Swansea University E-Theses

A rapid prototyping environment for computational engineering simulation.

Turner-Smith, Edward Alan

How to cite:

Turner-Smith, Edward Alan (2002) *A rapid prototyping environment for computational engineering simulation..* thesis, Swansea University.
http://cronfa.swan.ac.uk/Record/cronfa42915

Use policy:

This item is brought to you by Swansea University. Any person downloading material is agreeing to abide by the terms of the repository licence: copies of full text items may be used or reproduced in any format or medium, without prior permission for personal research or study, educational or non-commercial purposes only. The copyright for any work remains with the original author unless otherwise specified. The full-text must not be sold in any format or medium without the formal permission of the copyright holder. Permission for multiple reproductions should be obtained from the original author.

Authors are personally responsible for adhering to copyright and publisher restrictions when uploading content to the repository.

Please link to the metadata record in the Swansea University repository, Cronfa (link given in the citation reference above.)

http://www.swansea.ac.uk/library/researchsupport/ris-support/

A Rapid Prototyping Environment for Computational Engineering Simulation.

By

Edward Alan Turner-Smith

B.Eng., M.Sc.

Thesis submitted to the University of Wales in partial fulfilment of the requirements for the Degree of ***Doctor of Philosophy***

Department of Civil Engineering

University of Wales Swansea

March 2002

C/Ph/237/02

ProQuest Number: 10821305

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 10821305

Published by ProQuest LLC (2018). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

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

To My Family

DECLARATION

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

Signed ___ ___ (candidate)

Date 5/3/2003

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 5/3/2003

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 the summary to be made available to outside organisations.

Signed ___ (candidate)

Date 5/3/2003

SUMMARY

This thesis investigates the design and development of an environment for computational engineering simulations with the aim of providing rapid prototyping capabilities. The Parallel Simulation User Environment was developed under a number of projects including CAESAR, MEDUSA and JULIUS, which were funded by the EC under the ESPRIT initiative. This thesis focuses on the author's efforts to harness a collection of computational simulation tools and produce a seamless integrated environment. Some of the main software tools developed by the author include graphical user interfaces, application integration modules and communication libraries.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to Professor N. P. Weatherill for his guidance and support, during my research activities that provided the material contained within this thesis and throughout my time in Swansea.

I would also like to give a very special "thank-you" to Dr. M. Marchant for his friendship and assistance on numerous occasions. His support was greatly appreciated and proved a valuable asset to me during my research.

Thanks also go to my colleagues J. Jones, M. Sotirakos and Y. Zheng, who were an enjoyable group to work with. Without them, I believe the PSUE package would not have been the success it is today.

I would like to thank my family, for all their support, dedication and guidance throughout my education. This thesis is a tribute to them.

Thanks also go to Claire, Graham and Janet, for their encouragement, during the writing of this thesis.

CONTENTS

1 Introduction ____ *1*

1.1 Introduction ____ **2**

1.2 Aim and Scope of this Thesis ____ **5**

1.3 Objectives of the PSUE ____ **6**

1.4 Implications of PSUE team development ____ **8**

1.5 Details of the Implementation ____ **9**

1.6 Philosophy of Approach ____ **11**

2 Graphical User Interfaces ____ *14*

2.1 Review of Graphical User Interfaces ____ **15**

2.1.1 Introduction ____ 15
2.1.2 Design of Control ____ 17
2.1.3 Workspace Elements ____ 22
2.1.4 Colour and Sound ____ 27
2.1.5 International Considerations [10] ____ 29
2.1.6 Errors [7][5] ____ 30
2.1.7 Utilising Help [6][4][7][12] ____ 31

2.2 Implementation of GUI Guidelines ____ **33**

2.2.1 Introduction ____ 33
2.2.2 Design of Control ____ 38
2.2.3 Workspace Elements ____ 44
2.2.4 Colour and Sound ____ 48
2.2.5 International Considerations ____ 48
2.2.6 Errors ____ 49
2.2.7 Utilising Help ____ 49

3 PSUE Main Interface ____ *51*

3.1 Overview of the Main Functionality **52**
3.1.1 Data Management 54
3.1.2 Geometrical Manipulations 55
3.1.3 Grid Generation 56
3.1.4 Solvers 57
3.1.5 Computing Platforms 60
3.1.6 System Tools 66
3.1.7 Data Analysis 68

3.2 Resource Management **69**
3.2.1 Colour Preferences 69
3.2.2 Font Preferences 70

3.3 Process Control **72**

3.4 Pathfinder **74**
3.4.1 Pathfinder View 74
3.4.2 History View 76

3.5 On-line Help **77**

3.6 Integration of Applications **79**

4 Communication Library ***80***

4.1 Overview and Requirements **81**

4.2 Comparison of Various Techniques **86**
4.2.1 IPC Facilities 86
4.2.2 Methodology 87

4.3 Data Set Manipulation **90**
4.3.1 Data Types 90
4.3.2 Shared Memory Segment Breakdown 95

4.4 Shared Memory Communication System **96**
4.4.1 Initialisation of the PSUE Main Interface 96
4.4.2 Initiation of a PSUE Module 97
4.4.3 PSUE Module updates Main Interface 98
4.4.4 Main Interface updates PSUE Module 98
4.4.5 Closure of a PSUE Module 99
4.4.6 Closure of the PSUE Main Interface 99

4.4.7 Error within the PSUE Main Interface ____ 100

5 Application Integration ____ 101

5.1 Overview and Requirements ____ 102

5.2 Philosophy ____ 104

5.3 Development Stages ____ 106

5.3.1 Application Button Integration ____ 106

5.3.2 Application Initiation ____ 108

5.3.3 Data Transfer ____ 108

5.3.4 Automation ____ 109

5.4 Techniques Employed ____ 111

5.4.1 Script Files ____ 111

5.4.2 Application Initiation Procedures ____ 117

5.4.3 Pipes and Sockets ____ 118

5.5 Application Integration Library ____ 122

5.5.1 Connection and Disconnection with the PSUE ____ 122

5.5.2 Sending and Receiving Arbitrary Data ____ 123

5.5.3 Extended Data Extraction Facilities ____ 123

5.6 Full Example ____ 125

5.7 Summary ____ 130

6 Application Tool Wrapper ____ 131

6.1 Introduction ____ 132

6.2 Architecture ____ 134

6.2.1 Techniques Employed ____ 136

6.3 Examples ____ 139

6.3.1 Example 1 – Surface Grid Definition ____ 139

6.3.2 Example 2 – Complex Geometry Definition ____ 140

6.4 Summary ____ 145

7 Test Cases _____ *146*

7.1 Introduction _____ **147**

7.2 Test Case 1 – Dassault Falcon _____ **148**

7.2.1 Geometry Manipulation _____ 148

7.2.2 Preparation for Grid Generation _____ 149

7.2.3 Grid Generation and Partitioning _____ 150

7.2.4 Grid Quality Evaluation _____ 152

7.2.5 Parallel Flow Solver _____ 153

7.2.6 Grid Adaptation _____ 154

7.2.7 Final Solution _____ 155

7.2.8 Summary _____ 156

7.3 Test Case 2 – Airbus A3XX _____ **158**

7.3.1 IGES Import _____ 158

7.3.2 Geometrical Repair _____ 158

7.3.3 Grid Generation _____ 159

7.3.4 Electromagnetic Solver _____ 159

7.3.5 Data Analysis _____ 160

7.3.6 Summary _____ 161

7.4 Test Case 3 – F16 Fighter _____ **162**

7.4.1 Geometry Preparation _____ 162

7.4.2 Parallel Grid Generation _____ 162

7.4.3 Parallel Solver and Adaptation _____ 163

7.4.4 Virtual Reality Data Analysis _____ 163

7.4.5 Summary _____ 164

8 Conclusions _____ *166*

8.1 Overview _____ **167**

8.2 Discussions _____ **169**

8.3 Future Developments _____ **172**

9 References _____ *174*

LIST OF FIGURES

Figure 1-1 - Overview of the PSUE Components 13
Figure 2-1 - General Layout of Key Areas of a Module 34
Figure 2-2 - General Layout of Key Features in Popup Dialog Windows 35
Figure 2-3 - Typical Size and Appearance of Buttons, Text Fields and Other Widgets 36
Figure 2-4 - Competency Switch 37
Figure 2-5 - Typical View of a Status and Message Window 38
Figure 2-6 - Small Section of the Pathfinder Module 39
Figure 2-7 - A Typical Push Button 40
Figure 2-8 - A Typical Toggle Button 40
Figure 2-9 - A Typical Option Button 40
Figure 2-10 - A Typical Text Entry Field 40
Figure 2-11 - A Typical List 40
Figure 2-12 - Typical Example of Defaults and Specifications 41
Figure 2-13 - Example of Ghosted / Unavailable Functionality 42
Figure 2-14 - A Progress Indicator Using a Percentage Bar 43
Figure 2-15 - Comparison between the two different types of interfaces 44
Figure 2-16 - Panel Showing Close Proximity of Related Functionality 45
Figure 2-17 - Pulldown Menu Shown with Tear-off Functionality 46
Figure 2-18 - Example of an Option Menu 46
Figure 2-19 - The Help Window with Index Facility 50
Figure 3-1 - The PSUE Main Interface 53
Figure 3-2 - The Data Management Functionality 54
Figure 3-3 – The Geometry Manipulation Functionality 55
Figure 3-4 – The Grid Generation Functionality 56
Figure 3-5 - The Solver Functionality 57
Figure 3-6 - The Flow Solver Initiation Panel 58
Figure 3-7 - The Flow Solver Generic Settings Panel 59
Figure 3-8 - The Computing Platforms Functionality 61

Figure 3-9 - The Parallel Tools Functionality 61
Figure 3-10 - The Remote Connection Tool - RECON 62
Figure 3-11 - The Machine Connection Panel for RECON 63
Figure 3-12 - The CODINE Connection Panel 64
Figure 3-13 - The Optimisation Control Panel 65
Figure 3-14 - The System Tools Functionality 67
Figure 3-15 - The Data Analysis Functionality 67
Figure 3-16 - Typical View for the Colour Resource Manager 70
Figure 3-17 - Typical View of the Font Resource Manager 71
Figure 3-18 - Typical View of the Process Control Panel 72
Figure 3-19 - Typical View of the Pathfinder Panel 75
Figure 3-20 - Typical View of the History Panel 76
Figure 3-21 - Typical View of the On-Line Help Facility 78
Figure 4-1 - Client / Server Topology 81
Figure 4-2 - Daemon Controlled Topology 82
Figure 4-3 - Specific Data Sets Transferred Between the PSUE and Modules 84
Figure 4-4 - Communication Methods and the Type of Data Transmitted 87
Figure 4-5 - Comparison of the Socket and Shared Memory Communication Methods 89
Figure 5-1 - Structure of the PSUE Integration System 105
Figure 5-2 - Example of Three User Applications Integrated into the Main Functionality Region. 107
Figure 5-3 - Application Initiation Panel 109
Figure 5-4 - Comparison of Application Integration with and without Automation 110
Figure 5-5 - Flow Diagram of the Main Part of the Script File Parser 115
Figure 5-6 - Flow Diagram of the Automation Part of the Script File Parser 116
Figure 5-7 - Comparison of the Two Application Initiation Methods 118
Figure 5-8 - Connection Procedure for the Pipe System 120
Figure 5-9 - Connection Procedure for the Socket System 121
Figure 5-10 - View of the Grid Generation Functionality Region 126
Figure 5-11 - Original Main Routine of the Tetragrid Application 127

Figure 5-12 - Final Main Routine of the Tetragrid Application _____ 128
Figure 6-1 - Comparison of the Application Integration and Tool Wrapper _____ 136
Figure 6-2 - The Tool Wrapper Configured for a Surface Mesh Definition _____ 139
Figure 6-3 - The Tool Wrapper Configured for a Complex Geometry Definition _ 141
Figure 7-1 - Falcon Geometry in the Geometry Builder _____ 149
Figure 7-2 - Falcon Geometrical Configuration with Grid Sources _____ 150
Figure 7-3 - Surface and Volume Grids for the Falcon Geometry _____ 151
Figure 7-4 - Typical View of the Histogram Chart of the Grid Quality Statistics __ 153
Figure 7-5 - Grid Adaptation Module Showing the Point Cloud Around the Falcon155
Figure 7-6 - Final Solution of the Airflow Around the Dassault Falcon _____ 156
Figure 7-7 - Summary of Data Flow for Falcon Test Case _____ 157
Figure 7-8 - Summary of Data Flow for A3XX Test Case _____ 161
Figure 7-9 - Summary of Data Flow for F16 Test Case _____ 164

LIST OF TABLES

Table 1-1 – Members of the Development Team 8

Table 2-1 – Typical Guidelines for a Graphical User Interface 15

Table 4-1 – The Shared Memory Segments and their Content 95

Table 4-2 – Actions Performed during Initiation of a PSUE Module 97

Table 4-3 – Actions Performed when Updating the Main PSUE Interface 98

Table 4-4 – Actions Performed when the Main Interface updates a PSUE Module 98

Table 4-5 – Actions Performed during the Closure of a PSUE Module 99

Table 5-1 - Script Files Controlling Functionality Regions 107

Table 5-2 - Connection and Disconnection Routines 122

Table 5-3 - Arbitrary Send and Receive Routines 123

Table 5-4 - A Selection of the Extended Data Extraction Routines 123

Table 6-1 - Advantages and Disadvantages of the Application Integration and the Tool Wrapper 135

Table 6-2 - Example Set of Key Tool Wrapper Items 137

Table 7-1 - A Summary of each Stage's Tasks and Data Transfer for Test Case 1 157

Table 7-2 - A Summary of each Stage's Tasks and Data Transfer for Test Case 2 161

Table 7-3 - A Summary of each Stage's Tasks and Data Transfer for Test Case 3 164

GLOSSARY

API	Application Protocol Interface
ASP	Application Service Provider
AVS	Advanced Visual Systems – Proprietary post-processing software
CAD	Computer aided design
CAESAR	CEC Esprit Project – Clusters of Computationally Intensive Applications for Engineering Design and Simulation on Scalable Parallel Platforms
CEM	Computational Electro Magnetics
CFD	Computational Fluid Dynamics
Check Button	A widget that allows the user to switch options on and off.
CODINE	Proprietary load balancing software
Dialogs	Windows that are sub-windows of the primary window of the application
Ensight	Proprietary post-processing software
FEA	Finite Element Analysis
Flite3D	UWS grid generation and CFD solver suite
Front-End	A GUI for a specific application
FTP	File Transfer Protocol
GAA	Generic Authorisation and Access
GSI	Grid Security Infrastructure
GUI	Graphical User Interface
HCI	Human Computer Interaction
HPCN	High Performance Computing and Networking
IP	Internet or Intercommunication Protocol
IPC	Inter Process Communications
IRIS/Explorer	Proprietary post-processing software

JULIUS	CEC Esprit Project – Joint Industrial Interface for End-User Simulations
MEDUSA	CEC Esprit Project – Multi-disciplinary Engineering Design via Unitary Software Applications
Menubar	A menu that is always located at the top of the primary window of an application
Menus	Widgets that help group other widgets – used as menubars, option menus, pulldown menus and popup menus
Motif	Window library based on X Windows
MPI	Message Passing Interface
MPP	Multi-Processor Platform
MSC/Patran	Proprietary pre-processing software
MSC/Nastran	Proprietary FEA software
OpenGL	Open Graphics Library from SGI
Options Menu	A menu widget that allows the user to select one of a multiply number of options
Panels	Windows that are sub-windows of the primary window of the application
ParaGraph	Front-end to MPI
PC	Personal Computer – Usually Intel Technology
Popup Menu	A menu widget that groups options and presents them in a list that appears where the mouse is located
PSUE	Parallel Simulation User Environment
Pulldown Menu	A menu widget that is part of a menubar or option menu
Push Button	A button widget that activates a particular task or action.
PVM	Parallel Virtual Machine
Radio Button	A button widget that when multiples are grouped together into a radio box allow the user to switch between multiple choices
SCCS	Source Code Control System
SDK	Software Development Kit
SMP	Shared Memory Platform

Socket	Bi-directional data stream for transferring data using a particular network protocol
SSP	Storage Service Provider
TCP	Transmission Control Protocol
Text Field	A widget that allows the entry of text and / or numerical values
TLS	Transport Layer Security
UDP	User Datagram Protocol
UNIX	Operating systems used by professional workstations, MPPs and SMPs
VPN	Virtual Private Network
Widget	A Motif component of a window system that could be a button, label, frame, drawing area, text areas etc
XML	Extensible Markup Language
XPVM	X-Based Parallel Virtual Machine
X Windows	Window based system for UNIX platforms

1 INTRODUCTION

1.1 Introduction

Computers have now been used in research and industry for quite some time however the technological advancement of computers is still very rapid. In recent years, there has been a large increase in new software and extensive developments to existing packages supporting multidisciplinary computational engineering. To carry out most computational simulations, an end-user is required to employ a collection of these software packages, usually creating a complex sequence of events. As computer platforms rapidly increase in performance and reduce in cost, research and industrial sectors are increasingly able to exploit High Performance Computing and Networking (HPCN). However such facilities can further complicate the simulation process.

The industrial sector is progressively increasing product complexity, yet demands a reduction in the time required for each stage of the simulation and requires the ability for rapid execution of problem set-up. The complexity of the product definitions and simulation requirements also increases the volume of computational data involved. This in turn generates communication bottlenecks and platform limitations that may also constrain the extent of the problem even further.

Clearly, there is a requirement for a multidisciplinary engineering environment that encompasses all of the key stages and controls the overall computational simulation and product design. Such an environment would need to allow for the integration of arbitrary application software with the efficient but transparent communication of computational data between the key stages. The bottlenecks and limitations associated with the computational data and engineering software should be minimised by using facilities such as direct inter-process communications and shared memory management.

At the time that the research for this thesis was started, use of computers in computational engineering was widespread. However, in order to perform a typical solution of an engineering problem, a number of stages would be required, e.g.

geometry creation and manipulation, pre-processing, analysis and finally post-processing. Often, each of these stages would require the execution of a number of programs. This process is usually a cumbersome activity and so the research behind this thesis is based upon the concept of integrating all of the modules and tools. This would produce a single easy-to-use environment.

Such an integrated environment that can also be referred to as a problem solving environment, would need to incorporate many of the details discussed so far. An integrated environment should provide some generic functionality but most importantly allow for the direct integration of the software that it targets. The environment should assist in the seamless connection between separate modules that should also include data communications.

A successful integrated environment would need to be able to assist the simulation engineering process and would provide a single unified interface. An environment should improve the efficiency of the solution process and increase the effectiveness of the tools and should also speed up training of users and the progression of the users skills. Finally, the environment should generate a tool that is greater than the sum of all of its constituent parts.

Some of the major challenges of computational engineering that can be overcome with an environment include:

- Easier and faster execution of the total simulation process.
- Seamless connections between separate software packages and hardware platforms.
- Users hidden from differences between data structures.
- Automatic data transfer.
- Management of data volume and bottlenecks.

This thesis, "A Rapid Prototyping Environment for Computational Engineering Simulation", explores just some of the possibilities that are currently available for the

exploitation of computing technology. In the author's own opinion, 'Computational Engineering Simulation' can be considered as the ability for engineers to model multi-disciplinary engineering problems on computers, usually to a high degree of accuracy. The main benefits of using a computer to analyse a model as compared to conventional methods, e.g. computational fluid dynamics compared with wind tunnel measurements, are the time taken to get results, hardware requirements, manpower requirements and therefore costs. Also in the author's own opinion, a 'Rapid Prototyping Environment' is a tool that will allow the user to set-up and execute a particular engineering problem as swiftly as possible and allow amendments to the problem definition to be carried out easily.

The rapid prototyping environment presented in this thesis is known as the "Parallel Simulation User Environment" (PSUE) [1]&[2]. The PSUE was initially designed under a CEC ESPRIT project called "CAESAR" (Clusters of Computationally Intensive Applications for Engineering Design and Simulation on Scalable Parallel Platforms) which was to ascertain the feasibility of such an environment. A second CEC ESPRIT project called "MEDUSA" (Multi-disciplinary Engineering Design via Unitary Software Applications) involved further development of the environment with the PSUE as the core software. Due to the success of the PSUE development, a third CEC ESPRIT project was initiated called "JULIUS" (Joint Industrial Interface for End-User Simulations) which aimed to produce a new, more intelligent engineering environment based on parts of the PSUE environment.

1.2 Aim and Scope of this Thesis

The aim of this thesis is to present the author's involvement in the development of the Parallel Simulation User Environment (PSUE). Since the author has not developed all of the PSUE software, and the PSUE encompasses a large number of computational engineering tools, only certain areas of the PSUE will be presented within this thesis. The main modules that will be discussed include:

- The main PSUE interface which controls and communicates with all the other modules of the PSUE.
- The communication library used to control and transfer data.
- The application integration facility that allows any arbitrary application to be integrated into the environment.
- The application tool wrapper that combines the application integration facility and the communication library.

Chapter 2 discusses the study and implementation of graphical user interfaces. Chapters 3 through to 5 discuss the main PSUE interface, the communication library and the application integration, respectively. Chapter 6 discusses the tool wrapper, which is an extension and unification of the communication library and application integration facilities. Chapter 7 presents three test cases showing typical use of the PSUE and finally, Chapter 8 concludes the research covered by this thesis.

1.3 Objectives of the PSUE

As previously mentioned, the PSUE was initially developed under the CAESAR project, and was intended to be a demonstrator project in order to ascertain whether the concept of an integrated environment was feasible. The main intentions of this project were to integrate the tools required to carry out a full solution of an engineering problem into a single easy-to-use environment. The environment was also required to be applicable to various multi-disciplinary engineering applications and to utilise the fast growing High Performance Computing and Network (HPCN) capabilities of computers.

The main tools required for a simulation that were integrated within the environment include:

- Geometry importation, creation and manipulation.
- Topology construction.
- Grid generation.
- Solver initiation and control.
- Visualisation of results.

During the initial development of the environment a number of key objectives were identified by industrial end-users [3]:

- Reduction of problem set-up time.
- Reduction of user training period.
- A modular framework for continued development.
- Integration of arbitrary engineering simulation applications.
- Exploitation of High Performance Computing and Networking (HPCN).

The development of the PSUE continued under the MEDUSA project, in which further modules where integrated within the environment which involved further pre-processing, analysis, post-processing and load balancing facilities.

The JULIUS project proposed a fully integrated engineering environment based on the PSUE functionality and successes. The main development that the author was involved with, during this project, was the extension and amalgamation of the communication library and the application integration capabilities to produce the application tool wrapper.

1.4 Implications of PSUE team development

Since the PSUE has been developed over a number of years and within a number of projects, the software has been designed and written by a number of individuals. The members of the development team are shown in Table 1-1 below.

Table 1-1 – Members of the Development Team

Name	Position	Main Responsibilities
Prof. N.P. Weatherill	Team Leader	Overall Project Leadership
Dr. M.J. Marchant	Team Supervisor	Project Management Team Advisor
Dr. O. Hassan	Team Consultant	Grid Generation
Dr. Y. Zheng	Software Developer	Geometry Manipulation Grid Generation Interface
Mr. E.A. Turner-Smith	Software Developer	Main PSUE Interface Communication System Application Integration Grid Generation Tool Wrapper
Mr. M. Sotirakos	Software Developer	Grid Analysis Parallel Tools
Mr. J. Jones	Software Developer	Help System Parallel Visualisation
Mr. R. Said	Research Student	Grid Generation

This thesis describes the sections of the PSUE that have been designed and developed by the author. In certain areas, the author has worked in collaboration with other developers or has taken over certain responsibilities. As this occurs the thesis will explain which parts were the responsibility of the author and which were the responsibilities of another member of the development team.

Chapter 7, which presents a number of test cases, covers all aspects of a simulation and therefore covers more areas of the PSUE than the author was responsible for. Further details will be given within the chapter.

1.5 Details of the Implementation

In recent years, PC based machines have become extremely powerful. However at the time of this research only UNIX based machines were powerful enough to execute most engineering simulations required by the industrial sector.

Generally UNIX machines, compared to PC machines, had faster and more powerful processors, multi-processor facilities, greater memory capabilities [14], faster and larger file storage, and superior system boards and were generally produced to a higher specification.

Most PC machines run Microsoft DOS or Microsoft Windows as an operating system whereas UNIX machines all operate variations of the UNIX operating system. The UNIX operating system may be considered superior to the Microsoft operating system since it provides, amongst other facilities:

- True multi-tasking capabilities.
- Better process control.
- A stable and robust environment.
- Extended inter-process tools

The opinions presented so far within this chapter led to the decision that the PSUE should be developed within a UNIX environment and possibly converted to a Microsoft environment in the future if necessary.

Within the UNIX operating system a standard has been established for most window programs to follow. This involves the use of the X Window system with a very useful extension known as the Motif widget libraries. Motif provides an easy to use toolkit and produces a windows program that gives an aesthetically pleasing representation using three-dimensional aspects. All of the PSUE window modules have been developed using the X Window system and the Motif widget library.

The graphical manipulation of data within the PSUE has been developed using the OpenGL graphics library originating from the Silicon Graphics UNIX platform. OpenGL provides efficient and powerful facilities to present and manipulate data in real time on a standard UNIX based machine. OpenGL is becoming the standard for graphics on such machines and therefore was used for all graphics within the PSUE development.

UNIX operating systems contain a number of very powerful Inter-Process Communication (IPC) facilities [15] such as shared memory, semaphores, message queues, pipes and sockets. These provide an excellent method of communicating data efficiently between separate processes and have been used throughout the development of the PSUE, giving the environment its seamless presentation.

The shared memory capabilities have been used extensively to transfer the usually very large data sets between the various modules of the PSUE. Since the data sets of the PSUE are quite specific, a communication library based on shared memory has been implemented within the PSUE development and provides the first line of data transport. Shared memory allows the data owned by one process to be accessed by any number of other processes.

As mentioned above, the PSUE has been developed using the X/Motif window system and OpenGL, which have been developed in the C programming language [16-20]. However, a number of tools that have been integrated into the PSUE had already been developed in the Fortran77 programming language. Therefore, the PSUE has been developed using both the C and Fortran77 programming languages [14].

1.6 Philosophy of Approach

Since the development goals of the PSUE, described previously in the chapter, were so extensive, a number of software developers would be required. A major software development project requiring a number of personnel to work simultaneously can be very difficult to manage. Initially for this reason, the development work was broken up into a series of modules that would finally reunite to produce the final software package.

However, as the software development continued it soon became clear that a general modular framework had many advantages over a fully integrated system:

- Easier software programming – Generally, each software developer may have their own module to develop and so a Source Code Control System (SCCS) is not as necessary as if all developers were updating the same source code files simultaneously.
- A reduction in process overheads – In a fully integrated system the resources required for all aspects of the package would be loaded at all times, however with a modular system, only the resources for a particular module that is running would be loaded.
- A reduction in the complexity of the interfaces – Instead of having one large interface that contains all options and functionality, a module would only have the options and functionality pertaining to that specific module. Therefore, the interface is less cluttered with controls that are redundant at that specific time with respect to the user's current project.
- Multiple task access – Since multiple modules may run simultaneously, and due to the excellent multi-tasking capabilities of the UNIX operating system, each module could be executing tasks. For example, a user could be setting up a new geometrical problem definition at the same time as generating a finite volume grid on another problem definition at the same time as post-processing yet another problem.

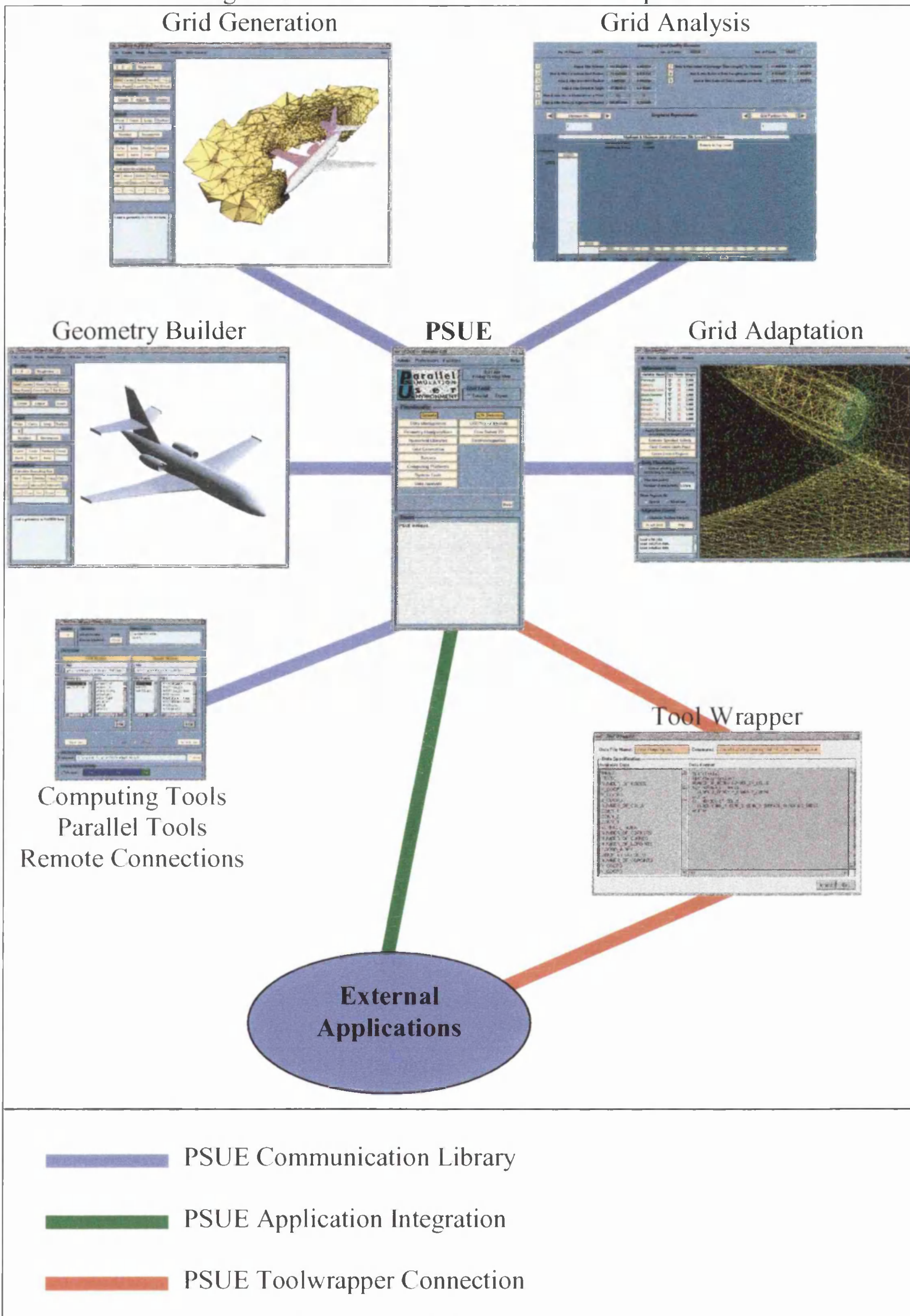
- Multiple problem definitions – Similarly to the previous point, since the user may initiate multiple copies of the same module, the user is able to process multiple problem definitions simultaneously. For example, generating three finite element grids on different problem definitions all at the same time.

The software consists of a number of modules that are integrated into an environment using a communication library that utilises the Inter-Process Communication (IPC) facilities. A simple menu driven control interface, the main PSUE Interface, controls all of the other modules and manages the communication library. It also controls the arbitrary application integration since there is also the communication between these packages and the PSUE itself.

The key components of the PSUE consist of the main control interface, geometry editor and builder, grid generators, grid quality assessments and grid adaptation facilities. Links to HPCN are provided, where appropriate, within these modules and when coupled with the ability to integrate arbitrary engineering application software, to the PSUE through the communication library, the environment becomes a powerful tool encompassing many of the present requirements specified by industry.

Figure 1-1 shows an overview of the PSUE components.

Figure 1-1 - Overview of the PSUE Components



2 Graphical User Interfaces

2.1 Review of Graphical User Interfaces

2.1.1 Introduction

The PSUE, so far, has been presented as an environment however as far as the end-user is concerned, it can be seen as basically a Graphical User Interface (GUI). This section is a review of material relating to the layout and method of building a graphical user interface [4-13]. This documentation provided a compact set of guidelines that were used whenever and wherever possible in the design of the PSUE.

There has been a vast amount of work in this area, especially more recently with the advancement of the graphical capabilities of computers. Graphical User Interfaces are becoming more widespread and so an interface must be of a very high standard to succeed. The window manager used by SUN Microsystems called "OPEN LOOK" followed the guidelines [4] as shown in Table 2-1 below.

Table 2-1 – Typical Guidelines for a Graphical User Interface

Goal	Approach	Benefit
Easy to learn.	Based the interface on a small number of simple concepts.	The system is easy to learn initially, and users can get work done right away.
Easy to learn new applications for experienced users.	Provide a consistent interface. For example, both the window manager and applications work the same way.	The user can leverage knowledge and learn new applications.
Efficient for experts.	Minimise keystrokes and mouse travel.	Increased productivity.
Distinctive user interface.	Design a visually uncluttered and consistent look.	The open look UI becomes a signature that people recognise.
Easy migration for other interface uses.	Harmonise with other user interfaces.	Users can move easily among the major user interfaces with minimal retraining.

2.1.1.1 Consistency

Consistency [12][6][7] helps users to move from an application that they are familiar with to another application, bearing in mind that the standards must be very good to start with. In practise, consistency is difficult to uphold, since many variations can be applied to a single operation and interpretation is different for various users. For example, an hourglass [7] used for showing that the computer is busy could be shown as:

- An hourglass on its own.
- An hourglass with sand moving to indicate time left.
- Figures to show estimated time of completion.
- An icon flashing at a particular rate.

For applications with many sets of "windows", large duplications generated in the name of consistency may be very confusing for the user and they may lose their place within the overall package. The main areas that need to be consistent are such things as mouse button actions [12]. For example, if mouse motion while certain mouse buttons are depressed produces a particular type of transformation, then this should be the same within all graphical windows.

Users expect components to behave consistently across all applications - push buttons always perform an action and option buttons always provide selections [12]. Due to this, when users want to perform an action they look for a push button, usually in a menu, but they do not look for an option button. Components that are provided should be used when appropriate rather than creating new ones and the look of a component should not be altered so drastically that its type is unrecognisable.

The layout of the application windows [12] should be designed according to the natural use of order and the natural scanning order of the people who will be using the application:

- Design for the natural use of order. Consider the tasks that the user will perform within the application.

- The natural scanning order is most important when arranging small groups of components to help the user find the correct component for the task. The most important and most common commands should come first. In most cases, this order is from left to right and from top to bottom [11]. However, this may be different for international considerations.

2.1.1.2 Levels of Competence [4]

When users are learning an application they require assistance to be able to complete tasks and understand what they are actually doing. However, as these users familiarise themselves with the application, they no longer need as much help and may want to work faster. Below are three levels of competence together with possible actions to assist the user who is at that level:

1. *Novice users* – The number of possibilities or routes to take should be kept small and the novice user should be able to carry out a few small tasks easily to build confidence and to reduce anxiety. Step by step on-line tutorials together with comprehensive manuals may be very useful for complicated sections of an application or for the whole application.
2. *Knowledgeable intermittent users* – Recognition rather than recall is very important. The user may have forgotten how to do a certain task, but if there is an icon / button / menu for this task then the user will know which way to go. Consistent sequences of actions, meaningful messages, and frequent prompts will inform the user that they are progressing correctly.
3. *Experts* – They seek to get their work done rapidly. They demand rapid response times, brief and less distracting feedback, and the capacity to carry out tasks with just a few keystrokes or selections. Shortcuts, abbreviations, hot-keys, macros etc are all good accelerators.

2.1.2 Design of Control

2.1.2.1 Usability [12]

The application should never destroy the user's motivation. Error messages should be informative but never sarcastic and allow retrieval of information wherever and

whenever possible. Users should be informed at all times if the application is busy and preferably how much longer they have to wait. After an application has performed a task the user should be informed of the result - good or bad. Users understand faster and better if they can see what they can do and see changes as they do them.

The abbreviation of input commands [8] can be very useful, especially for the expert user. Some abbreviation strategies are:

- Simple truncation (e.g. PR for print)
- Vowel drop with simple truncation (e.g. DVLP for develop)
- First and last letter (e.g. ST for sort)
- First letter of each word in a phrase (e.g. WYSIWYG [12])
- Standard abbreviations from other contexts (e.g. QTY for quantity)
- Phonics (e.g. XQT for execute)

2.1.2.1.1 User's Memory [8]

The user should not be expected to remember too many items of information at the same time, so usually four items of information should be the most. Users remember how to retrieve items easier than actually remembering the items themselves. For example, instead of remembering a file name, an application may list the last files accessed.

2.1.2.1.2 Computer Jargon [8]

Computing jargon such as “PORT” and “RAM” etc. apply also to commonly used words and so care should be taken when using such words in dialogs, since the “user's language” may not fully understand the terms.

2.1.2.2 Modifications [12]

User control is very important. The user should tell the application what to do, not the application prompting the user all the time for responses. A program that always shows an initial screen, that always requires confirmation of actions that assumes

data-entry in particular ways, soon becomes tedious. Users should be allowed to change the way they use the system e.g. date entry as DD/MM/YY or MM/DD/YY or even just a default where the application gets the date from the computer.

The application should be user modifiable [12] since different users will have different requirements such as inputs by icons or words etc. These sorts of entities should be gathered into the user's preference section and stored so that every time the user accesses the application the preferences do not have to be changed. Sensible default values should be initially provided and then changed if so desired by the user.

2.1.2.2.1 User Flexibility [12]

Good user-application interaction should also allow user flexibility. No matter how well an application is designed, some users will not like parts of it and they will want to change these elements. For example, from simple elements like the colours and fonts, up to complicated elements like the default values. Users should be allowed to adjust elements of an application because it increases their sense of control over it. The following attributes of an application should be considered for user customisation:

- Application parameters
- Colours
- Fonts
- Default values
- Labels
- Messages
- Help information

2.1.2.2.2 Short Cuts [8]

Short cuts for expert users are very important since they will not need so many prompts. The most basic short cut is a "hot key", however such a short cut must not be essential for the application's operation, but they must be reasonably easy to discover by beginners.

2.1.2.3 Indicating Action of Components [12]

When an application behaves as expected and the user is not surprised by the results of the actions, the user can complete tasks quickly. The first step to consistent interaction is to provide cues to the result of every action. This means that the component's shape, label, and graphics should indicate actions of components. It also means that the actions and interactions of components should remain consistent so the user always knows what to expect.

