# A USER INTERFACE TOOLKIT EXTENSION FOR COOPERATIVE PROBLEM SOLVING


by

Regina Huntington, B.App.Sc. (Computer Technology) *Victoria.*

*School of Computer and Information Science,*
*Faculty of Applied Science,*
*University of South Australia.*


A thesis submitted to the Faculty of Applied Science and Technology,
in partial fulfilment of the degree of
Master of Applied Science in Computer and Information Science.

October 1995.

# Table of Contents

**List of Figures** v

**List of Tables** vi

**Glossary** vii

**Abstract** viii

**Declaration** ix

**Acknowledgements** x

**Chapter 1. Introduction** 1

1.1 Objectives . . . 2

1.2 Methodology . . . 2

1.3 Thesis organisation . . . 4

**Chapter 2. Related Work** 6

2.1 Cooperative problem solving . . . 6

2.1.1 A new taxonomy . . . 7
2.1.2 Visualisation . . . 8
2.1.3 Multiple views . . . 9
2.1.4 Filtering . . . 9
2.1.5 Highlighting . . . 10
2.1.6 Summation . . . 10
2.1.7 Advising . . . 10
2.1.8 Problem structuring . . . 11

2.2 Toolkits . . . 11

2.2.1 Toolkit extensions . . . 12
2.2.2 Why choose InterViews? . . . 12
2.2.3 Existing toolkits . . . 13

2.3 Frameworks . . . 14

2.3.1 Framework example . . . 14
2.3.2 Existing frameworks . . . 15

2.4 Summary . . . 16

**Chapter 3. Cooperative Problem Solving Framework** 17

3.1 Cooperative software requirements . . . 17

3.2 Information presentation requirements . . . 19

3.3 Information presentation framework . . . 21

3.3.1 Framework components . . . 23
3.3.2 Examples of derived components . . . 25
3.3.3 Class hierarchy . . . 26
3.3.4 Sample configuration . . . 27
3.3.5 Cooperative problem solving techniques . . . 28

3.4 Summary . . . . . 29

**Chapter 4. Cooperative Software Application** **30**

4.1 Small batch robotic welding . . . . . 30
4.1.1 Welding process parameters . . . . . 30
4.1.2 Welding robot setup . . . . . 31

4.2 Relevant work in robotic welding . . . . . 32
4.2.1 Autonomous approaches to setup . . . . . 32
4.2.2 Support approaches to setup . . . . . 32
4.2.3 Cooperative approaches to setup . . . . . 33

4.3 The existing welding environment . . . . . 33
4.3.1 Yasnac ERC robot controller interface . . . . . 34
4.3.2 Welding Data Monitor interface . . . . . 35
4.3.3 A typical scenario - existing environment . . . . . 36

4.4 Requirements for a robotic welding environment . . . . . 37
4.4.1 Existing functions . . . . . 37
4.4.2 New functions . . . . . 38

4.5 Summary . . . . . 39

**Chapter 5. Design and Implementation** **40**

5.1 The InterViews toolkit extension . . . . . 40
5.1.1 Framework classes . . . . . 40
5.1.2 Derived classes . . . . . 41
5.1.3 C functions . . . . . 44
5.1.4 Sample linkage of toolkit components . . . . . 45
5.1.5 Facilities of the toolkit extension . . . . . 46

5.2 The prototype cooperative software environment . . . . . 47
5.2.1 InterfaceManager subsystem . . . . . 50
5.2.2 RobotManager subsystem . . . . . 52
5.2.3 DataManager subsystem . . . . . 53
5.2.4 Panel functionality . . . . . 54
5.2.5 A typical scenario - the new environment . . . . . 56
5.2.6 Demonstration . . . . . 57

5.3 Evaluation of the framework . . . . . 57
5.3.1 Framework design . . . . . 57
5.3.2 Framework implementation effort . . . . . 58

5.4 Evaluation of the toolkit extension . . . . . 58
5.4.1 Cooperative software techniques . . . . . 59
5.4.2 Toolkit extension application effort . . . . . 60

5.5 Evaluation of the cooperative software environment . . . . . 60
5.5.1 Evaluation against requirements . . . . . 61

5.6 Summary . . . . . 63

**Chapter 6. Future Work and Conclusions** **64**

6.1 Enhancements . . . . . 64
6.1.1 Object-oriented implementation . . . . . 64
6.1.2 Advanced program debugging . . . . . 65
6.1.3 Knowledge base of cases . . . . . 65
6.1.4 Parameter limit alerting . . . . . 65
6.1.5 Multimedia data presentation . . . . . 65
6.1.6 User–configured information presentation . . . . . 66

6.2 Future Work . . . . . 67
6.2.1 High-level cooperative problem solving techniques . . . . . 67
6.2.2 Other testbeds . . . . . 68
6.2.3 Study of problem solving . . . . . 68

6.3 Conclusion . . . . . 68
6.3.1 Cooperative problem solving taxonomy . . . . . 69
6.3.2 Information presentation framework . . . . . 69
6.3.3 Toolkit extension . . . . . 70
6.3.4 Summary . . . . . 70

**Bibliography** **72**

**Appendix A. Framework Class Reference** **76**

**Appendix B. Panel Class Reference** **83**

# List of Figures

Figure 1-1 Typical setup procedure for a complex manufacturing process . . . . . 3
Figure 2-1 InterViews FileChooser . . . . . 13
Figure 3-1 Information presentation framework design. . . . . 23
Figure 3-2 Information presentation class hierarchy . . . . . 27
Figure 3-3 Composition of a sample information presentation subsystem . . . . 28
Figure 4-1 Overview of the robotic welding environment. . . . . 34
Figure 4-2 Yasnac ERC robot controller panel . . . . . 35
Figure 4-3 Welding Data Monitor (WDM) . . . . . 36
Figure 5-1 Sample WDM data format file . . . . . 41
Figure 5-2 Sample WDM data file. . . . . 42
Figure 5-3 Readout. . . . . 42
Figure 5-4 Histogram . . . . . 43
Figure 5-5 Alarm . . . . . 44
Figure 5-6 Sample code for linking toolkit components . . . . . 46
Figure 5-7 Overview of Panel within robotic welding environment . . . . . 48
Figure 5-8 The Panel subsystems. . . . . 50
Figure 5-9 Sample glyph hierarchy for *InterfaceManager* subsystem. . . . . 51
Figure 5-10 The Panel cooperative software environment . . . . . 52
Figure 5-11 Sample object-instance graph for *DataManager* subsystem . . . . . 54
Figure B-1 Sample WDM data format file . . . . . 85
Figure B-2 Sample WDM data file. . . . . 85

# List of Tables

Table 2-1 Taxonomy of CPS techniques . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
Table 3-1 Summary of requirements for cooperative software . . . . . . . . . . . . . 17
Table 3-2 How cooperative software requirements are to be met . . . . . . . . . . . 20
Table 5-1 Interface comparison: Yasnac ERC vs Panel, by no. of steps . . . . . . 55

# Glossary

| | |
|---|---|
| abstract class | a class designed to factor out behaviour common to a range of classes; not to produce instances of itself. |
| class responsibilities | actions a class is expected to perform, or knowledge it maintains. |
| concrete class | a class designed to be instantiated so it can be used directly in an application. |
| CPS (Cooperative Problem Solving) | cooperation between a human and a computer to solve a problem or perform a task using complementary capabilities. |
| data stream | a series of values arriving over a period of time representing a single measured quantity. |
| GUI (Graphical User Interface) | a system of interaction between a computer and its user based on graphical icons. |
| framework | a reusable design for a subsystem, which specifies the way a set of components work together. |
| subsystem | a self-contained part of a system that performs a set of related functions. |
| toolkit | a set of related and reusable classes designed to provide useful, general-purpose functionality. |
| toolkit extension | a relatively small number of related classes derived from a base toolkit for a more specialised purpose. |
| ULOC (Un-commented Lines Of Code) | a method of measuring the implementation effort of a piece of software. |
| WDM (Welding Data Monitor) | a device used to measure and record welding process data. |
| widget | a primitive building block of a graphical user interface. |

# Abstract

Decision making in complex, reactive systems often requires a human operator to interpret a large quantity of information in real time and react accordingly. Automation of such tasks is obviously desirable, but full automation is often not achievable in the short term, because the models of both the decision-making process and the behaviour of the underlying system are often incomplete. In such situations, cooperative problem solving may provide a viable alternative to full automation. Cooperative problem solving in this context refers to the cooperation between a human and a computer to solve a complex, ill-defined problem. Research in the area of cooperative problem solving has to date concentrated on the provision of high-level support for decision making through the use of artefacts such as advisory systems or critiquing systems. There is currently an absence of tools to support the low level activities that are necessary for effective decision making in ill-defined, reactive environments. This thesis demonstrates how support can be provided for low-level activities in these environments, then describes an efficient mechanism for the implementation of that support.

The proposed support mechanism is an information presentation framework implemented as an extension to a graphical user interface toolkit. This framework defines five types of toolkit component and the ways in which these components can interact to enable data acquired from a reactive system to be presented to the user in a timely and appropriate manner. The application developer combines a number of toolkit components of each type into an information presentation subsystem according to the needs of the application. In order to validate the framework, an extension to the InterViews user interface toolkit was developed and used to build a prototype cooperative environment for robotic welding.

# Declaration

I declare that this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge, it does not contain any materials previously published or written by another person except where due reference is made in the text.

______________________________

Regina Huntington,

16 July, 1996.

# Acknowledgements

When I first became interested in undertaking a research degree, I was directed to Jacquie Jarvis (CIS) and Dennis Jarvis (CSIRO Division of Manufacturing Technology), as they were involved in a research activity (fault diagnosis in manufacturing systems) which most closely approximated my interests. Jacquie Jarvis and Dennis Jarvis, together with Doug Seeley (CIS), were responsible for defining the project which is the subject of this thesis. The initial intention was to develop a diagnostic system for welding using a cooperative problem solving approach and the project began with three advisers contributing different areas of expertise - Dennis Jarvis (cooperative problem solving), Jacquie Jarvis (fault diagnosis) and Bruce Thomas (HCI). However, once the project was under way, the focus shifted from welding diagnosis to cooperative problem solving frameworks and Jacquie Jarvis felt that her involvement was no longer required.

I would like to thank all three of my advisers for their support and encouragement throughout the project, and for providing such wonderfully different, complementary viewpoints. I would also like to thank Doug Seeley for facilitating such an interesting and rewarding project.

The viability of this project depended on access to a suitable testbed in order to evaluate the framework. I am extremely grateful to the CSIRO Division of Manufacturing Technology for providing me with access to their robotic welding environment. Special thanks must also go to the members of their Welding Group for their assistance during the development of the prototype cooperative software environment and its subsequent evaluation. Other staff members at CSIRO also assisted me with the nuts and bolts of my research. In particular, I would like to thank Ian Dick for providing the welding monitor and Laurie Care, Peter Lewis, and Richard Frost for battling with the computer systems on which my demonstrations depended. I owe you all many birthday cakes for that. I must also thank Chris Wood, now sadly no longer with us, for heroically coming to my rescue to install InterViews. I will fondly remember him.

Once mastered, InterViews is a very flexible and powerful tool. I would like to thank Upi Weston (Flinders University) for his assistance in helping me to understand the intricacies of InterViews. Thanks also to my colleagues for contributing valuable criticism of my thesis, especially Elena Trichina and Gary Gibson.

I am grateful to the School of Computer and Information Science for providing an internal scholarship to help support me through my period of full time study.

Special thanks go to Steve Hunter for his support, and to Jack Grozev for his encouragement and warm company at the office. Which reminds me to thank the Motorola Australia Software Centre for providing a pleasant and productive final write-up environment for several months.

# Chapter 1. Introduction

Cooperative problem solving (CPS) refers to the cooperation between a human and a computer to solve ill defined problems; the software used is known as cooperative software. An ill defined problem in this context is a problem which cannot be fully specified before an attempt is made to solve it, and for which there is no single, optimal solution to be found [13]. A good example of an ill defined problem is product design, where the designer is required to be an active participant in a complex and evolving problem solving process. This process may involve other people with different perspectives and a range of software systems. For example, there may be marketing, engineering, or production roles to be taken into account [4]. Often, multiple solutions will be the norm; traditional decision support systems only provide a single, optimal solution to the user and are therefore of limited use in such situations. Alternative approaches such as genetic algorithms can provide multiple solutions, but they do not involve the user in solution generation [15].

CPS has traditionally been applied to static design tasks, such as kitchen layout, user interface design, and manufacturing cell design. However, the author supports the view that a CPS approach is also suitable for reactive decision-making tasks such as power grid management, air traffic control, and intensive care patient monitoring. In these and other reactive work environments, decisions must be made rapidly and are based on the assimilation of large quantities of information.

The problems faced by decision makers in these reactive environments relate to the unpredictability of event arrival, the need to respond before the next event, and the complexity of control procedures. The tasks often require an experienced operator under normal circumstances and may become too difficult under a heavy workload, such as during an emergency. There may be many different procedures to remember and some procedures may involve gathering information in order to diagnose a problem. Although some degree of automation is desirable - for reliability, safety, or economy - the processes involved in these reactive environments are often not specifiable or predictable enough for full automation in the near term. Consequently, a human operator is still necessary and should be supported, rather than replaced by software.

Research into cooperative software has so far concentrated on developing cognitive models of problem solving or decision-making, and on providing high-level support by means of advising systems, checklists, and other problem structuring techniques [13]. However, there are many lower-level activities that a cooperative environment could easily perform, particularly in a reactive environment. Such activities might include: design retrieval, data acquisition, data presentation, and data storage.

## 1.1 Objectives

The main objectives of this thesis are:

1. to define effective support for low-level activities in reactive environments;
2. to describe efficient mechanisms for implementors of CPS systems to support these activities.

