Turbines. In *Proceedings of* 16^{th} *IMACS World Congress*, 2000.

[34] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Int. Journal of Supercomputing Applications*, 11(2):115–128, 1997.

[35] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computer Infrastructure.* Morgan-Kaufmann, 1999.

[36] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Open Grid Service Infrastructure WG, Global Grid Forum, 2002.

[37] D. Foulser. IRIS Explorer: A Framework for Investigation. *Computer Graphics*, 29(2):13–16, 1995.

[38] M. Fowler and K. Scott. *UML Distilled.* Addison-Wesley, 1997.

[39] G. Fox, D. Gannon, and M. Thomas. A Summary of Grid Computing Environments. *Concurrency and Computation: Practise and Experience (Special Issue)*, 2003.

[40] 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.

[41] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPCD-'01)*, 2001.

[42] N. Furmento, W. Lee, A. Meyer, S. Newhouse, and J. Darlington. ICENI: An Open Grid Service Architcture Implemented with Jini. In *SuperComputing '02*, 2002.

[43] Fabrizio Gagliardi, Bob Jones, Mario Reale, and Stephen Burke. European DataGrid Project: Experiences of deploying a large scale Testbed for e-Science applications. In *Performance 2002 Tutorial Lectures Book "Performance Evaluations of Complex Systems: Techniques and Tools"*, 2002.

[44] 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):11–23, 1994.

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

[46] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Massachusetts, 1995.

[47] The Globus Alliance. See website at `http://www.globus.org`.

[48] C.A. Goble, S. Pettifer, R. Stevens, and C. Greenhalgh. Knowledge Integration: In silico Experiments in Bioinformatics. In *The Grid: Blueprint for a New Computing Infrastructure Second Edition.* Morgan Kaufmann, 2003.

[49] C.E. Goodyer, M. Berzins, P.K. Jimack, and L.E. Scales. Grid-Based Numerical Optimisation in a Problem Solving Environment. In S.J. Cox, editor, *Proceedings of Second UK E-Science All-Hands Meeting, Nottingham*, 2003.

[50] 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.*, 11(2):129–143, 1997.

[51] gViz: Visualization Middleware for e-Science.
See website at `http://www.visualization.leeds.ac.uk/gViz/`.

[52] 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.

[53] Ken Hawick, Gregory V. Wilson, et al. High Performance Computing and Communications Glossary.
See website at `http://nhse.npac.syr.edu/hpccgloss/`.

[54] Wolfgang Hocheck, Cay S. Horstmann, Gary Cornell, Paul Houle, Doug Lea, Matthew Austern, and Alexander Stepanov. The Colt Distribution. Open Source Libraries for High Performance Scientific and Technical Computing in Java.
See web site at: `http://tilde-hoschek.home.cern.ch/ hoschek/colt/`.

[55] Andreas Hoheisel and Uwe Der. An XML-based Framework for Loosely Coupled Applications on Grid Environments. In *Lecture Notes in Computer Science*, pages 245–254. Springer Verlag, 2003.

[56] E. N. Houstis, S. B. Kim, P. Wu S. Markus, N. E. Houstis, A. C. Catlin, S. Weerawarana, and T. S. Papatheodorou. Parallel ELLPACK Elliptic PDE Solvers. In *Proceedings of the Intel Supercomputer Users Group Conference*, 1995.

[57] E. N. Houstis, J. R. Rice, E. Gallopoulos, and R. Bramley. *Enabling Technologies for Computational Science: Frameworks, Middleware, and Environments.* Kluwer Academic Publishers, 2000.

[58] Andrew Hunt and David Thomas. *The Pragmatic Programmer.* Addison-Wesley, 2000.

[59] IONA. ORBacus CORBA ORB.
See web site at: `http://www.iona.com/products/orbacus_home.htm`.

[60] Ivar Jacobson. *Object-Oriented Software Engineering.* ACM Press, 1992.

[61] The Java Programming Language™. See web site at: `http://java.sun.com`.

[62] The JavaBeans Framework.
See website at `http://java.sun.com/products/javabeans/`.

[63] Java API for XML Processing (JAXP).
See website at `http://java.sun.com/xml/jaxp`.

[64] The JDOM Project. See web site at: `http://www.jdom.org`.

[65] C.R. Johnson, S. Parker, D. Weinstein, and S. Heffernan. Component-Based Problem Solving Environments for Large-Scale Scientific Computing. *Journal on Concurrency and Computation: Practice and Experience*, 14(Grid Computing Environments Special Issue 13-14):1337–1349, 2002.