It also means that interactions should be simple. As interactions become complicated, it also becomes difficult to visually represent the interaction. Complicated interactions and components create the possibility for more errors. Even the most complicated concepts can be clarified by careful organisation, so if an application's interactions seem complicated, they should be reorganised for simplicity.

2.1.2.3.1 Showing Labels [10]

One of the best indicators of the action of a component is its label and this can be either text or graphical. Labels should be chosen carefully to indicate the action of each component. Components that perform actions should be labelled with active verbs. Components that present options should be labelled with nouns. You should also label component groups, including panels, with nouns to indicate the contents of the group. E.g chdir standing for "Change Direction" is to obscure, perhaps the full label should be used in these circumstances.

2.1.2.3.2 Showing Defaults [7]

The application should use default values for common settings or obvious selections. For example:

- The default value for a text area should be in the text area in the selected state whenever text entry is requested.
- The default selection in a list should be set in the selected state whenever a list selection is requested.

- The default radio button should be filled in a panel at application start-up time. In any case, once the state is changed, the new state should take the place of the default until the state is reset.

However, certain changed values may need to be stored by the user e.g. colours or fonts, so the application should make this possible.

2.1.2.3.3 Showing Unavailable Components [4]

As the state of the application changes, certain components become inappropriate. For example, minimisation of a window is inappropriate when the window is already minimised. In such cases, the application should make the inappropriate components unavailable - this is also called disabling the components. Disabled components should be visually de-emphasised, usually by greying the label of the component.

2.1.2.3.4 Showing Action Using Graphics [11][12][4]

Many components also include a small graphic symbol following the label to indicate the action of the component. Cascade buttons should use an arrow graphic that points in the direction the cascading menu will appear. Option buttons should use a rectangular graphic to distinguish them from push buttons.

2.1.2.4 Feedback [12]

Another important element to user interaction is providing feedback about the current state of the application. This is done, as described in the previous section, by using labels and graphics and by keeping the interface consistent. The application should also dynamically indicate the state of the application's actions. For example, the mouse pointer shape changes to indicate when and where special actions can occur.

2.1.2.4.1 Showing Progress and Response Times [4]

If an action takes a long time to complete, the user may mistake the delay to mean that the system or the application has stopped working. For actions that take a long time to complete, the application should indicate that there will be a delay with a working

dialog and the mouse pointer should change to an hourglass. If the application can track the progress of long actions, it should try to update the working dialog with the progress of the action, such as in a percentage complete dialog.

Recommended response time [5] guidelines:

- Users prefer shorter response times.
- Longer response times (greater than 15 seconds) are disruptive.
- Shorter response times lead to shorter user think times.
- A faster pace may increase productivity, but error rates may also increase.
- Response time should be appropriate to the task:

- Typing, cursor motion, mouse selections	50 to 150 milliseconds
- Simple frequent tasks	less than 1 second
- Common tasks	2 to 4 seconds
- Complex tasks	8 to 12 seconds

- Users should be advised of long delays.
- Modest variability in response time is acceptable.
- Unexpected delays may be disruptive.
- Empirical tests can help to set suitable response times.

2.1.2.4.2 Providing Warnings [7]

Certain actions can cause destructive results, such as closing an application before saving changes in the current file. Applications should not disallow such destructive actions; instead, they should warn the user of the consequences with a warning dialog. The warning dialog must allow the user to rectify, cancel or ignore the destructive action, however too many warning dialogs can be disruptive to the user's main task. Warning dialogs should be reserved for truly destructive actions.

2.1.3 Workspace Elements

2.1.3.1 Graphical Layout [11]

The constraints of screen width and length, display rate, character set, and highlighting techniques strongly influence the graphical layout of workspace

elements. Great care and planning must be carried out to ensure a friendly graphical user interface is produced which contains all of the required functionality represented consistently.

2.1.3.1.1 Screen Layout [12][13]

Some of the more general guidelines are as follows:

1. Layout guidelines:
    - Balance.
    - Regularity, symmetry, predictability.
    - Sequential.
    - Proportion.
2. Font guidelines:
    - For text use lower case with an upper case initial letter.
    - Use proportional spacing.
    - Stay with one font style.
3. Use of words guidelines:
    - No jargon.
    - Use short familiar words.
    - Use complete words; avoid contractions, short forms.
    - Use positive terms.
    - Do not include punctuation for abbreviations, mnemonics and acronyms.
4. Text on screen guide lines:
    - Keep text up to 30-35 characters wide.
    - Use short sentences of familiar words.
    - Place a full stop at the end of each sentence.

2.1.3.2 Menu Design [12]

There are four main types of menus:

- Pulldown Menus - are pulled down from a cascade button. Cascade buttons should always be available in the context that they are needed. Menus can also contain

cascade buttons so that a number of menus may be nested. The menubar is a horizontal collection of cascade buttons.

- Tearoff Menus - are a combination of a tearoff button and another menu that is usually a pulldown menu. A tearoff button contains a dashed line graphic representing perforations. When the tearoff button is activated, the menu changes into a dialog box. A tearoff menu is useful when you do not want the menu to disappear after a menu selection.
- Popup Menus - are context sensitive, but give no cue to their existence. They are popped up when the user presses a particular mouse button (usually the right hand button) over a component with an associated popup menu. Popup menus should only be used to provide shortcuts, since new users of an application may not realise or remember that they exist.
- Option Menus - provide a means of making a selection from a set of choices and takes up only a very small space. An Option Menu is popped up from an option button, which is distinguished by a bar graphic usually on the right side of the button.

Menus are composed of titles, elements, mnemonics, and accelerators. A menu's title should be unique to avoid confusion and the title should clearly indicate the purpose of the menu. The most basic menu elements include labels, separators, push buttons, toggle buttons, and cascade buttons. Either a text label or a graphical icon can identify the elements and a mnemonic provides a quick way to access menu elements from the keyboard. While the location cursor is in a menu or menubar, pressing the mnemonic letter of an element activates that element. An accelerator provides a way to access menu elements from the keyboard without posting the menu. Accelerators are useful to the experienced user for saving time when using frequently used components. The application should provide accelerators primarily as a matter of utility, not design conformity.

The application should use the following guidelines [5][12] when designing menus and menu systems:

- Menu structures should be kept simple.
- Menu elements should be grouped appropriately.
- Menu elements should be listed according to order of use.
- Destructive actions should be separate.
- Mnemonics and accelerators should be provided where appropriate.

2.1.3.3 Dialog Design [5]

Dialog boxes can limit how a user can interact with other windows in order to force the order of interaction. These limitations, which are called modes, are described in the following:

- Strive for consistency
- Offer informative feedback
- Design dialogues to promote completion and closure of the dialog
- Offer simple error handling
- Permit easy reversal of actions
- Support internal focus of control
- Reduce short-term memory load

2.1.3.3.1 Mode of the Application [5][12]

There are a number of different modes that a dialog may have:

- Modeless - Allows interaction with the secondary window and all other windows.
- Primary modal - Does not allow interaction with any ancestor of the window.
- Application modal - Does not allow interaction with any window created by the same application even if the application has multiple primary windows.
- System modal - Does not allow interaction with any window on the screen. This includes windows from all other applications and any icon box.

2.1.3.3.2 Common Dialog Box Actions [12]

Dialog boxes have a combination of the following elements:

1. *Yes* - Must indicate an affirmative response to a question posed in the dialog box and then close the window.

2. *No* - Must indicate a negative response to a question posed in the dialog box and then close the window.
3. *OK* - Must cause the application to apply any changes and perform related actions specified by the components in the dialog box and then dismiss the dialog box.
4. *Close* - Should cause the current dialog box to be closed without performing any of the actions specified by the components in the dialog box.
5. *Apply* - Must apply any changes and perform related actions specified by the components in the dialog box.
6. *Retry* - Must cause the task in progress to be attempted again. This action should be used in message boxes that report an error.
7. *Stop* - Must end the task in progress at the next possible breaking point. This action should be used in a working dialog.
8. *Pause* - Must cause the task in progress to pause. This action should be used in a working dialog and in combination with "Resume".
9. *Resume* - Must cause a task that has previously paused to resume. This action should be used in a working dialog and in combination with "Pause".
10. *Reset* - Must cancel any user changes that have not been applied to the application.
11. *Cancel* - Must close the dialog box without performing any dialog box actions that have not yet been applied to the application.
12. *Help* - Must provide any help for the dialog box.

2.1.3.3.3 Determining Dialog Box Location and Size [13]

The application should size and place dialog boxes so that they do not obscure important information in other windows of the application. The initial size of a dialog box should be large enough to contain the dialog components without crowding or visual confusion, but otherwise should be as small as possible. Once location has been determined for one dialog box, all other dialog boxes should be located in the same way - preferably towards the centre of the main active window.

2.1.3.4 Data Display and Entry [5][7]

2.1.3.4.1 Data Display

High-level objectives for data display:

- Consistency of data display.
- Efficient information understanding by the user.
- Minimal memory load on user.
- Compatibility of data display with data entry.
- Flexibility for user control of data display.

2.1.3.4.2 Data entry

High-level objectives for data entry:

- Consistency of data-entry transactions.
- Minimal input actions by user.
- Minimal memory load on user.
- Compatibility of data entry with data display.
- Flexibility for user control of data entry.

2.1.4 Colour and Sound

2.1.4.1 Colours [12][10][5][6]

Colour is most important for making displays more interesting for the user. Colour is useful for highlighting, grouping, distinguishing, and drawing attention. The following guidelines are recommended:

- All captions should be one colour and all data-entry fields another.
- The active window should be a different colour to an inactive one.
- Warning and error messages should be in red.
- When the status of an item changes then its colour should change.

Main window areas should follow the general rules of:

- Workspace background - neutral, lightly saturated colours.
- Window background - neutral, lightly saturated colours.
- Selection/input area - bright, highly saturated colours.

The best method of designing is to first design in monochrome and then add colour. Colour is a powerful visual stimulus that means that unfortunate choice combinations will be noticed:

- A screen should not contain any more than four different colours. Research has demonstrated that the more colour codes used, the longer it takes the user to distinguish each one.
- Different shades of the same colour should be used and different variations for emphasis.
- Different shades of grey are acceptable colour variations.
- Warm colours should be used to indicate actions and brightness for emphasis. Black, blue or white backgrounds are the most affective.
- Incompatible combinations of colours should be avoided. Examples are blue and yellow, red and green, green and blue, red and blue.
- High colour contrast should be used for character and background pairs. Examples are white on black, white on blue, or black on white.
- Light blue should be confined to background areas.
- Red and green are hard to see on the periphery of the visual field and so should not be used in areas away from the centre of attention.
- Colours should be assigned to user expectations. Blue for cold, red for hot and danger, green for go, amber for wait.
- Bright colours should be used sparingly.
- Users should be allowed to set their own colours.
- Colour blindness should be considered when choosing colours. (E.g. red↔green)

2.1.4.2 Sound [12]

Sounds within an application should be kept to a minimum, using them only for confirmations, errors, cancellations, interrupts, etc as required. Sounds are important to inform the user that the application is still operating correctly. However, sound should not be a requirement for the standard operation of the application, since some users may turn their sound level down or may be hard of hearing.

Some of the most important aspects of sound are for:

- Information about physical events (e.g. Successful save)
- Information about invisible structures (e.g. Computer to computer communication)
- Information about dynamic change (e.g. Running of a process)
- Information about abnormal structures (e.g. Errors)

All of these aspects, and others, have many different reasons for being produced and so different sounds may be required for the different circumstances. For example, if a physical event occurred such as a successful save, the sound would preferably be different than another physical event such as a cancellation.

2.1.5 International Considerations [10]

The most important functions needed to develop an international product are summarised as:

- Appropriate character sets.
- Multiple orientation and direction of text.
- A way to translate all visible text.
- A way to localise the date, units, formats, colours, and symbols.

2.1.5.1 Country-Specific Text

Obviously, all text cannot be translated for all possible languages for users that may use the application. However, in some locale, order of reading differs and some thought should be given to whether it is possible to allow user flexibility to request order of reading.

2.1.5.2 Country-Specific Data Formats

- Thousand Separators - The comma, period, space, and apostrophe are all examples of valid separators for units of thousands and are shown in the following examples: 1,234,567 1.234.567 1 234 567 1’234’567

- Decimal Separators - The comma, period, and the centre dot are examples of valid separators for decimal fractions as shown in the following examples: 5,324 5.324 5·324
- Positive and Negative Values - Various countries indicate positive and negative values differently. The symbols + (plus) and - (minus) can appear either before or after the number. Negative numbers can be enclosed in parentheses in applications such as a spreadsheet.
- Date formats - Most countries use the Gregorian calendar but others do not and dates can be formatted differently based on the locale. The hyphen, comma, period, space, and slash are all examples of valid separators for the day, month, and year. In numeric date formats, the month and day fields can be reversed, and in some cases, the year field can come first. For example, the 4th of August 1992 can be written as either 4/8/92 or 8/4/92 depending on locale. In addition, users in other countries sometimes place the year first, so June 11, 1992 could be 920611 or 921106.
- Time formats - Time formats can change based on locale. The colon, period, and space are examples of valid separators for hours, minutes, and seconds. The letter h can separate hours and minutes. There is both the 12-hour or 24-hour notation. For 12-hour notation, a.m. or p.m. can appear after the time, separated by a space. The following example shows a number of valid time formats: 1830, 18:30, 04 56, 08h15, 11.45 a.m., 11.45 p.m., 13:07:31.30, 13:07:31

2.1.6 **Errors** [7][5]

Error messages should be informative and help the user to correct the problem. Actual correction should be an easier operation than the operation that caused the error. To prevent errors, dialogues should guide the user and in certain circumstances if the error message is going to give the solution, perhaps the application should assume to continue with that solution. Clear instructions should be provided throughout the application and allow the user to exit from any process or the application at any time. When an error occurs the user should be told how the application interpreted the

situation, and allow the user to define how detailed the error messages are to be. For example:

- Simple error message - A number was expected.
- Not so simple error message - "T" was entered, but a number was expected.

Brief simple but accurate sentences should be used so that a user can immediately understand and act upon the message.

2.1.6.1 Error-Message guidelines

Errors should be posted using dialog boxes and should follow these guidelines:

- Product:
  - As specific and precise as possible.
  - Constructive – an indication of what needs to be done should be given.
  - A positive tone should be used not condemnation.
  - Multiple levels of messages should be considered.
  - Grammatical form, terminology, and abbreviations should be kept consistent.
  - Visual format and placement should be kept consistent.
- Process
  - A message quality-control group should be established.
  - Messages should be included within the design phase.
  - All messages should be kept in a file.
  - Messages should be reviewed during development.
  - Frequency data for each message should be collated.
  - Messages should be reviewed and revised over time.

2.1.7 Utilising Help [6][4][7][12]

Even in the most intuitive application, the purpose of a component or the way to do a task can be hard to figure out for a new user. The application should provide help mechanisms for all of its aspects. The menubar should store the location for the help pulldown menu but short cuts should also be available, e.g. F1.

Users avoid using help because they have difficulty in finding the actual information they require; however an index and search system helps tremendously. Once they have reached the required area of help, relevant comprehensive instructions should be given to allow the user to continue. There should be no difficulty in switching between help and the working context, and the help interface must also be as simple as possible, to operate.

The most important aspect of the help facility is the quality and layout of help information, and that it must be available at all times.

2.2 Implementation of GUI Guidelines

2.2.1 Introduction

The principles gathered from section 2.1, the review of graphical user interfaces, were used to steer the design and development of the various facilities that make up the PSUE. These guidelines were initially quite relaxed but later tightened as diverging styles, in the modules, became apparent.

2.2.1.1 Consistency

As mentioned in the previous section, there are various separate modules that are included within the PSUE. Each module has been designed to be in either of two states, running in the integrated environment of the PSUE or as a standalone application. This has required that each module must be totally self-contained but still allow the sharing of information with others. To be consistent, if a feature of one module changes, e.g. the layout of the file selection box, the same changes must occur in all other modules.

Each of the PSUE modules is designed to do different tasks and therefore has different requirements, however, due to the guidelines they should be generally laid out in a similar manner. The basic layout followed by the PSUE modules is:

(a) The menubar is always at the top of the window and spans the entire width of the module's main window.

(b) The main panel that holds the principal control and manipulation buttons is located on the left-hand side of the module's main window. All buttons and controls are laid out in a typical left-to-right and top-to-bottom style.

(c) Any graphical manipulation window is located on the right-hand side of the module's main window. The main PSUE interface was exempt from this guideline since it was anticipated that its graphical viewer would be used quite infrequently.

(d) Each module has a status and messages window that is located at the bottom of the main panel within the module's main window.

(e) All popup windows are laid out in a typical left-to-right and top-to-bottom style and have a consistent layout of control buttons, i.e. Apply, Close, Help etc.

(f) All cursors, hourglasses etc are consistent across all modules.

(g) Buttons, text entry fields, toggle buttons etc all have similar respective sizes, layouts and actions across all modules.

Figure 2-1 shows the general layout of the key areas of a module that involves points (a) through to (d) from the list above. Figure 2-2 shows point (e) from above that covers popup dialog windows and Figure 2-3 shows point (g) from above that covers buttons, text fields and other such standard widgets.

Figure 2-1 - General Layout of Key Areas of a Module

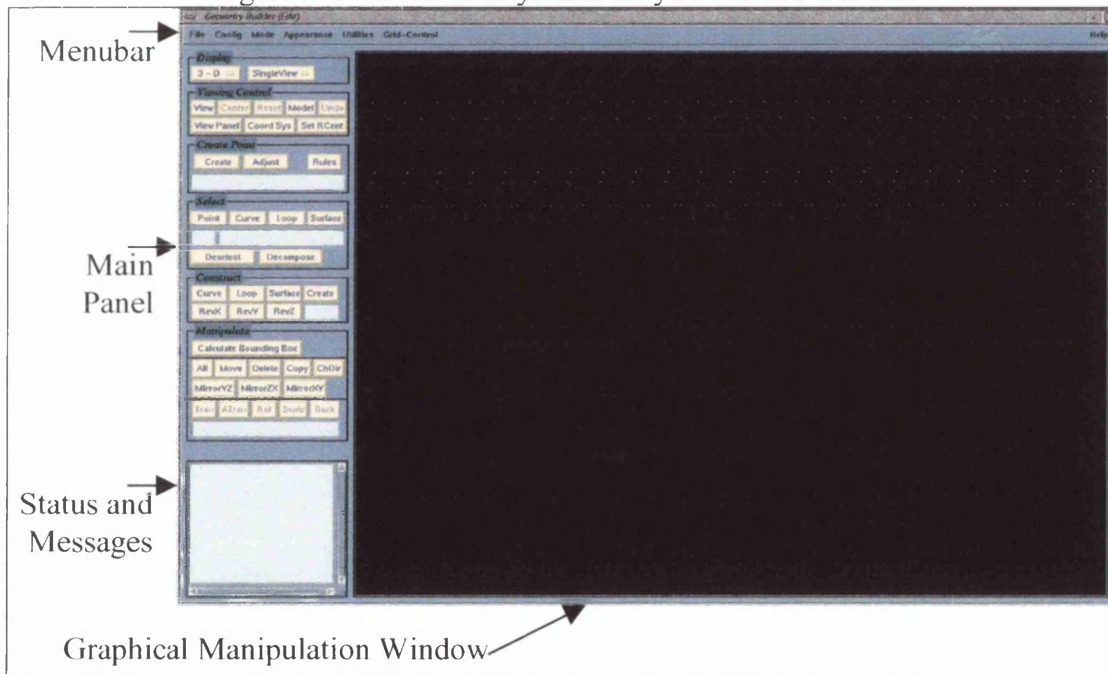


Figure 2-2 - General Layout of Key Features in Popup Dialog Windows

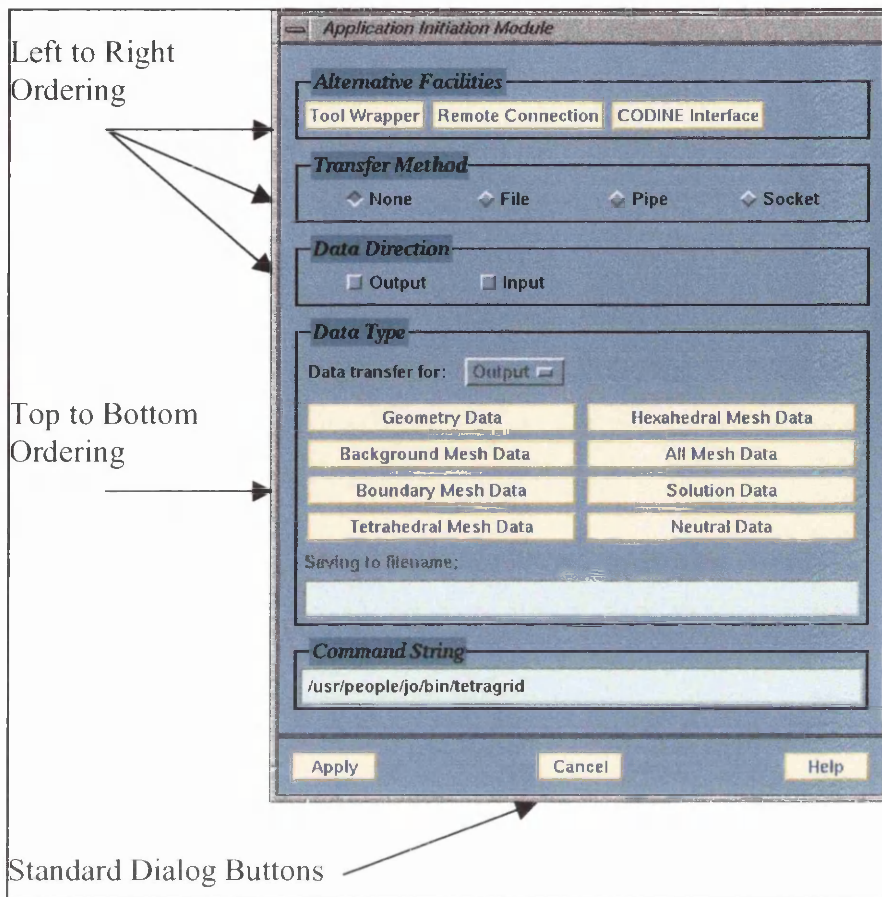
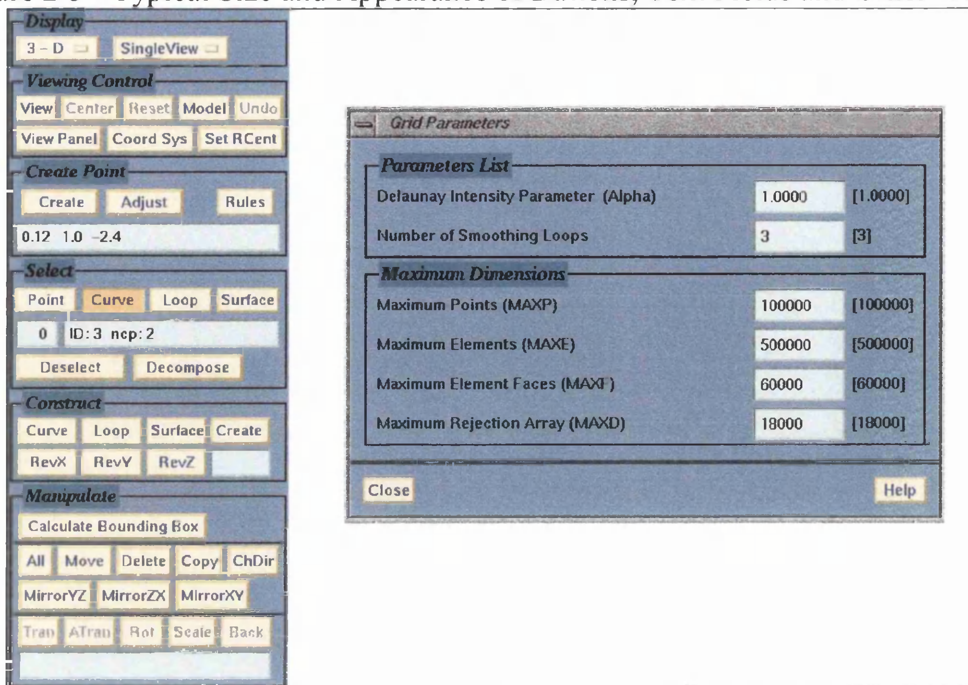


Figure 2-3 - Typical Size and Appearance of Buttons, Text Fields and Other Widgets



Font and colour resources are consistent throughout all of the modules. The main guidelines for the colours follow the original guidelines mentioned in section 2.1.4.1 and also allow for the user to change the colour of just about any component found throughout all the modules. The fonts have been kept simple and only three are used throughout all of the modules. The main font is used for all buttons, text entry fields, status windows etc. and another font is used to highlight sections and frames that group controls of a particular manner. The third font is a title font and is used quite seldom within the modules.

The use of the mouse is consistent throughout all of the modules, where for general user interfaces, the left mouse button is used for almost all operations such as button selection, slider movement etc. On UNIX systems, the middle mouse button defaults to being able to paste any previously selected text into the given window, text field etc. and so the modules continue to allow such pasting of text.

Within the graphical user interfaces the mouse is also used to transform and manipulate the geometrical data. All transformations use the mouse consistently, such that in transformation mode the mouse buttons have the following actions:

- Left button – Translate the geometrical data.
- Middle button – Rotates the geometrical data.
- Right button – Zoom in and out of the geometrical data.

While in an editing mode the mouse buttons have the following actions:

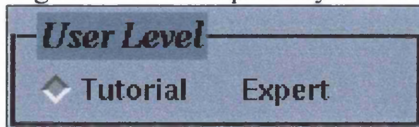
- Left button – Selects geometrical entities.
- Middle button – Deselects geometrical entities.
- Right button – Complete / Confirmation of creation.

One of the most important areas that must be consistent throughout an application is the help feature. For this reason a single help facility was developed that could be individually accessed by each of the modules. This ensured that the help facility appeared and acted consistently throughout all modules. This help facility would also allow the integration of the user's own notes and messages.

2.2.1.2 Levels of Competence

Initially it was planned that various levels of competence would be built into the PSUE modules at all stages. However, due to a large unforeseen amount of other functionality that would need to be integrated into the modules, the levels of competence functionality proposals where greatly reduced. The levels of competence would involve a novice level and an expert level, and would only have an influence within the main PSUE module. Figure 2-4 shows the functionality switch found within the main interface.

Figure 2-4 - Competency Switch

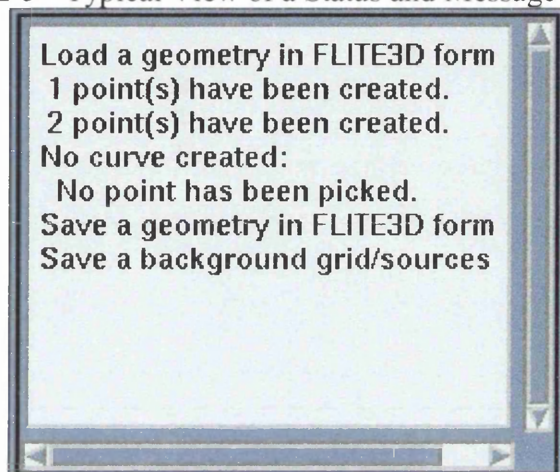


2.2.2 Design of Control

2.2.2.1 Usability

As mentioned above, all modules have a status and messages window and a typical example is shown in Figure 2-5. This is used to inform the user of the success or failure of key operations. All messages are as informative as possible without generating copious amounts of data that the user would soon tire of.

Figure 2-5 - Typical View of a Status and Message Window



While a module is busy the user is shown that the application is busy by use of a standard hourglass icon and / or by a popup message box that continues to update the user on how far it is through the particular task.

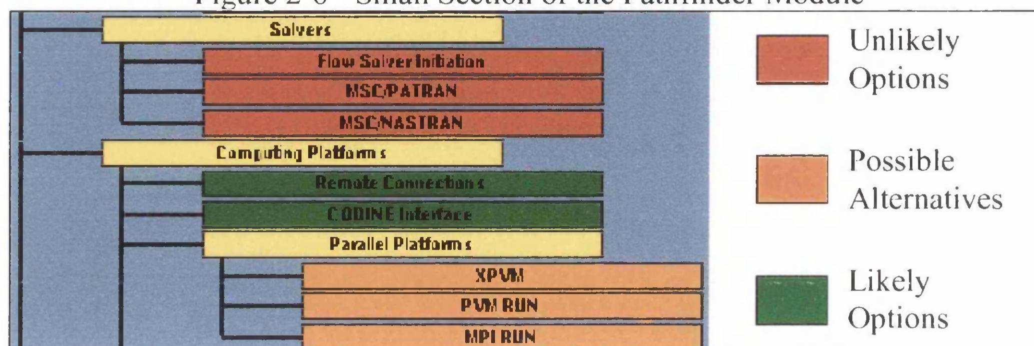
No extreme computer jargon is used within any of the modules. Since the user of such an application would have a basic understanding of computer hardware, certain simple computer jargon words, such as memory, have been used but kept to a minimum.

2.2.2.2 Modifications

The guidelines suggest that an application should not tell the user what to do, however, one of the key design aspects of the PSUE and its modules is the ability for

it to guide the user through particular tasks. It could be seen that there is a thin line between telling a user what to do and helping the user come to an informed decision on what they are likely to need to do next. The PSUE attempts to address this issue, mainly within the "Pathfinder Module" that is shown in Figure 2-6. The "Pathfinder Module" is discussed in further detail in the next chapter.

Figure 2-6 - Small Section of the Pathfinder Module



As mentioned in section 2.2.1.1 of this chapter, the PSUE modules allow users to change fonts and colours of almost all components. These changes can be stored as the user's specific preferences that may be recalled each and every time the user starts any of the PSUE modules.

As an extension to the novice / expert level of competence functionality, short-cut keys are available within graphical windows. Short-cut keys were not introduced into the main window interfaces because of the amount of complexity, not to mention the modules diverse functionalities.

2.2.2.3 Indicating Action of Components

To provide a consistent feel for control widgets within the modules, the following show how each widget has been used:

- Push Buttons – Selection or activation of operations.
- Toggle buttons – Selection of multiple options.
- Option buttons – Selection of large number of options.

- Text Fields – Entry of text and or numerical values
- Lists – Selection of singular and / or multiple items.

Figure 2-7 through to Figure 2-11 show typical examples of each of the above types of widgets.

Figure 2-7 - A Typical Push Button

Figure 2-8 - A Typical Toggle Button

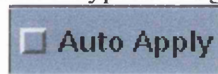


Figure 2-9 - A Typical Option Button

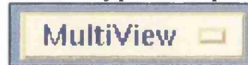


Figure 2-10 - A Typical Text Entry Field

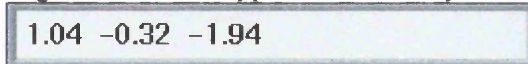
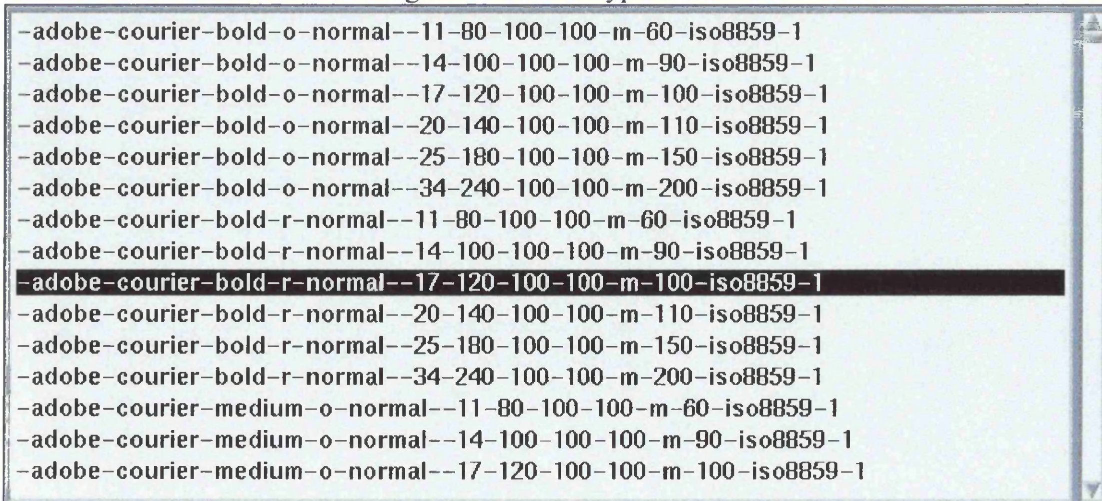


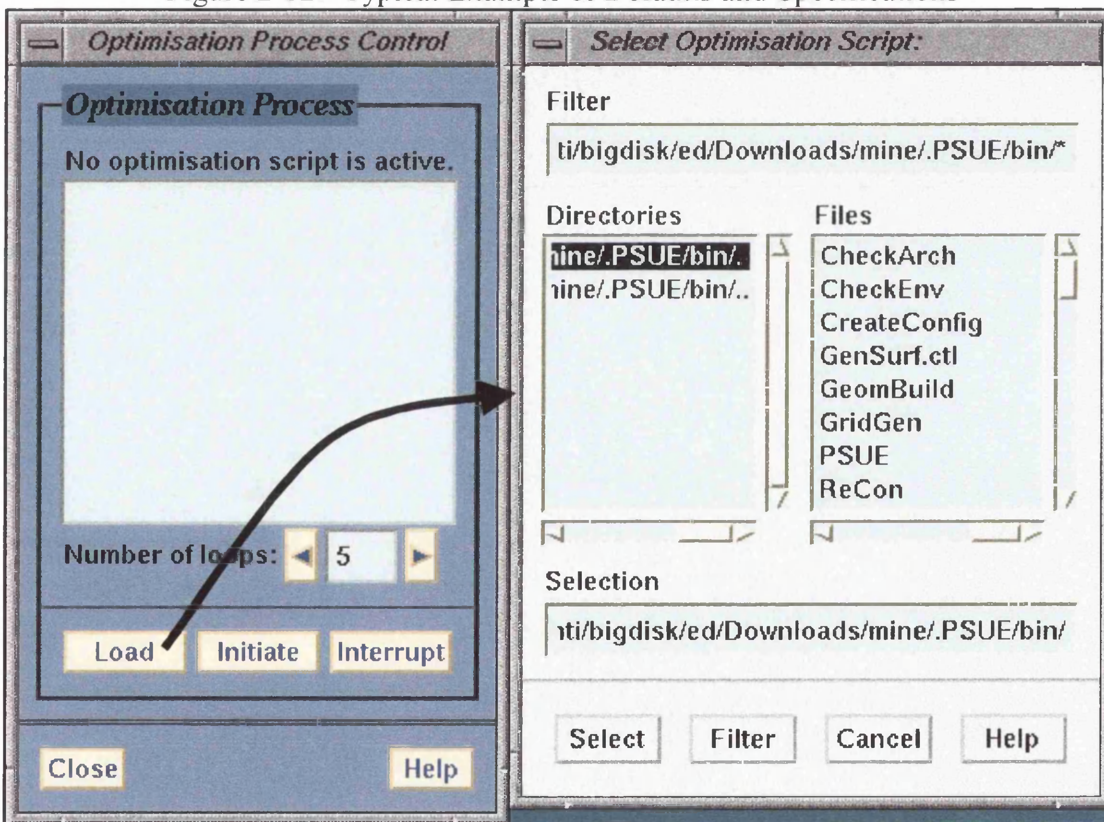
Figure 2-11 - A Typical List



Throughout all of the modules, text labels have been used for all control widgets, such as the buttons and no graphical icons have been used due to requirements of other development features.

Default values of text are provided for most input data entries whereas more complicated input details such as directory and filenames are left blank. These entries are facilitated by use of buttons that initiate a file selection box process for directory and / or filename selection. Figure 2-12 shows a typical example of default values and button use for filename specification.

Figure 2-12 - Typical Example of Defaults and Specifications

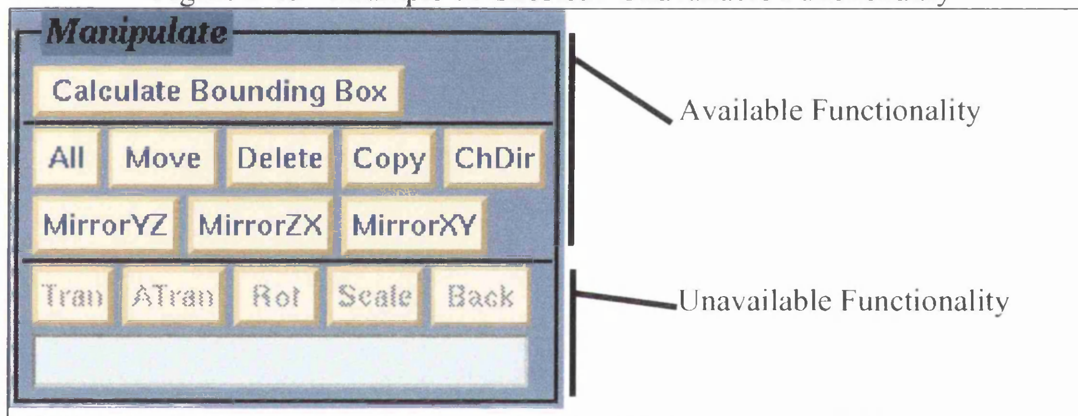


Throughout all of the modules there are many initial values found within all of the different functionality and each one is associated with toggle buttons, option buttons

and entry fields. The specific widget control associated with every default value is set to show the corresponding value or setting.

One of the biggest standards within all graphical user interfaces is the appearance of components that are not available either temporarily or permanently. These components are disabled and are visually represented as "ghosted" or "greyed" out and an example may be seen in Figure 2-13. As certain processes are carried out, particular tasks may become available and their corresponding components are then enabled and become visually equivalent to normal components. If at any stage a particular action results in certain tasks being inappropriate, the corresponding components should be deactivated as before.