The proposed mechanism is the extension of a graphical user interface (GUI) toolkit - InterViews [27] - to support reactive decision-making. The toolkit extension will be based on a framework [14], which is a set of cooperating components that form a reusable subsystem design. Frameworks define how the different types of components are connected and work together, rather than just the behaviour of each component. While toolkits reuse code, frameworks reuse designs.

A typical GUI toolkit already provides much of the required functionality of a CPS software environment; the framework proposed in this thesis addresses a low-level activity which is not already provided by a GUI toolkit: information presentation. The framework's design defines five types of component and the links between them to support the flow and processing of incoming data, from external data acquisition devices to the screen. These components have only generic functionality which needs to be specialised by the developer to suit a particular application domain.

The toolkit extension implements the information presentation components which are then used in the same way as the existing interface components. Together, the information presentation components and the interface components should provide a CPS toolkit that is no more difficult to use than the original GUI toolkit.

## 1.2 Methodology

This thesis is attempting to:

1. identify low-level activities that need to be supported in CPS systems for reactive environments;
2. devise a generic framework which directly supports the implementation of these activities in CPS systems.

The first task is addressed by a literature review. The second task involves two steps:

1. designing the framework and implementing the toolkit extension;
2. evaluating the usefulness of the framework, by using the toolkit extension to develop a prototype CPS system for a real world application.

Robotic welding setup was chosen as the real world reactive domain required to evaluate the toolkit extension, because it is an ill defined task [18] that requires the problem solving skills and domain expertise of a human operator. Setup is ill defined because:

- each new job being set up will be slightly different to previous jobs due to variations in initial conditions (such as work piece position or metal surface finish); this requires a change in welding input parameters;
- the relationships between process inputs and process outputs are not well understood, or only understood empirically by practitioners;
- setup usually requires an experienced operator to interpret and act on a large amount of - often incomplete - information in real time in order to make suitable parameter modifications.

The operator is required to setup the robot using an interactive, iterative procedure of programming and welding that consumes materials, robot time, and person time. These factors contribute to the expense of setting up small-batch manufacturing jobs and have primarily restricted the use of robotic welding to mass production applications, such as vehicle assembly.

The setup stage of robotic welding (and many other manufacturing processes) is described in Figure 1-1.

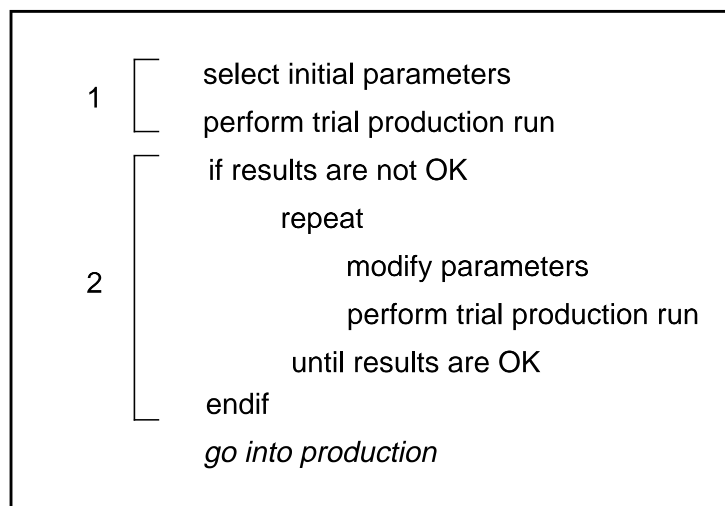


Figure 1-1 Typical setup procedure for a complex manufacturing process

Note that there are two distinct planning phases in the above setup procedure:

1. a planning phase, in which initial process parameters are selected and tried;

2. a replanning phase, in which process parameters are modified because the results of the trial production run were not satisfactory.

This results in two very different planning problems and makes robotic welding an interesting evaluation domain. The first problem relates to off-line planning, which is essentially a static design problem, with the design artefact being a proposed set of welding parameters. The second problem involves monitoring process conditions and adjusting process parameters in order to converge on a successful set of welding parameters. To date, computer assistance for setup has addressed the first, off-line, planning phase, typically using a knowledge based approach [1]. There has been no indication in the literature of significant work addressing the second phase.

The artefacts of this research include a framework for information presentation, a toolkit extension that implements the framework, and a prototype CPS software environment that applies the extension to a specific domain, i.e. robotic welding. The purpose of the prototype software environment is to evaluate how well the toolkit extensions support the low-level activities required in a reactive domain. Related issues that were deemed to be beyond the scope of this thesis include:

- what are the human factors issues of a CPS system;
- what is the ideal CPS software environment for robotic welding;
- how can the expertise required for robotic welding be reduced;
- to what extent can robotic welding setup be automated.

## 1.3 Thesis organisation

The first part of this thesis proposes the information presentation framework and the second part evaluates the framework in a reactive environment.

In the first part, Chapter 2 examines the background of cooperative problem solving, toolkits, and frameworks. Existing strategies in CPS are classified in order to determine which techniques need to be supported in reactive domains. Then the advantages of frameworks, toolkits (including InterViews), and toolkit extensions are explained, followed by a discussion of some existing toolkit extensions and frameworks. Chapter 3 describes the requirements of cooperative environments and defines the main artefact: the information presentation framework.

In the second part, Chapter 4 outlines the requirements of support for robotic welding. Chapter 5 describes the implementation of the toolkit extension and the prototype software environment based on the framework design. The success of the framework is evaluated as a way of applying cooperative software to a reactive decision environ-

ment. Finally, Chapter 6 discusses future work and enhancements to conclude the thesis.

# Chapter 2. Related Work

This chapter outlines the research problem of supporting reactive decision tasks and investigates why some researchers advocate a CPS approach. Section 2.1 describes the principles and benefits of CPS and discusses related work. Sections 2.2 and 2.3 describe software toolkits and frameworks, the mechanisms that will be used to support CPS. Some examples of these mechanisms are provided, along with a discussion of the reasons for choosing InterViews as the base GUI toolkit.

## 2.1 Cooperative problem solving

Cooperative problem solving is defined by Rettig [36] as a strategy in which "the software and the person using it are partners in the task at hand, bringing complementary strengths and weaknesses to bear." In practical terms, the nature of this partnership depends on both the application area and the user's expertise in that area. In this thesis, the user is assumed to have experience in monitoring and controlling a series of domain-specific events. The objective of the cooperative software developer should be to identify the likely strengths and weaknesses of both the human and the computer. Fisher [13] suggests that human strengths include using common sense, defining goals, and decomposing problems into sub-problems; the computer provides external memory, ensures consistency, hides irrelevant information, and summarises and visualises information.

The author supports the view that cooperative software techniques can offer significant productivity, reliability, or safety improvements in reactive decision tasks. Cooperative software can provide many levels of assistance to a human operator performing these tasks. Essentially, by taking over routine aspects of the task, cooperative software can reduce the operator's cognitive load - the demand on the brain to process and act on information. For example, a cooperative software environment may prompt the operator through the appropriate procedures for the current situation using a dynamic checklist. This allows the operator to concentrate on the ill-structured aspects of a task, those for which humans are better suited than machines.

A software approach to reactive decision tasks is especially useful when the option of improving existing hardware or equipment is not practical in the short term. For example, welding robots could be improved by replacing the existing interface to the controller with a graphical touch screen, but such equipment cannot be incorporated in a closed, proprietary system. In this case, improvements to the operator's tools can only be provided through software on a separate computer.

### 2.1.1 A new taxonomy

This section defines a new taxonomy of cooperative software techniques, based on the work of a number of different researchers who are applying CPS to reactive environments. Some researchers focus on the problems related to a particular application, while others are concerned with more general cooperative support issues. This thesis is concerned with generic support for reactive systems, so the work on domain-specific support is less relevant. Domain-specific approaches tend to employ rule bases or algorithms, usually with little consideration of the operator's strengths or weaknesses. Alternatively, the generic approaches tend to consider how the human and computer can best cooperate.

Rencken [34] refers to all forms of support as *adaptive aiding* and has classified a wide range of techniques according to the general strategy of reducing an operator's workload:

- *transformation* of the nature of the task;
- *partitioning* of different tasks to either human or computer;
- *allocation* of the same tasks to both human and computer.

Rencken's approach is to use allocation, in which the software is able to make a limited set of decisions similar to those made by the human. Activities are then allocated dynamically to the decision maker - human or computer - with the least workload.

Rencken classifies CPS approaches as partitioning. He therefore does not classify his own work as CPS, because allocation refers to the human and computer performing the same activities independently of each other. Instead, this thesis classifies a range of techniques in the literature according to the level of assistance provided by the software. This new taxonomy of cooperative techniques makes it possible to define a generic framework for low-level support. These classifications, shown in Table 2-1,

are listed in order from low-level (i.e. passive) assistance towards high-level (i.e. active) assistance.

**Table 2-1 Taxonomy of CPS techniques**

| *Technique* | *Proponents* | *Description* |
|---|---|---|
| Visualisation | Crawford [11]<br>Bennett [2] | Qualitative or quantitative information is represented graphically, which may include the use of three dimensional graphics or animation. |
| Multiple Views | Crawford [11]<br>Kempf [23]<br>Jarvis [22]<br>Wybo [48] | User is presented with different views of a situation, for example: view may vary in the point of focus, degree of detail or degree of abstraction. |
| Filtering | Wybo [48]<br>Villanueva [45]<br>Schwuttke [41] | Incoming data is restricted according to some criteria such as capacity, urgency or user needs - either dynamic or static filtering. |
| Highlighting | Nann [30] | Emphasis is placed on the most important information without suppressing the rest, for example: using visual or aural alarms. |
| Summation | Schwuttke [41] | Information is sorted or arranged in such a way that it provides a pertinent summary of high-level events. |
| Advising | Nann [30]<br>Harbour [17]<br>Spelt [43] | Complete or partial solutions are presented to the user for assessment. |
| Problem Structuring | Fisher [13]<br>Clarke [10] | Domain knowledge is used to impose structure on the problem, for example: presenting prompts and tools to the user as appropriate to the current stage of the task. |

As the form of assistance becomes more high-level, it tends to require more domain knowledge to be encoded in the software. Although cooperative software generally leaves the expertise to the human, it may be possible to identify and encode well defined aspects of the domain knowledge to reduce the overall workload of the operator. Even with advisory systems, there remains an emphasis on presenting suggestions in a way that assists operators and still gives them the final say.

The following sections describe examples of each type of technique; some authors describe work that fits more than one category.

### 2.1.2 Visualisation

Work in visualisation is generally concerned with operator interfaces to industrial or environmental tasks that are characterised by a potential for unexpected situations.

Crawford [11] proposes the Intelligent Graphical Interface Project which will experiment with various quantitative visualisation techniques for operator interfaces. The project will employ expertise from human factors, artificial intelligence, and computer graphics to determine the safest and most effective techniques.

Bennett's work in optimising data visualisation techniques [2] is more fully developed. His approach, *Representation Aiding*, is designed to get information to the operator more effectively using mimicry and abstraction. For example, mimicry might represent the flow-rate of liquid in a pipeline as moving bands of colour on a schematic diagram of the pipe. Abstraction involves representing low-level physical parameters as higher-level functional quantities. This approach also contains elements of summation, as described below. Bennett's aim is to determine the most effective ways to present information in complex, dynamic environments.

### 2.1.3 Multiple views

In work using multiple views, either the system or the operator determines which views of the current situation are the most appropriate.

The Intelligent Graphical Interface Project described by Crawford [11] proposes to investigate the use of multiple levels of data abstraction and different views of a system. Kempf [23] proposes a universal support environment with hierarchical views that provide increasing structural detail about the subject system. These are both examples of the operator controlling the view.

Jarvis [22] and Wybo [48] suggest that the system should have some degree of control over the viewpoint. Jarvis proposes that manufacturing support systems should consist of a range of tools and should take a workbench approach, presenting the most relevant view and the most appropriate tools to the users. However, users can proceed manually if they so desire. Wybo proposes the use of intelligent agents to monitor an evolving situation and keep the user informed. In this system, a dynamically changing *scene* combines different graphical formats, such as maps or schematics, into a single relevant view of the situation.

### 2.1.4 Filtering

Approaches to filtering - i.e. restricting certain types of data - range from restricting data at all times, to restricting it only when too much is arriving.

Wybo [48] proposes using intelligent agents to filter incoming data according to its relevance to the situation. The system combines a reactive database and a real time information manager to monitor the data.

Villanueva [45] proposes combining a real-time expert system with a decision support system to remove less important information using compression techniques.

Schwuttke [41] describes an interface that uses *Dynamic Trade-off Evaluation* to perform intelligent filtering under heavily loaded situations. The system determines the most important incoming data and restricts it, either by reducing the number of monitored channels, reducing the data sampling rate, or both. The remaining data channels are then rearranged dynamically and presented to the operator using domain-specific rules. This approach is similar to summation.

### 2.1.5 Highlighting

Highlighting is concerned with drawing the user's attention to critical information.

Nann [30] proposes a system which highlights the most important information about a detected failure, allowing the operator to assess and direct further study of the problem. Highlighting is implemented as a list of scenarios that can explain the current failure, with an associated confidence level to indicate the likelihood of each scenario. The list is then ordered by the system from the highest to the lowest confidence level.

### 2.1.6 Summation

Summation can take many forms, but generally involves producing a meaningful overview of the raw data.

In Schwuttke's Dynamic Trade-off Evaluation [41], if restrictions are imposed, then the system dynamically rearranges the visual grouping of the remaining channels that are presented. The data is presented textually in a compact format that allows comparison between different groups of parameters, for example: anomalous parameters can all be grouped together into an error log summary. This approach is similar to highlighting.

Bennett's Representation Aiding [2] uses a form of summation by combining related streams of information into a single geometric representation called a Configural Display. For example, a rectangle can be used to represent a combination of four variables which must remain within a larger rectangle representing the limits of those variables. This allows an operator to more easily monitor all variables at once.