[66] 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. Wiley, 1999.

[67] Katarzyna Keahey and Dennis Gannon. PARDIS: CORBA-based Architecture for Application-Level PARallel DIStributed Computation. In *Proceedings of Super Computing '97*, November 1997.

[68] Rohit Khare. On the diffusion of Christopher Alexander's *'A Pattern Language'* into Software Architecture, 1995.

[69] Richard Koch, Dirk Olmes, et al. TeXShop.
See website at `http://www.uoregon.edu/ koch/texshop/texshop.html`.

[70] Sriram Krishnan and Dennis Gannon. XCAT3: A Framework for CCA Components as OGSA Services. In *roceedings of HIPS 2004, 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2004.

[71] Sriram Krishnan, Patrick Wagstrom, and Gregor von Laszewski. GSFL: A Workflow Framework for Grid Services, 2002.

[72] Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):643–662, 2001.

[73] Gregor von Laszewski, Ian Foster, Jarek Gawor, Peter Lane, Nell Rehn, and Mike Russell. Designing Grid-based Problem Solving Environments and Portals. In *34th Hawaiian International Conference on System Science*, Maui, Hawaii, 3-6 2001.

[74] Andy Lawrence. AstroGrid: the UK's Virtual Observatory. In S. J. Cox, editor, *Proceedings of Second UK E-Science All-Hands Meeting, Nottingham*, 2003.

[75] Frank Leyman. Web Services Flow Language (WSFL) 1.1. Technical report, IBM Software Group, 2001.

[76] Li and Baker. A Review of Grid Portal Technology. In Jose Cunha and O.F. Rana, editors, *Grid Computing: Software Environments and Tools.* Springer Verlag, 2004.

[77] 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. In *Proceedings of Super Computing '00.* Super Computing 2000, 2000.

[78] H. D. Lord. Improving the Application Environment with Modular Visualization Environments. *Computer Graphics*, 29(2):10–12, 1995.

[79] Vijay Menon and Anne E. Trefethen. MultiMATLAB: Integrating MATLAB with High-Performance Parallel Computing. *Proceedings of Super Computing '97*, 1997.

[80] Bram Moolenaar. Vim the editor. See website at `http://www.vim.org`.

[81] MPI Forum. MPI: A Message-Passing Interface Standard. Technical report, Message Passing Interface Forum, 1995.
See web site at: `http://www.mpi-forum.org`.

[82] NASA Information Power Grid.
See website at `http://www.nas.nasa.gov/About/IPG/ipg.html`.

[83] National Computational Science Alliance.
See website at `http://www.vcsa.uiuc.edu/Projects/Alliance`.

[84] National Partnership for Advanced Computational Infrastructure.
See website at `http://www.npaci.edu`.

[85] J.A. Nelder and R. Mead. A Simplex Method for Function Minimization. *Computer Journal*, 7:308–313, 1965.

[86] P. Newton and J. C. Browne. The CODE 2.0 Graphical Parallel Programming Language. In *Proceedings of the ACM International Conference on Super Computing '92*, 1992.

[87] J. Novotny, M. Russell, and O. Wehrens. GridSphere: A Portal Framework for Building Collaborations. In *1st International Workshop on Middleware for Grid Computing (at ACM/IFIP/USENIX Middleware 2003)*, 2003.

[88] J. Novotny, S. Tuecke, and V. Welch. An Online Credential Repository for the Grid: MyProxy. In *Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10)*. IEEE Press, 2001.

[89] K. Nowiński, B. Lesyng, M. Niezgódka, and P. Bala. Project EUROGRID. In *PIONIER 2001 Conference Proceedings*, pages 187–191, 2001.

[90] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Greenwood, T. Carver, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics Journal*, 2004.

[91] Roberto Piola. JChart, Java Charting Component.
See web site at: `http://www.ilpiola.it/roberto/jchart/index_e.html`.

[92] Press, Flannery, et al. *Numerical Recipes in C: The Art of Scientific Computing*, chapter 10, pages 408–412. Cambridge University Press, 1992.

[93] T. Priol and C. Ren. Cobra: A CORBA-Compliant Programming Environment for High-Performance Computing. In *Proceedings of Euro-Par'98*, 1998.

[94] Ptolemy II.
See website at `http://ptolemy.eecs.berkeley.edu/ptolemyII`.