Figure 2-13 - Example of Ghosted / Unavailable Functionality



2.2.2.4 Feedback

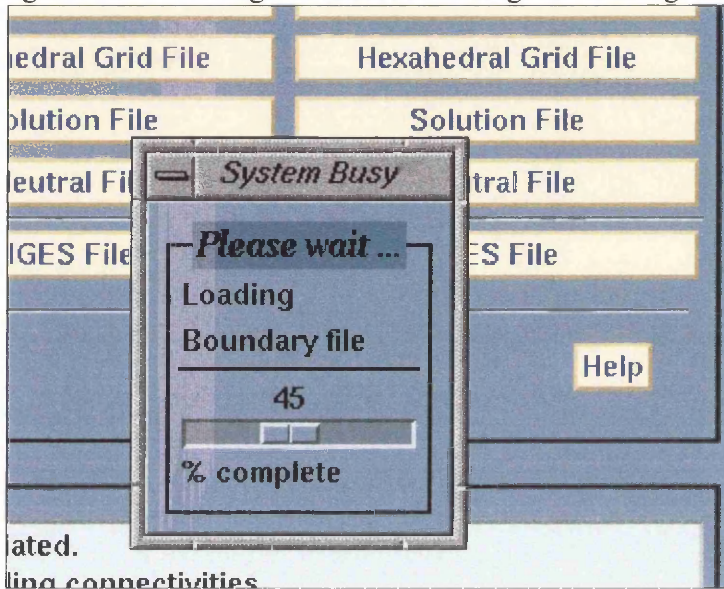
As mentioned previously, when a module is busy feedback is given to the user to signify this fact. At all times the pointer is changed to an hourglass icon and where appropriate a dialog box is shown. The dialog box contains a message to signify what is actually taking place and progress indicator. Depending on the task one of two methods is used for displaying the progress indicator:

(a) If the length of the operation is indeterminable, a small bar moves back and forth to signify that the application is still working.

(b) If the length of the operation is known or can be closely estimated, a percentage bar is used and is updated at regular intervals during the operation.

Figure 2-14 shows a typical progress indicator with a percentage bar as described in option (b) above.

Figure 2-14 - A Progress Indicator Using a Percentage Bar



Some times the delays may be quite short and under these circumstances, popup dialog boxes can appear and disappear too quickly for the user to acknowledge them but long enough to distract them. When this is the situation the dialog box should not be used, just the pointer changing to an hourglass icon should be sufficient.

Direct feedback after a certain key task has been performed should be given each and every time. This should use the standardised status and messages window found within each module as discussed previously in this section. However, major critical errors should be highlighted by high visibility error dialogs that must be acknowledged before the user can continue. The error message should also appear in the status window so that a complete history of the user's actions is maintained within one key area.

The application should take care not to present too many message dialogs that require confirmation, only critical and very important information messages should require confirmation.

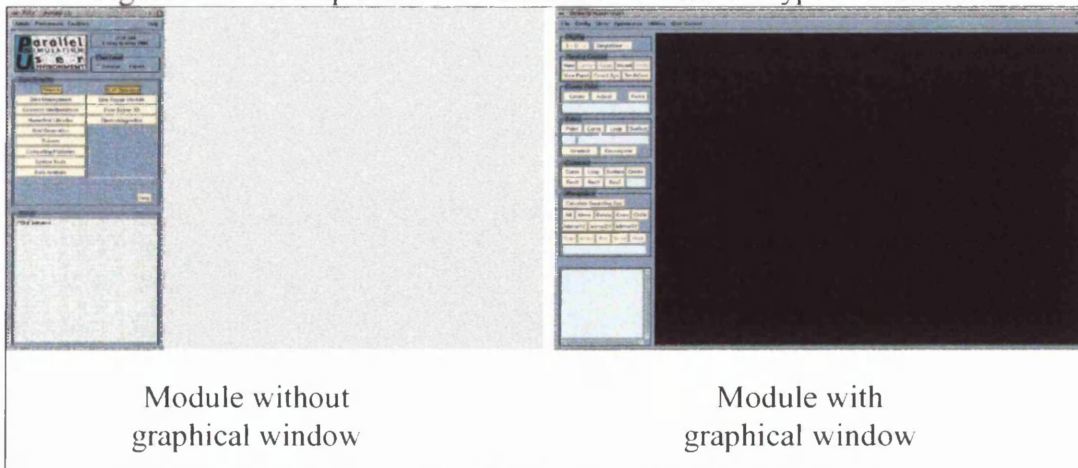
2.2.3 Workspace Elements

2.2.3.1 Graphical Layout

Since each of the different modules is used for very different tasks and therefore has different functionality, the layout of each module is different to the next but they all follow the general guidelines discussed previously in this section. These involve a narrow main panel on the left-hand side of the main window with a messages and status window below it and the main graphical region, if present, on the right-hand side.

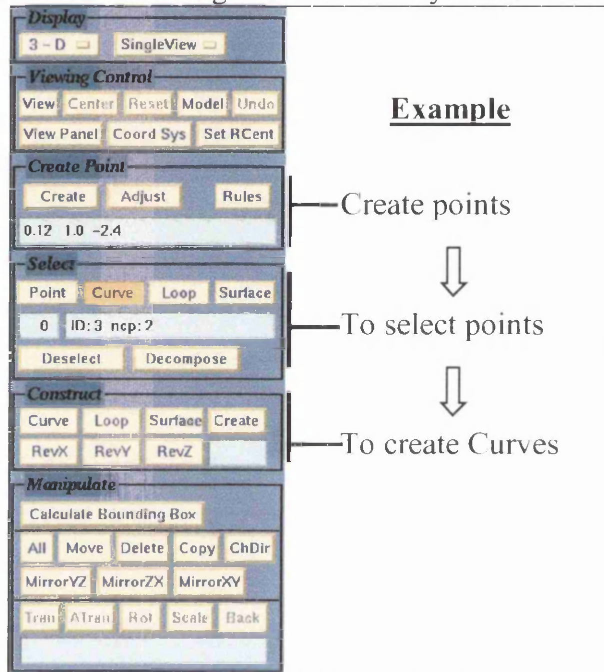
When using a graphical region, it is usual to use the entire screen and so such modules interrogate the window manager in order to ascertain the maximum size that the main window may be set to. However, modules that do not use graphical regions are kept as narrow and compact as is reasonably acceptable. Figure 2-15 shows the difference between the two types of interfaces and the amount of space that they both take up.

Figure 2-15 - Comparison between the two different types of interfaces



If a panel is created that contains a large number of control widgets, they are laid out so that widgets that are likely to be the used in conjunction with each other are close, ensuring that the user is not “jumping” around the panel unnecessarily. Figure 2-16 shows an example of the layout of related functionality. All of the screen layout guidelines described previously are followed by all of the modules.

Figure 2-16 - Panel Showing Close Proximity of Related Functionality



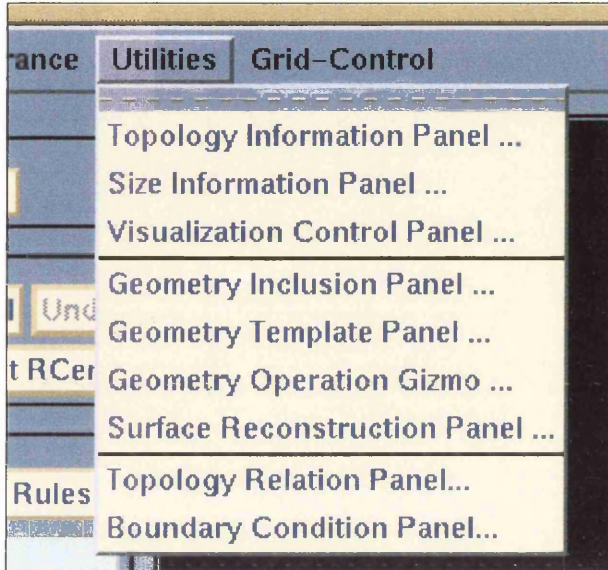
2.2.3.2 Menu Design

Almost all of the menus used by the modules are pulldown menus from the menubar that keeps a certain level of consistency throughout the application, however option menus are used occasionally.

As shown in Figure 2-17, every pulldown menu is given the functionality of a tearoff menu so that the user has complete freedom to treat the menu as either a pulldown menu or a tearoff menu. This allows the user to access the functionality held on the menu in a variety of ways. They may use it as a pulldown menu if they only want to

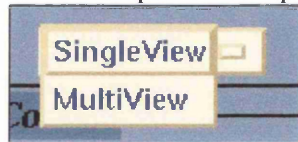
access the functionality occasionally, or as a tearoff menu for continuous use of particular options. The specific use of the menu may change as the user uses the module.

Figure 2-17 - Pulldown Menu Shown with Tear-off Functionality



The option menus are used within panels when two or more options are available and there is not much space available within the panel. The option button is always initialised to a specific setting and an example is shown in Figure 2-18.

Figure 2-18 - Example of an Option Menu



Between the different modules, menubar pulldown menus are kept as consistent as is practicably possible, allowing the user to know where certain functionality is within one module once they know where it is in another.

2.2.3.3 Dialog Design

Four different types of dialog boxes have been used within the modules:

(a) Utility panels that give the user access to further functionality that would not otherwise fit neatly in the main panel. These are modeless dialogs and therefore allow the user to continue accessing all other areas of the module.

(b) Informative windows that provide the user with particular important feedback. These are also modeless dialogs.

(c) Critical warnings and error message boxes. These are both primary modal dialogs that therefore require the acknowledgement or dismissal of the dialog before any other functionality maybe accessed.

(d) Application busy dialogs. These are also primary modal dialogs that require the user to wait until the task is completed upon which the dialog will automatically close and return control back to the primary window.

All dialog boxes appear at the centre of the primary window unless the window is placed mostly off-screen. In which case the dialog box will appear at the edge of the screen, at a point closest to the centre of the primary window whilst ensuring that the whole of the dialog box is visible.

2.2.3.4 Data Display and Entry

Data display is kept to a minimum so that the user does not become overloaded with data. Most data is presented sequentially or coupled to allow the user to interact and store particular values if necessary. Due to the nature of simulation engineering large data sets are used but seldom seen or accessed as such.

Entry of data follows the same philosophy as data display and usually coexist so that and data displayed can generally be directly edited and updated.

2.2.4 Colour and Sound

2.2.4.1 Colours

As mentioned previously, all modules use exactly the same colour scheme that has been designed to follow the guidelines in the first chapter very closely. The key aspects include the lightly saturated background being "light steel blue", highlighted headings using a darker shade of the blue, all text is black and high contrast colours are used for highlighting and selection.

The default colours may be changed at any time using the main interface resource panels and saved as the user's preference. This also allows the user to adjust the colours for any visual impediment that they may have.

2.2.4.2 Sound

The PSUE has been designed primarily for the UNIX platform and most large UNIX vendors produce machines that are not sound orientated. Therefore sound has not been used at all since any audio cues that may become expected by users on some machines might not be heard on other machines causing confusion. Similarly, if a user is used to no sound and then is subjected to audio cues, they too may become confused.

2.2.5 International Considerations

Very little direct international considerations were undertaken during the development of the PSUE modules as it was anticipated that these would be relatively easy to incorporate at a later stage if needed.

The text used for all labels and buttons uses the machine's settings for international character sets and therefore should undertake whatever setting the machine is configured to. However, data formats are restricted to using no punctuation characters, for example commas or apostrophes.

The date and time is presented on the top of the main interface at all times but is used in long format and therefore would not be expected to change with international adjustments. The only other situation in which date and time is used is with file creation and modification times. These are controlled solely by the operating system and therefore depend directly on the international settings of the specific platform.

2.2.6 Errors

All errors are reported directly to the user and are kept as informative as possible. All key information is displayed in the dialog box to aid the resolution of the problem. The messages are kept as constructive as possible and if the situation can be reversed the appropriate instructions are given.

If a large amount of data is expected to be output as either errors or messages, log files are used and the message dialog points the user to the appropriate file.

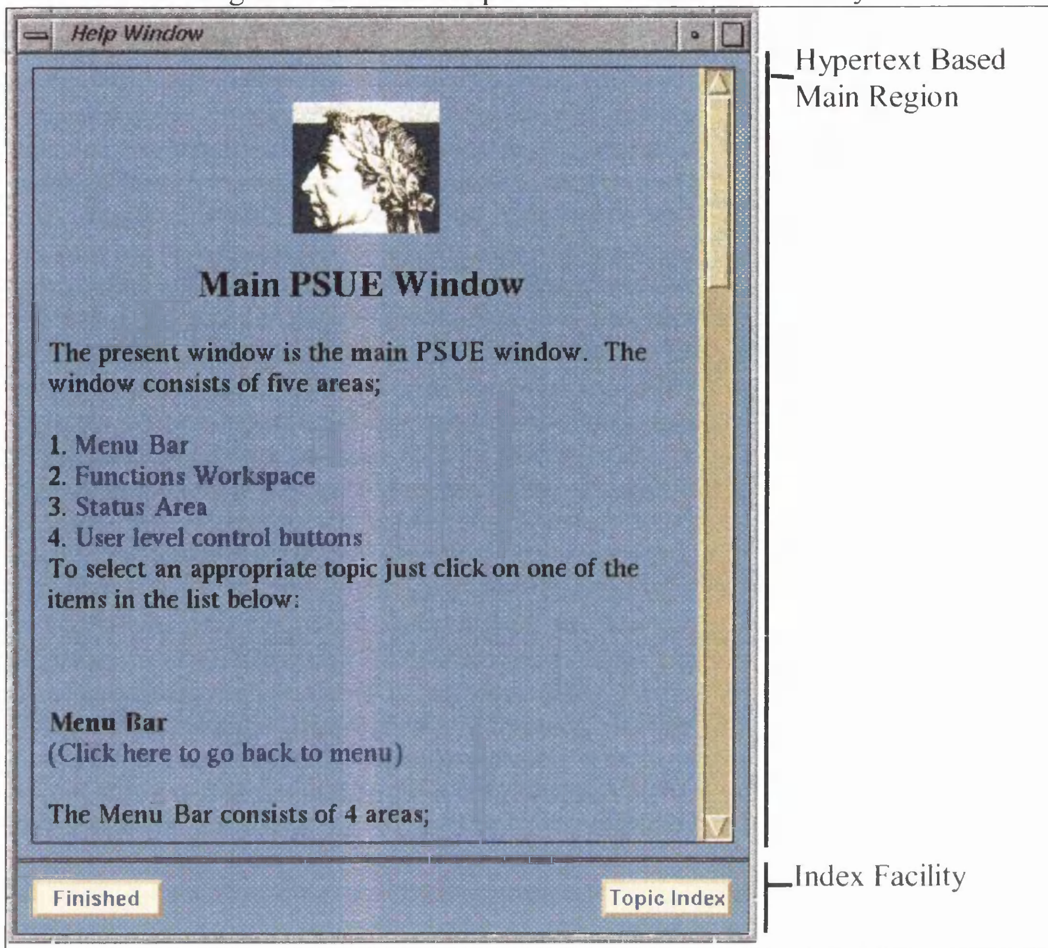
2.2.7 Utilising Help

Help is provided within every module of the PSUE and may be accessed from the menubars that provide a choice of access methods. Each and every major dialog also provides access to context sensitive help

A help index is one of the methods provided by the functionality under the menubar but the help dialog also provides direct access to the help index for the particular module.

The help dialog has been kept particularly simple to ensure no confusion may occur and it also allows for the user to configure and edit the help system and contents at a system and user level. Figure 2-19 shows the simplified help window with the module specific index facility.

Figure 2-19 - The Help Window with Index Facility



3 PSUE MAIN INTERFACE

3.1 Overview of the Main Functionality

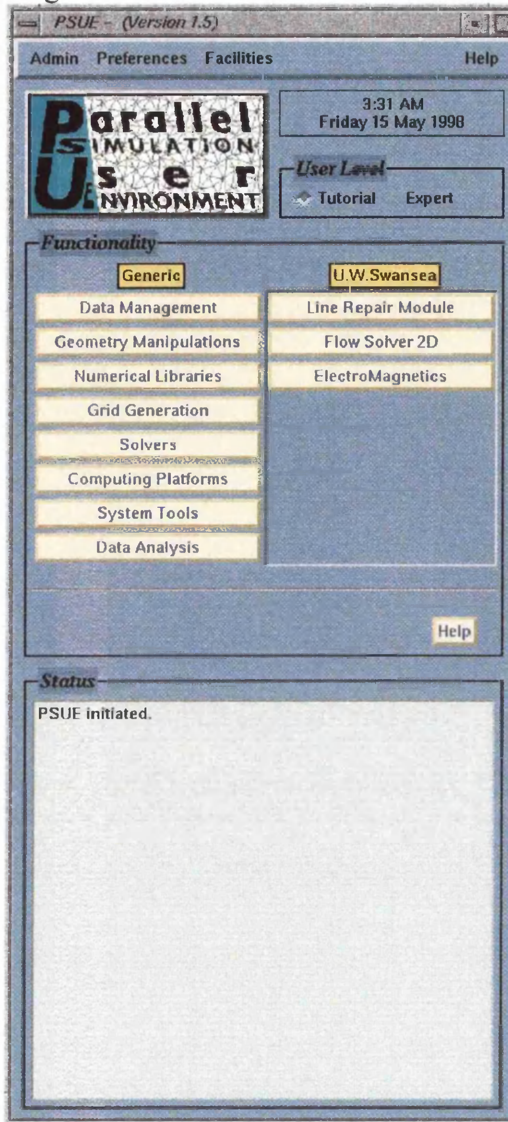
The Parallel Simulation User Environment, as discussed in the initial chapter, consists of a set of modules. These modules are coupled into a seamless environment by the main graphical user interface that primarily controls the modules and other processes and also manages data communication.

The main interface, shown in Figure 3-1 below, can be separated into three regions, which are the menubar, the main functionality region, and the message window. The message window displays all information messages, warnings and errors so that the user is always aware of the current situation. The menubar provides access to a number of general features such as:

- Save and load state facilities.
- Colour and font resource management.
- Process control.
- Pathfinder.
- On-line help.

The main functionality region is the control centre for all the PSUE modules and is divided into two regions, the generic functionality on the left and the user's functionality on the right. Both sets of functionality are arranged into a system of menus that group various categories of applications together. The user's functionality region is reserved for the applications that the user will integrate into the environment and this is discussed later in the chapter.

Figure 3-1 - The PSUE Main Interface



The generic functionality consists of the following main categories:

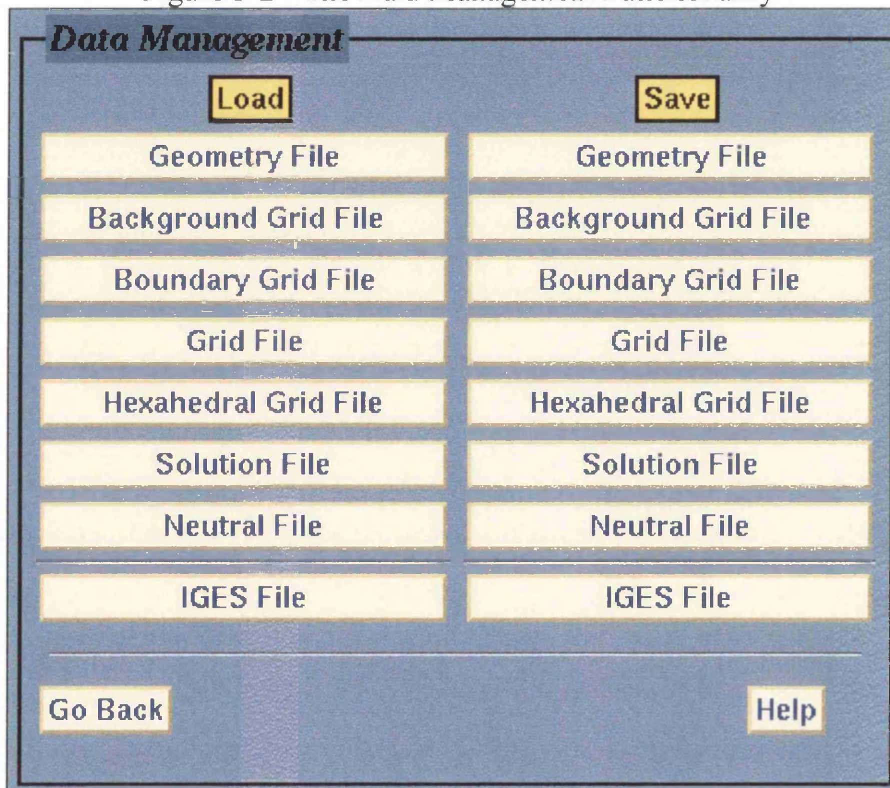
- Data Management
- Geometrical Manipulations
- Numerical Libraries
- Grid Generation
- Solvers
- Computing Platforms
- System Tools
- Data Analysis

The selection of one of these categories will display the specific functionality of that type. The following sections describe the functionality within each of the categories.

3.1.1 Data Management

The data management section is used to contain all of the file input and output facilities for the main interface. This section is the only section of the main interface functionality region that does not have a region for the user's applications. The loading capabilities are found on the left and the saving capabilities on the right.

Figure 3-2 - The Data Management Functionality



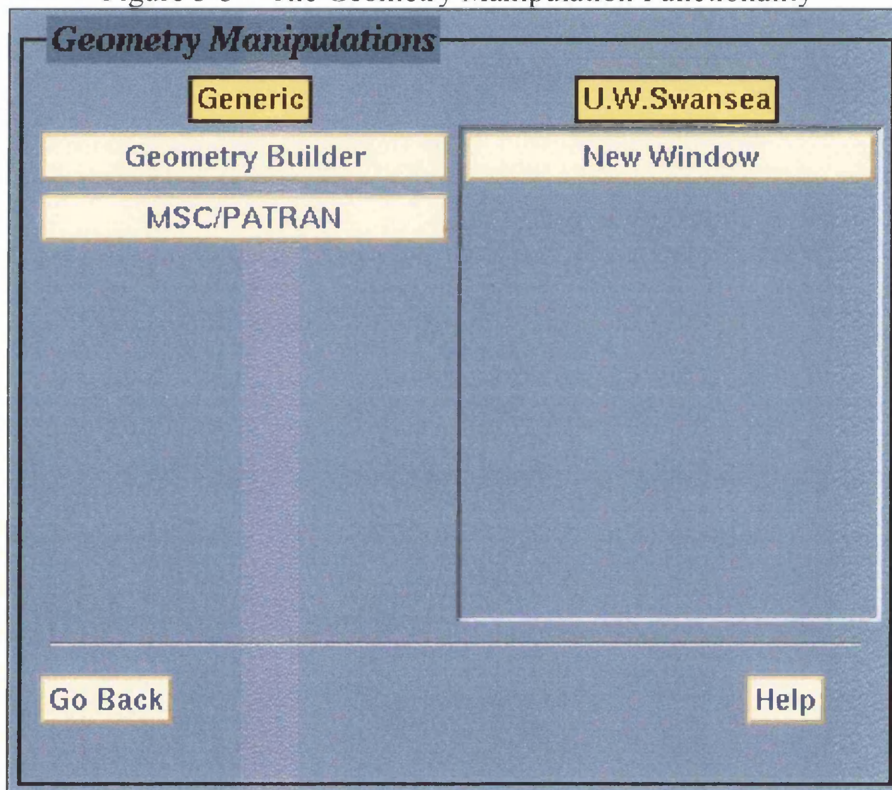
A number of specific PSUE file types are contained within this section but there is also provision for exportation of data to other packages. For example, under the MEDUSA project, a filter system was developed to create, append and read MSC/Patran database files. Figure 3-2 shows the data management functionality and

the PSUE file types that include geometry files, background grid definitions for controlling grid generation, boundary and volume grids, along with neutral and solution data files.

The figure also shows the IGES import and export facility, however, it should be noted that this underlying technology was provided by a project partner, IPK, from the CAESAR project. The MSC/Patran functionality mentioned above is not shown as the functionality is only provided if the main interface detects that the software is available on the current platform.

3.1.2 Geometrical Manipulations

Figure 3-3 – The Geometry Manipulation Functionality



There are only two options under the generic functionality region of the geometrical manipulation section that is shown in Figure 3-3. One of these options is the PSUE module "Geometry Builder" which deals with CAD creation, manipulation and repair.

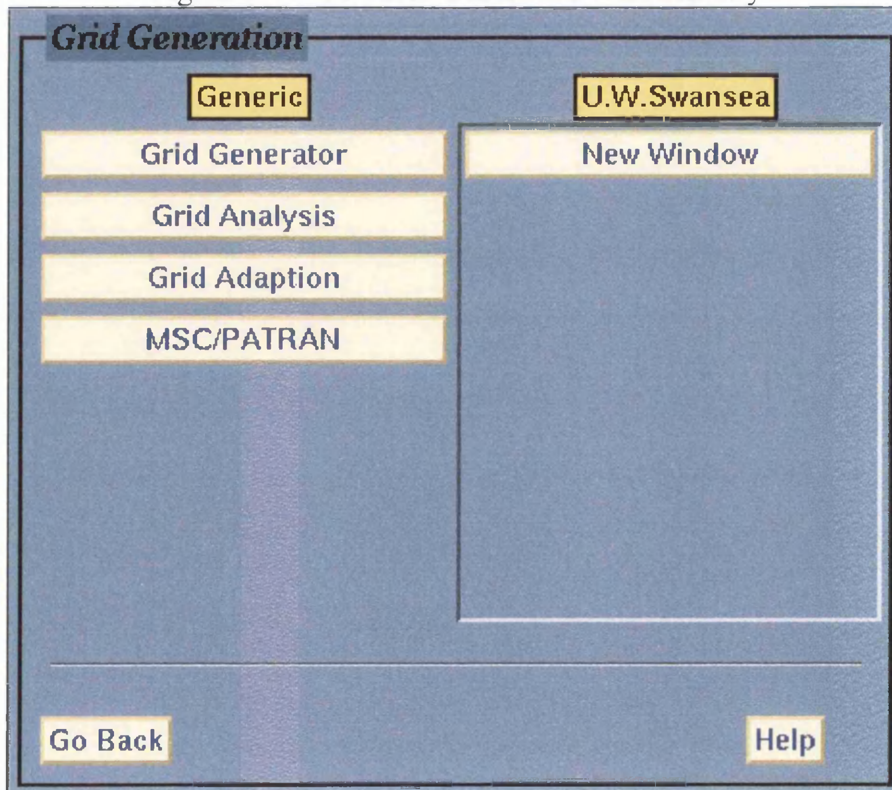
The other option is a close link with MSC/Patran and was developed specifically under the MEDUSA project.

Upon selection of MSC/Patran, the PSUE interface will automatically create a new MSC/Patran database file with all of the current geometrical data that is in the main interface. MSC/Patran will then be initiated, by the PSUE, in such a way that it will automatically load the database that was just created. This provides a very easy and efficient method of accessing the functionality of MSC/Patran.

3.1.3 Grid Generation

The grid generation section shown in Figure 3-4, provides access to three more of the PSUE modules and once again, MSC/Patran for exportation of geometrical and grid data entities. The PSUE modules provided are the “Grid Generator”, the “Grid Analysis” and the “Grid Adaption” modules.

Figure 3-4 – The Grid Generation Functionality

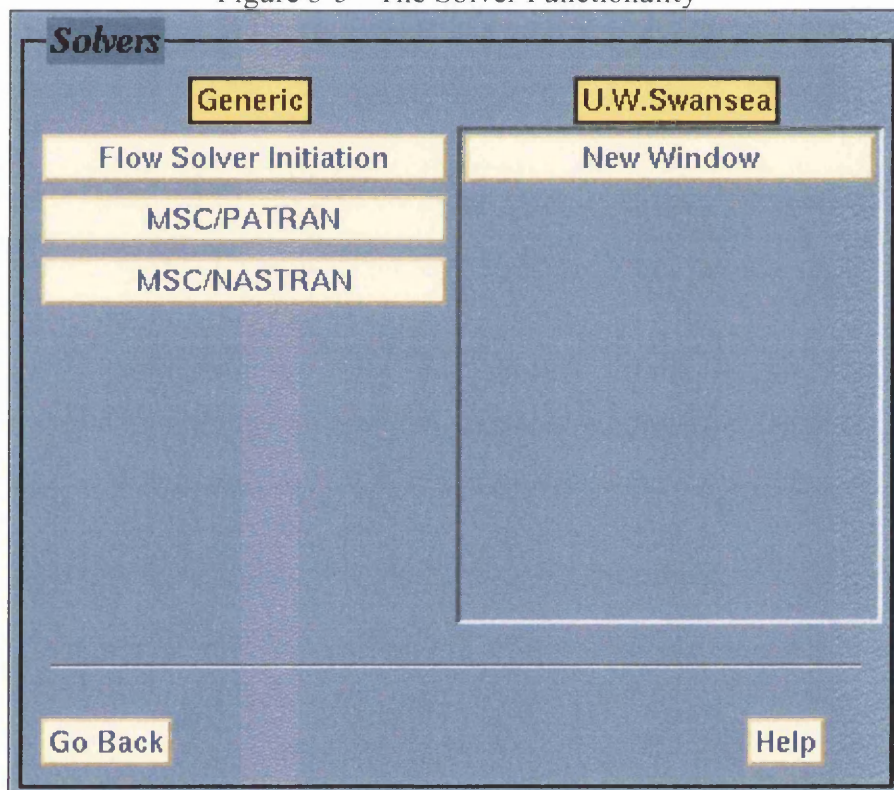


The grid generator, utilising the FLITE3D grid generators, is used to create 2D isotropic and viscous triangular grids, triangular surface grids and tetrahedral volume grids. The grid analysis module takes all of the aforementioned grids, checks them and then provides an extensive display of grid quality measurements.

3.1.4 Solvers

The section reserved for solvers, shown in Figure 3-5, contains three options: the flow solver initiation panel, MSC/Patran and MSC/Nastran. The MSC/Patran option works in the same way as discussed in the previous sections and transfers all geometrical and grid entities from the PSUE to MSC/Patran. The MSC/Nastran option opens a panel for job submission to MSC/Nastran.

Figure 3-5 - The Solver Functionality



3.1.4.1 The Flow Solver Initiation Panel

Within the scope of the PSUE development, it was not possible to integrate any specific solver into the PSUE interface as a generic module. Therefore the flow solver initiation panel was developed as a step towards such functionality. This panel is completely generic and is controlled using a text file that the user provides. Within this control file, the user can specify any number of solvers and user-defined parameters. Upon initiation of the flow solver panel, the control file is loaded and the appropriate options are added to the panel. The panel will allow the user to set appropriate parameters and then create a parameter input file ready for the solver or to create the parameter file and initiate the solver immediately.

Figure 3-6 shows a typical view of the flow solver initiation panel and Figure 3-7 shows the generic solver settings panel. The user settings panel appears the same as the generic settings panel with just the revised labels and default values.

Figure 3-6 - The Flow Solver Initiation Panel

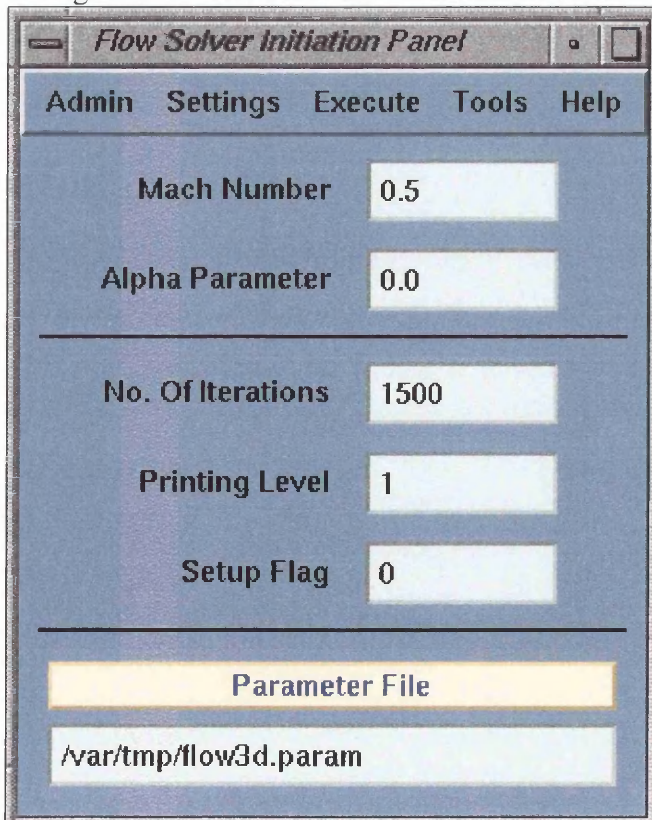


Figure 3-7 - The Flow Solver Generic Settings Panel

Generic Algorithm Settings

Setting	Value
Courant Number – C.F.L.	2.0
Boundary Relaxation	0.1
Second Order Dissipation Coefficient	0.4
Fourth Order Dissipation Coefficient	0.2
Enthalpy Damping Coefficient	0.1
Residual Smoothing Coefficient	0.0
Number of Multi Stages	3
1st Stage Coefficient	0.6
2nd Stage Coefficient	0.6
3rd Stage Coefficient	1.0
4th Stage Coefficient	0.0
5th Stage Coefficient	0.0

Close | Reset | Help

3.1.5 Computing Platforms

The computing platforms section, shown in Figure 3-8, contains three options for starting panels and another two options that provide access to other sections. The panels that may be initiated from this section are the "Remote Connection Tool", the "CODINE Interface" and the "Optimisation Panel" modules of the PSUE that are discussed below.

The Parallel Platforms section provides access to public domain software for executing tasks on parallel and networked platforms. The software that has been included within this section is XPVM, PVM Run, and MPI Run.

The Parallel Tools section, shown in Figure 3-9, provides access to further sections and options that altogether involve the following items:

- Domain Decomposition section, within which no generic functionality exists since the development requirements did not include such a facility.
- Parallel Grid Generation, for creation of grids in parallel.
- Grid Analysis, for statistical analysis of grid quality measurements for partitions.
- Parallel Solver, for execution of the FLITE3D parallel solver.
- Performance Monitoring section, which contains access to the public domain software of XPVM and ParaGraph.

Figure 3-8 - The Computing Platforms Functionality

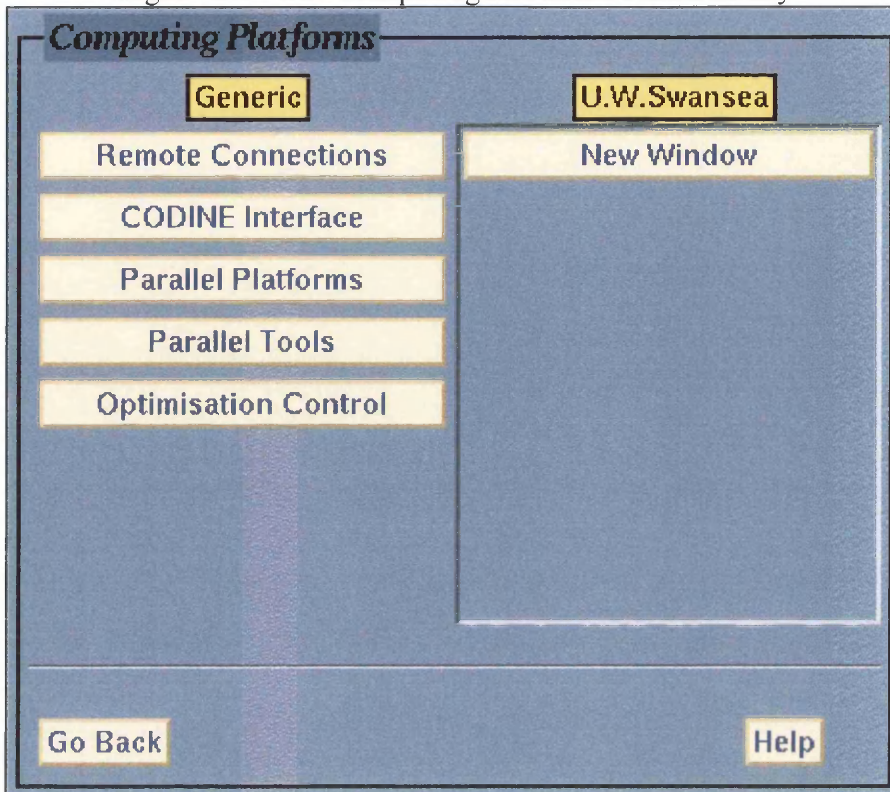
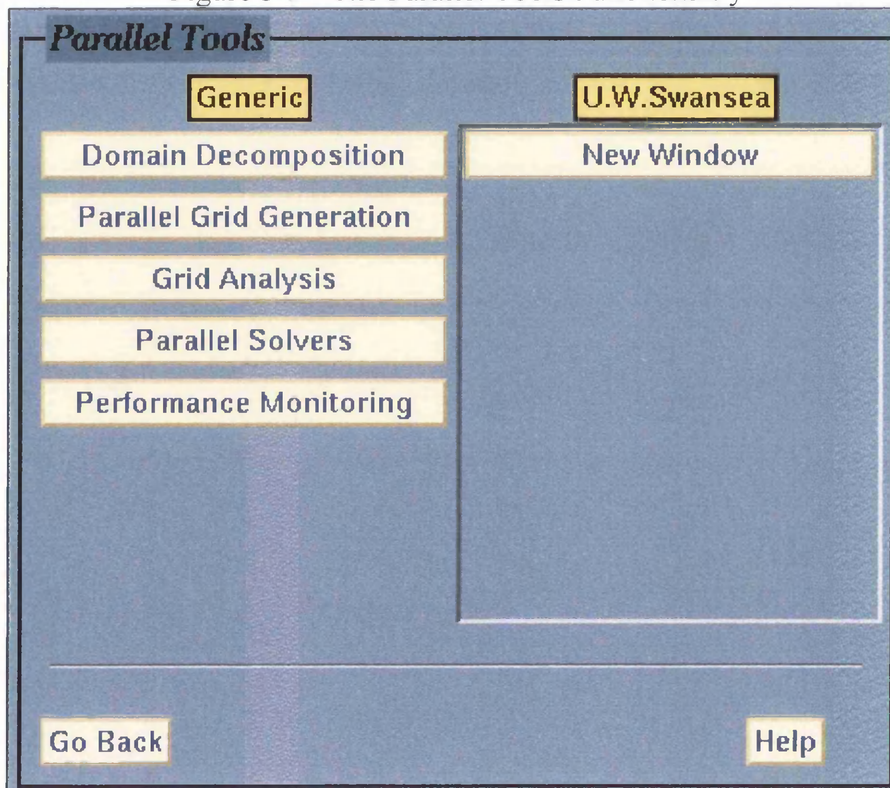


Figure 3-9 - The Parallel Tools Functionality



3.1.5.1 Remote Connection Tool

The remote connection module is a utility that allows a user to connect to a remote machine using a username and password. Once connected data files may be transferred back and forth between the local and remote machines. The module also provides the ability to initiate any number of jobs on the remote platform and provides an indicator to show how busy the remote platform currently is. Figure 3-10 shows the remote connection tool and Figure 3-11 shows the selection of the remote machine to connect to.