### 2.1.7 Advising

The following advisory systems give suggestions, but all systems leave the final decisions to the operator.

Nann's advisory system [30] pre-processes the incoming data and devises solutions if possible, but also presents information that allows the operator to assess and direct further study of the situation.

Harbour [17] proposes a system that can monitor future task conditions and operator variables, then advise operations managers if these conditions are likely to lead to human error. This system can also suggest alternatives and allow the manager to choose a course of action.

Spelt [43] describes an advisory system that assists the operator in diagnosing alarms in complex systems. Spelt uses a hybrid architecture consisting of a self-learning neural net and an expert system which can be trained to detect patterns leading to alarm conditions, then explain the cause of the alarm to the operator.

### 2.1.8 Problem structuring

No problem-structuring approaches have been used in dynamic replanning tasks; however, two examples of support for design tasks have been reported.

Fisher [13] has devised a general architecture for domain-oriented design environments such as designing a user interface. These environments provide a work space, a palette of components for building the solution, domain-oriented checklists to act as a prompt, and an incremental specification editor to help define the problem along the way.

Clarke and Smyth [10] have developed ideas for a cooperative system which structures a problem differently from Fisher, based on principles of human-to-human cooperation. Rather than leading the user through a task, the system works on the same design task in parallel with the user. The system is not intended to produce good designs itself; instead, it has been shown to prompt users' imaginations and help them define and express their goals in a clear and structured way.

## 2.2 Toolkits

According to Gamma *et al* [14], a software toolkit is "a set of related and reusable classes designed to provide useful, general-purpose functionality." Toolkits are often object–oriented, because object-oriented methodologies, particularly the inheritance mechanism, have been shown to be well suited for the development of toolkits [26]. Toolkits enable programmers to avoid "reinventing the wheel" by providing solutions to common programming problems. For example, a graphical user interface toolkit defines widgets - the building blocks of a user interface - such as buttons, scroll bars, windows, and menus. The application programmer can compose the classes into systems or derive more specialised classes from them.

A GUI toolkit may also use a Widget Level Theory [40] to encourage consistent interfaces. A Widget Level Theory refers to a model for both the appearance of a set of widgets and how they should be applied for best user performance. For example, a

model might specify layout appropriateness in terms of placing frequently used widgets together and using a left-to-right sequence in keeping with the task sequence description. The result is that the “look and feel” of an application is derived from the widget set. One existing Widget Level Theory is built into the Motif [12] widget set. Consistent interfaces built from a widget set can help users learn and use a suite of applications more easily, while the interfaces remain flexible enough to satisfy a variety of individual preferences.

### 2.2.1 Toolkit extensions

A toolkit extension is a relatively small number of related classes derived from a toolkit for a particular purpose which can be reused by other programmers. Extending an existing toolkit is easier than building a complete, new toolkit and allows a developer to take advantage of relevant features of the base toolkit. Toolkit extensions can encapsulate behaviour that addresses a more specific application than the original toolkit.

### 2.2.2 Why choose InterViews?

InterViews version 3.1 [27] is a graphical user interface toolkit written in C++ that encapsulates the X Windows System [39] graphical toolkit. InterViews has several advantages as a base toolkit for CPS.

InterViews is object–oriented, so it allows extensions to be easily constructed using inheritance. InterViews also emphasises the separation of the user interface from the application code, which makes it possible to modify the interface or the application independently.

Like most GUI toolkits, InterViews contains many predefined components and a variety of composition mechanisms, so it offers a convenient way to implement complex user interfaces - or to implement simple ones quickly. This makes InterViews suitable for developing proof-of-concept prototypes. One such component is the FileChooser (Figure 2-1), which is a widget that facilitates browsing and managing a file system. Another useful component is the IOHandler, which features the detection of data arriving at ports and the control of timing loops.

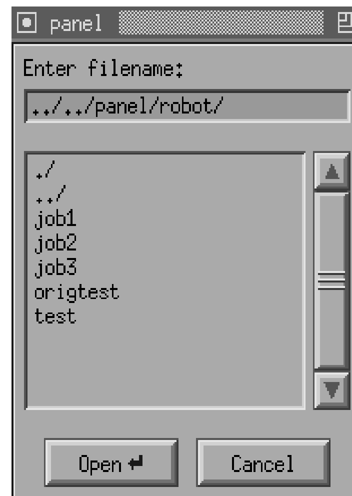


Figure 2-1 InterViews FileChooser

One of the more useful base classes available in InterViews is the *Glyph*, which is a visible representation of data. Glyphs are composed using a geometry model based on the TeX [24] boxes and glue model. In this model, both the glyphs (equivalent to boxes) and the "glue" (invisible layout glyphs) that bind them together are flexible, so an entire composition - the glyph hierarchy - can be stretched and shrunk dynamically to accommodate window resizing and data updating. Updating the top glyph of a hierarchy automatically draws all glyphs from which the top glyph is composed. Glyphs are a firm base from which to develop cooperative interface components because they provide simple but powerful mechanisms for display, composition, and extension.

### 2.2.3 Existing toolkits

To date, there have been no toolkit approaches to CPS reported in the literature. InterViews has been used to build research prototypes in other domains, for example: a diagram editor [7], a document editor [8], and a groupware toolkit [37].

Commercial software libraries are available that can perform some information processing aspects of CPS. At least one library, ILOG *Rules* [20], addresses real time data processing. However, the data collected by a *Rules* application is intended to be used for inference rather than data presentation. Another ILOG product, *Views* [21], is a graphical interface library suitable for operational data monitoring which defines data display widgets with external event handling capabilities. *Rules* is a rule base that generates C++ code, while *View*s is a library rather than a toolkit; both products offer fixed functionality, so neither product is suitable for explicit extension to support CPS. However, both products would be a useful source of specialised components for a cooperative software environment.

InterViews is extensively used within the computer science research community. One comprehensive example of an extension is Roseman's real-time groupware toolkit [37]. The requirements for a useful groupware application, along with the associated software techniques, have been encapsulated in the extension classes. Roseman recognises the benefits to the resulting application of basing it on coherent and well principled components.

## 2.3 Frameworks

Gamma *et al* define a software framework as "a set of cooperating classes that make up a reusable design for a specific class of software. The framework dictates the architecture of an application." A framework can form the design of a subsystem, which is a self-contained part of a system that performs a set of related functions.

The key benefit to a programmer is that the framework predetermines the way in which components will work together, i.e. the links between them, not just the functionality of the individual components. The link between one class and its associate can be defined as a member function of the class, or as a member function of a third, controlling class.

A framework allows a programmer to concentrate on the specific needs of an application, since the design decisions common to the application domain have already been made. A disadvantage to this scheme is the loss of flexibility in cases where the predetermined design does not suit a particular application. However, a good framework should be flexible and extensible enough to cover a large subset of similar applications. As stated previously, whereas toolkits are about reusing code, frameworks are about reusing designs.

### 2.3.1 Framework example

An example of a framework is described by Pree [33]. Pree likens different types of component in a framework to specially shaped plugs. Each plug is designed to fit into the matching interface "shape" on another plug. Any components that are subclasses of these plugs will inherit the plug's shape and be able to fit into the interface without trouble. The example application is a basic electronic mailing framework. Three of the classes defined in the framework are Mailer, Employee, and DesktopItem. Subclasses of Employee represent specific types of employees, such as Manager and Secretary. Subclasses of DesktopItem may include TextDocument and DrawingDocument. A member function such as TransferItem is defined in the Mailer class as:

```
TransferItem(DesktopItem* item,
                   Employee* sender,
                   Employee* receiver)
```

This member function mails the given item of type DesktopItem from the sender to the receiver, both of which are of type Employee. This member function will not need to be changed in order to deal with subclasses of either DesktopItem or Employee. It will work with all current subclasses of these two classes, such as DrawingDocument, Secretary, and Manager, as well as all future subclasses. The Mailer implements a generic mailing system that does not deal with specific types of component, but with the abstractions Employee and DesktopItem.

This example illustrates the mechanism of defining the "plug" interface between classes of a framework. Once a member function is defined for a base class, any of its future subclasses can use this inherited function's interface to pass messages of the same type. In addition, if any base classes are defined to be passed as parameters of this member function, they are considered part of the interface. So the function will also work with future subclasses of the parameter classes, because the interface recognises their "shape".

### 2.3.2 Existing frameworks

Fisher describes a generic architecture for domain-oriented design environments that employs a problem structuring approach to CPS. Design can be a complex, ill-defined decision task, in which a "good enough" solution to a problem is being sought. But design differs from dynamic replanning, since it deals with assessment criteria that are difficult to quantify. Therefore there is less emphasis on measuring situation data and more emphasis on satisfying rules.

The architecture consists of five main components:

- a *construction kit* for direct manipulation of the design, including a palette of items;
- an *argumentative hypertext system* containing issues, answers, and arguments about the design domain;
- a *catalogue* of pre-stored designs;
- a *specification component* which allows a specification to be described in incremental stages by the user;
- a *simulation component* that does "what-if" analyses.

While this architecture is not an object-oriented framework as defined by Gamma, it does describe a reusable design for cooperative software environments. Problem structuring is implemented in the construction kit by specifying sub-tasks and checklists and by prompting users through the design process. Other cooperative techniques are implied in the architecture. For example, the construction kit uses visualisation by

presenting an appropriate depiction of the design, such as the plan view of a kitchen. The specification component uses the partial specification to filter information from the hypertext argumentation component. The simulation component presents multiple views of the artefact being designed.

Fisher focuses on problem structuring in the design task. In contrast, the work described in this thesis focuses on information presentation in dynamic replanning, particularly the flow and processing of the information.

## 2.4 Summary

This chapter has discussed CPS and the mechanisms proposed to support CPS in a reactive environment. A new taxonomy of CPS techniques is defined based on a study of existing CPS work in reactive environments. An object-oriented toolkit solves common problems for application developers; a toolkit extension can solve more specific problems while retaining the features of the base toolkit. A framework can provide a reusable design for application developers of cooperative systems. By implementing the framework as a toolkit extension, a common solution for many similar application domains is available to developers that should improve the quality of applications while reducing the effort of development. The following chapter describes a framework that supports CPS.

# Chapter 3. Cooperative Problem Solving Framework

This chapter describes the main artefact of the thesis: a framework that is a reusable design for the information presentation aspect of CPS. Section 3.1 begins with some requirements of a cooperative software environment based on the work presented in section 2.1. Then section 3.2 explains which requirements can be met by the Inter-Views toolkit and which requirements demand a toolkit extension. This is followed in section 3.3 by the functional requirements for the object-oriented toolkit extension, then the design of toolkit classes that will meet these requirements. These class designs, together with the member functions for combining them into an information presentation subsystem, constitute the framework.

## 3.1 Cooperative software requirements

The aim of a cooperative software environment is to make optimum use of both the operator's and the computer's capabilities. The developer needs to separate the well defined activities of a dynamic task from those that require human expertise. The set of requirements summarised in Table 3-1 is intended to encourage this separation, by each addressing a well defined aspect of a typical dynamic task. The requirements are generally complementary, and some requirements overlap with others.

**Table 3-1 Summary of requirements for cooperative software**

| *Requirements* | *Benefits* |
|---|---|
| graphical interface | ease of learning and use |
| data retrieval and presentation | decision making support |
| information highlighting | draws attention to important information |
| information storage | later review or reference |
| integration of tools | ease and consistency of use |
| abstracted functionality | focus on task, not tools |
| cases repository | access to past solutions |
| extensibility | most appropriate tools |

**A Graphical interface** is important for ease of learning and use. The operator should be presented with all controls appropriate to the current task, rather than being expected to remember and type commands. To achieve this, a graphical, pointer driven interface is much more suitable than a textual display. In addition, the appearance of each control element should remind the operator of its purpose. For example: a control element may be a scroll bar used for navigating through the pages of a document. There

are many well documented design strategies for graphical interfaces. For instance, menus should be shallow rather than deep (no more than three levels are recommended [40], p110) and should be supplemented by shortcuts for experienced users, such as buttons or type-ahead hot keys ([40], p119). Ease of use translates to faster user reaction, while ease of learning is helpful for domain experts who may not be familiar with computers. A keyboard may still be necessary for information input. The pointing device may be a mouse or a trackball, although a touch screen is often more suitable for factory floor conditions.

**Data retrieval and presentation** - such as equipment status or the results of trials - is required as a basis for analysis and decisions. The cooperative software environment should obtain data from machines and acquisition devices, process it, and present it during operation. The operator should have flexible retrieval and presentation and have information from different sources accessible in one place. Some examples of presentation devices include bar histograms, three dimensional contour graphs, needle gauges, and flow meters. The use of animation to represent change can be an effective way to communicate status information [2].

**Information highlighting** draws the operator's attention to important information and frees them from having to monitor easily defined conditions in the presented information. The operator's expertise can then be focused more effectively on information processing that is less well defined. Highlighting may involve audio alarms, visual colour highlighting, or other means of drawing the operator's attention to the most important information as it arrives. There may be a variety of strategies for determining what is the most important information, such as predefined user-specified rules, expert systems, or adaptive filters.

**Information storage** enables the operator to review incoming data immediately after the event or activity that generated it. The cooperative software environment should store incoming data in either raw or processed form and provide a replay function. Data often arrives quickly or arrives when the operator is busy, so an immediate review process can supplement direct observation. Ideally the replay function would include variable speed settings for slowing down or speeding up the presentation of data, along with other enhancements that follow the video player metaphor, such as fast forward and rewind.

**Integration of tools** is another requirement for ease of use. The operator should not have to deal separately with external software tools to perform the task. Instead, the cooperative software environment should integrate access to all the necessary software and hardware tools within one consistent interface. This requirement makes the cooperative software environment analogous to an integrated development environment

such as Borland C++ [6], which provides an editor and a set of tools that support the interactive procedure of writing, compiling, and debugging C++ code. The window manager of a graphical operating environment makes integration easier to implement.