[95] Omer F. Rana, Maozhen Li, David W. Walker, and Matthew Shields. An XML Based Component Model for Generating Scientific Applications and Performing Large Scale Simulations in a Meta–Computing Environment. In *International Symposium on Generative and Component Based Software Engineering (GCSE), Erfurt, Germany.* Proceedings available on CD-ROM, 1999.

[96] S.M. Rao, D. R. Wilton, and A. W. Glisson. Electromagnetic Scattering by Surfaces of Arbitrary Shape. *IEEE Trans. Antennas Propagat.*, AP-30:409–418, 1982.

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

[98] Michael Russell, Gabrielle Allen, Ian Foster, Ed Seidel, Jason Novotny, John Shalf, Gregor von Laszewski, and Greg Daues. The Astrophysics Simulation Collaboratory: A Science Portal Enabling Community Software Development. *Journal on Cluster Computing*, 5(3):297–304, 2002.

[99] Douglas C. Schmidt. Evaluating Architectures for Multithreaded Object Request Brokers. *Communications of the ACM*, 41(10):54–60, 1998.

[100] Andreas Schreiber. The Integrated Simulation Environment TENT. *Concurrency and Computation: Practice and Experience*, 14(Grid Computing Environments Special Issue 13-14), 2002.

[101] SCIRun: A Scientific Computing Problem Solving Environment. Scientific Computing and Imaging Institute (SCI), `http://software.sci.utah.edu/scirun.html`, 2002.

[102] A. D. Scurr and A. J. Keane. The Development of a Grid Based Engineering Design Problem Solving Environment. In Peter M. A. Sloot, Chih Jeng Kenneth Tan, Jack Dongarra, and Alfons G. Hoekstra, editors, *International Conference on Computational Science*, volume 2329 of *Lecture Notes in Computer Science*, pages 881–889. Springer, 2002.

[103] A. D. Scurr, A. J. Keane, and S. J. Cox. Data-Centric Approach to a Grid Based Engineering Design Problem Solving Environment, 2001.

[104] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima. Ninf: A Network-Based Information Library for Globally High Performance Comput-

ing. In *Proceedings of the 1996 Parallel Object-Oriented Methods and Applications Conference*, February 1996.

[105] James N. Siddall. *Optimal Engineering Design, Principles and Applications.* Marcel Dekker, Inc., New York and Basel, 1982.

[106] Jon Siegel. OMG overview: CORBA and the OMG in enterprise computing. *Communications of the ACM*, 41(10):37–43, 1998.

[107] Aleksander Slominski and Gregor von Laszewski. Scientific Workflows Survey. See website at `http://www.extreme.indiana.edu/swf-survey/`.

[108] 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*, 3(3):33–43, 1996.

[109] 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)*, 1997.

[110] The Triana Project. See web site at: `http://www.trianacode.org`.

[111] The UNICORE Forum. UNICORE: UNiform Interface to COmputing REsources. See website at `http://www.unicore.org`.

[112] Visual Numerics. Java Numerical Library.
See web site at: `http://www.vni.com/products/wpd/jnl`.

[113] W3C. eXtensible Markup Language.
See web site at: `http://w3.org/XML/`.

[114] W3C. OSD: The Open Software Description Format, 1997.
See web site at: `http://www.w3.org/TR/NOTE-OSD`.

[115] W3C. Web Services Description Language (WSDL) 1.1. Technical report, W3C, 2001.

[116] W3C. Simple Object Access Protocol (SOAP) 1.2. Technical report, W3C, 2003.

[117] Nigel Warren and Philip Bishop. *Java in Practice: Design Styles and Idioms for Effective Java.* Addison-Wesley, 1999.

[118] S. Weerawarana, E. Houstis, J. Rice, et al. PDELab: an object-oriented framework for building problem solving environments for PDE based applications. Technical Report 94-21, Computer Sciences Department, Purdue University, 1994.

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

[120] Thomas Williams et al. Gnuplot: command-driven interactive function plotting program. See website at `http://www.gnuplot.info/`.

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

[122] H. Wright, K. Brodlie, J. Wood, and J. Procter. Problem Solving Environemnts: Extending the Role of Visualization Systems. In *Proceedings of the European Conference on Parallel Computing (EuroPar 2000)*, 2000.

[123] M. Young, D. Argiro, and S. Kubica. Cantata: Visual Programming Environment for the Khoros System. *Computer Graphics*, 29(2):22–24, 1995.