Figure 3-10 - The Remote Connection Tool - RECON

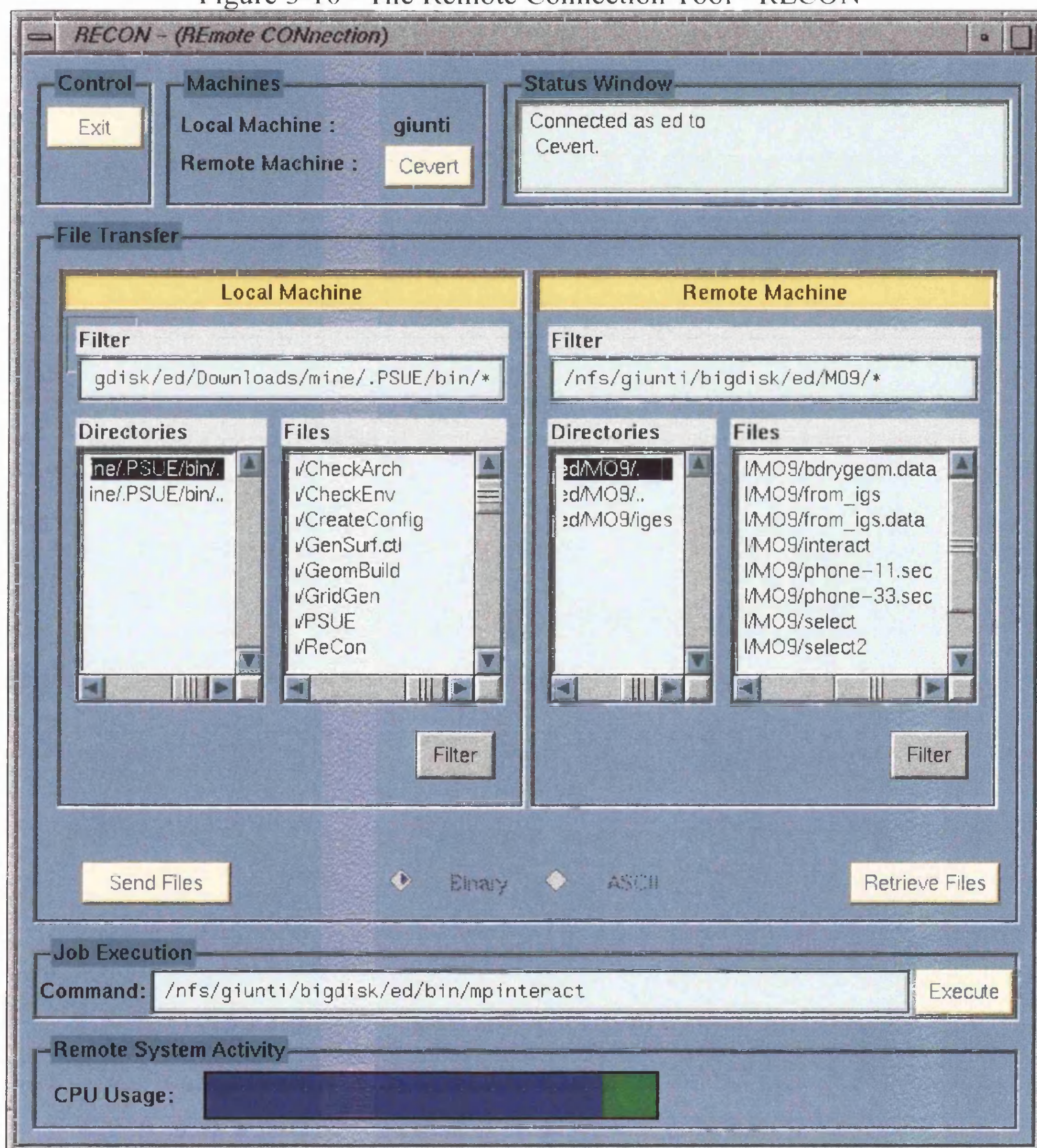
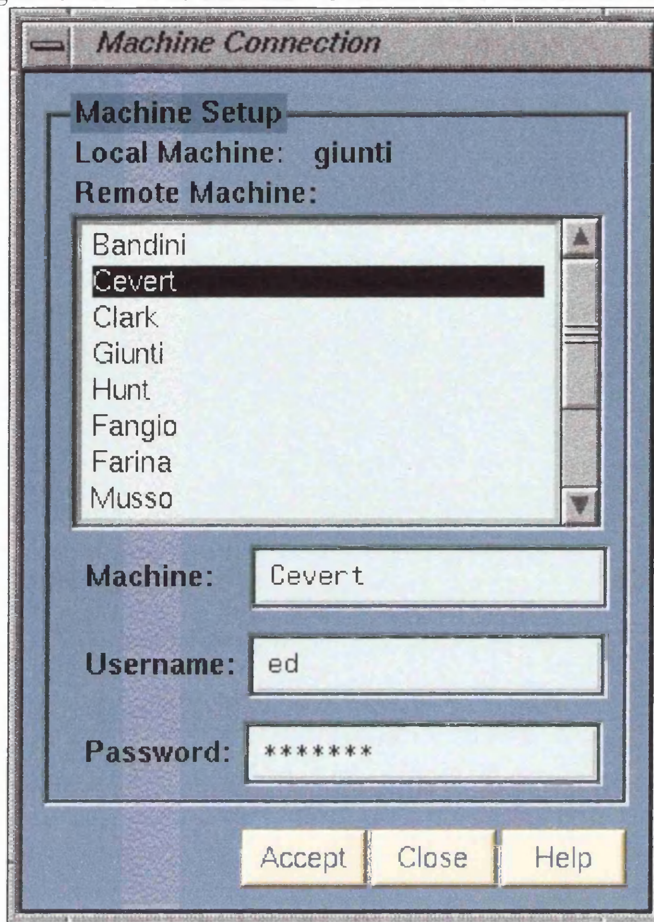


Figure 3-11 - The Machine Connection Panel for RECON



The entire module uses a specially designed communication interface using inter-process communications, TCP/IP and FTP (File Transfer Protocol). The module communicates using these standards to ensure that any machine may be linked to as these should work throughout the industry.

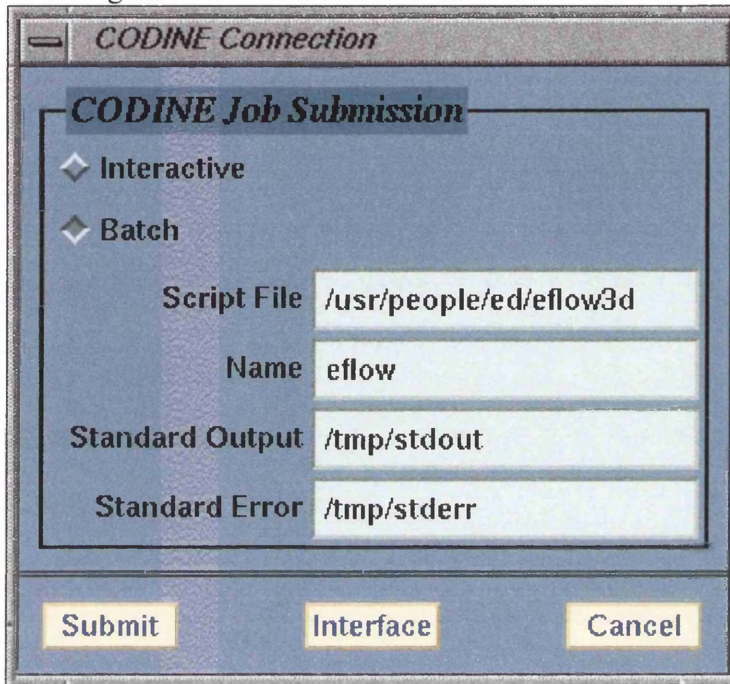
3.1.5.2 CODINE Interface

The CODINE Interface module was developed under the MEDUSA project and provides extensive links with CODINE that is a load-balancing package. It was integrated in such a way that the user does not come into contact with CODINE's interfaces directly and hence does not need to learn how to operate the software.

Figure 3-12 show the CODINE connection panel along with some example entries. The script file is the CODINE specific script file used to submit the proposed job to

the CODINE job queue. Once the job has been submitted it would be referred to by its name that is also specified within the connection panel and any program output is redirected to the specified files.

Figure 3-12 - The CODINE Connection Panel



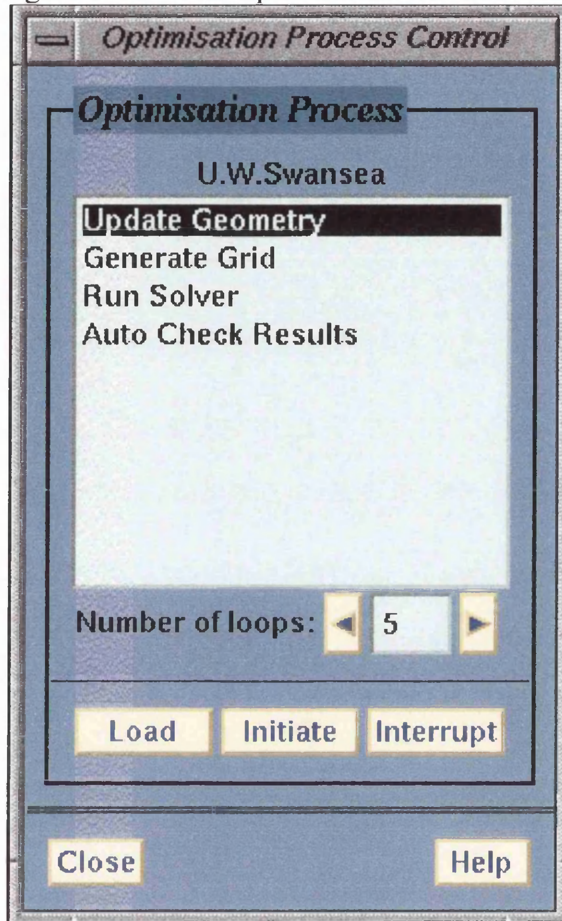
The CODINE interface is far more complicated than the CODINE connection panel since there is a wealth of functionality available to the user and administrator. However, once the software is configured it is expected that little intervention is required and therefore the connection panel forms a good bond to the CODINE software.

3.1.5.3 Optimisation Control Panel

The optimisation control panel is an extension to the PSUE main interface that was developed under the MEDUSA project. Figure 3-13 shows the optimisation control panel that is initiated from the main interface at any time. The interface uses the same script file technology as the user's functionality as described in section 5.4.1 and

therefore enables an arbitrary number of applications to be inserted into the optimisation loop.

Figure 3-13 - The Optimisation Control Panel



In the example in Figure 3-13, four steps have been used which will:

(a) Update the geometry

(b) Re-generate the computational grid

(c) Run the solver

(d) Check the results

Optimisation loops that are more complicated, which could contain conditional circumstances for example, can be implemented by using appropriate scripts or programs as a particular step within the optimisation process.

The number of loops of the optimisation may be specified that would normally be used as the maximum extent of the optimisation process. Once an optimisation script file has been loaded and the number of loops set, the "Initiate" button will start the optimisation process and the "Interrupt" button will interrupt and stop the whole process after the current activity has completed.

3.1.6 System Tools

This section is primarily for the user to integrate any miscellaneous applications into the environment but it also seemed like an ideal location to include the functionality that is found in the menubar. Therefore the following options are available:

- Colour Preferences – opens the colour resource management panel.
- Font Preferences – opens the font resource management panel.
- Process Communications – opens the process communications panel.
- Pathfinder – opens the pathfinder panel.
- Object Viewer – opens the object viewer that is a basic visualisation tool.

Figure 3-14 shows the functionality available under the "System Tools" Functionality.

Figure 3-14 - The System Tools Functionality

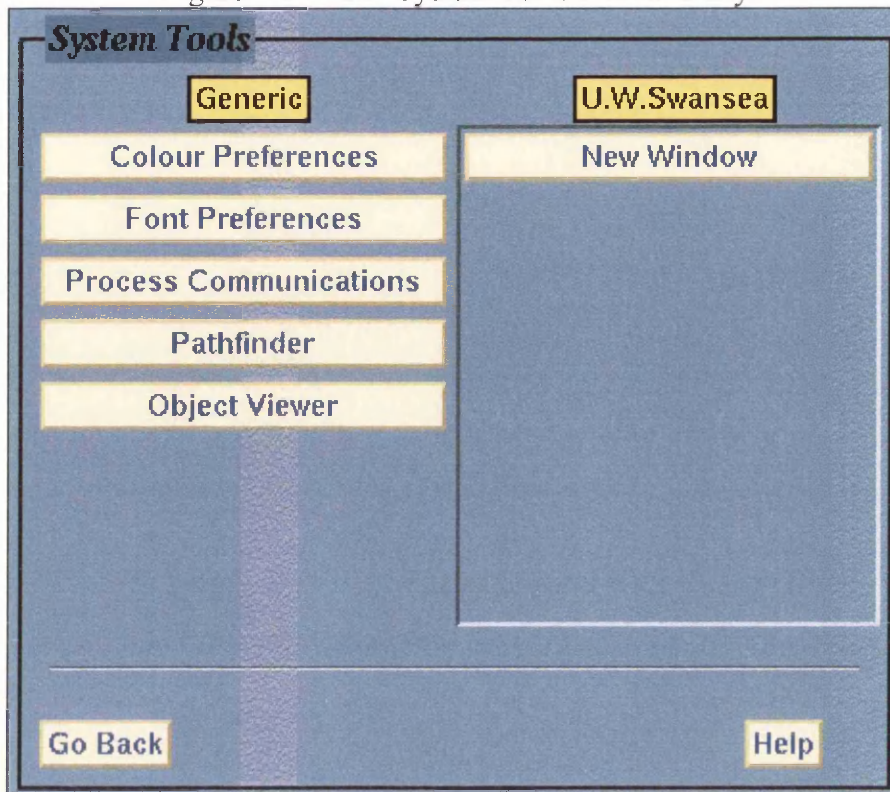
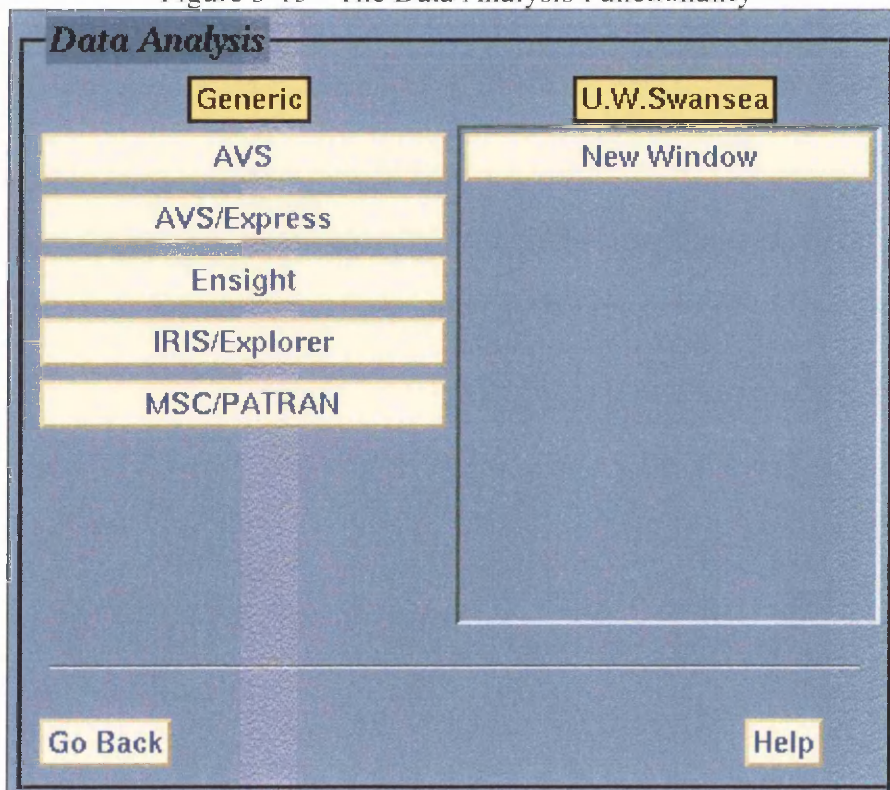


Figure 3-15 - The Data Analysis Functionality



3.1.7 Data Analysis

The data analysis section, shown in Figure 3-15, provides a number of links to proprietary software since it was decided that within the development time of the PSUE only very basic post-processing capabilities would be possible. Such a tool would never be able to compete with existing proprietary software and so the time was spent developing close links with some of the existing software such as:

- AVS and AVS/Express
- Ensight
- IRIS/Explorer
- MSC/Patran

3.2 Resource Management

All graphical applications involve colours and fonts and an outline of how to use these resources was given in section 1.5. During the development of the PSUE interface and the rest of the modules, these guidelines were followed where possible, however, the development team also wanted to produce a distinctive window environment. Since many users have different tastes and styles, resource management was provided to allow the user to produce their own individual presentation of the PSUE windows.

Upon changing any resource within the main PSUE interface, the results within the interface will be visible immediately, however, the other modules of the PSUE will only show the updated resources when restarted. In other words, any module that is already running will retain the original resource settings.

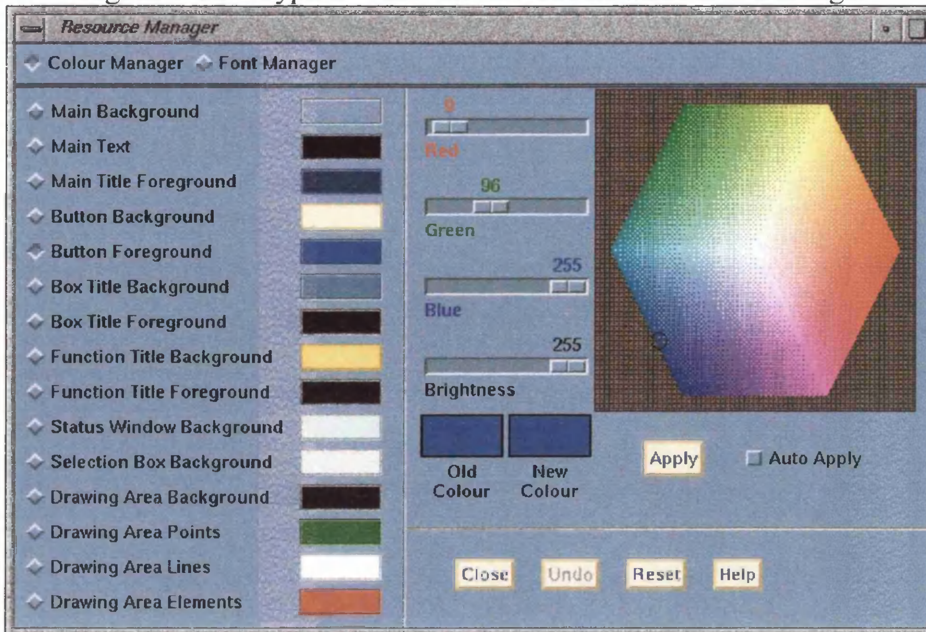
3.2.1 Colour Preferences

The colour management panel allows the user to change the colour of almost all parts of the PSUE interface. The colours are divided into a number of key descriptions:

- Main background colour for windows.
- Main foreground colour for windows (text).
- Title text colour (top of the main interface window).
- Background colour for buttons.
- Foreground colour for buttons.
- Background colour for frame labels.
- Foreground colour for frame labels.
- Background colour for functionality region labels.
- Foreground colour for functionality region labels.
- Background colour of the status window.
- Colours for the drawing area – background, nodes, edges and cells.

Figure 3-16 below, shows the typical view of the colour resource manager.

Figure 3-16 - Typical View for the Colour Resource Manager



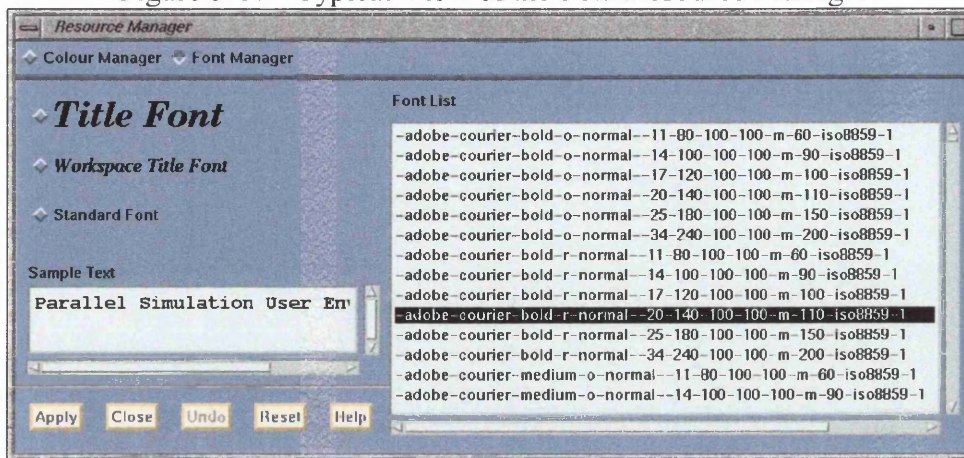
Whenever any change is made to the colour resources the new set-up is stored in a configuration file so that the settings will be retained upon subsequent initiation of the PSUE interface.

It should be noted that the author did not develop the hexagonal colour picker within the colour management panel.

3.2.2 Font Preferences

The font management panel allows the user to change the font of all regions of the main interface. There are only three options: the font for the main title, the font for framed regions and the font for all other text – labels, buttons, entry fields etc. The fonts available depend upon the platform that the PSUE is being displayed upon and which fonts have been installed on that platform. Figure 3-17 below shows a typical view of the font resource manager.

Figure 3-17 - Typical View of the Font Resource Manager

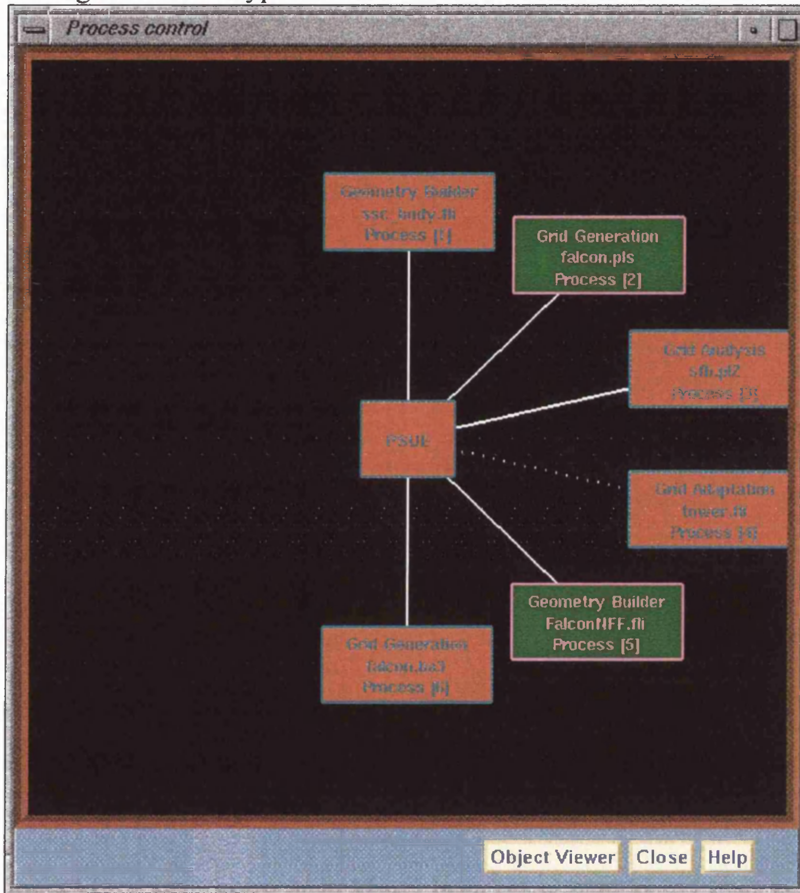


As with the colour management panel, any changes to the font resources are stored so that the same settings are retained for subsequent initiations of the PSUE interface.

3.3 Process Control

The process control panel allows the user to manipulate the processes and the memory management system of the communication library discussed in chapter 4. Figure 3-18 shows an example of how the process control panel may look during a session of using the PSUE main interface.

Figure 3-18 - Typical View of the Process Control Panel



A representation of the main PSUE interface is always shown in the centre of the screen and as modules are initiated, a representation of the process is added as a "satellite" with a communications link between the PSUE and itself. The information in the module's box is the module name; the last data loaded within that module or the data that may have been passed to it; and the process number. The process number is also displayed after the module name on the main window of the module itself. This is so that the user can identify processes when they want to transfer data between them.

The user can select and deselect processes and so when a process exits or updates the main interface, all selected processes will also be updated with the relevant data. To stop any communication between the PSUE and a module, the line connecting the representations of the PSUE and the module may be selected, toggling the connection on and off.

3.4 Pathfinder

The pathfinder is a utility that indicates to the user the functionality that is available and the next logical step forward, as well as providing a history of what has already been done. The panel is divided into two distinct regions, the pathfinder view and the history view, but if the user is working in expert mode then only the history view will be visible. This is so that the panel uses less of the screen up, since an expert user will probably not need the pathfinder view as they will already now what is available and what they want to do.

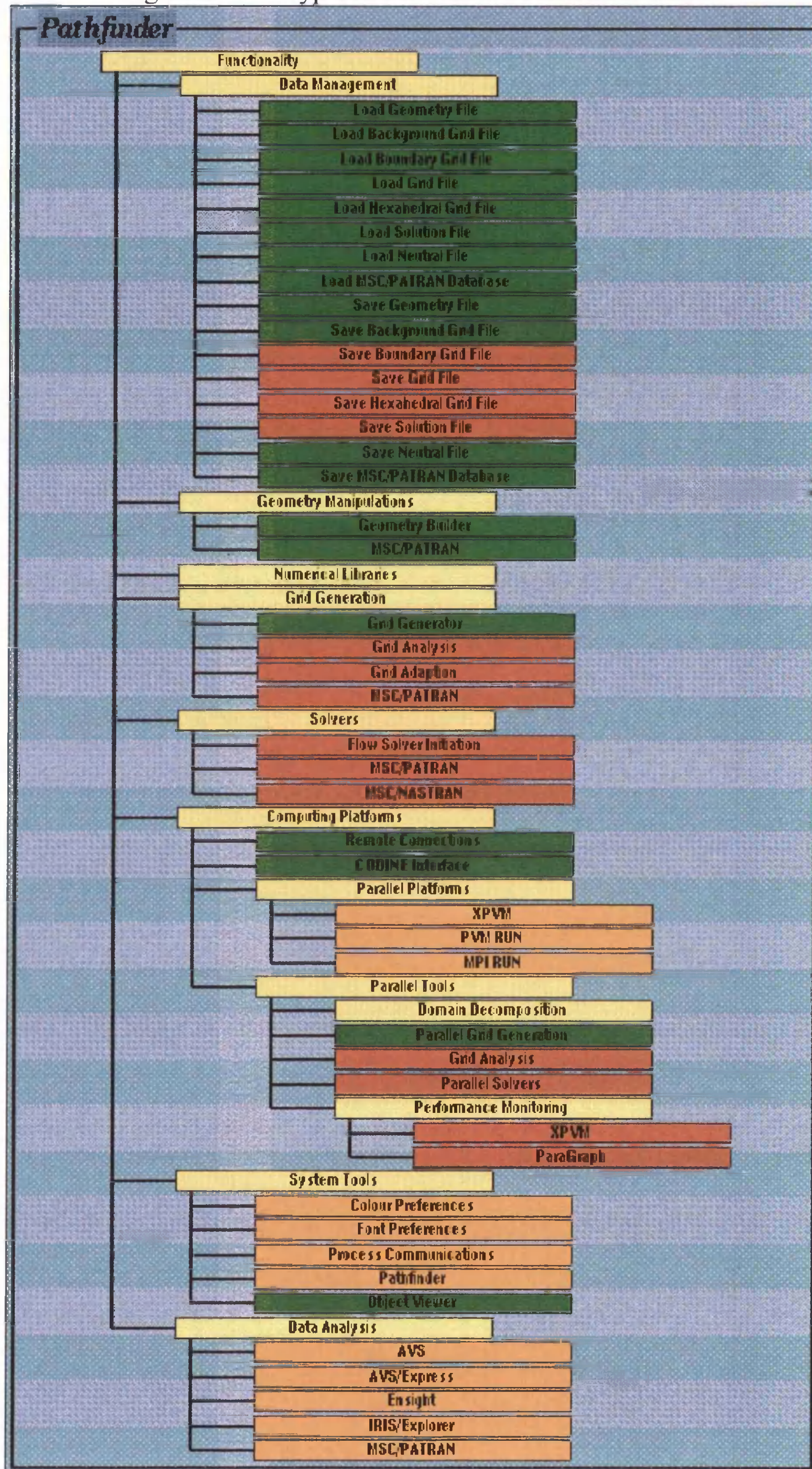
3.4.1 Pathfinder View

The pathfinder view shows all of the generic functionality of the PSUE as a tree structure to help show where the "original" buttons are within the main PSUE functionality region. Each of the separate options of functionality is coloured according to the availability of the particular function:

- Green infers that the option is available and likely to be a possible next step.
- Orange infers that the option is available but not necessarily the ideal next step.
- Red infers that the option is unavailable.

Figure 3-19 shows a typical view of the pathfinder panel. The colours of the options will change as data is created, loaded or removed, to indicate what new functionality is available or unavailable. If the colour of an option is green or orange then it may be selected, in the same way as a button, however, options that are coloured red may not be selected even though the "original" button in the main interface functionality region may be selected.

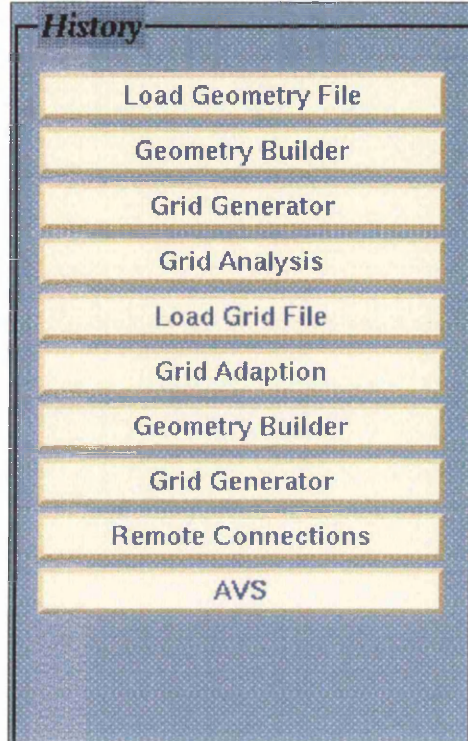
Figure 3-19 - Typical View of the Pathfinder Panel



3.4.2 History View

The history view is updated every time a PSUE functionality button is selected, and allows the user to have a quick link to the same functionality. The buttons will activate the appropriate modules, panels or interfaces as if the "original" button had been selected. Figure 3-20 shows a typical view of the history panel.

Figure 3-20 - Typical View of the History Panel

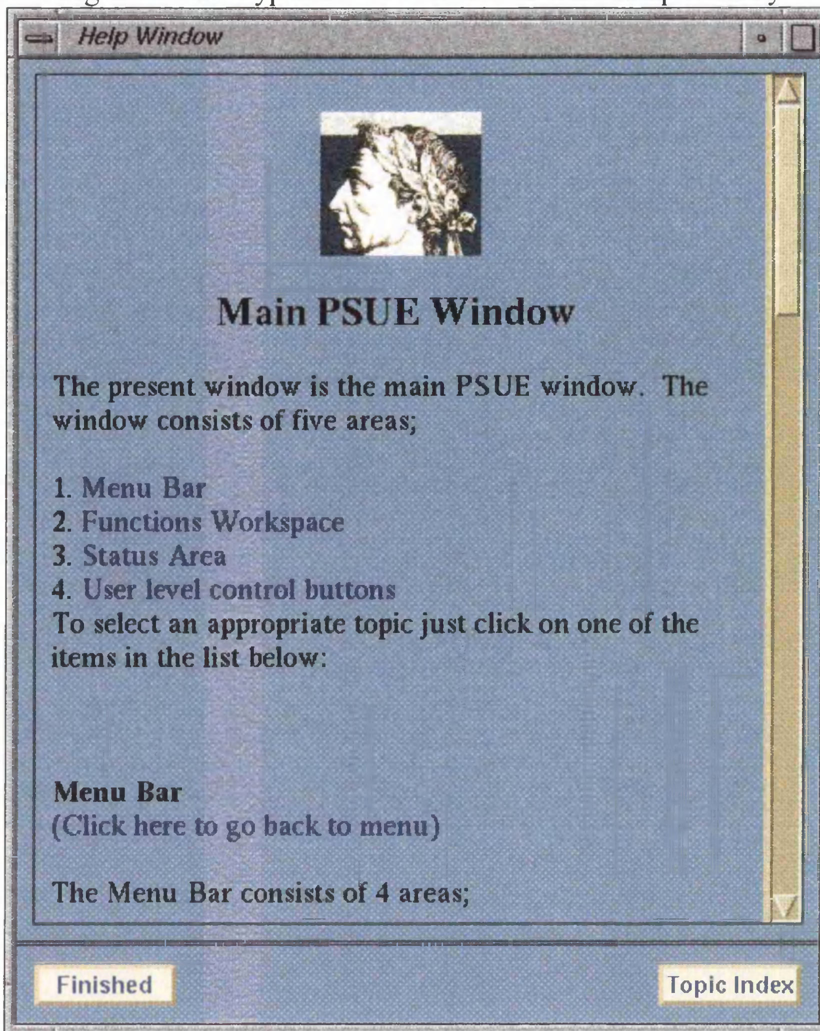


3.5 On-line Help

The on-line help facility is based upon a full hypertext concept allowing documents to link dynamically to other documents. The links to another document or to part of the existing document may be text itself or even pictures. All of the help pages are contained within text files that may be edited by the user or replaced by the users own copy of it. This allows the user to create their own help page or append particular notes to assist them in particular tasks later.

The help window system allows the user to access the help pages from any of the panels within the PSUE. When a help button is selected the help window will be shown, along with the appropriate information in the main panel of the help window. A typical view of the help panel is shown in Figure 3-21.

Figure 3-21 - Typical View of the On-Line Help Facility



It should be noted that the author did not develop the code directly associated with loading and displaying the help text and pictures.

3.6 Integration of Applications

As mentioned in the introductory chapter, the initial development under the CAESAR project required the integration of a number of commercial and confidential applications. The requirements capture for the project revealed that an alternative method of direct integration would have to be found, since this would take longer than the project would permit. This brought about the concept of the application integration library that allows each user of the PSUE to integrate their choice of arbitrary applications.

A complete description and analysis of the application integration facilities are covered in chapter 5, so this section gives only a brief outline.

At the beginning of this chapter, in section 3.1, the main functionality region was presented as two parts: the generic functionality and the user's functionality. It is through the integration of applications that a user may populate the right side of the functionality region with their own applications. Since we have seen that there is a hierarchical structure to the generic functionality menu, there is a corresponding region of user's functionality for each region of generic functionality. Text files that are loaded when the PSUE is initiated controls each region of the user's functionality.

The text files, which are stored in the user's own file store, are plain files that may be easily modified to integrate a new application. Upon loading a file, the PSUE ascertains whether it should be creating buttons, labels or pulldown menus. These entities are then presented in the appropriate region of user's functionality. The control file will also contain information about what applications to run when the button is selected and may also include details about the data to be transferred.

The key issue behind the integration of applications for the PSUE is the ability to transfer data to and from the arbitrary application. The PSUE transfers the data to the application using a "data handshake" and may be configured to use file, pipe or socket protocols to transfer the data.

4 COMMUNICATION LIBRARY

4.1 Overview and Requirements

As mentioned previously, the PSUE is separated into the main interface and a number of modules. Each of these modules is an application within itself and therefore to produce a fully integrated environment the main interface and modules must be coupled in some way. The reasons for maintaining all of the modules separately were discussed in section 1.6.

Most communication systems use one of two models:

- Client / Server Applications – One process acts as the server application and all other processes act as client applications (See Figure 4-1).
- Daemon Controlled Applications – All processes have equal status within the communication system but an extra process is continually running in the background controlling all communications between the other processes (See Figure 4-2).

Figure 4-1 - Client / Server Topology

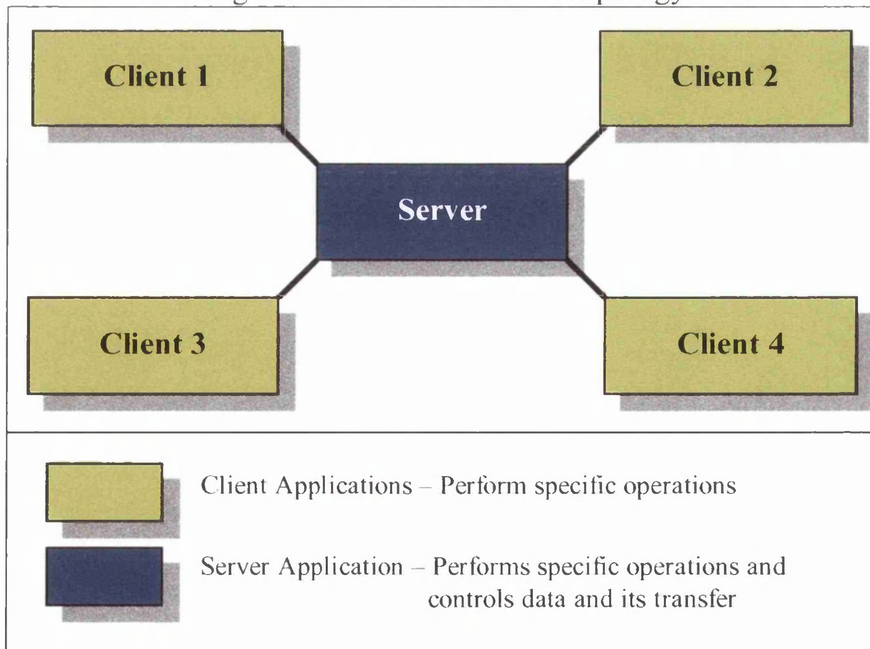
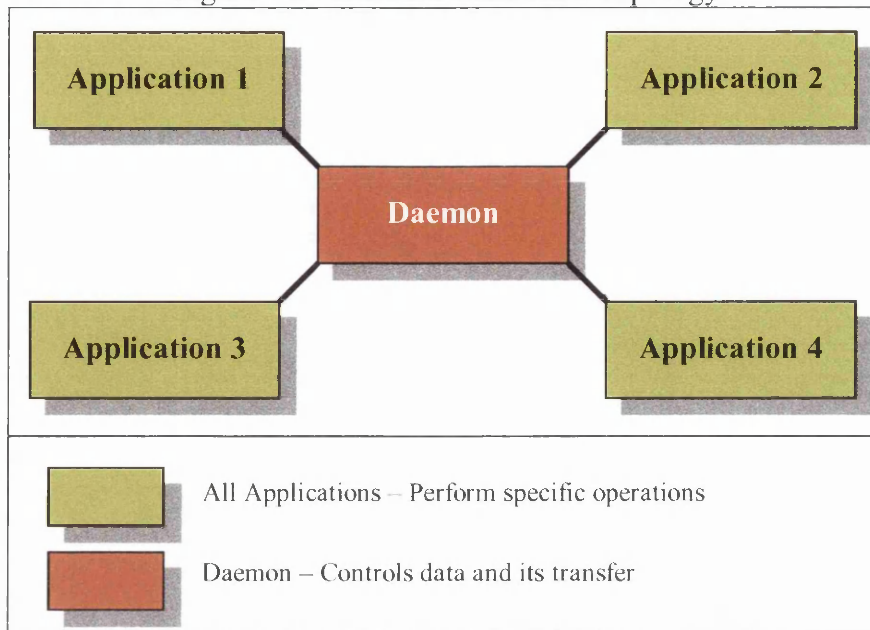


Figure 4-2 - Daemon Controlled Topology



The PSUE environment was developed as a client / server application with the main PSUE interface as the server application. This was because it was expected that the PSUE main interface would always be the starting point for any work within the PSUE environment. A daemon-controlled system was also eliminated due to the complexity and, in this case, the inefficiency of such a method. This inefficiency arises due to the expected large volume of data that would have to pass through the daemon no matter where its destination.