**Abstracted functionality** refers to a functional rather than physical view of the system's controls, i.e. the operator should be presented with high-level functional controls that relate to the problem solving task rather than the underlying tools. To achieve this, the cooperative software environment should perform as much low-level processing as possible, while hiding the details from the operator. The operator brings domain expertise to bear and uses the high-level controls to solve the problem. An example of abstracted functionality is to define a button or a keystroke macro to perform a function such as "replay last session" that requires several underlying commands.

**A Repository of cases** provides a source of experience for the operator in the form of previous problems matched with successful solutions. Previous solutions are a useful starting point for problem solving. Cases should be categorised by the characteristics of the problem they addressed. A case repository may be as simple as a keyword search of data files.

**Extensibility** enables the addition of both software and hardware components to the application. The operator should be presented with the most appropriate tools, whether to cover a variety of application domains, a range of equipment types, or to add new tools when they become available. Any equipment control subsystem should be weakly coupled with the rest of the system, i.e. connected by a narrow, well defined interface, so it can be easily replaced to suit other types of equipment. Extending the cooperative software environment to more than one hardware platform or operating system is also desirable and can be readily achieved within a cross-platform environment such as X Windows [39].

## 3.2 Information presentation requirements

The advantage of extending a GUI toolkit to support cooperative software is that many of the requirements in Table 3-1 can already be met by the existing toolkit and its underlying window manager and operating system. For example:

- the graphical widgets of a GUI toolkit provide basic components for building easy-to-use interfaces;
- the operating system can perform basic data storage;

- equipment control and tool access can be integrated by means of the window manager and by making all user controls available within one interface;
- abstracted functionality is assisted by InterViews, which provides high-level control widgets that hide the underlying commands, but will still depend on the way the application program is developed;
- a repository of cases can be provided either by basic operating system commands (e.g. data files and a keyword search) or by a stand-alone database that can be invoked as a system command within the interface.
- X Windows toolkits are all extensible and GUI toolkits in general are extensible, since they must be extensible to be useful; extensibility also depends on a good overall design;

Table 3-2 indicates which requirements are already satisfied by the base toolkit and its environment. The only requirements that demand a toolkit extension are the presenta-

**Table 3-2 How cooperative software requirements are to be met**

| *Requirements* | *Met by* |
|---|---|
| graphical interface | InterViews |
| data retrieval and presentation | InterViews extension |
| information highlighting | InterViews extension |
| information storage | operating system |
| integration of tools | operating system,<br>window manager,<br>InterViews |
| abstracted functionality | InterViews,<br>cooperative application |
| cases repository | operating system |
| extensibility | InterViews,<br>good design |

tion and highlighting of situation data in real time. The target cooperative application must be able to handle a potentially complex, dynamic, multimedia environment in which many different information sources need to be assimilated and acted upon. Overall, the cooperative application needs to coherently integrate disparate sources of information relating to a complex task.

The InterViews toolkit already provides classes for detecting the arrival of external data and for displaying static data and binary indicators. However, data retrieval and presentation involves other activities, such as: communicating with an external acquisition device, reading and interpreting the data, combining data from different sources, processing it, then displaying it dynamically in an appropriate format. Therefore, the InterViews toolkit must be extended to provide these information presentation capabilities.

The following list of requirements defines the flow and processing of multiple streams of data, from the external sources to the screen. A data stream is a series of values arriving over a period of time representing a single measured quantity. An information presentation subsystem should be able to:

- retrieve data from multiple communication ports;
- retrieve multiple streams of data through each communication port;
- apply a filter to a data stream;
- apply a calculation process to a data stream;
- combine streams using a calculation process;
- apply a limit alarm to a data stream;
- display a data stream in real time, i.e. with no noticeable delay to the operator;
- display a single data stream in more than one format simultaneously;
- provide different display techniques;
- store data, either in raw or processed form, as it is retrieved;
- replay stored data to the display devices;
- allow the user to start, stop, and replay a data presentation session;
- allow the user to choose processing and display techniques.

The next section describes the design of several component types that fulfill the above requirements for handling data.

## 3.3 Information presentation framework

The following framework is a reusable design for information presentation which supports the activities involved in transforming data into information. The framework is designed to be implemented as an object–oriented toolkit extension, which makes

possible the inheritance and specialisation of components. The framework contains the following components:

- *DataHandler* (abstract class, derived from IOHandler) links data streams to DisplayHandlers and optional Filters; retrieves and interprets data streams from a data acquisition device via a port; applies any given Filters then passes each data stream to the linked DisplayHandler(s);
- *Filter* (abstract class) removes data values from a data stream that match the given removal specification;
- *DisplayHandler* (concrete class) links data streams to Displayers and optional Processors; applies any linked Processors to incoming data streams and passes the result to linked Displayer(s);
- *Processor* (abstract class) performs a calculation on a data stream and/or merges two streams;
- *Displayer* (abstract class, derived from MonoGlyph) displays the values in a data stream dynamically.

This framework was designed to be implemented by any object–oriented GUI toolkit. The only assumption it makes about the toolkit is that primitive graphical widgets are available as building blocks for creating display components. Figure 3-1 shows the relationship between each component of the framework using a relational notation adapted from Rumbaugh [38]. The relational links indicate how many objects of each type can be linked to its associate. For example, a DisplayHandler object can be linked to many Displayers, but each Displayer object receives data from only one DisplayHandler.

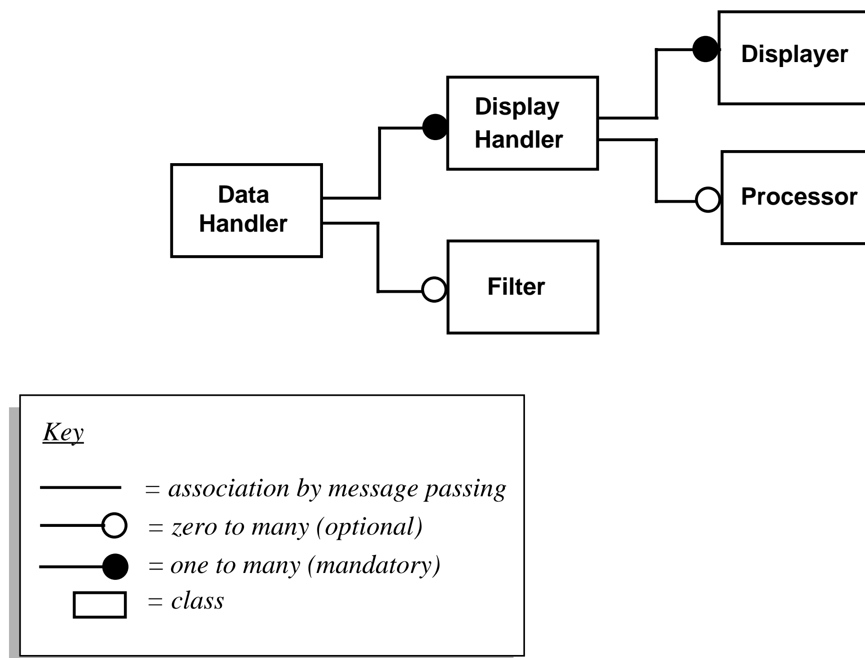


Figure 3-1 Information presentation framework design

The information presentation components (classes) have been designed to allow some flexibility in the retrieval, manipulation, storage, and display of feedback data. They can be combined in various configurations to suit particular application needs. The communication between components is one-way. For example, the DataHandler sends a message containing the next data value to the DisplayHandler by means of the DisplayHandler's `update` member function.

### 3.3.1 Framework components

All core framework components except the DisplayHandler are *abstract classes*. Wirfs-Brock [47] defines an abstract class as one designed to factor out behaviour common to a variety of classes, not to produce instances of itself. This encapsulates the class's behaviour for easy maintenance and allows the behaviour to be reused by means of inheritance. A *concrete class* is a class designed to be instantiated, so it can be used directly in an application. Abstract classes often define member functions that only implement default code (if any) and a function interface; these functions need to be specialised within subclasses to suit a particular application. As described in Pree's example framework in section 2.3.1, a member function is a way of defining a generic interface, or "plug shape" between the components of a framework that is inherited by all subclasses of these components.

DisplayHandler is defined in the framework as a concrete class because its behaviour (handling Displayers) does not depend upon the application. DataHandler is defined as an abstract class because its descendants need to encapsulate specific details about the data acquisition device. Filter, Processor, and Displayer classes are abstract because their descendant classes need to perform specific information presentation functions.

The framework components transfer data streams by message passing: DataHandler invokes the DisplayHandler's `update` member function when new data is available and has been filtered; DisplayHandler invokes the Displayer's `update` member function whenever new data is available and has been processed. Each DataHandler maintains a list of DisplayHandlers and Filters, while each DisplayHandler maintains a list of Processors and Displayers.

In the following description, IOHandler and MonoGlyph are InterViews classes which form base classes for DataHandler and Displayer respectively. The core framework components are described more fully in Appendix A. The term *class responsibilities* is defined by Wirfs-Brock to mean actions the class is expected to perform, or knowledge it maintains.

**DataHandler (abstract class, derived from IOHandler)** links data streams to DisplayHandlers or to storage files and handles the data from a port. There are no DataHandler methods defined for requesting and interpreting data from a data acquisition device, so specialised subclasses with this functionality must be defined to communicate with specific devices.

DataHandler's inherited responsibilities are to:

- accept notification of data ready on its port;
- accept notification of timer expiry, and to reset the timer.

DataHandler's specialised responsibilities are to:

- know the name of the port to which it is attached;
- link one or two named data streams to a Filter (optional) and to DataHandlers;
- start retrieving data from the data acquisition device through the port and pass it to linked Filters and DisplayHandlers;
- if specified, save retrieved data streams to a logfile;
- stop retrieving data;
- start replaying logged data the same way as live data streams;
- stop replaying logged data.

**Filter (abstract class)** removes unwanted values from a data stream. Concrete Filter subclasses need to be derived from it, for example, a clipping filter to remove transient values occurring above or below a realistic limit. Filter's responsibility is to:

- accept a data value and return either the value or a substitute value according to a filter specification.

**DisplayHandler (concrete class)** channels a data stream to Display devices. Links can be created between data streams, Processors, and Displayers. Data streams can be merged using Processors and can be sent to multiple Displayers. One DisplayHandler is needed for each data stream. DisplayHandler's responsibilities are to:

- link a named data stream from a DataHandler to Processors (optional) and Displayers;
- update any linked Processors and Displayers with either one or two data streams.

**Processor (abstract class)** performs some calculation on a data stream and /or merges two streams. Concrete Processors such as a multiplier need to be derived from it. Processor's responsibility is to:

- accept either one or two data values and perform a calculation on it/ them, then return the result.

**Displayer (abstract class, derived from MonoGlyph)** displays the values in a data stream.

Displayer's inherited responsibility is to:

- draw itself on the screen.

Displayer's specialised responsibilities are to:

- update itself with a given value;
- clear itself.

### 3.3.2 Examples of derived components

This section outlines six examples of derived (non-core) components, as illustrated in Figure 3-2. These components are described in more detail in Chapter 5 and Appendix B.

- *WeldMonHandler* (concrete class, derived from DataHandler) links data streams to DisplayHandlers or to storage files and handles the data from a welding monitor;

- *ClippingFilter* (concrete class, derived from Filter) removes values that fall below the filter threshold from a data stream;
- *Multiplier* (concrete class, derived from Processor) multiplies each value from one data stream with the corresponding value in another stream, thereby merging the data streams;
- *Readout* (concrete class, derived from Displayer) displays a data stream in the form of a numeric gauge to a given number of decimal places;
- *Histogram* (concrete class, derived from Displayer) displays a data stream as a dynamic, vertical bar histogram;
- *Alarm* (concrete class, derived from Displayer) lights up a visual indicator whenever a data value exceeds the specified alarm limit.

### 3.3.3 Class hierarchy

Figure 3-2 shows the relationship between the InterViews classes, the core framework classes and the example subclasses derived from them. The InterViews classes (in italics) are IOHandler and MonoGlyph. The core framework classes (in bold type) are DataHandler, DisplayHandler, Displayer, Filter, and Processor. The examples of subclasses are WeldMonHandler, Histogram, ClippingFilter, and Multiplier. The question marks indicate that further specific classes may need to be derived from the core classes according to the application's requirements.

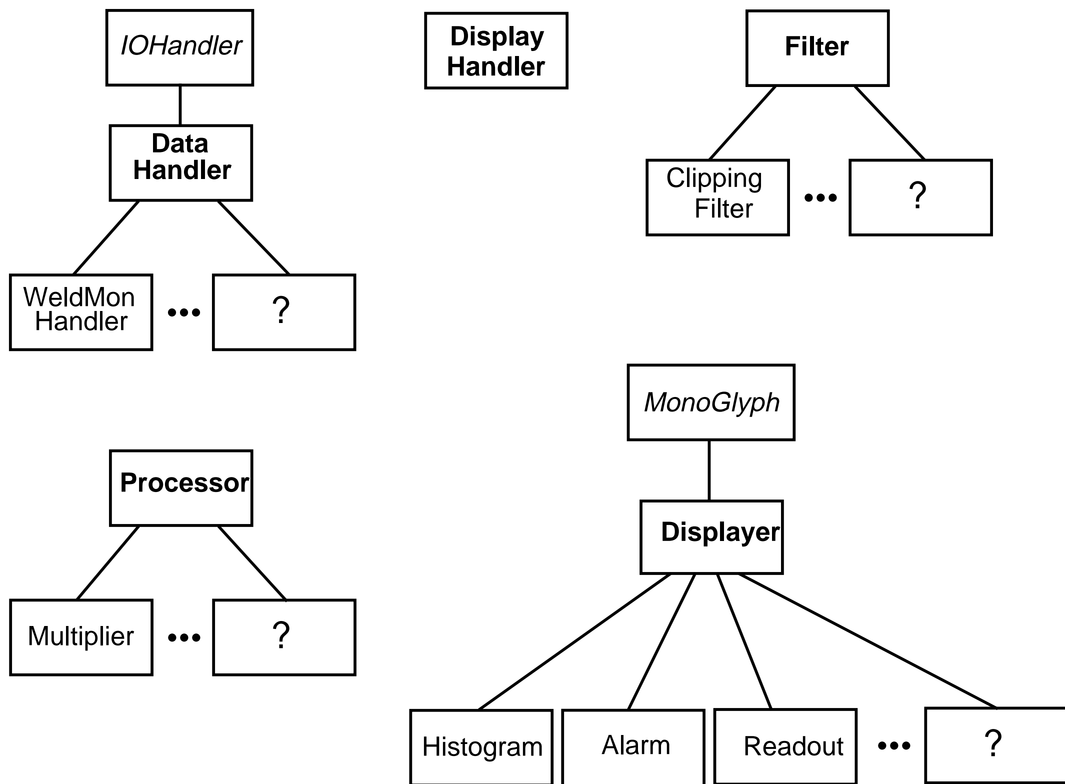


Figure 3-2 Information presentation class hierarchy

### 3.3.4 Sample configuration

Figure 3-3 shows a sample configuration of the above components in which multiplexed data streams from a single port are extracted and may then be merged by processing with other streams. An initial five data streams are combined into four resultant data streams (represented by four DisplayHandlers) that are displayed in seven Displayers of various types.

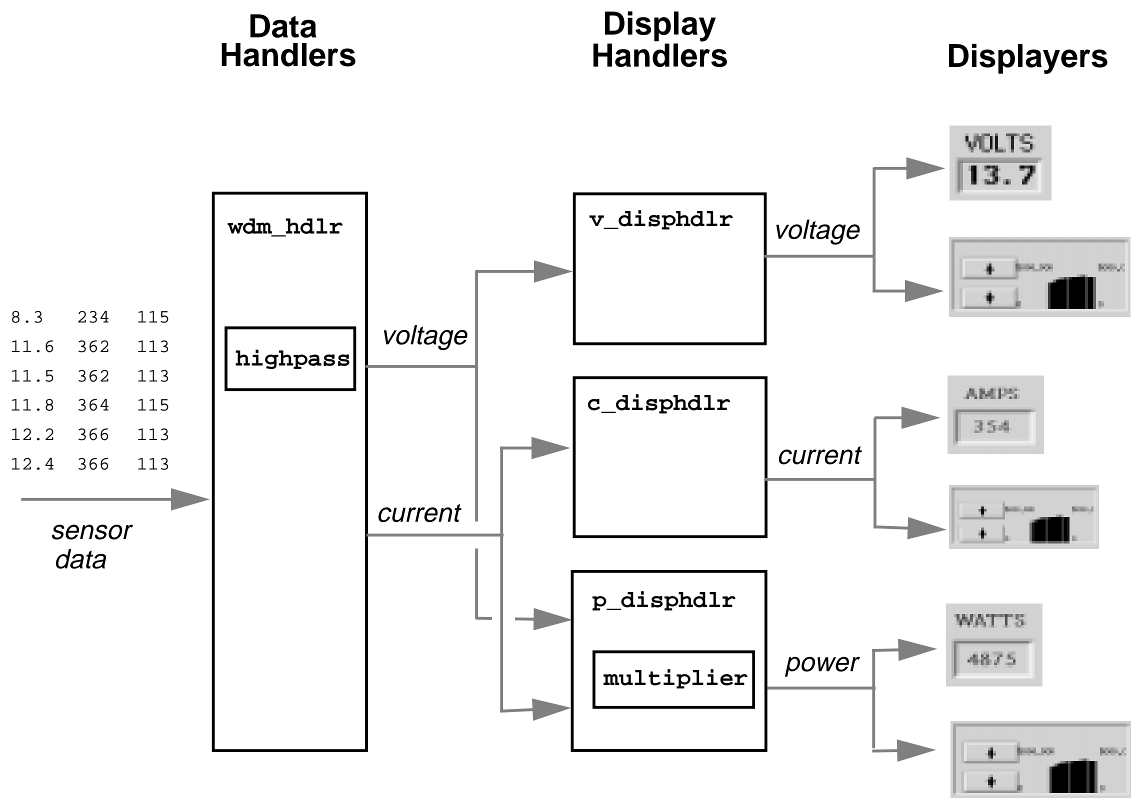


Figure 3-3 Composition of a sample information presentation subsystem

### 3.3.5 Cooperative problem solving techniques

This section explains how the framework helps the development of a cooperative software environment. This includes a description of how each of four different cooperative techniques are encapsulated in the framework.

**Visualisation** is potentially a more direct way of communicating information to the operator, provided no important information is lost. The Displayer components convert a stream of numbers into a visual representation of a physical quantity or quality. Displayers also enable different representation methods to be used for different purposes. For example, a histogram shows previous trends in the values of a parameter, while a needle gauge shows the parameter's current position within the allowable range of values. The ability of the DisplayHandler to send a data stream to different Displayers simultaneously enhances this visualisation, by allowing a single stream to be displayed in several forms at the same time.

**Filtering** removes unwanted forms of data. A Filter derivative can apply a filtering function to any incoming raw data stream to remove transient data - where such restrictions can be well defined in advance. For example, noise in the form of data acquisition

interference can obscure the desired data set, so a clipping filter can screen individual values which vary too much from the surrounding values. The benefit for users is a removal of extraneous data, which reduces the total amount of data to be interpreted.

**Summation** aims to provide a useful overview of raw data. There are several aspects of summation, as discussed in section 2.1.1. The ability of Processor objects to merge data streams provides a combinational aspect of summation. Related data streams can be merged then displayed as a single entity - reducing the amount of data while increasing the amount of information. In addition, different parameters can be viewed simultaneously for direct comparison, for example, the x-axes of several histograms can be aligned to provide a temporal comparison between the parameters. The benefit to users is an overall reduction in cognitive load, similar to that provided by filtering.

**Highlighting** draws the user's attention to important information. An Alarm can be associated with any data stream (which may be a combination of data streams) to light up when a value is out of range. Alarm highlights excessive parameter values during welding, such as high temperature, so the operator can concentrate on watching for less obvious fault conditions.

## 3.4 Summary

This chapter has defined a set of requirements for a cooperative software environment, many of which can already be met by InterViews, an OS or a window manager. Information presentation and highlighting are the only requirements that need to be met by a framework. The aim of the framework is not to explore novel visualisation techniques, so it does not define any new display devices. Nor does it define the most effective display and interaction techniques for a cooperative system. Instead, it is a reusable design for an information presentation subsystem. The framework components can be linked in particular ways to pipe data streams from data acquisition devices to the screen. The framework assists developers by defining a simple, coherent set of building blocks that can be assembled flexibly into a variety of information presentation subsystems. The following chapter describes the application domain in which the framework is to be validated - robotic welding setup.

# Chapter 4. Cooperative Software Application

This chapter discusses robotic welding as a suitable domain in which to validate the information presentation framework defined in Chapter 3. The primary source of information relating to robotic welding has been the Welding Group, Commonwealth Scientific and Industrial Research Organisation (CSIRO), Division of Manufacturing Technology (DMT), Adelaide. CSIRO DMT also provided the evaluation environment.

The framework is to be validated by extending the InterViews toolkit, then using the toolkit and the extension to build a prototype cooperative software environment. Section 4.1 explains why robotic welding setup was chosen as the testbed application and section 4.2 examines other work that addresses robotic welding setup. Sections 4.3 and 4.4 describe the existing setup environment and outline the requirements for a CPS software environment.

## 4.1 Small batch robotic welding

Setting up a robot to perform welding is an ill defined decision task that illustrates the problems faced by operators of complex, reactive systems. Setup is also an information-rich domain that comprehensively tests the cooperative software framework. The evaluation subject was based on a Motoman K6S robot (developed by Yaskawa Electric Corporation). The robot arm has six degrees of freedom and is controlled by a Yasnac ERC robot controller [49]. The Yasnac ERC is typical of most robotic welding controllers currently being used in manufacturing and can be described as a first generation robot. Welding jobs are performed by executing a procedural program on the controller. The main problem with controllers of the Yasnac ERC generation is that setup for small batches is very resource-expensive relative to the returns of a small number of products, hence small batches are often not economically feasible. This section explains why setup is resource-expensive.

### 4.1.1 Welding process parameters

The welding process is controlled by adjusting variable inputs to affect outputs. The three types of input parameter are:

1. *primary manipulable* - directly adjustable, for example: arc current, arc gap size, torch speed;
2. *secondary manipulable* - indirectly adjustable by employing empirically known relationships, for example: arc voltage, heat input;

3. *fixed* - parameters which cannot be adjusted during program tuning, for example: material, joint type, and joint gap width.

Output parameters, which include joint hardness, brittleness, and tensile strength, combine to characterise the quality of the welded joint.

There are many input parameters and some are closely coupled, so relationships between inputs and outputs can be non–linear and complex. Initial selection is usually guided by reference to published data such as material codes and standards. Specific procedural information is also available from a Welding Procedure, which is a formal record of the parameters that produced a desired result for a given joint configuration [31]. Welding Procedures document the steps necessary to achieve repeatable weld quality.

In addition, primary welding parameters such as torch speed need to be adjusted in-process to avoid fault conditions and maintain consistency under variable welding conditions. Whereas other manufacturing tasks (such as assembly of parts) are reasonably deterministic, welding parameters change dynamically and unpredictably. For example, the base metal expands as it heats, causing the joint gap to change width and require more or less filler metal to compensate. A human welder will use a combination of hand, eye, ear, and touch coordination along with empirical experience to monitor and control the welding process dynamically.

### 4.1.2 Welding robot setup

In order to set up a robot for welding, an operator is required to:

- write a program in the robot's procedural language that specifies the initial welding parameters;
- specify the motion path to be followed by the welding torch.

The problem being addressed in this thesis relates to programming and parameter tuning rather than path specification. Path specification is a problem domain in its own right [9],[43] in which the problems tend to be hardware related. Parameter tuning is an information rich, reactive task to which CPS can be readily applied.

At present, commercial robots are dull-witted assistants to humans: every action - including dynamic corrections - must be planned and programmed. At best the robots can be programmed to use sensory input to adapt their actions during a task, so they are a long way from being independent of their human operators. The more complex the task, the more dependent the robot and the more difficult it is to program; welding is a complex and indeterminate task.

The welding robot operator needs an understanding of both the welding process and the setup requirements and is required to interpret a large amount of feedback information from different sources during setup. Each new batch job may involve different materials, shapes, and welding methods, so an iterative trial and error procedure is the only way to devise a working program. Setup requires the labour of an experienced and hence expensive human operator, removes the robot from production, and uses consumable welding materials: the longer setup takes, the more expensive it is. Therefore, while domains such as power generation impose safety-related time constraints on operators, robotic welding setup imposes an economic time constraint. The expense of setup has mainly restricted the use of seam welding robots to large batches, in which the cost and time of setting up can be amortised over a large number of products.

The aim of applying CPS is not to reduce the amount of expertise required to perform setup or to determine how well it can be automated. The key problem is how to reduce setup time for an experienced operator, thereby making it more economical to use welding robots for small batches.

## 4.2 Relevant work in robotic welding

This section describes work that addresses full or partial automation of small batch robotic welding. There is also a discussion of work related to cooperative problem solving in the robotic welding domain.

### 4.2.1 Autonomous approaches to setup

Much of the work that addresses robotic welding is designed to make the next generation of robots more independent of the human operator. For example, automated process control involves dynamically controlling and correcting the welding parameters, the motion path, or both, using sensory feedback [3],[16],[28],[46]. Some researchers are developing mathematical models of the process [44] or encoding process knowledge as rules [42] to assist closed loop control with sensors.

However, first generation robots represent large investments for many manufacturers and will remain in use for years to come, so there is a need for tools to make them easier to use. Human operators need software support, since the hardware cannot be replaced in the near term.

### 4.2.2 Support approaches to setup

There is another body of work that recognises the short term need for human involvement and attempts to provide new tools for use in setup. A common approach is the use of expert systems to advise on process parameters [1], [32], however these are standalone, off-line systems which are often too general, providing knowledge over a range

of welding techniques. One exception [18] is an attempt to integrate a parameter database with off-line programming simulation in order to verify the program more completely. So far, no on-line knowledge base support has been provided. In general, there is a lack of work which specifically addresses the difficult procedure of specifying and verifying process parameters.

### 4.2.3 Cooperative approaches to setup

The only on-line cooperative support for robotic welding setup reported in the literature so far has been that of Reilly [35], which is a domain-specific approach. The system consists of a Macintosh personal computer (PC) running a customised software package that is capable of analysing and displaying real-time data from the welding process. Although Reilly is employing data presentation, the short term aim is to eliminate the need for post-weld inspection rather than to reduce the time taken during setup. Reilly further intends to incorporate control of the robot through the Macintosh-based software. The long term aim is to use the data collected to create an intelligent database and decision *making* module that uses rule based methods, not to support robot programming by an operator.

There are also several commercial packages, such as LabWindows [25] and Matlab [29], that provide real-time data visualisation facilities, which give operators some of the advantages discussed in section 2.1. LabWindows has been used to display robotic welding process data[1]. LabWindows provides tools for building virtual instrument panels, such as a GUI graphical editor and a library of predefined instrumentation widgets which can process and display data. Although these packages are able to provide a solution for robotic welding setup, they cannot be generalised easily to suit a range of similar applications. They provide tools rather than a design. A framework has the advantage of providing a generic design solution to the problem of manufacturing setup and other reactive tasks.