With the PSUE main interface as the server application, the other modules of the PSUE environment are the client applications communicating data back to the main interface. As the server application, the main interface then controls when and where to send the data to the next client application, if at all. Since the main interface itself utilises the data from the client applications there is not the inefficiency that would be expected in the daemon based communication system. For one client application or module to communicate with another client application, all data must be communicated through the server application, the main interface.

The communication system controls and transfers all data such as the internal process numbers, where to place the windows on the screen, through to actual geometrical and grid entity details, such as nodal coordinates.

The development of a communication system requires a large amount of planning and the key objective for the PSUE communication system was to ensure that the facility would be easily integrated into the main interface and all of the PSUE modules. The easiest method of providing such functionality is to create a library of functions that may be accessed by the server and client applications and hence the PSUE communication library was developed. Due to the modular structure of the PSUE environment and the number of developers involved, the library must be easily accessed and be as simple as possible to implement.

Due to the complexity of a communication system, ideally the communication library would be completely transparent to the user. However, since the PSUE main interface allows the expert user to influence where the data is transferred to and from, the communication library must allow some user manipulation.

The communication library must be fast, efficient and robust, however, if an error does occur, recovery of data is of the utmost importance. The communication system must be able to handle a number of different types of data, such as:

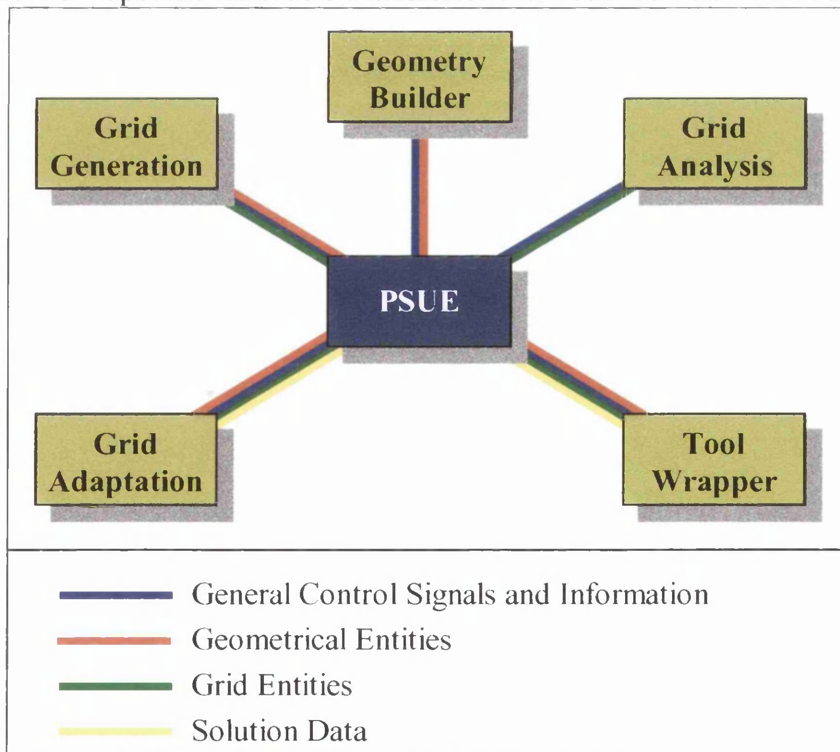
- Initial authentication tokens.
- Major event notifications.
- Actual geometrical entities:
  - Point coordinates.
  - Curve connectivity.
  - Surface connectivity.
  - Boundary conditions.
- Actual grid entities:
  - Nodal coordinates.
  - Cell connectivity.

- Boundary conditions.
- Material properties.

- Actual solution data:
  - Scalars.
  - Vectors.
  - Tensors.

Since there is a great deal of different types of entities and the number of such entities is expected to be very large (e.g. Millions of cells), the communication library must work intelligently to minimise the amount of data traffic. If all the data is transferred to all modules at all times the environment would very quickly grind to a halt. Figure 4-3 shows the different types of data transferred to and from each of the PSUE modules.

Figure 4-3 - Specific Data Sets Transferred Between the PSUE and Modules



Finally, the communication library must be easily expandable due to the ongoing development of the main interface and all of the modules. This requires a modular configuration of the data and a filter system to be built into the library.

4.2 Comparison of Various Techniques

The UNIX operating system provides a number of different methods in which communication of data may be carried out. The simplest method for data transfer is the use of files written and read from file stores. Most other methods may be collectively known as Inter-Process Communication (IPC) facilities and have significantly greater efficiency than file store usage.

4.2.1 IPC Facilities

4.2.1.1 Semaphores

A semaphore contains an integer value and any process that is connected to the semaphore may set or read the value. This is very useful for sending and receiving signals that are represented by a single value.

4.2.1.2 Message Queues

A message queue contains a text message and any process that is connected to the message queue may set or read the message.

4.2.1.3 Shared Memory

Shared memory is created in segments, much like normal memory, and all processes that are connected to the memory segment are able to read and write to the same block of memory.

4.2.1.4 Pipes

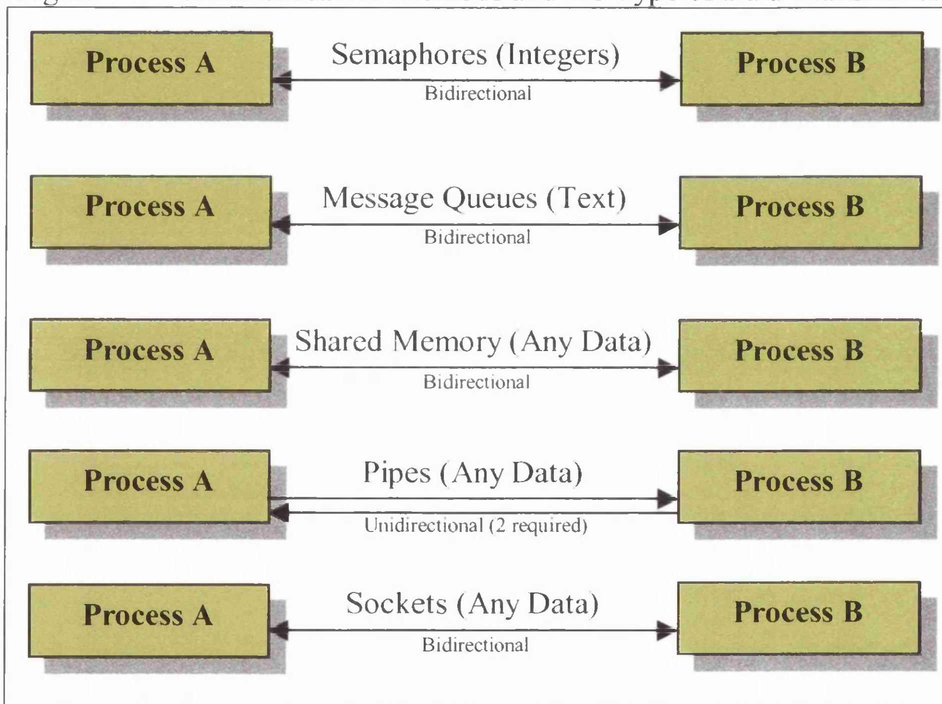
A pipe is a unidirectional data stream for sending or receiving arbitrary data by using a file store. An empty file is created and one process writes data to the file as the other process reads the data on a first in first out basis. Since no data is physically written to the file store a pipe is much more efficient than storing the actual data in a file.

4.2.1.5 Sockets

A socket is a bi-directional data stream for sending and receiving arbitrary data that utilises network protocols. This allows communication of data between multiple processes on different machines, which may be geographically distributed.

Figure 4-4 shows a comparison of the different communication methods. It shows that the sockets and shared memory methods provide the easiest method of communication since they allow any data type to travel bi-directionally.

Figure 4-4 - Communication Methods and the Type of Data Transmitted



4.2.2 Methodology

Two different communication systems were devised, one based on sockets and the other based on shared memory.

4.2.2.1 Socket Based System

As the PSUE main interface initiates a new module, it creates a new semaphore and socket connection to the module. The module then connects to both the semaphore and the socket. The semaphore is used to flag events between the main interface and the module, such as, a flag sent from the main interface to the module to inform the module that it should start sending any data that it has, back to the main interface. Any data that needs to be transferred to and from the module is done so by using the socket connection.

This method was found to be effective, however, the transfer of large data sets was very inefficient and large delays were incurred.

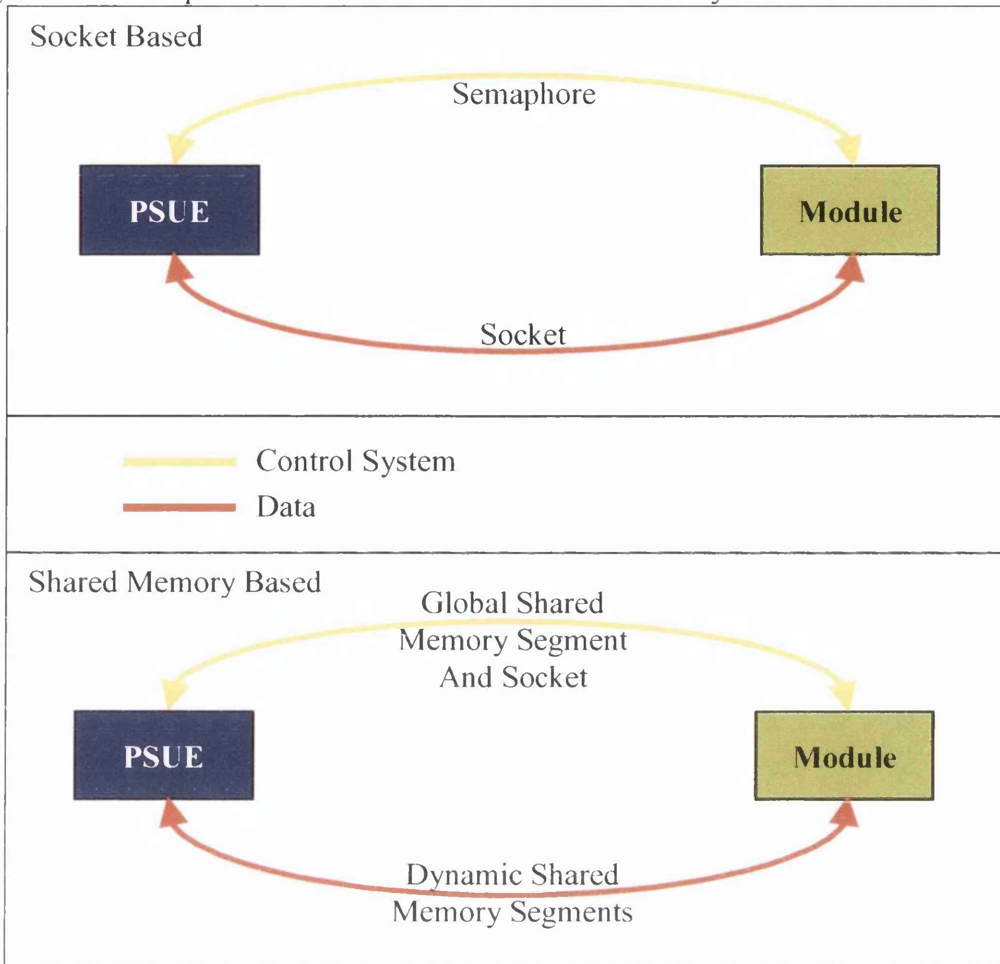
4.2.2.2 Shared Memory Based System

Upon initialisation of the PSUE main interface, a small shared memory segment is created, known as the global segment, which will hold the memory addresses to all other shared memory segments.

As the PSUE main interface initiates a new module, it creates only a socket connection to the module. The module then connects to the socket connection and also connects to the global shared memory segment. The socket is used to send basic information back and forth between the main interface and the module. The information consists of both integer values and text messages. When the main interface is required to send a data set to the module it will create the shared memory segments that it will need and populates the memory with the required data. The global shared memory segment is then updated so that the module may connect to the other shared memory segments and read the required data. When the module needs to send data the reverse occurs.

Even though this method is much more complicated than the socket based communication system, the efficiency for data transfer was much greater. Figure 4-5 shows the comparison between the two methods.

Figure 4-5 - Comparison of the Socket and Shared Memory Communication Methods



4.3 Data Set Manipulation

As a module loads or creates data, a data set is built within each module. From time to time these data sets will need to be passed through the communication system and it is anticipated that these data sets will become very large, e.g. 1+GByte of data. When creating a shared memory segment, the required size must be specified and memory of the machine is allocated to the shared memory segment. If only one shared memory segment was to be used for the entire data set the machine would need to find a block of free memory capable of holding the entire data. As the data sets get larger, the location of a single contiguous free memory region large enough is less likely, which in turn would cause a failure. Therefore, the data set is separated into clusters of data and provides the modular configuration that was discussed earlier in the chapter. In total, a data set that contains all types of data would require 21 separate shared segments (see section 4.3.2). Section 4.3.1 describes all of the different data types that are used within the PSUE environment and section 4.3.2 gives a breakdown of what data types are contained within each shared memory segment.

4.3.1 Data Types

The following tables show the configuration of the data types used throughout the communication system of the PSUE environment.

The coordinates object is used to describe the x, y, and z coordinates location such as a grid point.

Coordinates Object
X Coordinate
Y Coordinate
Z Coordinate

The vector object is used to describe the u, v, and w components such as a face normal direction.

Vector Object
X Component
Y Component
Z Component

The vertex object is used for geometrical nodes and contains a coordinate object to specify the actual x, y, and z components, a boundary condition, a material and property index and the ability to set grid source control specifications.

Vertex Object
Coordinates Object
Boundary Condition
Material Value
Property Value
Grid Source Values

The curve object is used for geometrical curves and contains a type flag to show how the curve is currently being used, the number of vertices used to make up the curve together with the array of the vertex indices, an index value for identification and a grid source flag for grid source specification.

Curve Object
Type
Number of Vertices
Array of Vertex Indices
Index
Grid Source Flag

The edge object is used to group geometrical curves and contains a type flag just like the curve object above, the number of curves used to make up the edge together with the array of the curve indices, and an index value for identification.

Edge Object
Type
Number of Curves
Array of Curve Indices
Index

The loop object is used to associate geometrical curves with geometrical surfaces. It contains the number of curves used to make up the edge together with the array of the curve indices. There is also the surface index that the loop is associated with and a surface specification that signifies if the surface is specified or not.

Loop Object
Number of Curves
Array of Curve Indices
Surface Index
Surface Specification

The surface object is used for geometrical surfaces and starts with a type flag to show how the surface is being used, e.g. boundary surface. The surfaces are represented as Coons patches and so the number of edges in the u and v directions are used followed by the array, of size u*v, edge objects, followed by the surface index for identification.

Surface Object
Type
Number of Edges in U Direction
Number of Edges in V Direction
Array of Edge Objects
Index

The cell object is used for generic grid cells and starts with the number of connectivities making up the cell. A boundary condition value may be stored and grid and geometry indices mat be specified to link the cell back to the original geometry. Finally the connectivity indices are given.

Cell Object
Number of Connectivities
Boundary Condition
Grid Index
Geometry Index
Array of Connectivity Indices

The line object is used to link multiple vertex objects together to create more complex grid control specifications such as line and triangular grid sources. The object contains the number of vertex objects to be used followed by the array of the vertex objects.

Line Object
Number of Vertices
Array of Vertex Objects

The discretisation object is used to associate grid point mappings on a given Coons surface patch. It contains the index to the grid point and a value of u and v (normalised) within the Coons patch. It is also used with curve objects in which case the v direction value is redundant.

Discretisation Object
Index
Normalised Value in U Direction
Normalised Value in V Direction

The curve tangent object is used to store the associated x, y and z tangents (or derivatives) on a curve object at the given x, y, and z coordinates.

Curve Tangent Object
X, Y, Z Coordinates
X, Y, Z Tangents

The surface tangent object is used to store the tangents on a surface object at the given x, y and z coordinates. The derivatives with respect to u and the derivatives with respect to v are stored along with second derivatives with respect to both directions.

Surface Tangent Object
X, Y, Z Coordinates
X, Y, Z Tangents with respect to U Direction
X, Y, Z Tangents with respect to V Direction
X, Y, Z Tangents with respect to both U and V Directions

The grid curve object is used to associate the positions of the grid points on an original curve along with the original curve discretisation. The number of grid points on the curve is stored followed by the discretisation objects for each of these grid points. The original curve description is contained as the number of original vertices followed by the curve tangent objects for each of these original vertices.

Grid Curve Object
Number of Redistributed Vertices
Array of Discretisation Objects
Number of Original Vertices
Array of Curve Tangent Objects

The grid surface object is used to associate the positions of the grid points on an original surface along with the original surface discretisation. The number of grid points on the surface is stored followed by the discretisation objects for each of these grid points. The original surface description is contained as the number of original vertices in the two parametric directions followed by the surface tangent objects for each of these original vertices.

Grid Surface Object
Number of Redistributes Vertices
Array of Discretisation Objects
Number of Original Vertices in U Direction
Number of Original Vertices in V Direction
Array of Surface Tangent Objects

The data set object controls the entire data structure model. It contains the total numbers of each object within the different areas of geometry, grid control specifications, grid definitions and solution data.

Data Set Object
Number of Geometrical Vertices
Number of Geometrical Curves
Number of Geometrical Loops
Number of Geometrical Surfaces
Number of Tri/Tet Grid Vertices
Number of Tri/Tet Grid Cells
Number of Tri/Tet Boundary Grid Cells
Number of Grid Curves
Number of Grid Surfaces
Grid Type
Number of Quad/Hex Grid Vertices
Number of Quadrilateral Cells
Number of Hexahedral Cells
Number of Background Grid Vertices
Number of Background Grid Cells
Number of Point Grid Sources
Number of Line Grid Sources
Number of Triangle Grid Sources
Number of Solution Vertices
Number of Solution Cells
Number of Variables per Vertex
Number of Variables per Cell

4.3.2 Shared Memory Segment Breakdown

The 21 shared segments that collectively cover an entire data set are broken down as shown in Table 4-1.

Table 4-1 – The Shared Memory Segments and their Content

Shared Segment	Description	Content
1	Numbers for all major data types	Data Set Object
2	Geometrical Vertices	Array of Vertex Objects
3	Geometrical Curves	Array of Curve Objects
4	Geometrical Loops	Array of Loop Objects
5	Geometrical Surfaces	Array of Surface Objects
6	Tri/Tet Grid Vertices	Array of Vertex Objects
7	Tri/Tet Grid Cells	Array of Cell Objects
8	Tri/Tet Grid Boundary Cells	Array of Cell Objects
9	Grid Curves	Array of Grid Curve Objects
10	Grid Surfaces	Array of Grid Surface Objects
11	Boundary Grid Vertices	Array of Vertex Objects
12	Boundary Grid Cells	Array of Cell Objects
13	Quad/Hex Grid Vertices	Array of Vertex Objects
14	Quadrilateral Cells	Array of Cell Objects
15	Hexahedral Cells	Array of Cell Objects
16	Background Grid Vertices	Array of Vertex Objects
17	Background Grid Cells	Array of Cell Objects
18	Point Grid Sources	Array of Vertex Objects
19	Line Grid Sources	Array of Line Objects
20	Triangle Grid Sources	Array of Line Objects
21	Solution Variables	Array of Real Values

4.4 Shared Memory Communication System

Before any description of the communication system may be started, it should be clarified how a process monitors system activity and hence knows when particular actions need to be taken. Since the main PSUE interface and all of the modules are X-window based applications, a special functionality has to be used. This functionality is a "time-out" routine that is called after a specific period. For example, the main interface will need to check for messages from all of the modules frequently, perhaps every $1/10^{th}$ of a second. Upon receiving a message the program can process the message and then go back to monitoring all the modules. The "time-out" routine must continue to call itself after the specified time delay.

4.4.1 Initialisation of the PSUE Main Interface

When the PSUE main interface is started it will initialise the communication system by creating a shared memory segment and two dummy sockets. The shared memory segment is known as the global shared memory segment as it will always be available to all processes and will only be destroyed when the main interface terminates. The segment contains space to store all of the other shared memory segment id numbers and the actual memory addresses. The creation of the shared memory segment involves the actual creation command and then the attachment of the main interface to the shared memory segment, which provides the memory address.

The two dummy sockets that are created are used to replicate unique sockets that are the actual sockets used to communicate messages back and forth between the main interface and the individual modules. Each module has two of it's own sockets, one for reading messages and one for writing messages. Even though sockets are bi-directional, two sockets are used for each module so that race conditions are eliminated. Race conditions occur when it is undefined which process will try to read from the socket before the other.

4.4.2 Initiation of a PSUE Module

When a module is initiated from within the main interface, Table 4-2 shows the actions that are performed.

Table 4-2 – Actions Performed during Initiation of a PSUE Module

Main Interface Actions	Module Actions
1. Module started using command line arguments that specify global shared memory segment id, and both dummy socket ids.	Non-existent
Waiting	2. Check command line arguments for connection details. 3. Attempts to connect to first dummy socket.
4. Creates shared memory segments. 5. Attaches to the shared memory segments. 6. Copies the local memory into the shared memory segments. 7. Detaches from the shared memory segments. 8. Creates socket for receiving messages. 9. Creates socket for sending messages.	Waiting
Waiting	10. Connects to socket for sending. 11. Connects to socket for receiving. 12. Sends module name. 13. Sends module process id.
14. Receives module name. 15. Receives module process id. 16. Sends main interface process id. 17. Sends PSUE ID for the module.	Waiting
Waiting	18. Receives main interface process id. 19. Receives PSUE ID for the module. 20. Attaches to the global shared memory segment. 21. Attaches to all other shared memory segments. 22. Copies shared memory data into local memory. 23. Detaches and deletes shared memory segments. 24. Send OK signal.
25. Receive OK signal.	Continues initiation.

4.4.3 PSUE Module updates Main Interface

Table 4-3 shows the actions that are performed by both the PSUE main interface and the module when the module updates the main interface with a data set.

Table 4-3 – Actions Performed when Updating the Main PSUE Interface

Main Interface Actions	Module Actions
Monitoring modules.	1. Creates shared memory segments as appropriate. 2. Attaches to all shared memory segments. 3. Copies local memory data into shared memory segments. 4. Detaches from the shared memory segments. 5. Sends UPDATE signal.
6. Receives UPDATE signal. 7. Attaches to shared memory segments. 8. Copies shared memory data into local memory. 9. Detaches and deletes shared memory segments. 10. Updates all other modules that require it. (See section 4.4.4)	Continues working as before.

4.4.4 Main Interface updates PSUE Module

Table 4-4 shows the actions that are performed by the PSUE main interface and any module that requires updating with the data set from within the main interface.

Table 4-4 – Actions Performed when the Main Interface updates a PSUE Module

Main Interface Actions	Module Actions
1. Creates shared memory segments as appropriate. 2. Attaches to all shared memory segments. 3. Copies local memory data into shared memory segments. 4. Detaches from the shared memory segments. 5. Sends DATA signal.	Monitoring connection to main interface.

Waiting	6. Receives DATA signal. 7. Attaches to shared memory segments. 8. Copies shared memory data into local memory. 9. Detaches and deletes shared memory segments. 10. Sends OK signal.
11. Receives OK signal.	Continues working as before.

4.4.5 Closure of a PSUE Module

When a module closes due to either user intentions or an error, the actions that are performed are shown in Table 4-5. Since the main interface is updated with the data set that was currently within the module, it is hoped that no actual data loss will occur even if an error causes a module to crash.

Table 4-5 – Actions Performed during the Closure of a PSUE Module

Main Interface Actions	Module Actions
Monitoring modules.	1. Updates the main interface as specified in section 4.4.3. 2. Detaches global shared memory segment. 3. Sends EXIT signal.
4. Receives EXIT signal. 5. Sends OK signal.	Waiting
Continues monitoring.	6. Receives OK signal. 7. Exits.

4.4.6 Closure of the PSUE Main Interface

Upon the shutdown of the main interface, it is expected that the user is actually shutting down the entire environment. Therefore, the main interface sends a QUIT signal to each module and waits for a confirmation OK signal. As each module receives the QUIT signal they send the OK signal back to the main interface and proceeds to close the application.

4.4.7 Error within the PSUE Main Interface

If an error was to occur within the main interface data loss is the primary concern and the modules that the user is working in should not be affected. Therefore, upon an error occurring, the main interface will send a DISC signal to all modules to inform them to disconnect from the communication system.

5 APPLICATION INTEGRATION

5.1 Overview and Requirements

During the development of the PSUE under the CAESAR project it was realised early on that for the environment to be a success a number of multi-disciplinary engineering applications would have to be integrated into the PSUE. The project did not provide adequate time in which to fully integrate all of the applications, therefore an alternative method to direct integration had to be devised. An application integration system was proposed, in which an arbitrary application maybe integrated into the environment with an almost invisible data connection.

The integration of almost any application must be achievable and the transfer of data between the PSUE and the application must be as simple as possible. A particular user must be able to integrate an arbitrary number of applications in a structured manner. This will allow a user to group particular applications together and provide a similar structure to the generic functionality of the PSUE as discussed in section 3.1.

The user may integrate their functionality within the groups provided by the generic functionality of the PSUE. Each section of functionality is controlled by a specific control file and allows the user to specify the application, data transfer type and data components required for the integration of the application. The control file for each functionality region is a text file stored in the PSUE configuration directory and is specific to each user of the PSUE. The text file format is designed to be as simple as possible but at the same time being able to include as much information as would be required to fully integrate an application.

The transfer of data between applications is perhaps one of the most important aspects of the application integration system. Data must be able to flow from the PSUE to the application and back again as seamlessly as possible. Once the data requirements have been specified the user should be able to use their application as if the application was part of the PSUE. There should be a choice of data transfer protocols and the type of data to be transferred should be customisable. The latter is usually a very complicated process and was not integrated into the basic application integration system but has

been utilised within the Application Tool Wrapper, which is discussed in the next chapter.

5.2 Philosophy

One of the main requirements for the integration of applications is the ability to integrate almost any application into the PSUE environment. This has required a very structured approach to the design of the application integration system involving a number of key aspects that include:

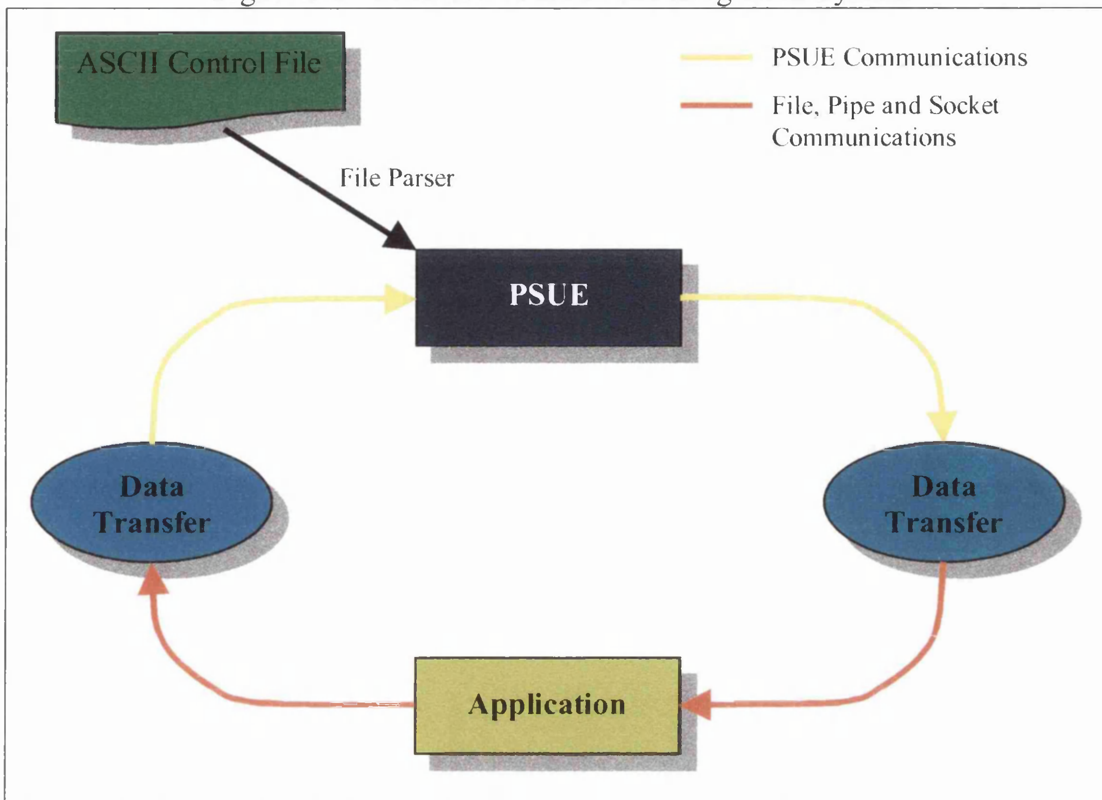
- Application configuration.
- Application and data control.
- Application initiation.
- Data transfer.

The integration of applications into the PSUE is implemented using the text files previously mentioned. These files contain the details of button labels, application executables, data transfer criteria and menu configuration. The PSUE controls the initiation of an application and any data transfer to and from the new process as shown in Figure 5-1.

The level to which an application may be integrated into the environment depends on a number of issues including;

- The type of application
- The configurability of the application
- Access to application source code

Figure 5-1 - Structure of the PSUE Integration System



The ASCII script file is created by the user and controls the appearance of applications within the PSUE interface. The initiation of an application requires the specification of data transfer information for communication from the PSUE to the application and also from the application back to the PSUE. This specification may be given during the initialisation of the application or within the controlling script files for an automated connection to the application.

5.3 Development Stages

Due to the structured nature of the application integration system, the development was easily separated into a series of stages. There were four stages in all that comprised the application button integration, application initiation, data transfer and automation.

5.3.1 Application Button Integration

The integration of applications into the PSUE is visible to the user through the appearance of push buttons, within the functionality region of the main interface. Figure 5-3 shows an example of the functionality region with three user applications integrated. The appearance of these buttons is activated by the creation of script files that also contain details of the applications to integrate. Since there is a series of generic functionality regions as discussed in section 3.1, each region has a corresponding script file that controls the applications in that region. Table 5-1 below shows the corresponding script file for each functionality region.

Figure 5-3 - Example of Three User Applications Integrated into the Main Functionality Region.

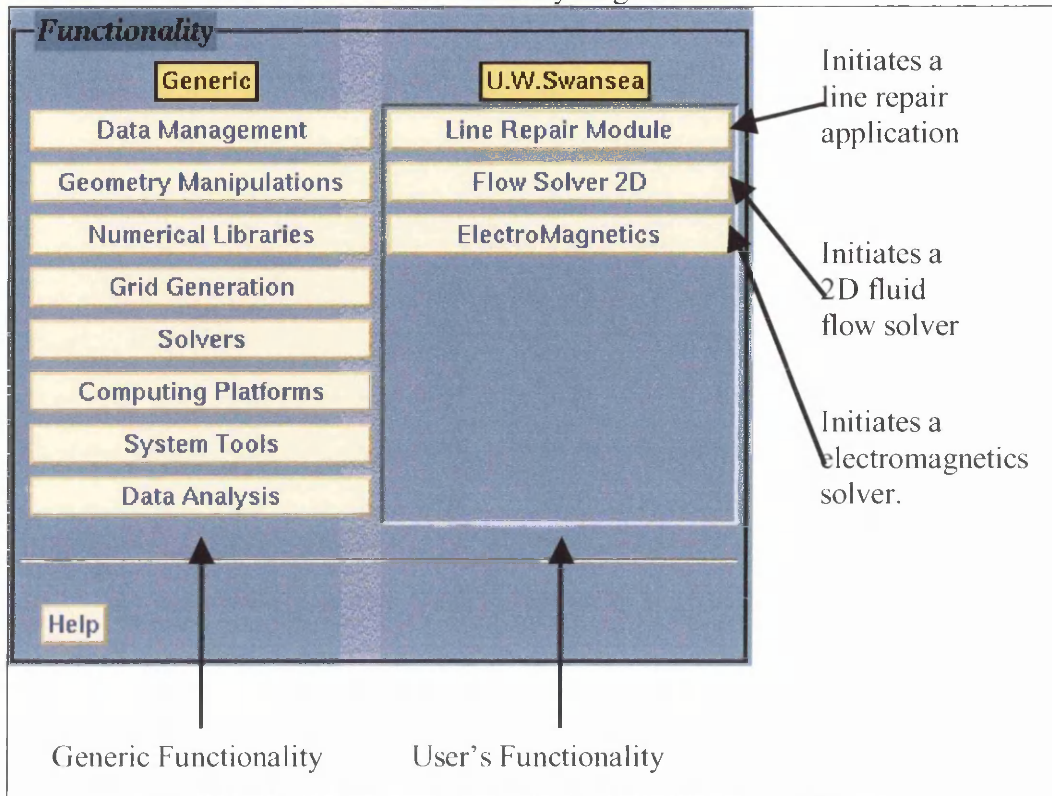


Table 5-1 - Script Files Controlling Functionality Regions

Functionality Region	Controlling Script File
Main generic functionality	PSUEscript.main
Geometry manipulations	PSUEscript.geom
Numerical libraries	PSUEscript.nlib
Grid generation	PSUEscript.grid
Solvers	PSUEscript.solv
Computing platforms	PSUEscript.comp
Parallel platforms	PSUEscript.plat
Parallel tools	PSUEscript.ptls
Domain decomposition	PSUEscript.dode
Performance monitoring	PSUEscript.pmon
System tools	PSUEscript.syst
Data Analysis	PSUEscript.data

This part of the development of the application integration system involved the creation of the data format for the script files and the production of the appropriate translator within the PSUE initialisation sequence.

5.3.2 Application Initiation

Once the application buttons are integrated into the PSUE window environment the next stage of the development was to initiate the application upon activation of the corresponding button. The initiation of applications actually came in two stages, now as a simple process initiation step and later, as data transfer was introduced, as a process initiation and manipulation step. The first stage initiation only required the immediate initialisation of the required procedure with absolutely no other interaction or manipulation. The second stage initiation required different technical routines to allow command line arguments to be issued followed by extensive communications between the PSUE and the newly initiated process. In this second stage of the initiation process, the process must continually be monitored so that if the PSUE looses contact with the application, appropriate steps for data recovery and a clean disconnection from the PSUE are carried out.

5.3.3 Data Transfer

The third stage of development of the application integration system was the ijtroduction of data transfer. This involved the ability to transfer data from the PSUE to the application upon its initiation and then the transfer of data from the application to the PSUE upon its completion. Three separate methods of transferring data were developed that use files, pipes and sockets. At this stage, due to the increasing complexity of the application integration system, a process initiation panel had to be developed that may be seen in Figure 5-5 below.

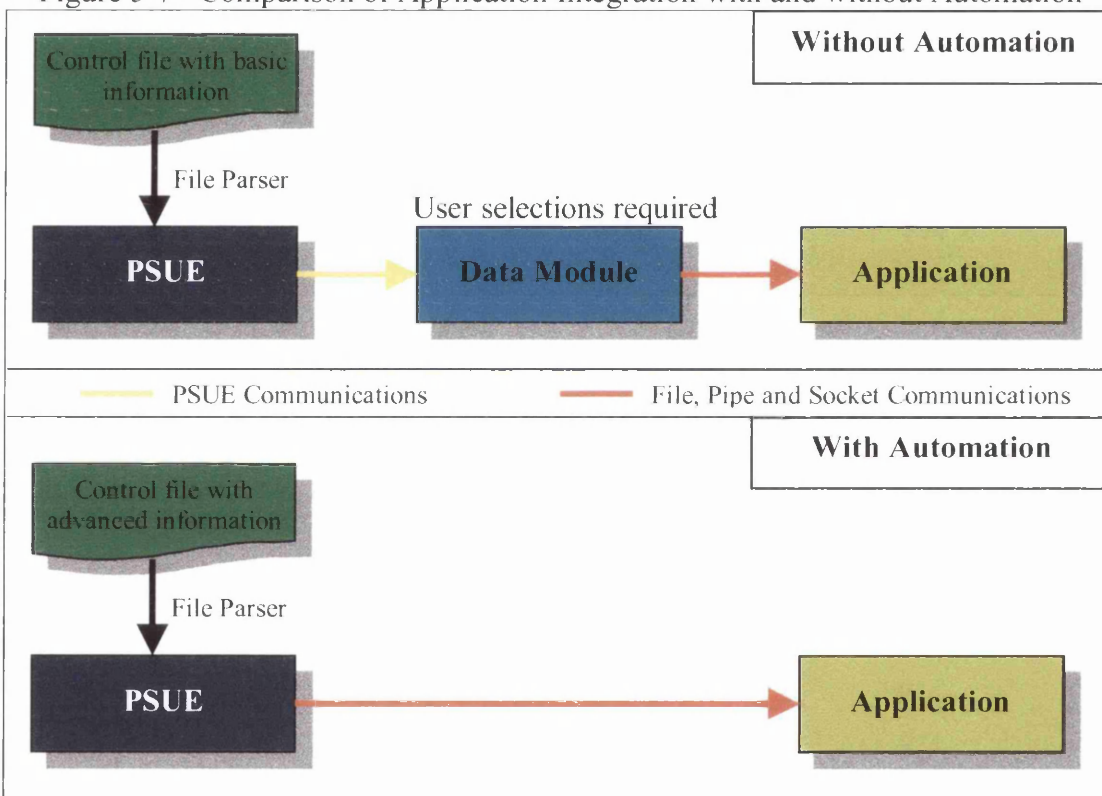
Figure 5-5 - Application Initiation Panel

5.3.4 Automation

The final stage of the application integration system was the automation step. The main development for this stage involved the format of the script files and the translator within the PSUE initialisation procedure. A series of new commands were integrated into the file format of the script files that would replace the need to open and use the application initiation panel. Since most applications would be set up to specifically use a particular communication method and data format every time that

application was initiated, all of the appropriate options would have to be set in the application initiation panel. The automation process allows the user to specify all the requirements only once within the script files. After doing this, upon selection of the application button within the main functionality regions, the application would be initiated immediately without using the application initiation panel, making the application integration a far shorter task. Figure 5-7 show a comparison between the two methods showing a far smoother integration of an application when using the automated approach.