## 4.3 The existing welding environment

The welding environment used to evaluate the cooperative framework is located at the CSIRO DMT. The environment consists of the Motoman welding robot, the Yasnac ERC controller [49], a teach pendant (a portable panel which is connected to the controller and can be used to move the robot arm), and a Welding Data Monitor (WDM). The WDM was developed by CSIRO DMT to acquire process data during welding.

[1] Welding Group, CSIRO DMT, Adelaide.

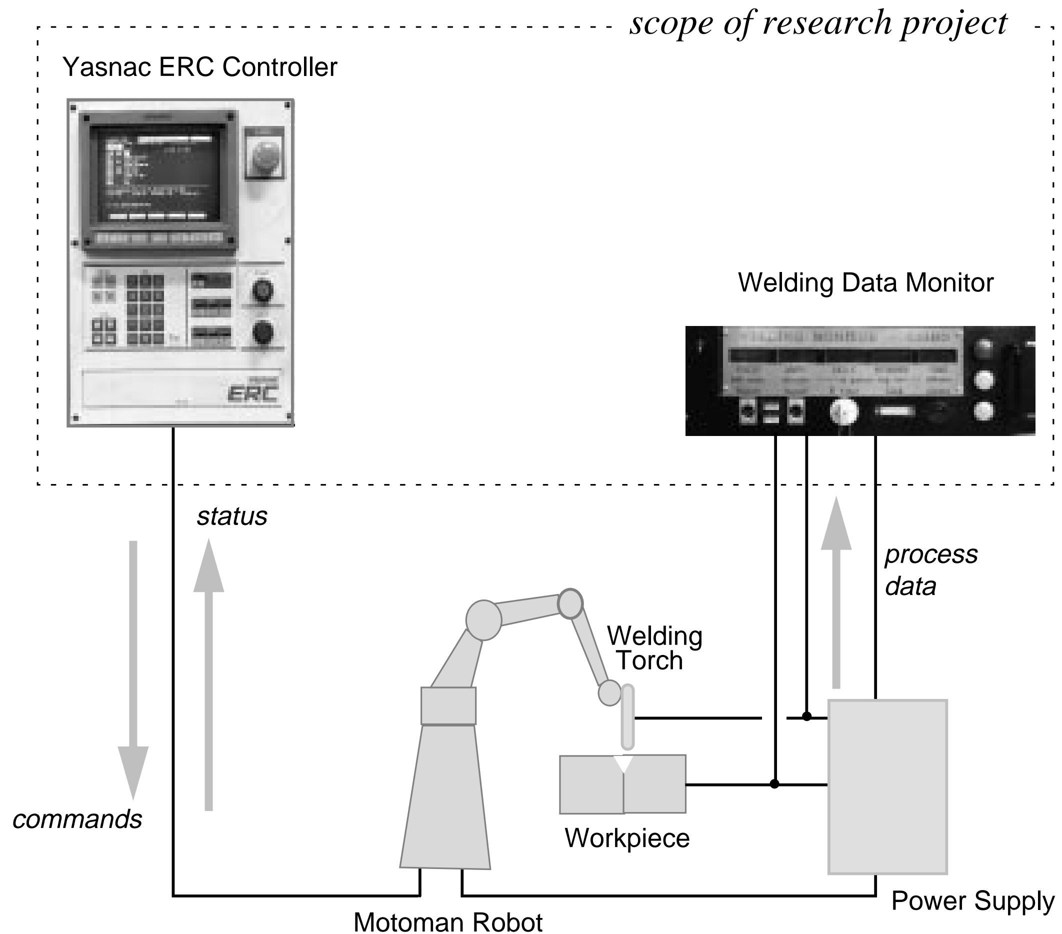


Figure 4-1 Overview of the robotic welding environment

Welding data is obtained from a separate data acquisition system, in this case the WDM. The WDM logs data on current, voltage, and temperature, then transmits the data on command via a serial link to a computer.

### 4.3.1 Yasnac ERC robot controller interface

Figure 4-2. depicts the Yasnac ERC robot controller interface, which is the existing interface to the controller's microprocessor and, in turn, to the Motoman robot. Cursor keys and function soft-keys F1 to F5 are used to navigate through menus for which there are no shortcuts. Navigating menus becomes a time consuming part of using the robot controller during setup. A keypad allows numeric input, but letters of the alphabet must be picked out one at a time with the cursor keys from a screen "keyboard" in order to edit job files or specify file names. File management on the controller is limited to loading, saving, and deleting. The storage of job files on a PC is the best existing alternative but requires a separate procedure for downloading them to the robot controller.

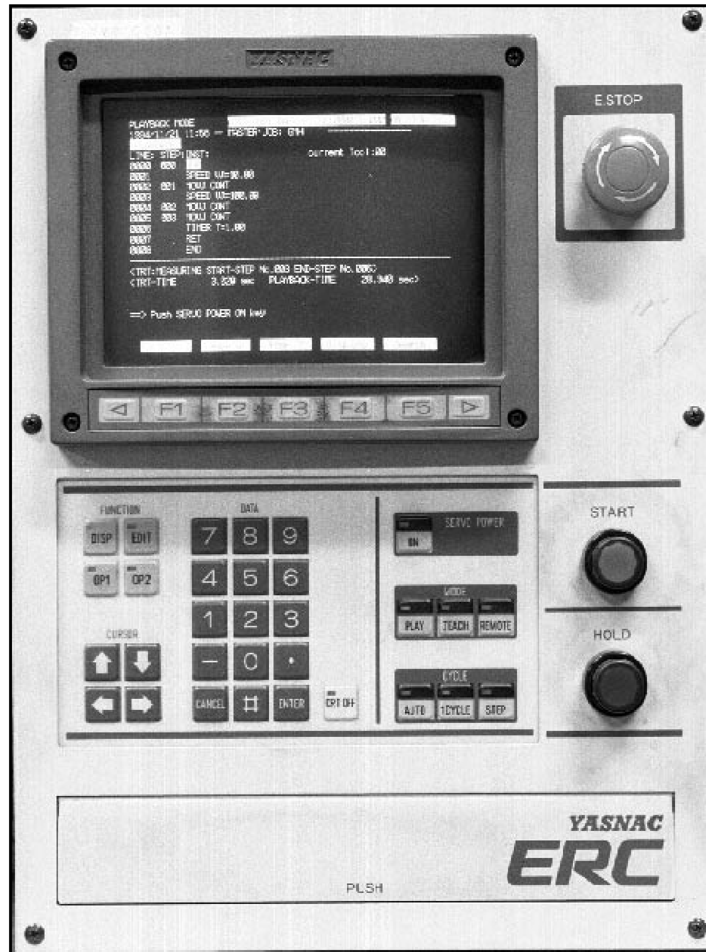


Figure 4-2 Yasnac ERC robot controller panel

## 4.3.2 Welding Data Monitor interface

The WDM has a separate control and display panel, which is controlled and configured by means of three buttons on the panel. Different commands are executed by pressing different button combinations. The monitor can also be remotely controlled from a computer connected via a serial link to start recording, stop recording, and send data to the computer.

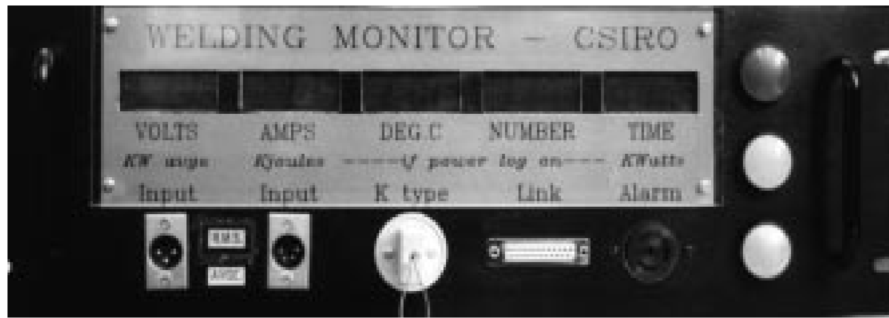


Figure 4-3 Welding Data Monitor (WDM)

### <u>4.3.3 A typical scenario - existing environment</u>

The setup procedure is a cycle as shown in Figure 1-1, that begins with the planning phase: an initial specification of parameters and a motion path. This is followed by the replanning phase: program execution (a trial weld), then assessment, then back to specification to make corrections.

The operator begins by creating a program job file on the controller that contains settings for the primary parameters such as voltage and torch speed. Initial selection of these parameters is guided by reference to published data. Then the operator specifies a motion path for the robot to follow when welding. This involves using the teach pendant to move the robot arm around the work piece while the controller records the sequence of coordinates in a file. The coordinates are then embedded within the program file by the operator.

After creating the files for the job, the operator executes them in a trial run, while observing the welding process using sight and hearing. Process data can be logged on the WDM from the arc power supply and a temperature sensor, with the instantaneous values displayed on the WDM numeric displays. If any obvious fault conditions are observed during welding, there is no need for further testing. Otherwise, the welded joint is assessed at the end of the run by examining the joint and by uploading the WDM process data to a computer for analysis.

The joint may be examined by a combination of visual inspection and other non–destructive testing methods, then finally by destructive testing. The WDM process data provides feedback regarding the actual values of input parameters such as current and voltage, since these parameters are specified to the controller as a number from 1 to 10 rather than absolute values.

If the joint is flawed, the robot program must be corrected and the cycle of specification, execution, and assessment is repeated until it works. Only then are the finished products welded as a batch job. At this stage the WDM can also provide basic quality

assurance by recording whether parameters such as heat input remained within acceptable ranges.

## 4.4 Requirements for a robotic welding environment

The new robotic welding environment needs to run on a computer that is connected to both the robot controller and the WDM by separate RS232 [19] serial links.

The robot controller is switched to remote mode in which it accepts commands from the computer using a proprietary remote communication protocol. All functions of the controller for robot system control and status monitoring are available in this remote mode. To initiate welding, the program files and positional coordinate files that constitute the robot's instructions for the job must be transferred to the robot controller before the execution commands are issued.

The purpose of a robotic welding environment should be to assist the operator to follow a loosely defined setup procedure that produces a satisfactory weld. The time constraints of this task are imposed for economic reasons, since the solution must be found by using as few resources as possible, although there is not the urgency of a power plant emergency. Monitoring the welding process is an on-line, dynamic activity, while program correction and robot control are off-line, static activities. The aim of the cooperative software must be to present relevant information - both static and dynamic - to the operator in a meaningful way, while providing the means to test, analyse and alter the program.

Two types of requirements are described: those which duplicate the existing functionality of the Yasnac ERC and those which provide new functions.

### 4.4.1 Existing functions

The following functions duplicate the existing functionality of the Yasnac ERC, including programming, control, and monitoring of the robot.

**Robot programming** is required by the operator to correct and re-run program files as quickly as possible during setup. This requires a file management scheme which allows all files to be edited on the remote computer, then downloaded automatically before execution.

**Control and status** of the robot system are required, for example: robot arm servo motor ("On", "Off"), program error status ("Reset"), and arc power ("On", "Off"). The Yasnac ERC's communication protocol specifies that robot commands can be sent in ASCII format to the controller, while system status must be determined by regularly polling the controller.

### 4.4.2 New functions

New functionality for a cooperative robotic welding environment is derived from the general requirements defined in section 3.1. Each requirement is discussed in terms of robotic welding setup.

**A Graphical interface** requires common functions, such as robot system commands "Run", "Stop", etc., to be available as buttons, especially if they are already implemented as buttons on the existing panel. Critical buttons should be prominently displayed. The software environment's interface should also avoid complex sequences of menus or keys for executing functions.

**Data retrieval and presentation** is required in addition to direct observation. The operator needs to have feedback data displayed during a test run, in order to assess the program and decide what corrections to make. This requires the software environment to coordinate and present dynamically changing information from various sources, including the robot, the welding monitor, and other sensors.

**Information highlighting** requires that the software environment draws the operator's attention to the most relevant information. For example, simple limit alarms on incoming data would allow a user to specify the acceptable range for parameters such as heat input to the weld, and be notified by means of an aural or visual indicator if a value is out of range.

**Information storage** provides the operator with a record of the acquired data. In cases where the operator needs to confirm the weld assessment, the operator should be able to conveniently review the acquired data immediately after program execution.

**Integration of tools** allows setup to be carried out from a single environment as much as possible. This requires integration of the robot with other tools, both software and hardware, such as welding monitors, sensors, knowledge bases, and expert systems.

**Abstracted functionality** specifies that the operator works at the level of welding-related activities rather than robot commands. The cooperative software environment should hide the details of the robot controller.

**Repository of cases** can save the operator significant amounts of time by providing ready access to previous jobs to use as templates for similar conditions, e.g. the same base metal and joint shape. A single welding job will consist of several job files: some specifying different aspects of robot movement; others containing parameter programming sequences. Job files should be stored according to the characteristics of the job.

**Extensibility** requires the software environment to be adaptable for new tools and different models of robot. This enables the most appropriate tools to be added to the cooperative software environment as they become available, and for the same environment to be reused with a wide range of robots as appropriate.

## 4.5 Summary

Small batch robotic welding setup is an effective test of cooperative problem solving issues, since it is an ill defined, reactive task that requires human involvement. Setup also generates a large amount of operational information. The existing tools that support the human operator in the setup task are inadequate. Related work in robotic welding focuses on developing better equipment rather than supporting the users of existing equipment. The following functions are required in a cooperative problem solving environment for robotic welding:

- robot programming and program management;
- control and status of robot systems;
- a graphical interface that mimics real button controls;
- presentation of welding process information from multiple sources;
- highlighting the most important information;
- storage of information for review;
- integration of the robot with other tools;
- abstracted functionality;
- repository of previous job descriptions and their program solutions;
- extensibility to other tools and robots.