Figure 5-7 - Comparison of Application Integration with and without Automation



5.4 Techniques Employed

A number of special techniques were employed throughout the application integration system, which include script files, initiation procedures, pipes and sockets. This section will describe each of these in further detail.

5.4.1 Script Files

Upon initialisation of the PSUE, all of the application integration script files are examined for their existence and content. As the window environment is created the script files give details of each of the buttons to be created and data is stored about the corresponding applications. The number of buttons that may be integrated into a particular region is unlimited and the specification of drop down menus is also possible. Once a functionality region is filled with buttons any further addition of application buttons or menu buttons will cause the functionality region to become a scrolled region. The user may then scroll up and down in order to access all of their applications. Under these circumstances, it may be better for the user to specify menus of buttons to group the applications and reduce the scrolling requirement.

At this first stage, the script files controlling application integration only involve application button names and the corresponding executable with its full path name. The script file starts with a line that gives the label that appears at the top of the functionality window and is followed by a menu command. The script files are set out using menus with the main menu at the top. Each menu contains both application buttons and their executables or other menu names. In order for the PSUE to know when a menu is being referenced the name must be "Menu" and the next line is the name and label of the menu, which must be specified later in the script file. A standard entry for an application has the application button label followed by the application executable.

Example 1 – This will produce two buttons labelled "Program 1" and "Program 2" that will initiate the applications "/usr/bin/program1" and "/usr/bin/program2" respectively. The label at the top of the functionality region will be "U.W.Swansea".

```
U.W.Swansea
Menu Main
{
Program 1
/usr/bin/program1
Program2
/usr/bin/program2
}
```

Example 2 – This will produce a series of menus and buttons and again the label of the functionality region will be "U.W.Swansea".

```
U.W.Swansea
Menu Main
{
Program 1
/usr/bin/program1
Menu
Structured Grid Generators
Program 2
/usr/bin/program2
Menu
Unstructured Grid Generators
}
Menu Structured Grid Generators
{
Program 3
/usr/bin/program3
Menu
Quad Grid Generators
}
Menu Quad Grid Generators
{
...
...
}
Menu Unstructured Grid Generators
{
...
...
}
```

The second stage of the script file development includes the automation sequences and commands. As discussed in section 5.3.4, the automation sequence allows the

user to avoid the application initiation panel and therefore the script files must allow for all possible selections within the application initiation panel.

The automation description for each application follows immediately after its existing reference for name and executable. The first entry flag to indicate whether this application has an automated description or not - a zero indicates no automation while a one indicates that there is automation and the description follows. The next entry may be one of NONE, FILE, PIPE or SOCKET, which indicates the method of transferring any data. If NONE has been specified then no further information is required and the application will initiate with no data transfer as soon as the application button is activated. If FILE has been specified then two further entries must be given, the first is the file type and the second is the filename. Obviously, this restricts the user to always using the same file to transfer data but this can be very useful if a temporary area us used, such as /tmp. If PIPE or SOCKET is specified then a pipe or socket, which are discussed further on in this chapter, is used to transfer the data to and from the application. There are three options available for pipe and socket transfers that are OUT, IN or BOTH. If the OUT option is specified then data will only be sent from the PSUE to the integrated application upon initialisation and will be sent in the data format that must be specified in the next entry. If the IN option is specified then data will only be received by the PSUE when the integrated application is completed and will receive the data in the format that must be specified in the next entry. If the BOTH option is specified then data will be transferred from the PSUE to the integrated application upon initialisation in the data format that must be specified in the next entry. Data will also be transferred from the integrated application back to the PSUE, when the application is completed, in the data format that must be specified in the next entry.

Whenever a data format must be specified as laid out in the above description, seven options exist:

- GEOMETRY – The full geometrical definition.
- BOUNDARY – The boundary grid definition.

- BACKGROUND – The background grid density definition.
- GRID – The full triangular and/or tetrahedral grid definition.
- HEXGRID – The full quadrilateral and/or hexahedral grid definition.
- RESULTS – The solution results definition.
- NEUTRAL – All definitions.

Example 3 – The following example shows the use of the automation commands and gives a description of each line on the right hand side.

Mixed Applications	Title string
Menu Main	Main menu
{	
Comments Form	First application name
/usr/bin/feedback	First application executable
0	No automation
Xterm	Second application name
/usr/bin/X11/xterm	Second application executable
1	Automated application
NONE	No data transfer (just start process)
Fileview	Third application name
/usr/people/jo/bin/fileview	Third application executable
1	Automated application
FILE	Use file transfer
NEUTRAL	Use the neutral data file type
/tmp/jobloggs.neu	Save to this filename
Grid Flow	Fourth application name
/usr/people/jo/bin/gfp	Fourth application executable
1	Automated application
SOCKET	Use socket transfer
BOTH	Data to be transferred both ways
GRID	Grid data to be sent out
RESULTS	Results data to be brought back in
}	

Figure 5-5 shows the flow diagram of the main part of the file parser and how it iterates until all information and menus have been loaded and built. Figure 5-6 shows the flow diagram of the file parser for the automation sequences.

Figure 5-5 - Flow Diagram of the Main Part of the Script File Parser

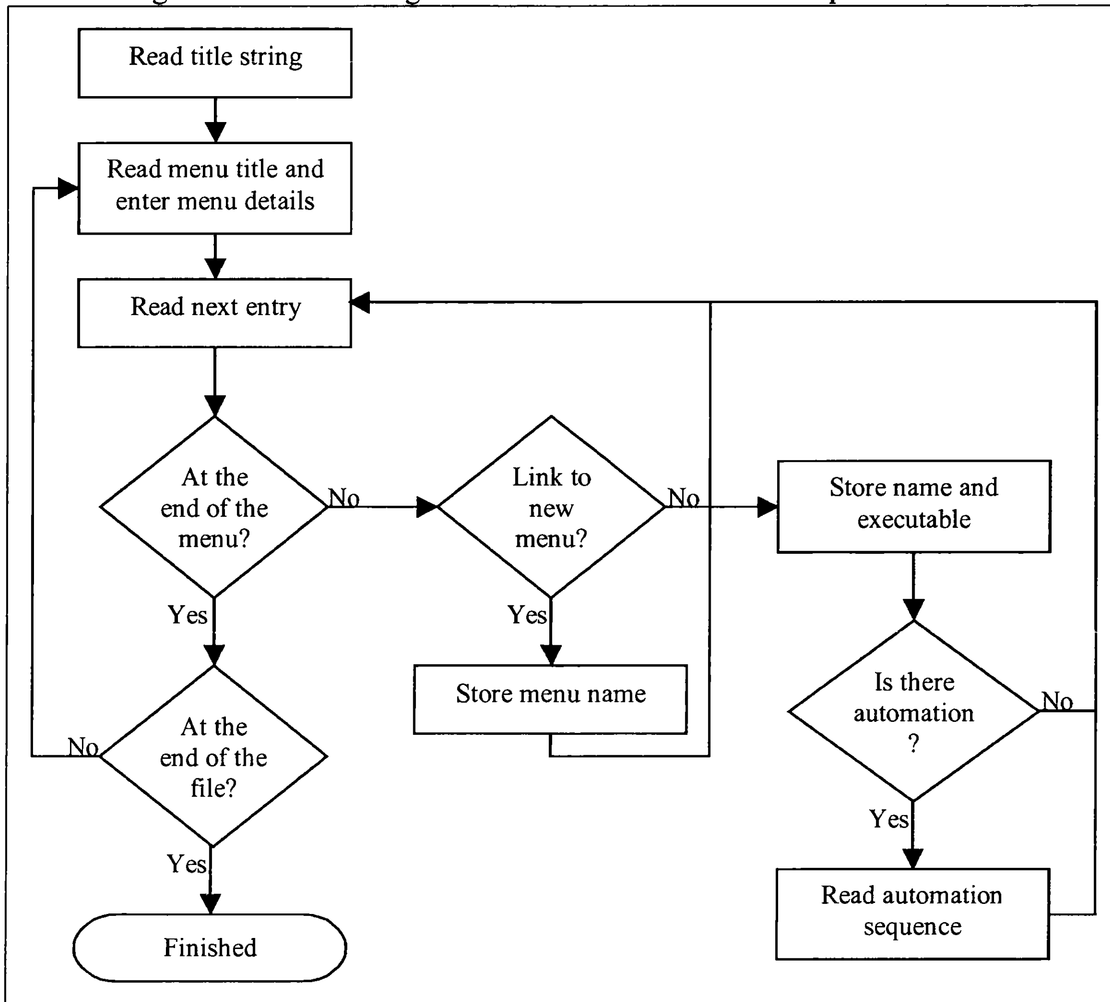
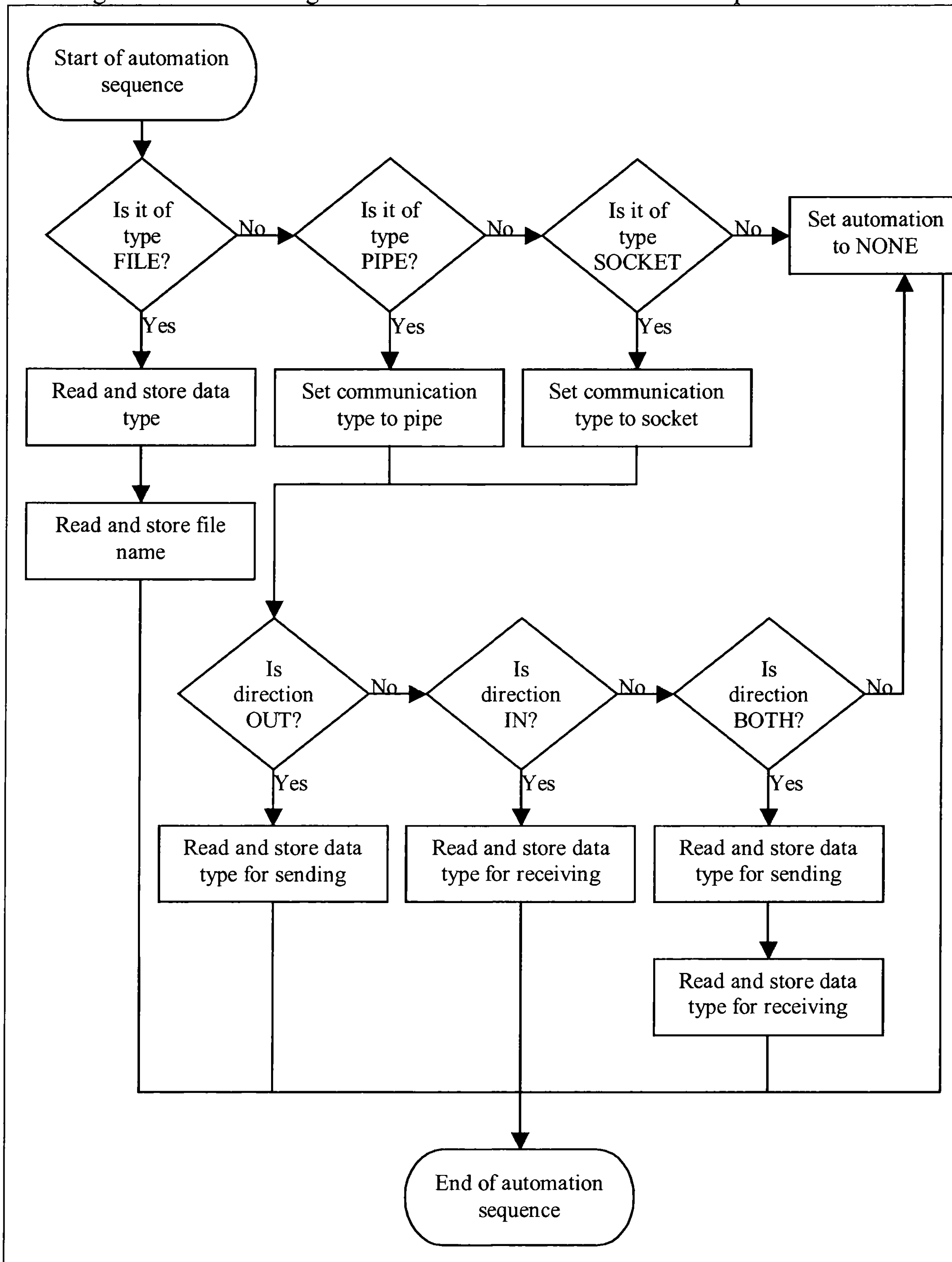


Figure 5-6 - Flow Diagram of the Automation Part of the Script File Parser

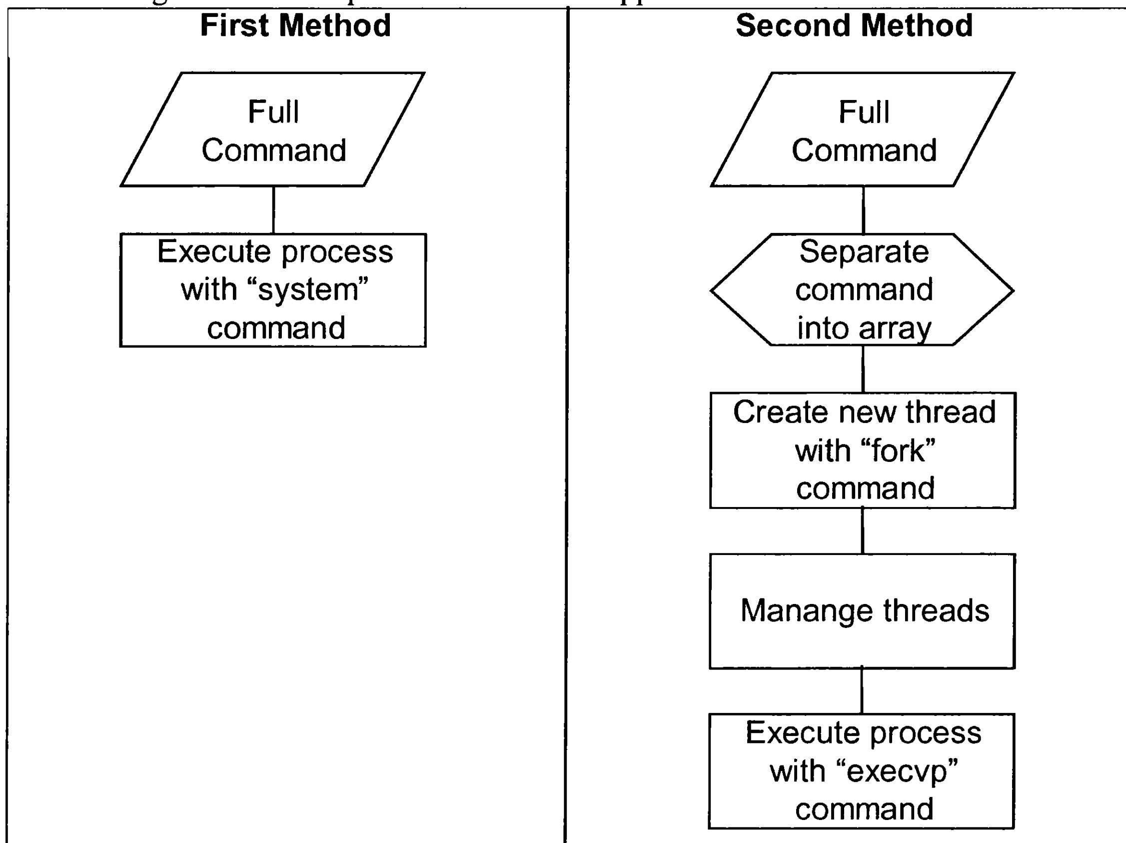


5.4.2 Application Initiation Procedures

As discussed in the previous section 5.3, the application initiation was developed in two stages. The first was a simple initiation of the application using the "system" command to execute the new process. This method has two major disadvantages that are (a) command line arguments for the new process are cumbersome to manage and (b) no system information is passed back to the PSUE to allow manipulation and communication. Even though this first method has these disadvantages it provided a fast and efficient method of initiating the processes, which is all that was needed at first.

The second stage of the application initiation development involved a number of other more complicated system facilities. Firstly, to allow the PSUE to know the identity of the new process it must create a new process thread (fork), where upon one process thread continues as the PSUE and the second process thread executes and therefore becomes the new application. The first thread stores the identity of the second thread that may then go on to be used for manipulation and communication. Another extension of the system facilities is the use of command line arguments. With the development of the file, pipe and socket data transfer methods, certain key information that is discussed in the next section must be passed to the new process. The easiest method of doing this is through the use of command line arguments, but the application may already be using command line arguments. To simplify the situation, the command line arguments of the application are broken down into an array that is then passed to a system facility (execvp) that can handle these arrays to execute the application. A comparison of the two methods is shown in Figure 5-7 below.

Figure 5-7 - Comparison of the Two Application Initiation Methods



5.4.3 Pipes and Sockets

Pipes and sockets were initially explained under section 4.2.1, and have some subtle differences. However, one of the key differences between them is the ability for sockets to work in an inter-platform capacity and that sockets transport data more efficiently.

If data is to be transferred in one direction only, then a pipe or socket works very well and causes very little inconvenience. Since pipes are unidirectional, if data is to be transferred in both directions then two pipes would have to be created for the process: one for sending data from the PSUE to the application and one for receiving data from the application to the PSUE. Since sockets are bi-directional, only a single socket connection needs to be made even if data is to be transferred in both directions.

In practise, when a pipe or socket connection is being used to transfer data to and from an application certain key information should always be transferred back, such as the total amount of data received. Therefore, if a pipe is to be used and data is only to be transferred in one direction, e.g. from the PSUE to the application, two pipes are still created. This is so that the data definitions may be transmitted in one direction and the other key information can be transmitted in the other direction. In other words, an application connected using the pipe system always has two pipes associated with it and an application connected using the socket system always has one socket associated with it. Figure 5-8 shows the connection procedure for an application using the pipe system and Figure 5-9 shows the connection procedure for an application using the socket system.

Figure 5-8 - Connection Procedure for the Pipe System

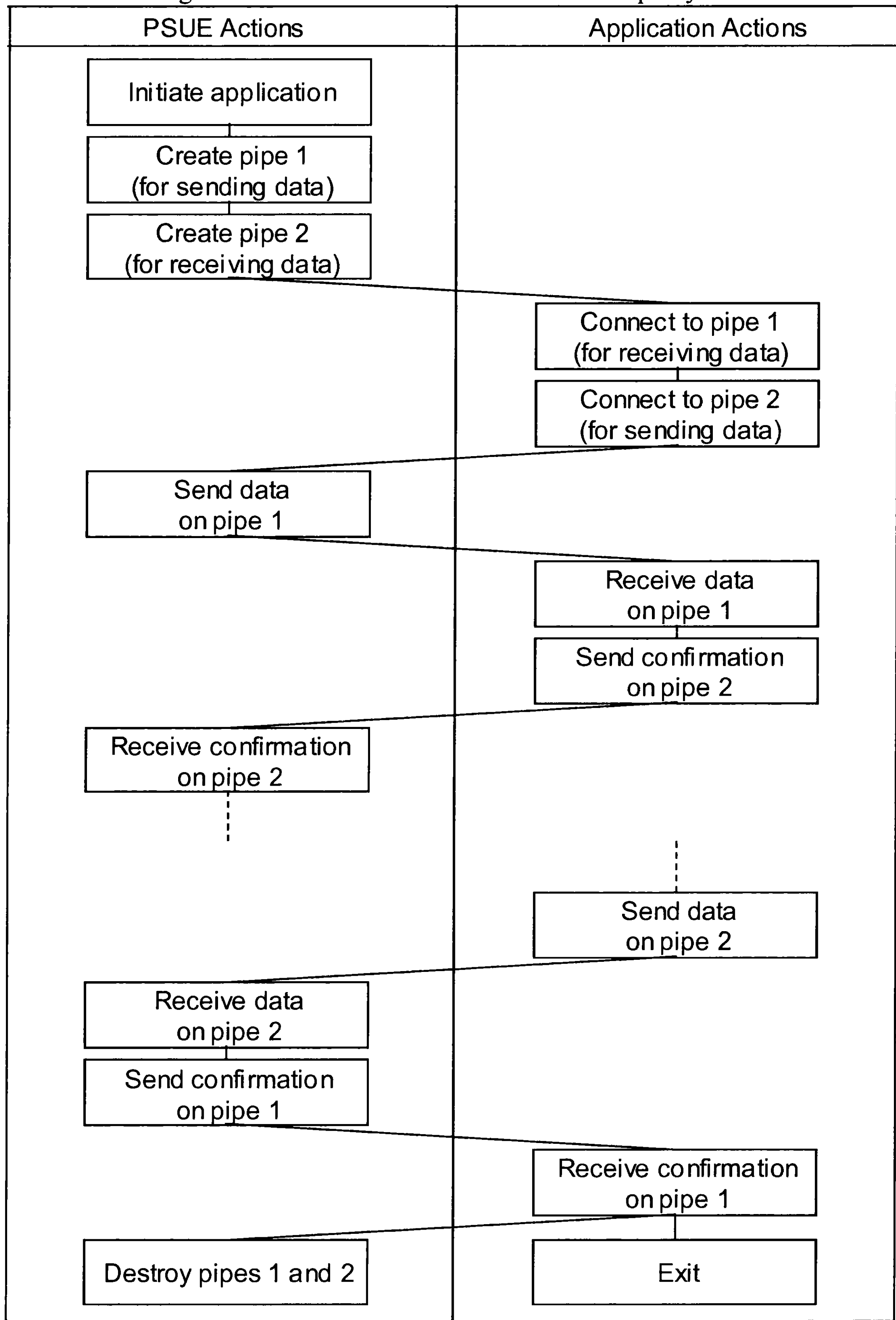
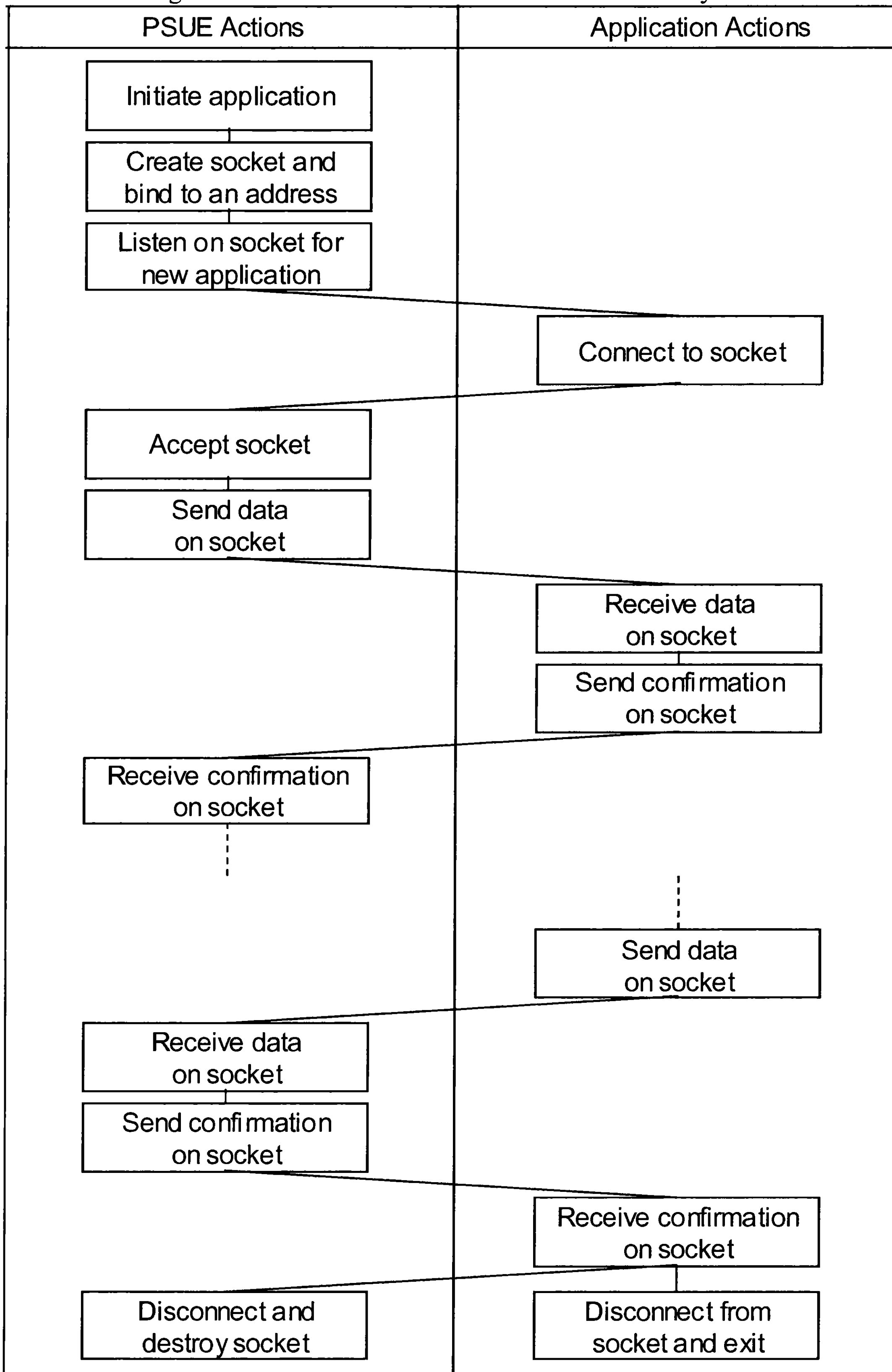


Figure 5-9 - Connection Procedure for the Socket System



5.5 Application Integration Library

When data is to be transferred using pipes or sockets the source code of the external application must be developed to incorporate the appropriate functionality. In order to assist the user with the integration of their application the application integration library was developed. The library provides the full range of routines, in both the FORTRAN and C languages, required to integrate an application into the PSUE when using pipes and sockets. The library contains utility routines to connect and disconnect pipes and sockets and also to send and receive data in various formats.

Once the source code developments have been made, recompilation of the application is required with the linking of the application integration library.

5.5.1 Connection and Disconnection with the PSUE

Since the FORTRAN and C languages use quite different methods of utilising command line arguments, separate routines had to be created for the two languages for this operation. The routines shown in Table 5-2 deal with the connection and disconnection of an application to the PSUE, using pipes and sockets.

Table 5-2 - Connection and Disconnection Routines

Routine	Description
ALFConnectPipe	FORTRAN routine to make a pipe connection to the PSUE.
ALCConnectPipe	C routine to a make a pipe connection to the PSUE.
ALFConnectSocket	FORTRAN routine to make a socket connection to the PSUE.
ALCConnectSocket	C routine to make a socket connection to the PSUE.
ALDisconnectPipe	Routine to disconnect a pipe connection (language independent)
ALDisconnectSocket	Routine to disconnect a socket connection (language independent)

5.5.2 Sending and Receiving Arbitrary Data

Once a pipe or socket connection has been made data may flow between the PSUE and the application. The core routines for transmitting the data back and forth are "read" and "write" routines, but these are rather cumbersome for the user to implement. Therefore some base sending and receiving routines, which can be seen in Table 5-3, were added to the library. All of these routines are language independent.

Table 5-3 - Arbitrary Send and Receive Routines

Routine	Description
ALReadData	Reads either an integer or real value from the given pipe or socket.
ALWriteData	Writes either an integer or real value to the given pipe or socket.
ALReadArray	Reads an array of either integers or real values from the given pipe or socket.
ALWriteArray	Writes an array of either integers or real values from the given pipe or socket.
ALEndRead	Sends confirmation of the data just received on the given pipe or socket.
ALEndWrite	Waits for confirmation of the data just sent on the given pipe or socket.

5.5.3 Extended Data Extraction Facilities

It was quickly found that the arbitrary data transfer routines described above still did not simplify the application integration enough. The next stage was the extended data extraction routines that would internally transfer all of the data definition and then provide access to the data storage as and when the user required it. Selections of these routines are shown in Table 5-4 and are again language independent.

Table 5-4 - A Selection of the Extended Data Extraction Routines

Routine	Description
ALRequestGridData	Receives the entire grid definition from the specified pipe or socket and stores all the data in a local data structure. Finally, it sends back confirmation to the PSUE.
ALGetAllNodes	Retrieves nodal ids and coordinates from the local data structure.
ALGetAllCells	Retrieves cell ids and connectivities from

	the local data structure of all cells with the specified cell type.
ALGetNodeData	Retrieves boundary conditions, material and property values from the local data structure.
ALGetCellData	Retrieves boundary conditions, material and property values from the local data structure of all cells with the specified cell type.

5.6 Full Example

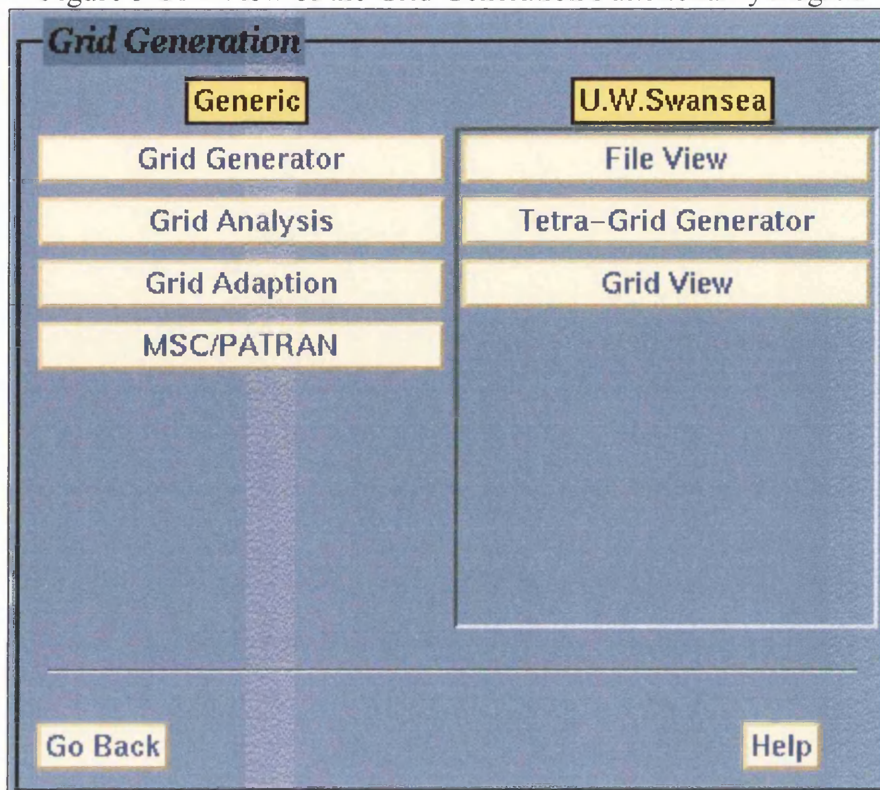
Within this section a full description of integrating an application into the PSUE will be given. This imaginary application will be a tetrahedral grid generator that will need the triangular boundary grid definition from the PSUE and, after generating an unstructured tetrahedral grid, it will need to transfer the full grid definition back to the PSUE before exiting.

The first stage will be to edit the script file for the relevant functionality region and add the appropriate entries for the grid generator, which will be named "tetragrid". Since there is a grid functionality region it makes sense to add the application to that particular area. Assuming that other applications are already in the script file, the final file listing would be as follows:

```
U.W.Swansea
Menu Main
{
File View
/usr/people/jo/bin/fileview
0
Tetra-Grid Generator
/usr/people/jo/bin/tetragrid
0
Grid View
/usr/people/jo/bin/gridview
0
}
```

Initially there is no automation sequence but as will be seen later in this section, the automation sequence will be added in. Figure 5-10 shows the view of the grid generation functionality region of the PSUE after loading the script file listed above.

Figure 5-10 - View of the Grid Generation Functionality Region



The next stage of integration is to plan what source code has to be developed within the existing source code of the "tetragrid" application. Generally, we will need to:

- Connect to the PSUE.
- Receive all of the boundary grid definition from the PSUE.
- Confirm receipt of the data with the PSUE.
- Execute the original tetragrid source code – i.e. generate the grid
- Send all of the new tetrahedral grid definition to the PSUE.
- Confirm that the PSUE received all the data.
- Exit.

This example will assume that the tetragrid application was initially written in the C language and that the communication method will utilise the socket system. Figure 5-11 shows the very simple source code of the original main routine of the tetragrid

application and Figure 5-12 shows the source code of the final main routine. Static arrays of memory have been used to simplify the overall source code.

Figure 5-11 - Original Main Routine of the Tetragrid Application

```
int main ( int argc, char **argv )
{
   float coords[100000][3];
   int connectivity[500000][4];
   int boundary[100000][3];
   int numPts, numTri, numTet;

   LoadData ( numPts, numTri, coords, boundary );
   TetraGrid ( numPts, numTri, numTet, coords, boundary, connectivity );
   SavaData ( numPts, numTri, numTet, coords, boundary, connectivity );

   exit;
}
```

Figure 5-12 - Final Main Routine of the Tetragrid Application

```
int main ( int argc, char **argv )
{
   float coords[100000][3];
   int connectivity[500000][4];
   int boundary[100000][3];
   int numPts, numTri, numTet;
   int connectedToPSUE, count, num;

   if ( connectedToPSUE = ALCConnectSocket ( argc, argv ) )
   {
      ALReadData ( &numPts, 'int' );
      ALReadData ( &numTri, 'int' );
      for ( count = 0; count < numPts; count++ )
      {
         ALReadArray ( coords[count], 'real', 3 );
      }
      for ( count = 0; count < numTri; count++ )
      {
         ALReadData ( &num, 'int' );
         ALReadArray ( boundary[count], 'int', num );
      }
      ALEndRead();
   } else LoadData ( numPts, numTri, coords, boundary );

   TetraGrid ( numPts, numTri, numTet, coords, boundary, connectivity );

   if ( connectedToPSUE )
   {
      ALWriteData ( numPts, 'int' );
      ALWriteData ( numTet, 'int' );
      ALWriteData ( numTri, 'int' );
      for ( count = 0; count < numPts; count++ )
      {
         ALWriteArray ( coords[count], 'real', 3 );
      }
      for ( count = 0; count < numTet; count++ )
      {
         ALWriteData ( 4, 'int' );
         ALWriteArray ( connectivity[count], 'int', 4 );
      }
      for ( count = 0; count < numTri; count++ )
      {
         ALWriteData ( 3, 'int' );
         ALWriteArray ( boundary[count], 'int', 3 );
      }
      ALEndWrite();
      ALDisconnectSocket();
   }
   else SavaData ( numPts, numTri, numTet, coords, boundary, connectivity );

   exit;
}
```

With the source code as above the application is ready to be integrated into the PSUE. Upon initialisation of the PSUE, the new application button will appear in the functionality region as previously shown in figure 5.6 and when activated will open the application initiation panel. Each time the user wishes to use the tetragrid application they will have to specify in the panel: a socket connection, that outward-bound data is of the boundary data type, that inward-bound data is of the grid data type and then select "Proceed". Since this would become quite tedious and some users may even make mistakes, it makes sense to automate this connection using the script files. Since the connection process will always be the same, e.g. there are no filenames that may change; no complications arise when automating this connection. The typical changes to the script file to provide an automated connection would produce a script as follows:

```
U.W.Swansea
Menu Main
{
File View
/usr/people/jo/bin/fileview
0
Tetra-Grid Generator
/usr/people/jo/bin/tetragrid
1
SOCKET
BOTH
BOUNDARY
GRID
Grid View
/usr/people/jo/bin/gridview
0
}
```

The application, tetragrid, has now been fully integrated into the PSUE and the whole process has been very quick and easy to implement. Unfortunately, a major disadvantage with the application integration system while using pipes and sockets is that it may not be able to fully utilise the integration system for applications with which the user does not have the source code readily available.

5.7 Summary

The application integration system allows the user to integrate an arbitrary application into the PSUE in a simple but powerful way. After an application has been "plugged in", it may be initiated any number of times by a simple click on its application button. The application buttons are grouped into regions of particular task types and can be set-up to utilise cascading menus that improve access to large numbers of applications.

The communication of data can use a number of different methods or the user may choose not to send any data at all, for example, when a system utility such as a calculator is to be initiated. The methods of communication include file transfer, dual pipes that are unidirectional or a socket that is bi-directional. The data may be sent just to the newly initiated application, or just from the application when it finishes or both. Different types of data may be transferred to and from the application and include geometry, surface mesh, volume mesh and solution data. The whole communication and initiation processes may be automated within the controlling script files.

The application integration system provides a highly versatile method of incorporating applications into the PSUE environment as seamlessly as possible. The partners of the CAESAR project all found the application integration facilities very useful and were able to gather their different engineering simulation tools together into the PSUE with great results.

6 APPLICATION TOOL WRAPPER

6.1 Introduction

The Application Tool Wrapper is the next generation tool for the integration of applications into the PSUE environment. It gives the user the ability to "wrap" any application with the capacity to transfer almost arbitrary data back and forth to the PSUE. It was developed under the JULIUS project.

We have seen that the application integration system allows great flexibility for integrating arbitrary applications, however it does have its limitations. The Application Tool Wrapper builds on the application integration facilities by allowing arbitrary data to be communicated to the application. It also allows for the transfer of data to proprietary applications for which source code is unavailable for adding application integration library code.

The Application Tool Wrapper was initially intended to replace the current Application Integration system, however, it was quickly noted that both facilities are very powerful tools with slightly differing capabilities. It was therefore decided to keep both facilities within the PSUE and have them run side by side. This gives the user the maximum amount of flexibility when deciding how to integrate their application.

During the initial design of the Application Tool Wrapper it was decided that the facility would maximise use of the previous technology employed by the PSUE. The PSUE communication library, used for internal modules, uses advanced communications and a complex data structure but gives far more control over the data. On the other hand, the application integration system has relatively simple communications and straightforward sets of data. The Application Tool Wrapper was anticipated to amalgamate the two methods providing all of the functionality of the internal data structure through relatively simple communication systems.

It should be noted at this stage that the Application Tool Wrapper that is described within this thesis is an early experimental prototype. The communications between

the actual Tool Wrapper and the application are very basic within this early version; however, it has been fully integrated into the PSUE communication library. Many other features that were intended to be included within the Tool Wrapper had not been completed at the time when the author's work finished with the module.

6.2 Architecture

The Application Tool Wrapper was developed as another module of the PSUE so that it could benefit from the entire data structure of the PSUE. It can be seen as another module, for example, like the grid generation module but instead of generating grids, its purpose is to communicate data back and forth to external applications. Since the Tool Wrapper is a module of the PSUE it utilises the communication library that represents efficient and comprehensive transfer of data to the module.

In a way, it may be seen as an amalgamation of the PSUE Communication Library and the Application Integration system. The communication library allows the efficient but comprehensive data sets to be transferred to the Tool Wrapper and the Application Integration side allows the initialisation and communication to new arbitrary applications. This concept follows the original architecture plans for the PSUE that all key functionality is provided in a modular way and drawn together into a unified environment.

As discussed in section 4, the communication library utilises shared memory as its core method for transferring data from one module to another, by way of the PSUE main interface. Since the tool wrapper is always initiated from the PSUE there is no overhead incurred by requiring the data to go through the PSUE main interface. It was intended that from the tool wrapper the data might be transferred using any of the original application integration methods; file, pipe and socket, however, at the time that this thesis was written only the file method had been implemented.

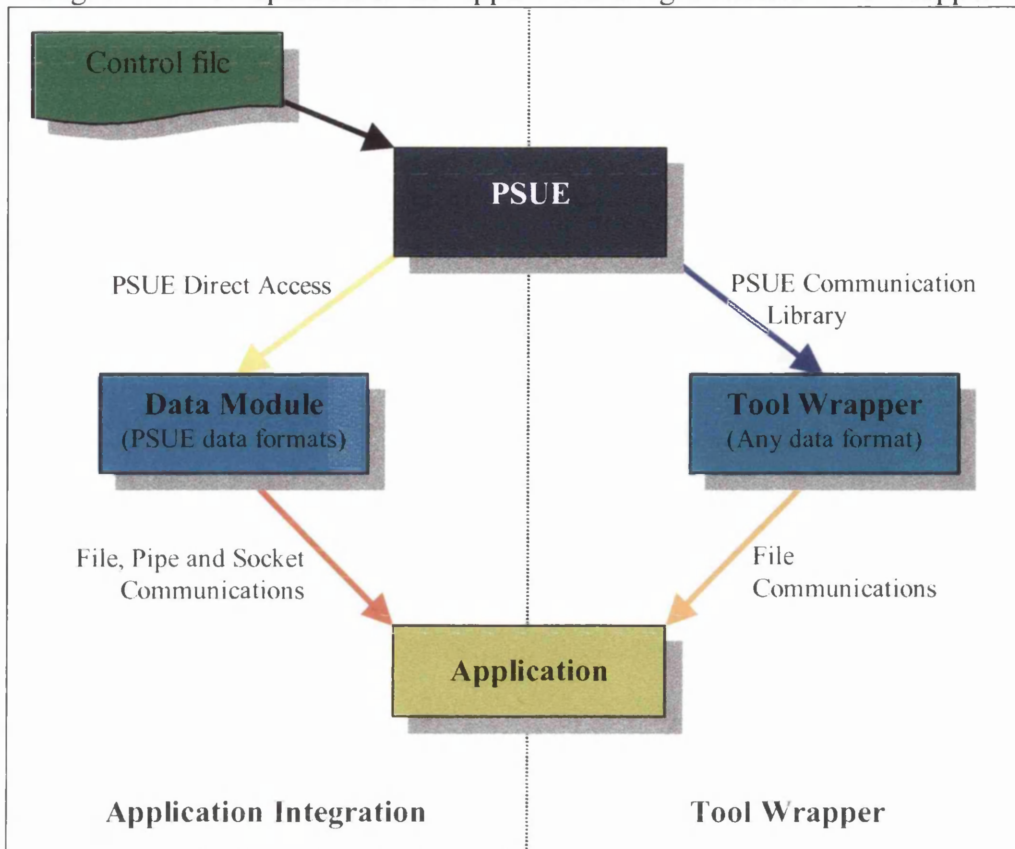
As previously mentioned above, the tool wrapper may be seen as a method to integrate applications and runs side by side with the application integration method. Long term, it is intended that the tool wrapper will replace the application integration system, but this will only happen when all of the useful functionality of the application integration system is implemented into the tool wrapper facility. Table 6-1 shows the current advantages and disadvantages of the two methods.

Table 6-1 - Advantages and Disadvantages of the Application Integration and the Tool Wrapper

Facility	Advantages	Disadvantages
Application Integration	Uses file, pipe and socket communications. May be automated.	Requires access to source code. Can only communicate using PSUE data formats.
Tool Wrapper	Does not require access to source code. Can communicate using arbitrary data formats.	Only uses files for communications. No automation.

As we can see from Table 6-1, there are currently some key issues differentiating the two methods of the application integration system and the tool wrapper - so when should the user implement which method? The application integration method allows the use of pipes and sockets that are far more efficient in transporting data than files, however the user is restricted to PSUE identified data formats and requires access to source code. Therefore, if the user has access to the source code of the application they wish to integrate and the PSUE data formats include the required data for the application, the user would be advised to utilise the application integration system. On the other hand if the source code is not available or the PSUE data formats do not provide an adequate data structure, the tool wrapper would need to be used. Figure 6-1 shows the comparison between the application integration system and the tool wrapper.

Figure 6-1 - Comparison of the Application Integration and Tool Wrapper



6.2.1 Techniques Employed

The general techniques employed such as the application initiation and data transfer have been described previously in section 5.4 since it was mentioned that existing technology would be used where possible. The major new technique employed by the tool wrapper is the scripting language used to control what data is sent through the communication channel.

This technique has been implemented using two separate procedures, which are the specification procedure and the interpretation procedure. The specification of the data to be transferred is carried out by selecting the required key data from a list, provided in the left-hand side of the application. As the user selects key items from the list they are automatically added to the "work area" on the right-hand side. The key items are

recognised command strings provided by the interface and therefore may be easily interpreted in the second procedure. The key items work in conjunction with a small number of key commands that are currently just "for" and "endfor". It is anticipated that far more key commands would be implemented such as: "if", "else", "endif", "while".

The key items are a series of identifies linked with each and every data item within the PSUE data structure. Table 6-2 shows an example set of the key items and their interpretation.

Table 6-2 - Example Set of Key Tool Wrapper Items

Key Item	Interpretation
TEXT()	This allows specification of any text or static data
INDEX	Index or entity number, e.g. node number
NUMBER_OF_NODES	The number of nodes
NUMBER_OF_CELLS	The number of cells
CONN_1	Connectivity node 1 for current cell
X_COORD	X coordinate for current node
X_U_TANGENT	X derivative in U direction for current surface
Y_V_TANGENT	Y derivative in V direction for current surface
Z_UV_TANGENT	Z second derivative for current surface
SURFACE_INDEX	Surface index for current entity
MATERIAL_INDEX	Material index for current entity
SOLUTION_NODE_2	Solution variable 2 for current node
SOLUTION_CELL_3	Solution variable 3 for current cell

The key items and key commands are used to build the data format required by the users application. It is kept as simple as possible and most of the data structure may be created by pointing and clicking on the specific key items. As a simple example, assume an application requires the number of nodes in a mesh and the corresponding x, y and z coordinates for each node.

The key structure would be:

```
NUMBER_OF_NODES
for NUMBER_OF_NODES
    X_COORD Y_COORD Z_COORD
endfor
```

However, if the nodes were not in order the index identity may also need to be sent to the application so that it may order them appropriately. The following key structure would be required:

```
NUMBER_OF_NODES
for NUMBER_OF_NODES
    INDEX X_COORD Y_COORD Z_COORD
endfor
```

The tool wrapper uses a number of key items for more than one use and one of these is the INDEX key item. The tool wrapper interprets each key item in the manner that it believes is right and so in the above structure it would send the index of the current node through the communication channel. However if the line "for NUMBER_OF_NODES" were actually "for NUMBER_OF_CELLS" it would assume that the index of the current cell is the intended data to be sent. This is the interpretation procedure.

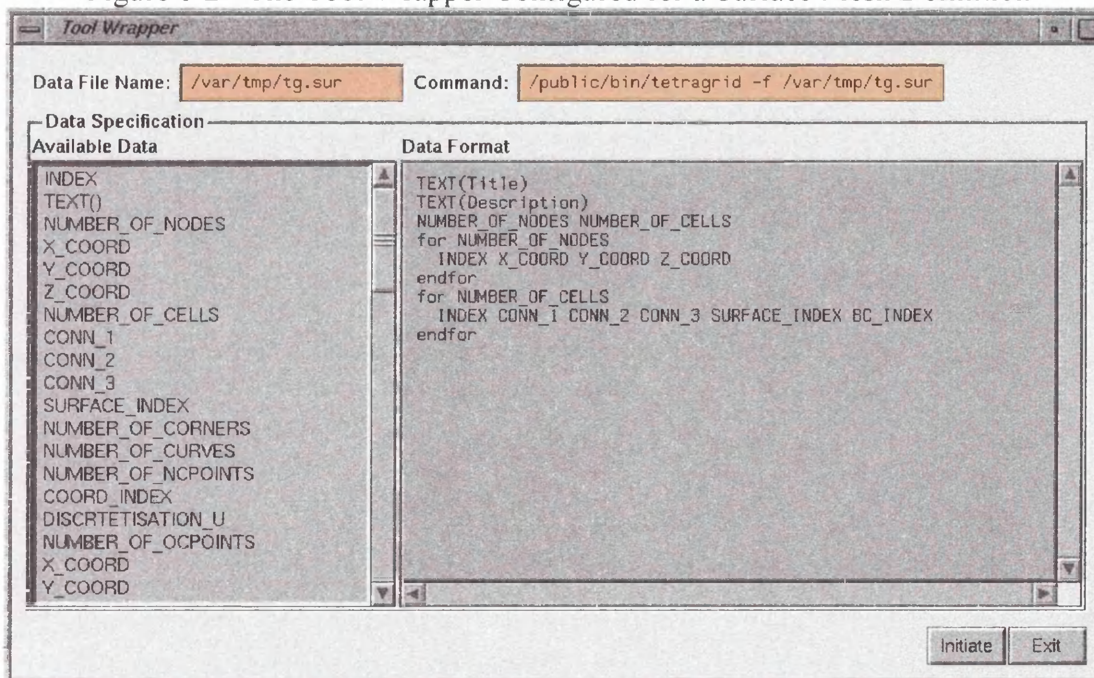
6.3 Examples

In order to help understand the procedures involved within the tool wrapper two examples are presented. The first represents the communication of a simple surface grid definition and the second represents a complex geometrical definition.

6.3.1 Example 1 – Surface Grid Definition

Figure 6-2 shows the tool wrapper interface with the key item list on the left-hand side and the work area on the right-hand side. The user specifies the data file name that will be used to transfer the data that in this case is named "/var/tmp/tg.sur". The command string is also specified, as "/public/bin/tetragrid –f /var/tmp/tg.sur" that allows the application, tetragrid, to automatically load the file since the application uses the "-f" option to do this.

Figure 6-2 - The Tool Wrapper Configured for a Surface Mesh Definition



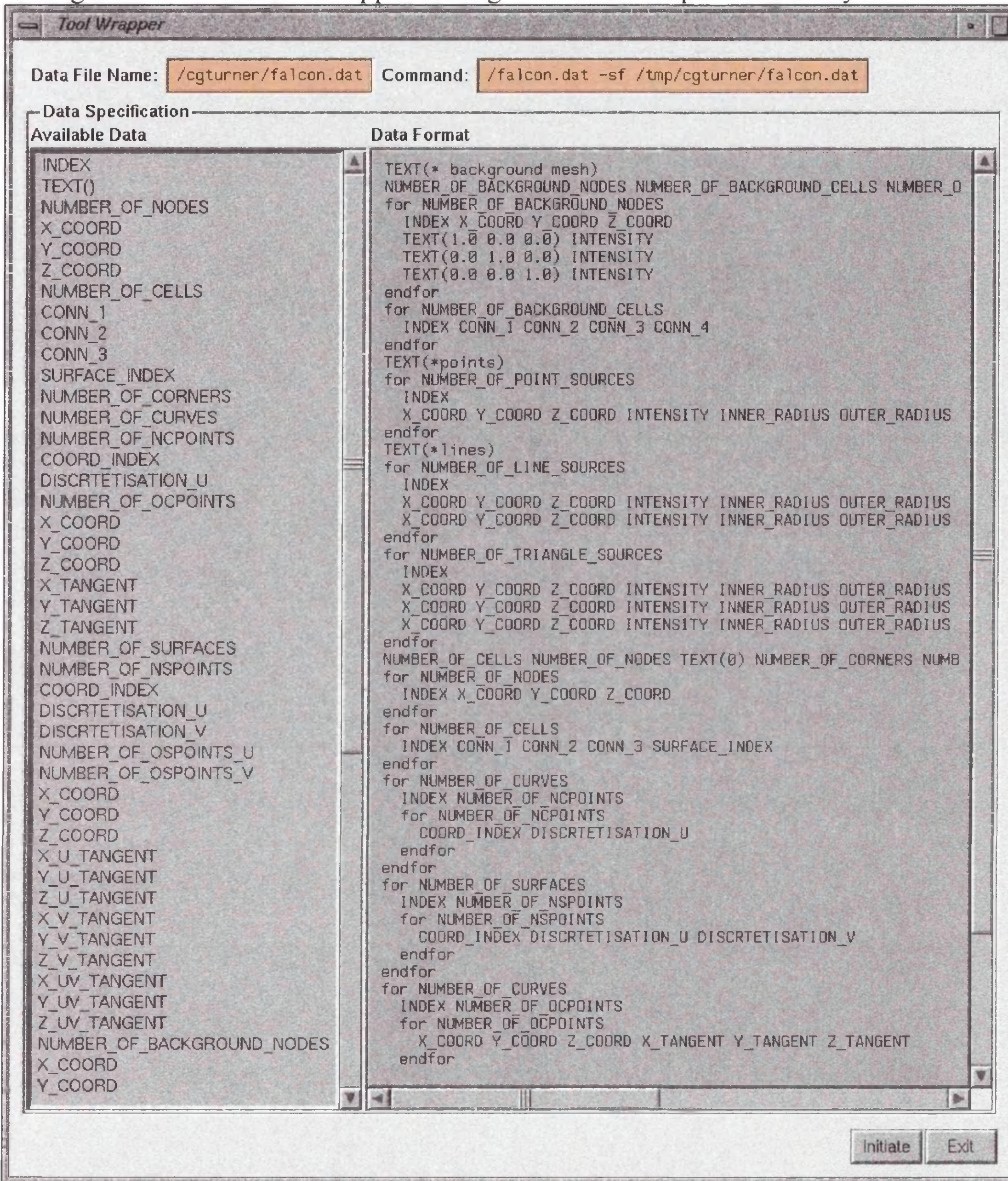
The key structure is shown below in blue along with a description of what each line represents:

```
TEXT(Title)
            This provides a line of text that would print "Title"
TEXT(Description)
            This provides a line of text that would print "Description"
NUMBER_OF_NODES NUMBER_OF_CELLS
            This provides the number of nodes and faces within the surface mesh
for NUMBER_OF_NODES
  INDEX X_COORD Y_COORD Z_COORD
endfor
            This enters a loop of size equal to the number of nodes, and for each of
            those nodes the node index, x, y and z coordinates are provided.
for NUMBER_OF_CELLS
  INDEX CONN_1 CONN_2 CONN_3 SURFACE_INDEX BC_INDEX
endfor
            This enters a loop of size equal to the number of cells, and for each of
            those cells the cell index, 3 connectivities, surface index, and boundary
            condition index are provided.
```

6.3.2 Example 2 – Complex Geometry Definition

Figure 6-3 shows the tool wrapper interface again but with a more complex set of key structures. The user specifies the data file name that will be used to transfer the data and the application to be initiated once again. This example shows some further functionality that the application integration system is not capable of. The key structures contain not only the geometrical definition but also the background mesh density grid and grid sources. If the application integration system was being utilised only one of the two data sets could be transferred or the neutral file format would have to be used, possibly including a lot of extra data that would not be necessary.

Figure 6-3 - The Tool Wrapper Configured for a Complex Geometry Definition



The key structure is shown below in blue along with a description of what each line represents:

```
TEXT(* backgound mesh)
```

This provides a line of text that would print "* background mesh"

```
NUMBER_OF_BACKGROUND_NODES...
 NUMBER_OF_BACKGROUND_CELLS...
  NUMBER_OF_POINT_SOURCES...
   NUMBER_OF_LINE_SOURCES...
    NUMBER_OF_TRIANGLE_SOURCES
```

This provides the number of nodes and cells in the background mesh density description and the number of point, line and triangle grid sources

```
for NUMBER_OF_BACKGROUND_NODES
  INDEX X_COORD Y_COORD Z_COORD
  TEXT(1.0 0.0 0.0) INTENSITY
  TEXT(0.0 1.0 0.0) INTENSITY
  TEXT(0.0 0.0 1.0) INTENSITY
endfor
```

This enters a loop of size equal to the number of background nodes, and for each of those background nodes the node index, x, y and z coordinates are provided. These are then followed by the given text strings and the intensity value at the nodal position.

```
for NUMBER_OF_BACKGROUND_CELLS
  INDEX CONN_1 CONN_2 CONN_3 CONN_4
endfor
```

This enters a loop of size equal to the number of background cells, and for each of those background cells the cell index and 3 connectivities are provided.

```
TEXT(* points)
```

This provides a line of text that would print "* points"

```
for NUMBER_OF_POINT_SOURCES
  INDEX
  X_COORD Y_COORD Z_COORD INTENSITY INNER_RADIUS  OUTER_RADIUS
endfor
```

This enters a loop of size equal to the number of point sources, and for each of those point sources the index, x, y and z coordinates are provided. These are then followed by the intensity value and the inner and outer radii of the point source.

```
TEXT(* lines)
```

This provides a line of text that would print "* lines"

```
for NUMBER_OF_LINE_SOURCES
   INDEX
   X_COORD Y_COORD Z_COORD INTENSITY INNER_RADIUS  OUTER_RADIUS
   X_COORD Y_COORD Z_COORD INTENSITY INNER_RADIUS  OUTER_RADIUS
endfor
```

This enters a loop of size equal to the number of line sources, and for each of those line sources the index is provided. Followed by the x, y and z coordinates, the intensity value and the inner and outer radii of the point sources at either end of the line source.

```
TEXT(* triangles)
```

This provides a line of text that would print "* triangles"

```
for NUMBER_OF_TRIANGLE_SOURCES
   INDEX
   X_COORD Y_COORD Z_COORD INTENSITY INNER_RADIUS  OUTER_RADIUS
   X_COORD Y_COORD Z_COORD INTENSITY INNER_RADIUS  OUTER_RADIUS
   X_COORD Y_COORD Z_COORD INTENSITY INNER_RADIUS  OUTER_RADIUS
endfor
```

This enters a loop of size equal to the number of triangle sources, and for each of those triangle sources the index is provided. Followed by the x, y and z coordinates, the intensity value and the inner and outer radii of the point sources at all three ends of the triangle source.

```
NUMBER_OF_CURVES NUMBER_OF_SURFACES
```

This provides the number of curves and surfaces within the geometry definition.

```
for NUMBER_OF_CURVES
   INDEX NUMBER_OF_NCPOINTS
   for NUMBER_OF_NCPOINTS
      COORD_INDEX DISCRETISATION_U
   endfor
endfor
```

This enters a loop of size equal to the number of curves and for each of those curves the index and the number of new curve points is provided. A second loop is entered of size equal to the number of new curve points for the current curve. Within this loop the coordinate index and the discretised distance in U along the curve is provided for the current point on the current curve.

```
for NUMBER_OF_SURFACES
  INDEX NUMBER_OF_NSPOINTS
  for NUMBER_OF_NSPOINTS
    COORD_INDEX DISCRETISATION_U DISCRETISATION_V
  endfor
endfor
```

This enters a loop of size equal to the number of surfaces and for each of those surfaces the index and the number of new surface points is provided. A second loop is entered of size equal to the number of new surface points for the current surface. Within this loop the coordinate index and the discretised distance in U and V across the surface is provided for the current point on the current surface.

```
for NUMBER_OF_CURVES
  INDEX NUMBER_OF_OCPOINTS
  for NUMBER_OF_OCPOINTS
    X_COORD Y_COORD Z_COORD
    X_TANGENT Y_TANGENT Z_TANGENT
  endfor
endfor
```

This enters a loop of size equal to the number of curves and for each of those curves the index and the number of old curve points is provided. A second loop is entered of size equal to the number of old curve points for the current curve. Within this loop the x, y and z coordinates and the x, y and z tangents in U are provided for the current point on the current curve.

```
for NUMBER_OF_SURFACES
  INDEX NUMBER_OF_OSPOINTS
  for NUMBER_OF_OSPOINTS
    X_COORD Y_COORD Z_COORD
    X_U_TANGENT Y_U_TANGENT Z_U_TANGENT
    X_V_TANGENT Y_V_TANGENT Z_V_TANGENT
    X_UV_TANGENT Y_UV_TANGENT Z_UV_TANGENT
  endfor
endfor
```

This enters a loop of size equal to the number of surfaces and for each of those surfaces the index and the number of old surface points is provided. A second loop is entered of size equal to the number of old surface points for the current surface. Within this loop the x, y and z coordinates, the x, y and z tangents in U, the x, y, z tangents in V and the x, y and z tangents in UV are provided for the current point on the current surface.

6.4 Summary

The overall design of the tool wrapper produces a highly versatile and configurable tool. It allows the PSUE the ability to integrate any application and to communicate key data between the PSUE and the application. The data format is very flexible and is specified by the user upon initiation of the application.

As mentioned, the tool wrapper is an early prototype designed for the JULIUS project but has already proved that the concept is feasible and would be very powerful once all of the planned developments have been made to the facility. These include further expansion of the data structures to include arbitrary entities that would allow for almost all eventualities. The other major planned developments include bringing the tool wrapper communication methods up to the same level as the application integration system. One further development that has not been mentioned so far is the implementation of data retrieval from the application back through the tool wrapper to the PSUE.

As has just been outlined, there is still a huge amount of required development work on the tool wrapper and until this is done, it would be expected that the application integration system would remain the key tool to use when integrating applications into the PSUE.

7 TEST CASES

7.1 Introduction

This section represents the results of the development of the PSUE. Since the development techniques and philosophy of the PSUE are new it is very difficult to make a direct comparison of the PSUE with any other system. The main method of evaluating whether the PSUE is a success or not is to examine the benefits of the system when carrying out test cases. Obviously, the target will be to meet the objectives of the PSUE as discussed in section 1.3.

The main test case that will be presented within this thesis utilises the Dassault Falcon aircraft geometry and will include all the steps required to reach a solution of the airflow around the aircraft.

Two further test cases will be presented but due to the confidentiality of the geometries, illustrations will not be shown. Both cases are aircraft configurations, the first is the solution of an electromagnetic solver on the A3XX airline jet and the second is the solution of the airflow around the F16 fighter jet. However, it must be highlighted that any arbitrary geometry and any multidisciplinary engineering application may be used. For example, the PSUE has been used to model the Thrust Super Sonic Car (SSC), an electrical transformer and power station cooling towers amongst many other applications.

The test cases described in this chapter cover all the procedures involved with carrying out an engineering simulation and therefore include descriptions of some functionality that was not developed by the author. This functionality has been included in the test case descriptions to provide a complete picture of how the PSUE operates overall.

7.2 Test Case 1 – Dassault Falcon

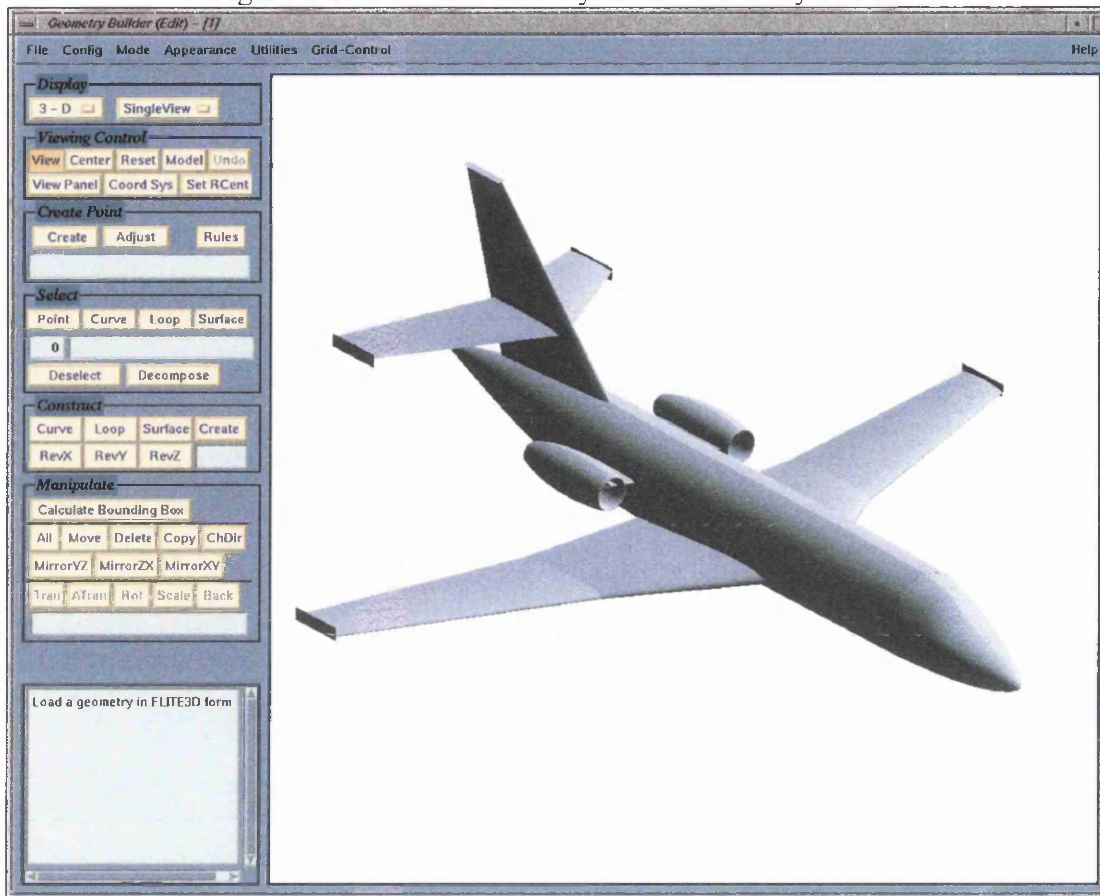
The objective for this first test case is to take the geometry of the Dassault Falcon, generate a tetrahedral grid, partition the grid, run a 3D flow solver with the configuration to convergence, adapt the computational grid and calculate the final results.

7.2.1 Geometry Manipulation

The original data for this case was provided as geometrical entities in the Flite3D geometry file format. Since this was employed as the base geometry format within the PSUE, as specified in the user manual [21], no geometry conversion is necessary and the file may be loaded into the PSUE directly. Figure 7-1 shows the Falcon geometry loaded in the geometry builder module of the PSUE.

As can be seen in the illustration, the geometry requires some additional parts, namely, a symmetry plane and a far field. This can be generated within the geometry builder [21] extremely easily with the template panel. A plane is created for the symmetry plane and a hemisphere for the far field.

Figure 7-1 - Falcon Geometry in the Geometry Builder



7.2.2 Preparation for Grid Generation

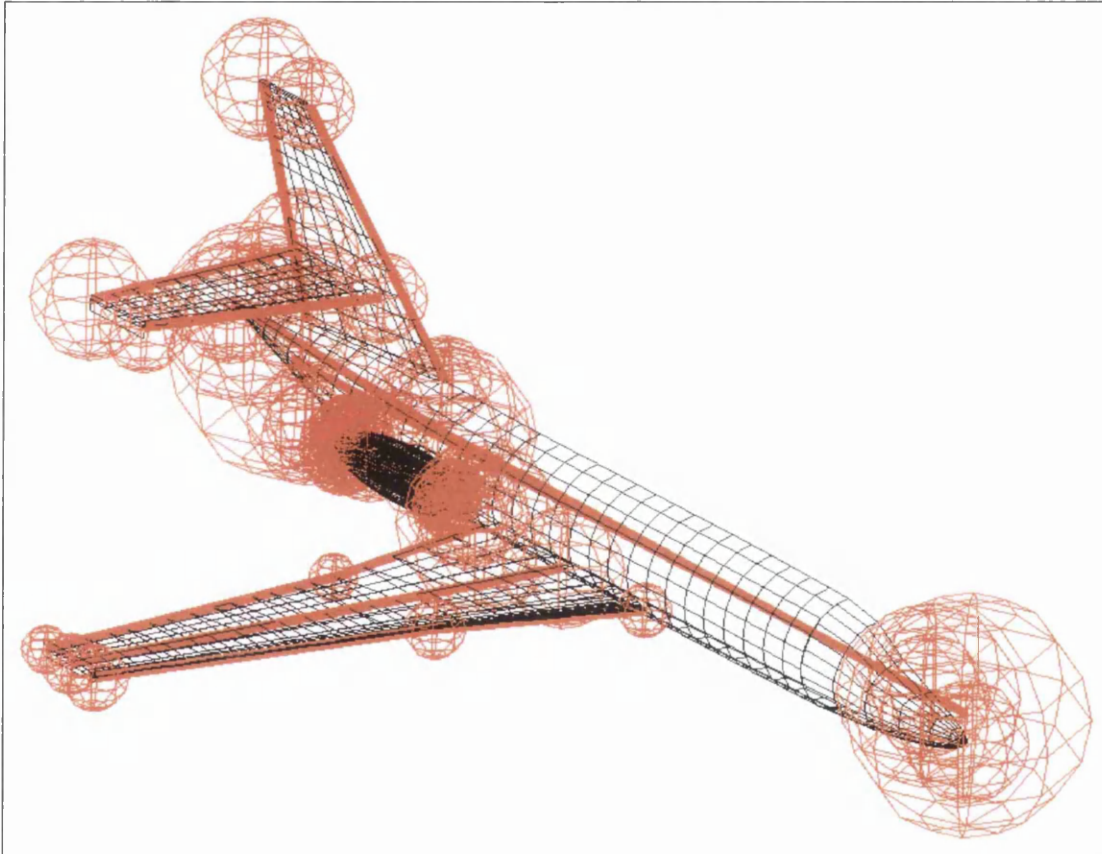
There are three stages for preparing the geometry for grid generation and these are: (a) checking the topology of the geometry, (b) setting up the boundary conditions and (c) setting up the controls for grid density.

An automatic topology generator is provided within the geometry builder that will generate and check the topology. If any problems occur the user will need to manually correct the geometry and/or the topology.

A boundary condition panel is provided to set values at the node level, line or curve level, or surface level. Using this panel, boundary conditions of grid nodes and cells may also be modified after the generation of a grid.

The final stage of the geometry preparation step is the specification of grid densities around the geometry. The grid generator [21] that has been provided as generic functionality within the PSUE is the FLITE3D system [21],[26] and uses the idea of sources to control the density of cells in specific regions. Sources come in three types, point, line and triangular. The grid sources can be created within the geometry builder and Figure 7-2 shows the final geometrical configuration with the grid sources.

Figure 7-2 - Falcon Geometrical Configuration with Grid Sources



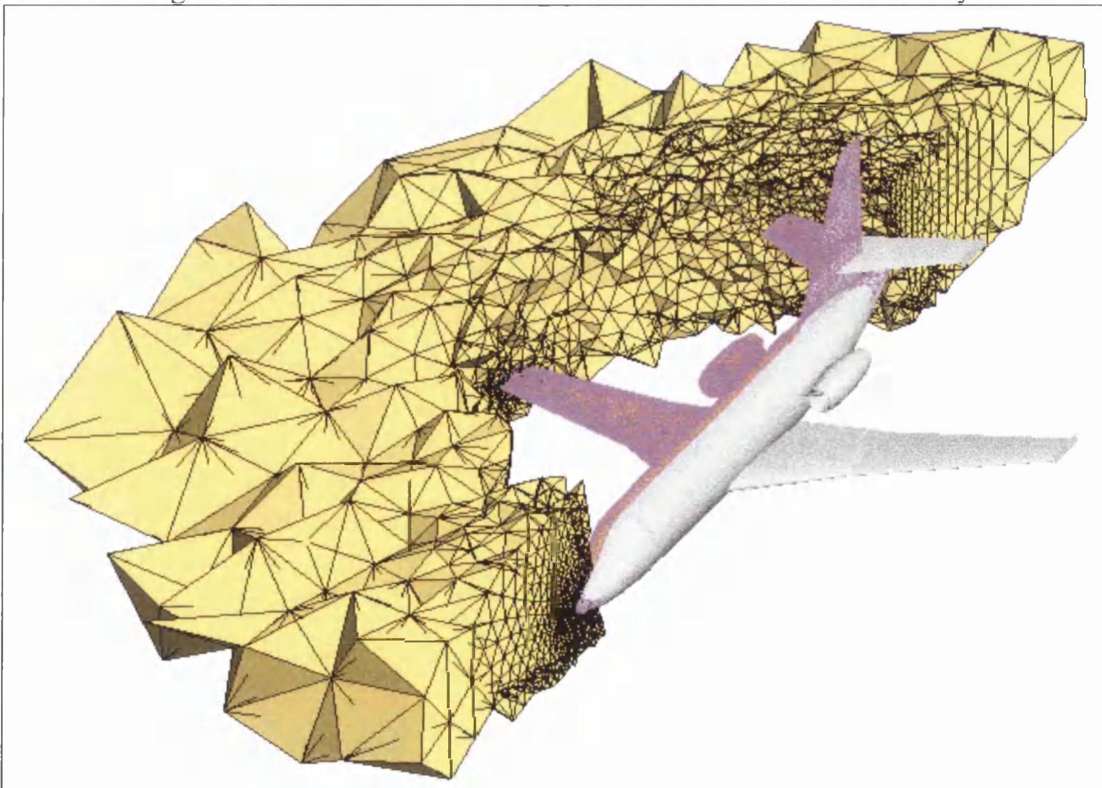
7.2.3 Grid Generation and Partitioning

All of the data has now been prepared ready for the grid generation step but the data is in the geometry builder module, not the grid generation module. The user would now exit the geometry builder that would automatically and transparently invoke the PSUE

communication library to pass all of the data definitions back to the main interface of the PSUE. The user now starts the grid generation interface that again automatically and transparently receives the data definitions held within the main PSUE interface. Hence, the user has changed modules without having to undertake any transferral of data themselves providing an almost seamless connection of modules.

Within the grid generation interface [21], since all data is already available, the user may immediately start the automatic surface grid generator, followed by the automatic tetrahedral grid generator. After this has completed the resulting grid definition may be readily viewed within the geometry viewer module within the PSUE. Figure 7-3 shows the resulting surface and volume grids for the Falcon geometry – the purple grid shows the surface grid on the actual aircraft surfaces while the yellow grid shows cutting regions through the volume grid.

Figure 7-3 - Surface and Volume Grids for the Falcon Geometry



Now that the initial volume grid has been generated, the user can exit the grid generation module that once again will send all the data back to the main PSUE interface using the communication library.

The nest step in the simulation procedure is to partition the new volume grid, however, the PSUE does not provide any generic functionality to do this and so an external application must be integrated. The PSUE functionality region provides an area specifically for this task, namely the Domain Decomposition area, and the script file that controls this region is named "PSUEscript.dode". The script file is created and is edited so that its contents are:

```
Partitioners
Menu Main
{
Swansea Partn
/usr/people/jo/bin/partn
1
SOCKET
OUT
GRID
}
```

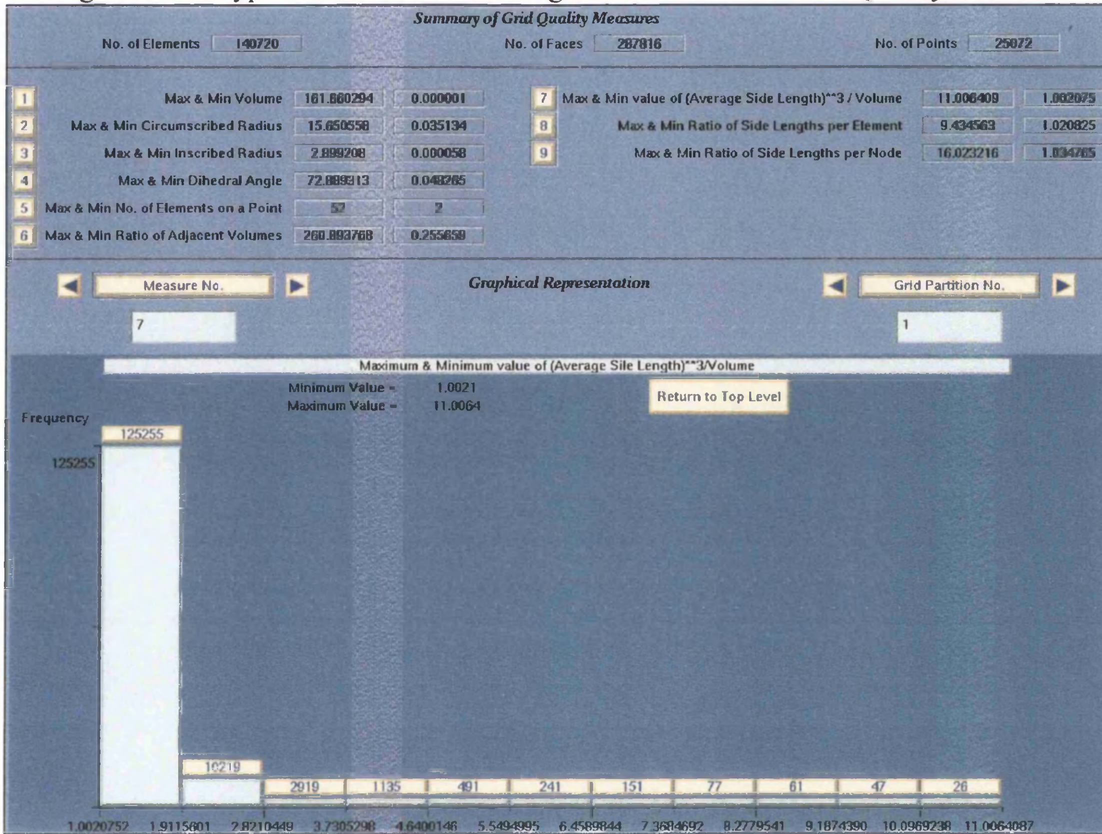
This will produce a single button that will be labelled "Swansea Partn" and will initiate the partitioning application "/usr/people/jo/bin/partn". The grid definition that was just generated within the PSUE grid generation module will be sent to the grid partitioner using a socket connection through the PSUE application integration library. The partitioner prompts for the number of domains to create and then continues to process the grid-partitioning algorithm. The application used is a research code written within UWS and is based on the Recursive Spectral Bisection method [29]. The output from the grid partitioner is a series of grid files – one for each domain.