The next chapter describes the implementation and validation of the information presentation framework, then evaluates the findings.

# Chapter 5. Design and Implementation

This chapter describes the implementation of the framework defined in Chapter 3. Section 5.1 describes the toolkit extension and section 5.2 describes the prototype cooperative software environment built from this extension. The prototype software environment -The Panel - required some domain-specific code to be implemented in addition to the toolkit extension classes. The Panel consists of three main subsystems: *InterfaceManager*, *RobotManager*, and *DataManager*. *DataManager* implements the reusable subsystem design of the framework. *RobotManager* implements the robot controller's communication protocol and is needed to make the prototype a valid demonstration of a cooperative problem solving environment. Then sections 5.3 to 5.5 evaluate three aspects of the framework: how well does the software environment support the operator? how well does the toolkit support the development of the software environment? how effectively does the design of the framework support developers and indirectly benefit users?

## 5.1 The InterViews toolkit extension

The toolkit extension classes are based entirely on the framework design specified in section 3.3. The instantiations of Filter and Processor were not implemented as classes because of the trivial nature of the chosen examples. Instead, the examples of Filter and Processor are C functions that are passed as pointers to the WeldMonHandler and DisplayHandler respectively.

### 5.1.1 Framework classes

The core framework classes implement the design specified in the information presentation framework. The features of each component and the ways in which they cooperate are described in more detail in section 3.3.1 and Appendix A.

- *DataHandler* (abstract class, derived from IOHandler) links data streams to DisplayHandlers and optional Filters; retrieves and interprets data streams from a data acquisition device via a port, while optionally storing the data to a log file; applies any given Filters then passes each data stream to the linked DisplayHandler(s);
- *Filter* (abstract class) removes data values from a data stream that match the given removal specification;
- *DisplayHandler* (concrete class) links data streams to Displayers and optional Processors; applies any linked Processors to incoming data streams and passes the result to linked Displayer(s);

- *Processor* (abstract class) performs a calculation on a data stream and/ or merges two streams;
- *Displayer* (abstract class, derived from MonoGlyph) displays the values in a data stream dynamically.

### 5.1.2 Derived classes

The following derived classes were specialised for the needs of robotic welding based on the above abstract framework classes; they are described in more detail in Appendix A:

- *WeldMonHandler* (concrete class, derived from DataHandler);
- *ClippingFilter* (concrete class, derived from Filter);
- *Multiplier* (concrete class, derived from Processor);
- *Readout* (concrete class, derived from Displayer);
- *Histogram* (concrete class, derived from Displayer);
- *Alarm* (concrete class, derived from Displayer).

**WeldMonHandler (concrete class, derived from DataHandler)** links data streams to DisplayHandlers and manages the data coming through the port from the WDM. WeldMonHandler interprets data by referring to a data format file. The "Format" file (Figure 5-1) is set up by the application developer to specify which column of the WDM data file (Figure 5-2) contains the named data streams.

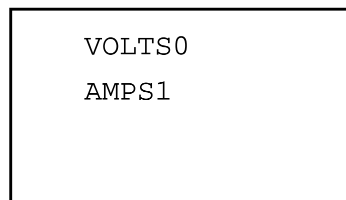


Figure 5-1 Sample WDM data format file

```
8.3234  115        3091
11.6362 113        4224
11.5362 113        4179
11.8364 115        4328
12.2366 113        4492
12.4366 113        4554
12.6366 113        4642
12.7366 113        4681
```

Figure 5-2 Sample WDM data file

The class's inherited responsibilities are to:

- know the name of the port to which it is attached;
- link one or two named data streams to an optional Filter and to DisplayHandlers;
- start retrieving data from the data acquisition device through the port and pass it to linked Filters and DisplayHandlers;
- save retrieved data streams to a logfile if specified;
- stop retrieving data;
- start replaying logged data the same way as live data streams;
- stop replaying logged data.

The class's specialised responsibilities are to:

- know the protocol for retrieving data from the WDM;
- know the format of data coming from the WDM.

**Readout (concrete class, derived from Displayer)** displays the values in a data stream as a digital gauge, as shown in Figure 5-3.

Figure 5-3 Readout

The class's inherited responsibilities are to:

- draw itself on the screen as a digital gauge to the specified precision, labelled with the given data stream name;
- update itself with a given data value to the specified precision;
- clear itself.

**Histogram (concrete class derived from Displayer)** displays the values in a data stream as a continuously updating bar histogram, as shown in Figure 5-4. The Histogram's height (resolution of values displayed) and width (amount of history) are variable. The scale can be changed by the user during a display session.

Figure 5-4 Histogram

The class's inherited responsibilities are to:

- draw itself on the screen as a vertical bar histogram to a given height;
- update its viewscreen with a bar that represents the new value;
- clear itself.

The class's specialised responsibility is to:

- allow a user to change its vertical scale.

**Alarm (concrete class is derived from Displayer)** lights up an indicator button whenever a value in the data stream goes above the allowable range. Alarm demonstrates the function of highlighting.

Figure 5-5 Alarm

The class's inherited responsibilities are to:

- draw itself on the screen as an indicator button and an input field;
- update itself by checking whether the value is out of range; if so, to light up its indicator;
- clear its limit.

The class's specialised responsibilities are to:

- allow a user to enter an upper limit;
- reset its indicator to "Off".

### 5.1.3 C functions

The following C functions demonstrate sample behaviour for Filters and Processors. These examples also illustrate the limitations of using C functions and the benefits of using subclass inheritance and member function interfaces as described in section 2.3.1.

**ClippingFilter** modifies values that fall above a fixed threshold. If the value falls below the threshold, the value is returned intact; if not, the threshold value is returned:

```
float clippingfilter(float value)
if value < threshold
        then return value
        else return threshold
```

**Multiply** takes two values as parameters, multiplies them together and returns the result:

```
float multiply(float value1, float value2)
return value1 * value2
```

The function interface for the WeldMonHandler's linkStream is:

```
void linkStream(const char* stream1,
                     const char* stream2,
                     DisplayHandler* disphdlr,
                     float (*clippingfilter)(float))
```

The `clippingfilter` function is passed as a pointer to `linkStream`. As defined here, this `linkStream` interface can only accept pointers to `clippingfilter`, not to any other kind of filter. Instead, if `clippingfilter` were a subclass, `linkStream` could be defined as:

```
void linkStream(const char* stream1,
                     const char* stream2,
                     DisplayHandler* disphdlr,
                     Filter* filter)
```

Now any subclasses of WeldMonHandler will have the benefit of inheriting a generic member function interface and be capable of linking streams between any subclass of Filter.

### 5.1.4 Sample linkage of toolkit components

Figure 5-6 demonstrates how the derived toolkit components can be linked to form a Data Manager subsystem. A WeldMonHandler called `wdm_hdlr` links the stream labelled "voltage" to a Filter `clippingfilter` and a DisplayHandler `v_disphdlr` and links "current" to `c_disphdlr`; then the DisplayHandlers link each stream to a Histogram and to a Readout.

In order to generate a data stream representing power, "current" and "voltage" are both linked to `p_disphdlr`, which in turn links them to `multiplier`, `p_histo`, and `p_readout`.

```
//** create weldmonhandlers and displayhandlers
wdm_hdlr = new WeldMonHandler(portname);
v_disphdlr = new DisplayHandler;
c_disphdlr = new DisplayHandler;
p_disphdlr = new DisplayHandler;

//** create histograms, height = 20 pixels, width = 30 bars
v_histo = new Histogram(20,30,layouts,widgets);
c_histo = new Histogram(20,30,layouts,widgets);
p_histo = new Histogram(20,30,layouts,widgets);

//** create readouts, precision = 1 decimal place
v_readout = new Readout(1);
c_readout = new Readout(1);
p_readout = new Readout(1);

//** link single data streams to filters and displayers
wdm_hdlr->linkStream("voltage","",v_disphdlr);
wdm_hdlr->linkStream("current","",c_disphdlr,
                      clippingfilter);
v_disphdlr->linkStream(v_histo);
v_disphdlr->linkStream(v_readout);
c_disphdlr->linkStream(c_histo);
c_disphdlr->linkStream(c_readout);

//** link voltage + current to multiplier --> power
wdm_hdlr->linkStream("voltage","current",p_disphdlr);
p_disphdlr->linkStream(p_histo,multiply);
p_disphdlr->linkStream(p_readout,multiply);
```

Figure 5-6 Sample code for linking toolkit components

### 5.1.5 Facilities of the toolkit extension

The toolkit extension components fulfill most of the requirements for an information presentation framework specified in section 3.2:

- A WeldMonHandler can retrieve multiple streams of data through a port from the WDM.
- Multiple DataHandlers can be created to handle multiple ports.

- A WeldMonHandler can apply a Filter to any data stream, for example to remove noisy (out of realistic range) data values.
- A DisplayHandler can apply a Processor to any data stream, for example: to find the average of the values in a data stream.
- A DisplayHandler can use a Processor to combine two data streams to produce a new data stream, for example: the multiplication of current and voltage to produce power; this includes combining data streams from different ports, i.e. from different data acquisition devices.
- An Alarm (Displayer subclass) can be applied to any data stream to act as a limit alarm.
- A variety of display components can be derived from the base class Displayer; for example: a histogram and a digital gauge.
- DisplayHandler allows any data stream to be displayed in two or more Displayer objects simultaneously for comparison.
- WeldMonHandler can store all data streams in a file and replay them for analysis.
- WeldMonHandler can provide the user with start, stop and replay control.

The only requirement these components do not meet explicitly is allowing the user to choose the presentation configuration. However, the design of the components allows this facility to be implemented at a higher level, for example, by defining a Chooser class that lets the user dynamically recombine presentation components. This facility is discussed in section 6.1.6 as an enhancement.

## 5.2 The prototype cooperative software environment

The Panel is a prototype cooperative software environment that supports robotic welding setup. As shown in Figure 5-7, the Panel closes the loop between the Yasnac ERC robot controller, the Motoman robot, and the Welding Data Monitor. Commands are sent from the Panel to both the controller and the WDM. The WDM sends welding process data back to the Panel, but the controller only sends back robot status information, such as whether the welding arc is “On” or “Off”. For this reason, the Panel does not use the framework components to present information coming from the robot controller.

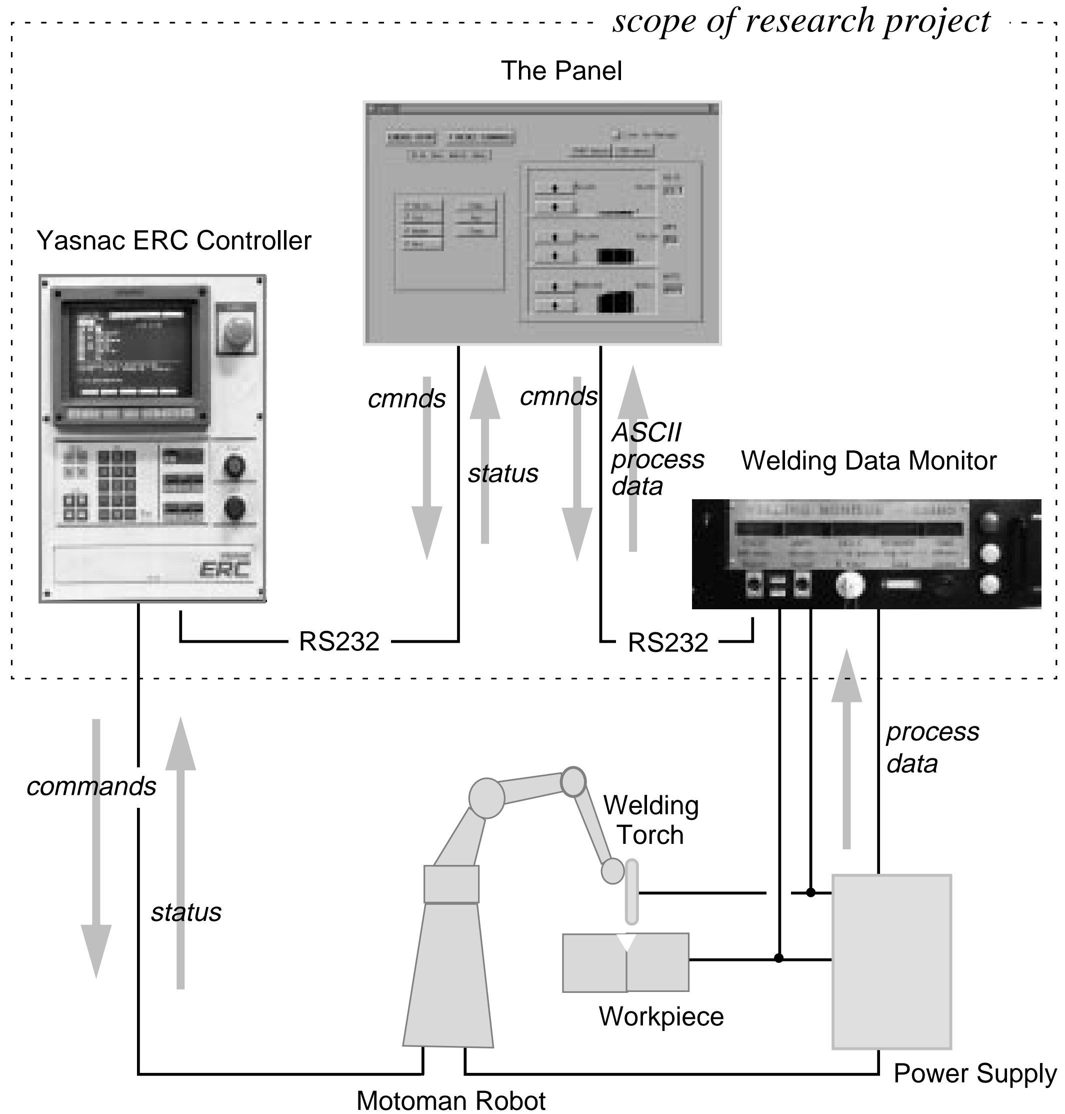


Figure 5-7 Overview of Panel within robotic welding environment

The Panel robotic welding software environment consists of three main subsystems, as shown in Figure 5-8:

- *InterfaceManager* arranges all interactive InterViews widgets and framework Displayers that appear on the Panel.
- *RobotManager* provides command and file exchange between the Panel and the robot controller.
- *DataManager* retrieves, prepares, and presents data from external sources such as the welding monitor.