7.2.4 Grid Quality Evaluation

Since each of the grid files is actually a computational grid in itself, they may be treated individually. Therefore, each may be sequentially loaded into the grid quality evaluation module [21] of the PSUE to check its quality statistics. Figure 7-4 shows the typical presentation of results for a grid within the grid quality evaluation module,

in the histogram mode. The user may move on to the next step of the simulation once they are satisfied with the quality of the grid and its partitions.

Figure 7-4 - Typical View of the Histogram Chart of the Grid Quality Statistics



7.2.5 Parallel Flow Solver

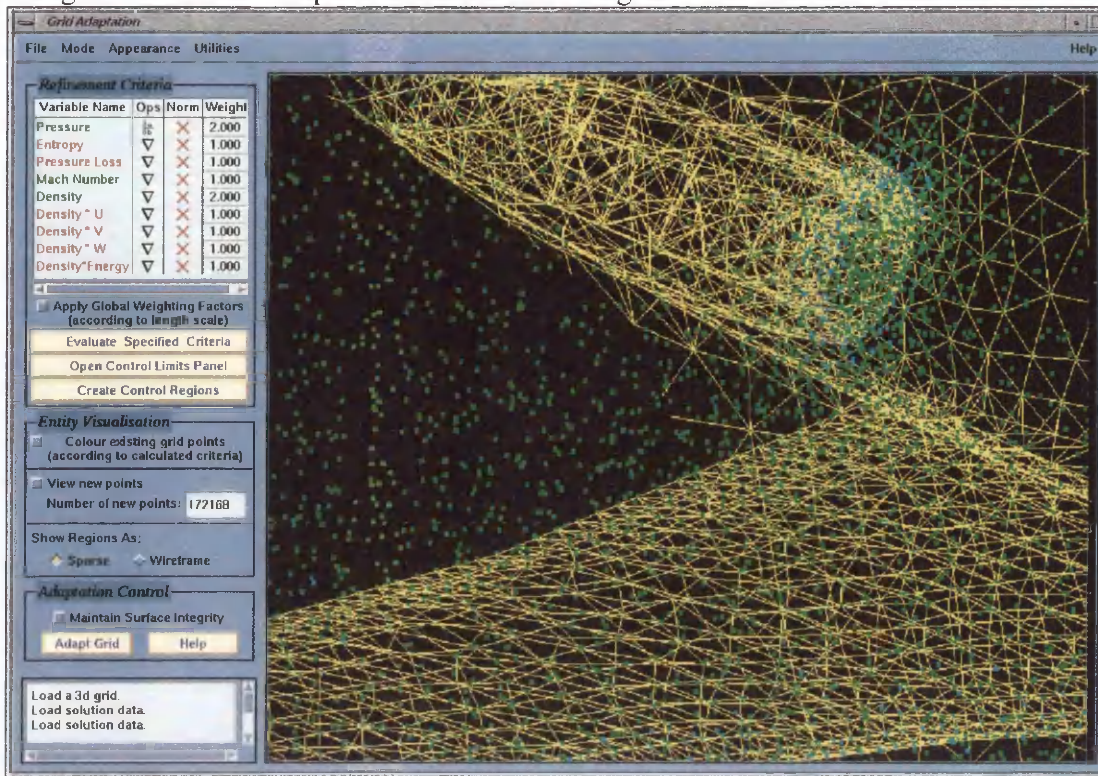
The next step is to solve the flow in parallel which involves starting an application on a remote parallel platform. The parallel flow solver is based on a research application written at UWS as a serial Euler flow solver. The parallel version of the solver was also developed at UWS. The remote connection tool of the PSUE is initiated and the remote machine specified within the interface. Firstly, the various grid files that were created from the grid partitioner must be transferred to the remote platform and then secondly, the parallel flow solver may be initiated. Both of these functions are carried out under the remote connection tool.

Once the flow solver has been directed towards the partitioned grid files it only needs to know the number of partitions and the number of iterations to run for. When the flow solver has finished the number of iterations that it needs to or sufficient convergence of the solution has been reached, the flow solver may be stopped and the solution data output to a file. The remote connection tool may then be used once again to retrieve the solution data file back to the local machine. This solution data may then be loaded back into the PSUE main interface to coexist with the original grid definition. The complete data set may then be transferred to one of the data analysis applications that have been integrated very closely with the PSUE or may be used to adapt the original computational grid.

7.2.6 Grid Adaptation

The grid adaptation module of the PSUE is actually an extension of the geometry builder module. The same viewing functionality is still available but there is also a grid adaptation algorithm incorporated that utilises the H-Refinement method [28]. This module has a series of tools to include and exclude certain regions for refinement and solution value range specification. Figure 7-5 shows a view of the point cloud around the Falcon geometry coloured by density variable.

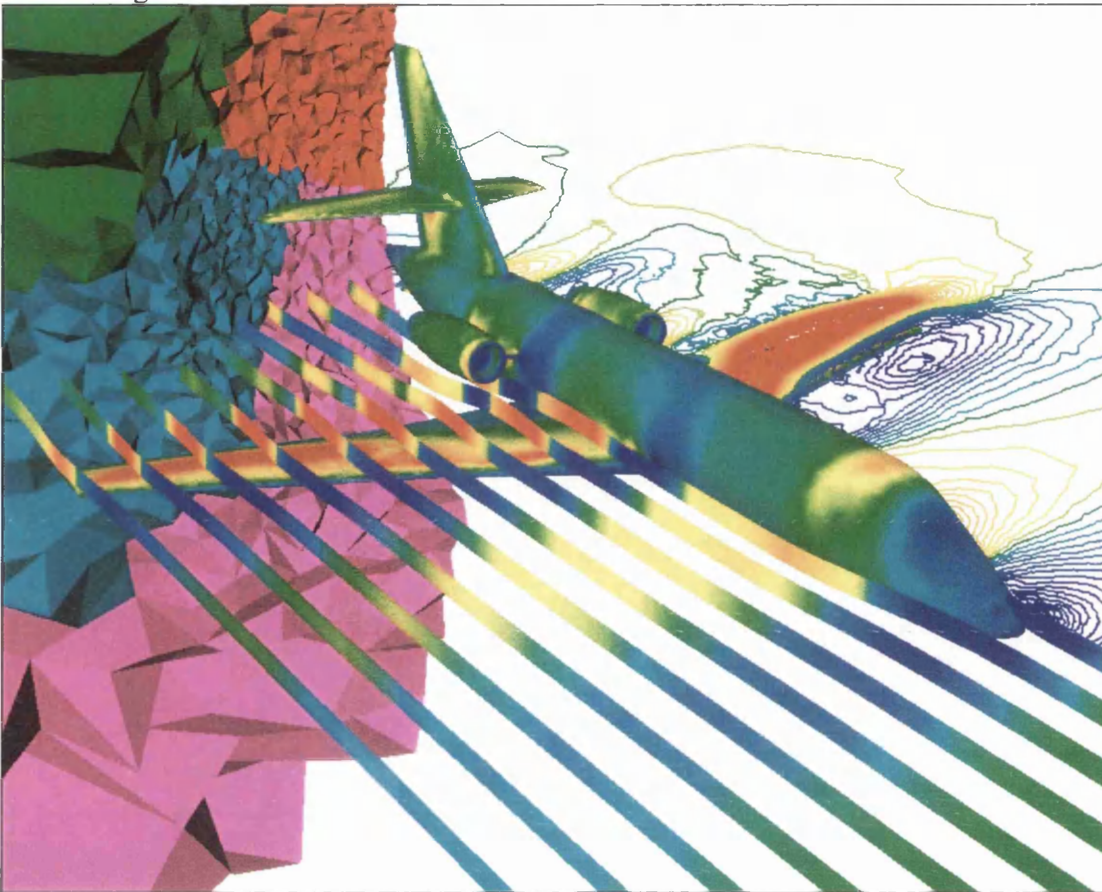
Figure 7-5 - Grid Adaptation Module Showing the Point Cloud Around the Falcon



7.2.7 Final Solution

Once the grid adaptation has been carried out the new grid must be repartitioned and solved using the relevant steps in sections 7.2.3 and 7.2.5. The final solution data may then be transferred with the full grid definition to a data analysis application. The application used in this case is the Ensight (Version 5) software provided by CEI (www.ceintl.com). Within this software, all sorts of post-processing may be carried and an illustration of the flow around the Dassault Falcon together with some of the partitions is shown in Figure 7-6. The figure shows the flow solution at a mach number of 0.7 on a mesh of approximately 146,000 cells partitioned into 16. The surface plot, contour plot and streamlines are all coloured by static pressure.

Figure 7-6 - Final Solution of the Airflow Around the Dassault Falcon



7.2.8 Summary

Figure 7-7 shows a summary of the data flow through the PSUE for the Falcon test case and Table 7-1 summarises each stage's tasks and data transfers.

Figure 7-7 - Summary of Data Flow for Falcon Test Case

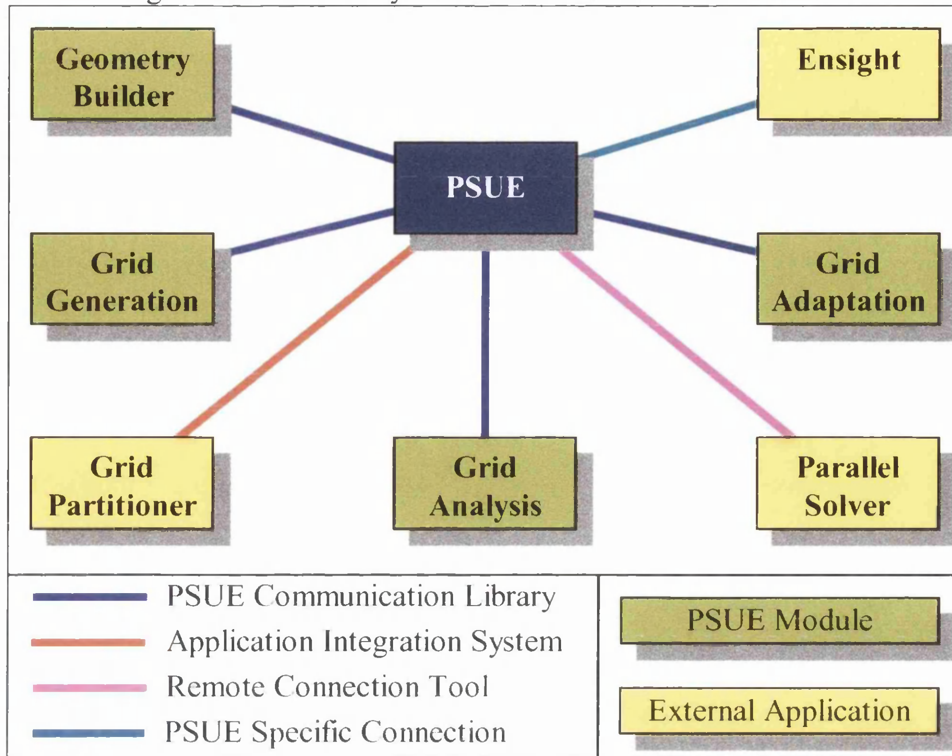


Table 7-1 - A Summary of each Stage's Tasks and Data Transfer for Test Case 1

Stage	Tasks	Data In	Data Out
Geometry Builder	Check topology Set boundary conditions Set grid controls	Geometry	Geometry Mesh Control
Grid Generator	Generate surface grid Generate volume grid	Geometry Mesh Control	Grid
Grid Partitioner	Partition grid	Grid	Multiple Grids
Grid Analysis	Check and analyse grids	Multiple Grids	None
Parallel Solver	Solve for flow solution	Multiple Grids	Solution Data
Grid Adaptation	Adapt grids	Geometry Grid Solution Data	New Grid
Grid Partitioner	Partition new grid	New Grid	Multiple New Grids
Parallel Solver	Solve for new solution	Multiple New Grids	New Solution Data
Ensight	Visualise solution data	Multiple New Grids New Solution Data	None

7.3 Test Case 2 – Airbus A3XX

Due to the confidential nature of the A3XX geometrical configuration, it is not possible to provide any illustrations of the aircraft, but since the general facilities of the PSUE have already been seen in the previous section, this should not cause a problem. This test case was carried out in conjunction with British Aerospace at their Sowerby research centre.

The main objectives for this test case are to import a geometrical configuration, undergo any required geometrical repair, followed by grid generation, an electromagnetic solver and the final analysis of the solution data.

7.3.1 IGES Import

Within the data management functionality of the PSUE main interface exists the ability to import files using the IGES format (developed by IPK (www.ipk.fhg.de) within the CAESAR project). The data files provided for this particular configuration were separated into a number of IGES files and therefore each file must be imported and then saved as the PSUE internal geometry format. The PSUE main interface can then be used to initiate the geometry builder module that is able to amalgamate all of the geometry files into a single configuration.

7.3.2 Geometrical Repair

Upon amalgamation of the geometry within the geometry builder module [21], a number of problems with the geometry are easily highlighted. These include:

- Duplication of surfaces
- Overlaps of surfaces
- Holes or gaps between surfaces

The duplication of surfaces is very easy to correct – one of the surfaces is deleted however the other two problems highlighted above are not so easy to overcome. The overlaps require the initial overlapping surface to be deleted and the resulting hole

may be covered with a new surface or treated as a hole or gap between the surfaces, which is discussed below.

The geometry builder provides functionality specifically for the closure of gaps and holes between surfaces. The surface reconstruction panel provides a number of options of how to knit together the surrounding surfaces.

Once all of the geometrical errors have been rectified the topology is created and the boundary conditions set. The final stage before grid generation is the specification of the grid sources.

7.3.3 Grid Generation

The geometry builder currently holds all of the data definitions and so these are transparently transferred back to the PSUE main interface using the communication library when the module has finished. The main interface is used to initiate the grid generation module [21] and again, all of the data is transferred automatically to the new module using the communication library.

The triangular surface mesh is generated, followed by the generation of the tetrahedral volume grid. This data is then transferred back to the PSUE main interface ready for being checked within the grid analysis module. Once the user is satisfied with the grid that has been generated the data may be sent to the solver.

7.3.4 Electromagnetic Solver

In this case, an electromagnetic solver (an internal code belonging to Sowerby Research Centre of British Aerospace) is being used and is an external application that has been integrated into the PSUE using the application integration system. The application has been configured to connect using the socket system and has been programmed to receive the full grid definition and then to send the solution data once the algorithm has finished.

The application has been integrated into the "Solvers" functionality region and the appropriate script file (PSUEscript.solv) has been edited as follows:

```
U.W.Swansea
Menu Main
{
EM Solver
/usr/people/jo/bin/emsolver
1
SOCKET
BOTH
GRID
RESULTS
}
```

Since all of the grid data is currently within the PSUE main interface, the user may immediately initiate the integrated application, emsolver, which will automatically have the grid definition transferred upon initialisation.

After the solver has finished, the resulting solution data is automatically transferred back to the PSUE before the solver exits. Once again, all of the data now resides within the PSUE main interface and all that remains is to visualise the results.

7.3.5 Data Analysis

Within the data analysis functionality region is the proprietary software, AVS/Express (www.avs.com), which is used in this particular test case. The data is automatically saved from the PSUE into the appropriate data formats relevant for this particular data analysis package. Upon initiation of AVS/Express, particular script files are automatically invoked to transparently load the newly created data files.

With all of the data in the data analysis package the user may set about visualising the results they need.

7.3.6 Summary

Figure 7-8 shows a summary of the data flow through the PSUE for the A3XX test case and Table 7-2 summarises each stages tasks and data transfers.

Figure 7-8 - Summary of Data Flow for A3XX Test Case

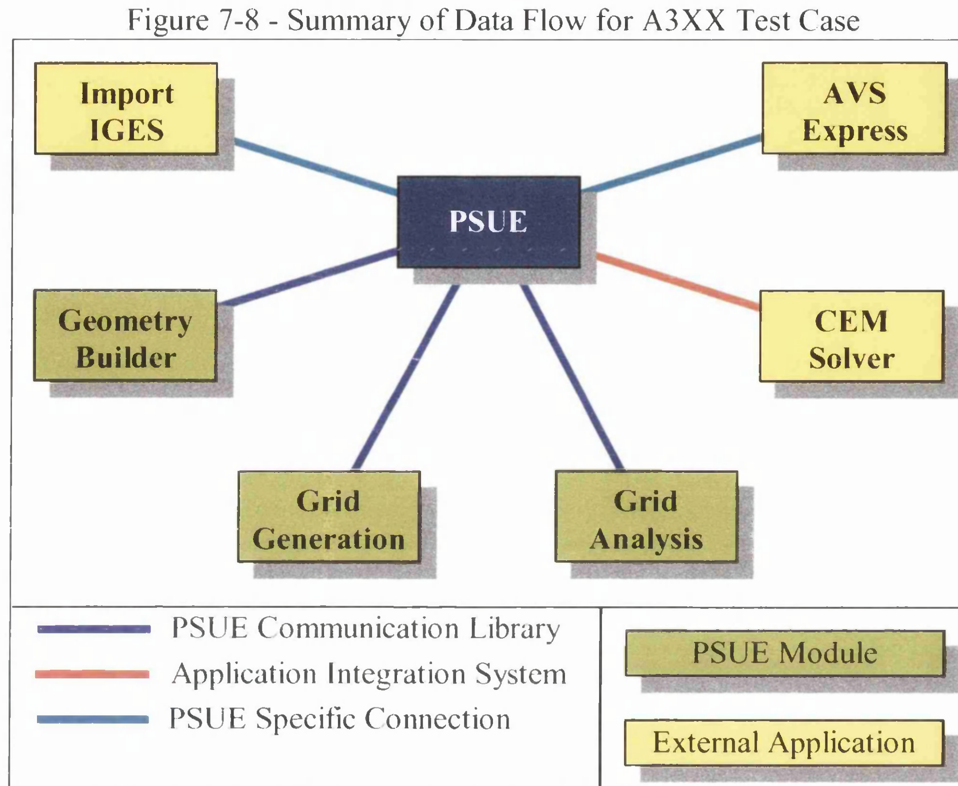


Table 7-2 - A Summary of each Stage's Tasks and Data Transfer for Test Case 2

Stage	Tasks	Data In	Data Out
IGES Import	Import IGES file	IGES file	Geometry
Geometry Builder	Repair topology Set boundary conditions Set grid controls	Geometry	Geometry Mesh Control
Grid Generator	Generate surface grid Generate volume grid	Geometry Mesh Control	Grid
Grid Analysis	Check and analyse grids	Grid	None
CEM Solver	Solve for EM solution	Grid	Solution Data
AVS/Express	Visualise solution data	Grid Solution Data	None

7.4 Test Case 3 – F16 Fighter

Due to the confidential nature of the F16 geometrical configuration, it is not possible to provide any illustrations of the aircraft, but since the general facilities of the PSUE have already been seen in a previous section, this should not cause a problem. This test case was carried out in conjunction with DASA at the University of Stuttgart.

The main objectives for this test case are to prepare the geometrical configuration, generate an unstructured grid in parallel, solve the airflow in parallel using a panel method flow solver, adapt and further solve the grid according to flow parameters and finally process the solution data.

7.4.1 Geometry Preparation

The geometry required extensive additions and manipulation in order to produce a topologically correct configuration and most of this work was carried out within the PSUE geometry builder module [21]. Upon creation of the topology and setting the boundary conditions, only the grid source configuration is required. The grid is to be generated in parallel however the algorithm used is essentially the same as the sequential grid generator. Therefore, the existing method for creating grid sources within the geometry builder may still be used for controlling the grid density within the parallel grid generator.

7.4.2 Parallel Grid Generation

Initially, the PSUE grid generation [21] module must generate the triangular surface grid. This requires the geometry builder to update the main PSUE interface with all of the data definitions. This is done using the communication library that continues to transfer the data to the grid generation module. Once the surface grid has been generated, the new boundary grid definition must be transferred back to the PSUE main interface via the communication library.

The parallel grid generator [27],[30],[31] is a relatively new module of the PSUE and therefore has not been totally integrated into the system. The data residing in the

PSUE main interface must be saved to a file that can be subsequently loaded in to the parallel grid generator. The parallel grid generator module is initiated from the PSUE generic functionality but without any data transfer.

The application loads the relevant data from the files saved by the PSUE main interface and begins to generate the unstructured tetrahedral grid in parallel. The resulting volume grid is stored in a file ready for the solver.

7.4.3 Parallel Solver and Adaptation

The parallel solver (an internal code belonging to DASA) is a panel method based flow solver that partitions the grid itself as a pre-processing stage. The solver is to be executed on a massively parallel platform – Cray T3E – and so the remote connection tool is used as in the first test case. Upon sufficient convergence of the flow solver, the data is output to file and transferred back to the local platform.

The PSUE main interface loads the newly acquired solution data and initiates the grid adaptation module. All of the required data definitions are transferred to the grid adaptation module, using the communication library, where the grid is refined in the relevant regions.

The refined grid and solution is then sent back through the same loop to the parallel platform in order to carry out further solver iterations. Once the final data has been calculated the data is retrieved back to the PSUE main interface for data analysis.

7.4.4 Virtual Reality Data Analysis

Due to heavy refinement of the grid in this particular test case, normal data analysis packages, such as Ensight, would have great difficulty visualising the solution data and facilities such as parallelised visualisation [32] would be necessary. Since this test case was being carried out at the University of Stuttgart and the site also hosted a large virtual reality facility (Virtual Environments Lab -

www.hlrs.de/organization/vis/velab), the large data set was processed using parallel visualisation techniques and viewed within a virtual reality environment [33].

7.4.5 Summary

Figure 7-9 shows a summary of the data flow through the PSUE for the F16 test case and
Table 7-3 summarises each stages tasks and data transfers.

Figure 7-9 - Summary of Data Flow for F16 Test Case

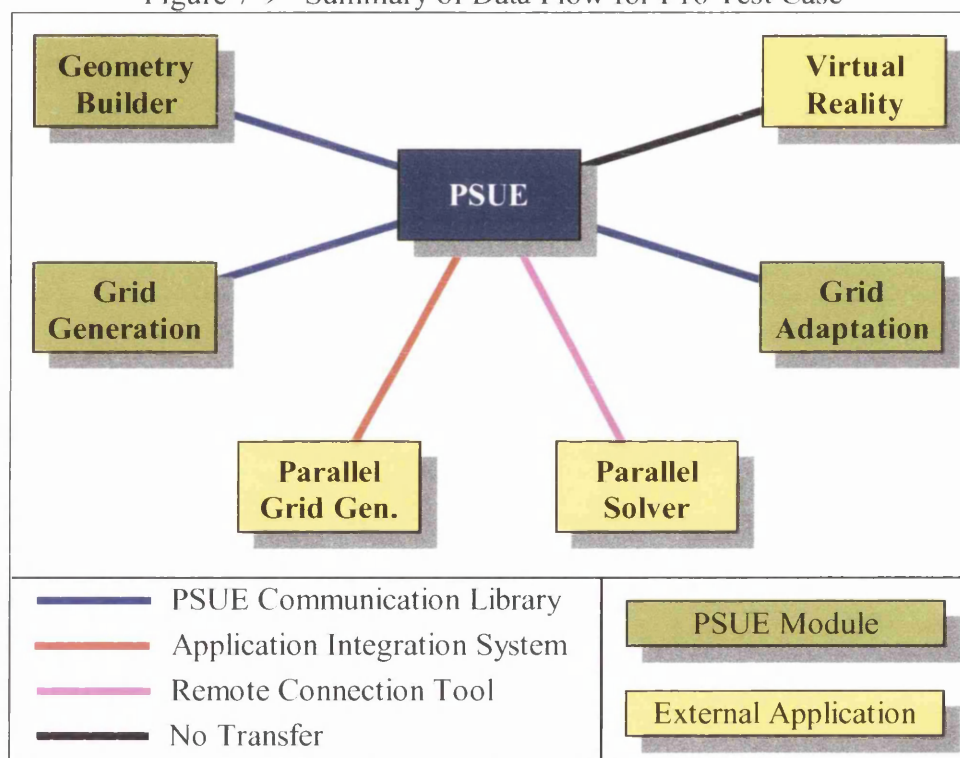


Table 7-3 - A Summary of each Stage's Tasks and Data Transfer for Test Case 3

Stage	Tasks	Data In	Data Out
Geometry Builder	Repair topology Set boundary conditions Set grid controls	Geometry	Geometry Mesh Control
Grid Generator	Generate surface grid	Geometry Mesh Control	Surface Grid

Parallel Grid Generator	Generate volume grid	Surface Grid Mesh Control	Volume Grid
Parallel Solver	Partition volume grid Solve for flow solution	Volume Grid	Solution Data
Grid Adaptation	Adapt Grid	Geometry Volume Grid Solution Data	New Volume Grid
Virtual Reality	Visualise solution data	None	None

8 CONCLUSIONS

8.1 Overview

This thesis has presented details of a computer environment for engineering simulation tools. The environment has been developed to fully utilise the existing capabilities of computing technology readily available within industrial and educational establishments.

The environment provides extensive capabilities for the integration of computational engineering applications. The ability to integrate any arbitrary application allows for a great flexibility within the environment and thereby creates a toolkit and provides a very powerful tool. This same tool provides access to the rapidly increasing capabilities of HPCN, possibly the technology that will grow most rapidly within information technology in the foreseeable future.

Although this thesis has presented details of the PSUE, a multi-disciplinary engineering application environment designed and developed by a team, it has concentrated almost entirely on the author's work in relation to the design and development of the PSUE and its functionality.

The PSUE was originally born in 1994 by Professor N. P. Weatherill and Dr M. Marchant. A design period of 1994-1998 saw development during 1995-1999 and has led to the design and development of the 6S Environment of the CEC ESPRIT project – JULIUS. However, due to the extremely rapid development of computer hardware and especially software, the PSUE and its modules are already out-dated and therefore redundant to a certain degree.

Many companies and institutes throughout Europe have used the PSUE however, due to a halt on further design and development of the PSUE it has failed to become a commercial success.

As part of the development of the PSUE and as a requirement for the ESPRIT projects, a number of detailed documents were written including a user manual [21], user tutorial [22], developer's guide [23], application integration handbook [24] and finally the system documentation [25].

8.2 Discussions

The main interface is the core application of the PSUE system and its modules. The design requirements for this software included the integration of all applications and the communication of data between those applications. The authors development of the main interface has met these requirements but to what extent?

The communication library that connects all of the PSUE modules together is extremely affective at providing an almost seamless flow of data between the modules. It also provides great flexibility for the experienced user to work with multiple problem definitions at any one time. However, a shared memory communication system is restricted by the computer hardware of the platform on which the application is running. There is a large overhead of memory for this particular communication system and on the average size workstation this limits the user on the amount of data they can access at any one time.

The user is able to integrate any number of arbitrary applications into the PSUE environment that in it self is a very useful feature. The PSUE also allows extensive automated data transfer between the PSUE and the applications, especially if the source code for the application is freely available. However a major disadvantage of the data transfer system is that only data formats specified by the PSUE may be used to transfer the data.

The PSUE provides the single unified interface that was originally suggested would be required to create a successful environment, along with the ability to assist in the simulation engineering process. The PSUE also fulfils this target and the other major challenges originally presented in this thesis due to the extensive work carried out on application integration, data transfer and management.

The original key objectives of the PSUE have also been met. The reduction in problem set-up time is quite considerable and for a particular case that would

normally take 2-3 weeks the user could expect to reduce that time to well under 1 week. Due to the general configuration of the PSUE interface and its modules, users learn quickly and efficiently, reducing training periods, however further work should be carried out in this area. The modular framework and integration of arbitrary applications have been fulfilled directly as a result of the architecture of the PSUE. Finally, the HPCN exploitation was quite extensive at the time the PSUE was developed but due to the rapid development of hardware systems and networks, is lagging behind technology.

One of the major technological advancements in recent time is grid technology [34][35], which promotes the use of universally wide computing resources to carry out most types of e-business and e-solutions. Grid computing produces and defines protocols, services and tools that allow scalable virtual organisations [34] to utilise idle computers on an international scale. The protocols cover major issues such as quality or service, job scheduling, co-allocation and accounting of resources. The services include storage service providers (SSP), application service providers (ASP), and Grid Security Infrastructure (GSI) that extends virtual private networks (VPN), Transport Layer Security (TLS) and Generic Authorisation and Access(GAA).

The grid allows the virtual organisations, which may span multiple institutions, to access various geographically distributed resources using resource management protocols that also allow co-allocation of the resources. The grid also provides, through the protocols, secure remote access to the computing and data resources and data management at all levels including transfer. The identification of the protocols and services is the first priority [34] of the grid architecture followed by the Application Protocol Interface (API) and Software Developmen Kit (SDK). APIs and SDKs allow developers to create complex applications that will interface with the grid architecture. They accelerate code development, enable code sharing and enhance application portability.

The development of APIs and SDKs and the internal structure of the grid architecture utilise the protocols and services of the grid and these facilities are termed "Middleware" [34]. Globus is an open source reference implementation of protocols [35] and is an example of Middleware facilities. However, it not only provides APIs and SDKs but also the protocols and services if they are not already available at the resource level. Globus also involves web service technologies [35] focussing on XML to address heterogeneous distributed computing.

These capabilities would extend the current HPCN and encourage geographically distributed abilities of the PSUE. An example implementation within the PSUE would be the parallel visualisation of data [32] within the third testcase. The geographically distibuted resources are pooled together to allow the visualisation of large datasets [36]. The grid allows the management and manipulation of the resources, security and integration of applications to allow the visualisation to happen.

The PSUE was initially designed and developed to be a prototype in order to ascertain whether the concept of a fully integrated engineering simulation environment is feasible. The author believes that the PSUE proves that this is feasible and the PSUE has been a major success and could lead to an environment that would easily be a major technological and commercial success.

8.3 Future Developments

It was mentioned in section 8.1 that design and development of the PSUE has been halted however if further development was to be made on the PSUE the author recommends the topics discussed below.

The main PSUE communication library and the application integration library have been developed independently. The environment would benefit greatly if a single system, encompassing all of the existing functionality, was developed to cover generic functionality modules and external applications. The communication system should be developed based on a socket system since the main disadvantage, the speed, has now been resolved and a socket system will enable a distributed PSUE. This would allow separate modules to be shown and even operated from another machine that may even be geographically distributed.

The first step to a completely unified application integration system is the Tool Wrapper, a module that would possibly allow as much integration for a proprietary application as an in-house application, for which the program source code is available. The Tool Wrapper is still very much a simple prototype however it has huge development prospects and managed appropriately would provide a great level of flexibility for the user to integrate almost any application.

The current links for HPCN should be improved and made far more transparent and automatic. A new development area for parallelisation is the visualisation of data for pre and post processing [32]. Also, the grid technology discussed in the previous section should also be researched and the appropriate capabilities integrated into the PSUE as standard functionality, at all levels.

As computer technology continues to rapidly increase, new areas for design should include artificial intelligence and virtual reality that can be used at almost all stages of an engineering simulation.

Going back to the base formation of a user interface, many extensions and new functionality may be introduced into the modules of the PSUE. Levels of competence, feedback, automatic completion of input data, short-cut keys, icons instead of text for controls, amongst other features, could all be enhanced to provide more intuitive and easier to use interfaces. As these features are developed and enhanced, further concepts will come to light, as this is the continual evolution observed within most computer interfaces.

9 REFERENCES

[1] N.P.Weatherill, M.J.Marchant, E.Turner-Smith, Y.Zheng, M.I.Sotirakos; A Parallel Simulation User Environment for Computational Engineering; 5th International Conference on Grid Generation in Computational Fluid Dynamics; Starkville, Mississippi, USA; 1-5th April 1996.

[2] N.P.Weatherill, M.J.Marchant, E.Turner-Smith, Y.Zheng, M.I.Sotirakos; The Design of a Graphical User Environment for Multi-Disciplinary Computational Engineering; ECCOMAS 1996; Paris; September 1996.

[3] N.P.Weatherill, M.J.Marchant, E.Turner-Smith, Y.Zheng, M.I.Sotirakos; PSUE – Requirements Capture; Civil Engineering Department, University of Wales Swansea; Official Report CR/941/96.

[4] Sun Microsystems Inc.; OPEN LOOK Graphical User Interface Functional Specification; Addison-Wesley Publishing Company; 1989; ISBN: 0201523655

[5] B.Shneiderman; Designing the User Interface - Strategies for Effective Human-Computer Interaction; Addison-Wesley Publishing Company; 2nd edition; 1992; ISBN: 0201694972

[6] B.Laurel; The Art of Human-Computer Interface Design; Addison-Wesley Publishing Company; 1990; ISBN: 0201517973

[7] J.R.Brown and S.Cunningham; Programming the User Interface: Principles and Examples; 1989; ISBN: 0471638439

[8] K.Cox and D.Walker; User Interface Design; 2nd edition; 1993; ISBN: 0139528881

[9] J.Preece et al.; A Guide to Usability: Human Factors in Computing; Addison-Wesley Publishing Company; 1992; ISBN: 020162768X

[10] E.Grandjean and E.Vigliani; Ergonomic Aspects of Visual Display Terminals; 1980; ISBN: 0850662117

[11] R.E.Eberts; User Interface Design; 1993; Prentice Hall; ISBN: 0131403281

[12] Apple; Human Interface Guidelines: The Apple Desktop Interface; Addison-Wesley Publishing Company; 1987; ISBN: 0201177536

[13] UIST; Fourth Annual Symposium on User Interface Software and Technology; 1991.

[14] M.I.Sotirakos and R.Said; Memory Management using C and FORTRAN; Internal Memo; Civil Engineering Department, Swansea University; January 1995.

[15] W.R.Stevens; Unix Network Programming; Prentice Hall Inc.; ISBN: 0-13-949876-1

[16] A.Nye; Xlib Programming Manual; O'Reilly & Associates Inc; ISBN: 1-56592-002-3

[17] A.Nye; Xt Toolkit Intrinsics Programming Manual; O'Reilly & Associates Inc; ISBN: 1-56592-003-1

[18] Open Software Foundation; OSF/Motif Programmers Reference; Prentice Hall Inc.; ISBN: 0-13-143166-8

[19] OpenGL Architecture Review Board; OpenGL Programming Guide; Wesley Publishing Company Inc.; ISBN: 0-201-63274-8

[20] OpenGL Architecture Review Board; OpenGL Reference Manual; Wesley Publishing Company Inc.; ISBN: 0-201-63276-4

[21] E.Turner-Smith, Y.Zheng, M.I.Sotirakos; PSUE – User Manual; Civil Engineering Department, University of Wales Swansea; Official Report CR/938/96.

[22] E.Turner-Smith, Y.Zheng, M.I.Sotirakos; PSUE – User Tutorial; Civil Engineering Department, University of Wales Swansea; Official Report CR/939/96.

[23] E.Turner-Smith, Y.Zheng, M.I.Sotirakos; PSUE – Developer's Guide; Civil Engineering Department, University of Wales Swansea; Official Report CR/942/96.

[24] E.Turner-Smith; PSUE – Application Integration Handbook; Civil Engineering Department, University of Wales Swansea; Official Report – ESPRIT MEDUSA.

[25] E.Turner-Smith, Y.Zheng, M.I.Sotirakos; PSUE – System Documentation; Civil Engineering Department, University of Wales Swansea; Official Report CR/940/96.

[26] N.P.Weatherill and O.Hassan; Efficient three-dimensional Delaunay Triangulation with Automatic Point Creation and Imposed Boundary Constraints; International Journal Numerical Methods in Engineering, Vol. 37; 1994.

[27] N.Verhoeven, N.P.Weatherill and K.Morgan; Parallelisation of an Unstructured Grid Generation Algorithm; Parallel CFD Conference, Pasadena, USA; July 1995.

[28] R. Löhner; Adaptive H-Refinement on 3-D Unstructured Grids for Transient Problems; *AIAA*-89-0365; 1989.

[29] Steven T. Barnard and Horst D. Simon; A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems; Concurrency: Practice and Experience, Vol. 6, No. 2.; April 1994.

[30] N.P.Weatherill, R.Said and K.Morgan; The Construction of Large Unstructured Grids by Parallel Delaunay Grid Generation; Numerical Grid Generation in Computational Field Simulations; Ed. M.Cross., B.K.Soni, J.F.Thompson, J.Hauser, P.R.Eiseman; Proceedings of the 6th International Conference, held at the University of Greenwich; July 1998.

[31] R.Said, B.Larwood, N.P.Weatherill, O.Hassan, K.Morgan; Parallel Delaunay Unstructured Grid Generation; 7th International Conference on Numerical Grid Generation in Computational Field Simulations; September 25-28, 2000; Chateau Whistler Resort Whistler, British Columbia, Canada.

[32] J.W.Jones and N.P.Weatherill; The Visualisation of Large Unstructured Grid Data Sets; Numerical Grid Generation in Computational Field Simulations; Ed. M.Cross., B.K.Soni, J.F.Thompson, J.Hauser, P.R.Eiseman; Proceedings of the 6th International Conference, held at the University of Greenwich; July 1998.

[33] L.Sastry, J.V.Ashby, D.R.S.Boyd, R.F.Fowler, C.Greenough, J.W.Jones, E.A.Turner-Smith and N.P.Weatherill; Virtual Reality Techniques for Interactive Grid Repair; Numerical Grid Generation in Computational Field

Simulations; Ed. M.Cross., B.K.Soni, J.F.Thompson, J.Hauser, P.R.Eiseman; Proceedings of the 6th International Conference, held at the University of Greenwich; July 1998.

[34] I.Foster, C.Kesselman, S.Tuecke; The Anatomy of the Grid: Enabling Scalable Virtual Organizations; International J. Supercomputer Applications, 15(3), 2001.

[35] I.Foster, C.Kesselman, J.M.Nick, S.Tuecke; The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration; Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[36] I.Foster, J.Insley, C.Kesselman, G.von.Lasewski, M.Thiebaux; Distance Visualisation: Data Exploration on the Grid; IEEE Computer Magazine, 32(12), 1999.