*InterfaceManager* is supported by InterViews and the operating system (OS); *DataManager* is built from the components of the information presentation framework; *RobotManager* is built from custom code which includes a serial communications

module and the encoded Yasnac command protocol. *RobotManager* demonstrates that the framework can be adapted to a practical application.

Figure 5-8 indicates the portions of the Panel which support static and/or dynamic decision-making. The two distinct halves of the Panel consist of:

1. a static cooperative software environment, supported by InterViews and OS;
2. a dynamic cooperative software environment, supported by InterViews, OS, and framework.

Static decision-making support refers to robot program modification, robot control and display of status. Dynamic decision-making support refers to information processing of welding data. This indicates that InterViews is suitable for supporting both types of environment.

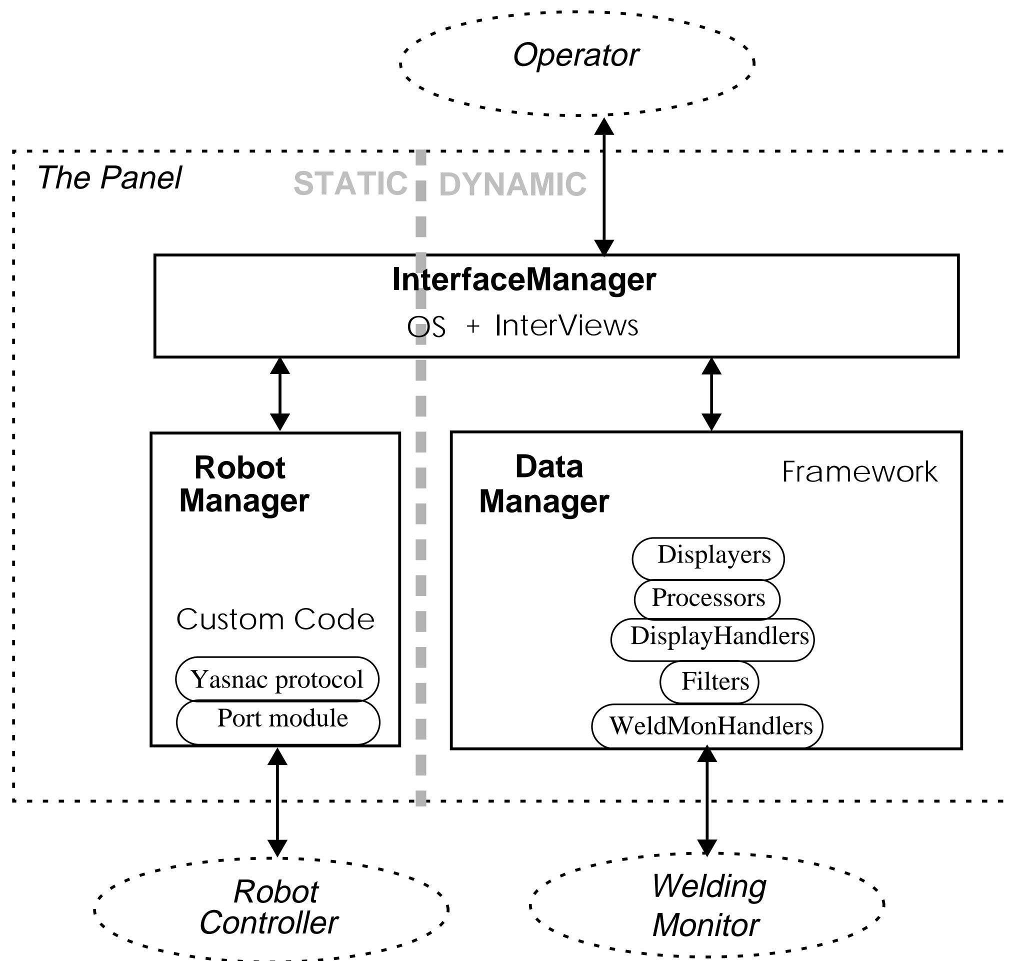


Figure 5-8 The Panel subsystems

### 5.2.1 InterfaceManager subsystem

*InterfaceManager* handles all interactive controllers and displayers that appear on the screen. The subsystem is built from InterViews components, such as buttons and menus, and toolkit extension display components, such as histograms and alarms. The toolkit extension components are built from low-level widgets by the *DataManager* subsystem, but their arrangement within the Panel interface is provided by *Interface-Manager* subsystem.

The interface widgets and display components are all subclasses of Glyph. As such, they can be readily combined into a glyph hierarchy like the one shown in Figure 5-9, which corresponds to the Panel interface shown in Figure 5-10. The compositional glyph protocol allows all glyphs in a hierarchy to be updated automatically by the top-level ApplicationWindow object whenever the window is resized or redisplayed.

The sample glyph hierarchy for *InterfaceManager* demonstrates the simplicity of the application developer's task in using the toolkit and extension components. No layout glyphs such as spaces are shown, since they are not necessary to provide *InterfaceManager*'s functionality.

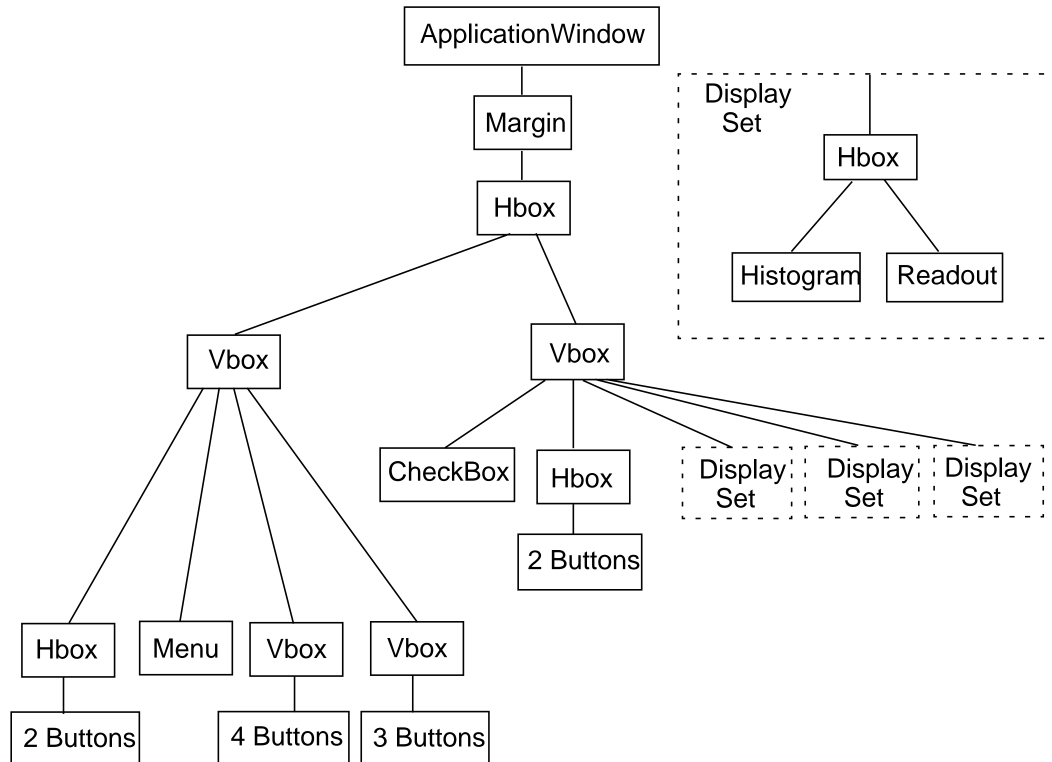


Figure 5-9 Sample glyph hierarchy for *InterfaceManager* subsystem

The layout of the Panel's interface (Figure 5-10) is a prototype intended to demonstrate possibilities, not to be a definitive solution. The left-hand side contains the *RobotManager* control elements (e.g. "Stop", "Servo", "Run") while the right-hand side contains the *DataManager* control elements (e.g. "Start Watch", "Stop Watch", various types of Displayers). The menu provides further control of both *RobotManager* ("File" menu, "Run" menu) and *DataManager* ("Watch" menu) functions.

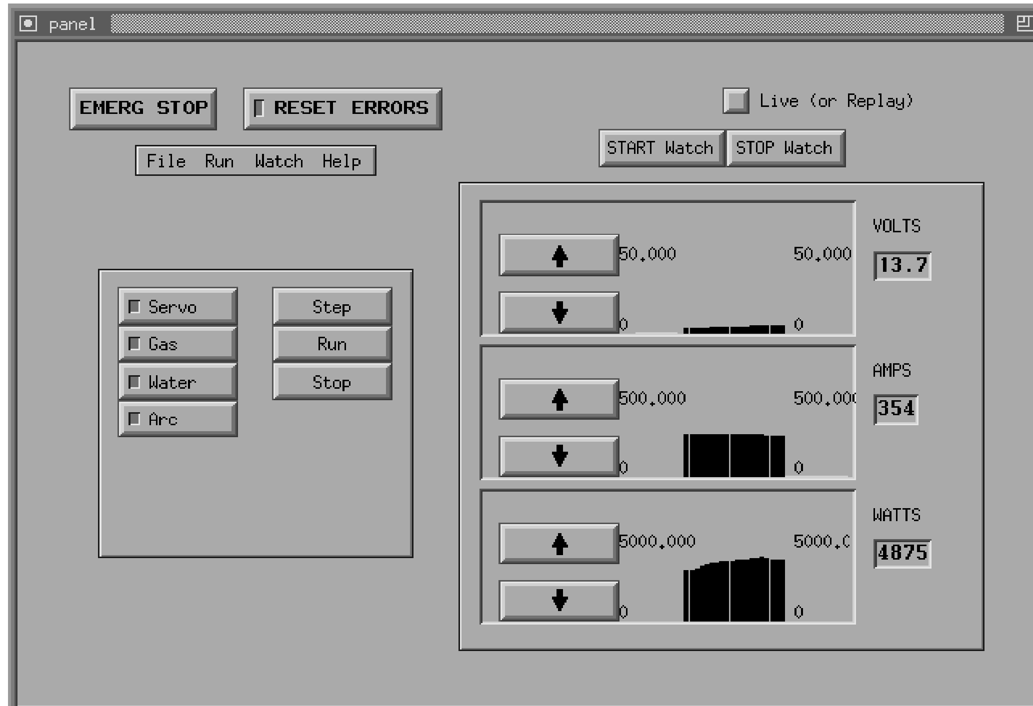


Figure 5-10 The Panel cooperative software environment

### 5.2.2 RobotManager subsystem

*RobotManager* supports strictly static decision-making within the cooperative software environment. This includes off-line programming, status, and control of the robot, i.e. activities performed when the robot is not welding. It does not include dynamic information presentation. Once the *RobotManager* initiates execution of a welding job, the operator has no dynamic control of the process apart from the ability to stop it.

*RobotManager* implements the Yasnac's handshaking protocol (consisting of request/ acknowledgement sequences) to provide command and file exchange between the Panel and the robot controller. In this protocol, the robot controller acts as a server, while the Panel acts as a client, i.e. the Panel initiates all exchanges. Each exchange consists of common elements, such as "send command", "wait for acknowledgement", or "start file send sequence". Files such as programs or coordinate files are exchanged by packing/unpacking them into multiple data frames of fixed size. A frame consists of a header, then a block of data, then a trailer containing a checksum character for error checking.

Each common element of the protocol is performed by a separate C function. Functions are then combined in different sequences to perform each of the required types of

exchange. The actual reading and writing of commands to the communication port is performed by a Port module that uses basic Unix file input/output functions.

A different RobotManager could be substituted to allow the Panel to control a different model of robot without affecting the other two subsystems.

### 5.2.3 DataManager subsystem

*DataManager* implements the cooperative framework design to retrieve data from the WDM and prepare it for display. *DataManager* is built from the InterViews extension classes: WeldMonHandlers, DisplayHandlers, Histograms, and Readouts. (Alarms are not shown in this example interface). Additional sources of data could be incorporated by deriving an appropriate DataHandler subclass, without affecting the rest of the Panel system.

A WeldMonHandler requests and receives serial data from the WDM. Data arrives as a series of lines of multiplexed ASCII values and is interpreted by the WeldMonHandler to form multiple simultaneous data streams. A WeldMonHandler can accept serial streams from multiple external sources simultaneously. Each data stream is passed on to the DisplayHandlers linked to the WeldMonHandlers, optionally via a Filter. Each DisplayHandler receives a single or double stream from a WeldMonHandler and may process the data via a Processor function. (In the Panel prototype, a maximum of two streams can be passed by DataHandlers, since data streams are passed individually in the DataHandler's `update` member function.) The DisplayHandler then sends each data stream onto the Displayers in its charge, which may be Histograms, Readouts, or Alarms.

A sample *DataManager* object-instance graph is shown in Figure 5-11. In this diagram, the WeldMonHandler, `wdm_hdlr`, is associated with three DisplayHandlers, which indicates that `wdm_hdlr` is handling three streams of data, one of which is filtered by `clippingfilter`. The DisplayHandler `p_disphdlr` is associated with `multiplier`. Then each DisplayHandler is associated with two Displayer subclasses.

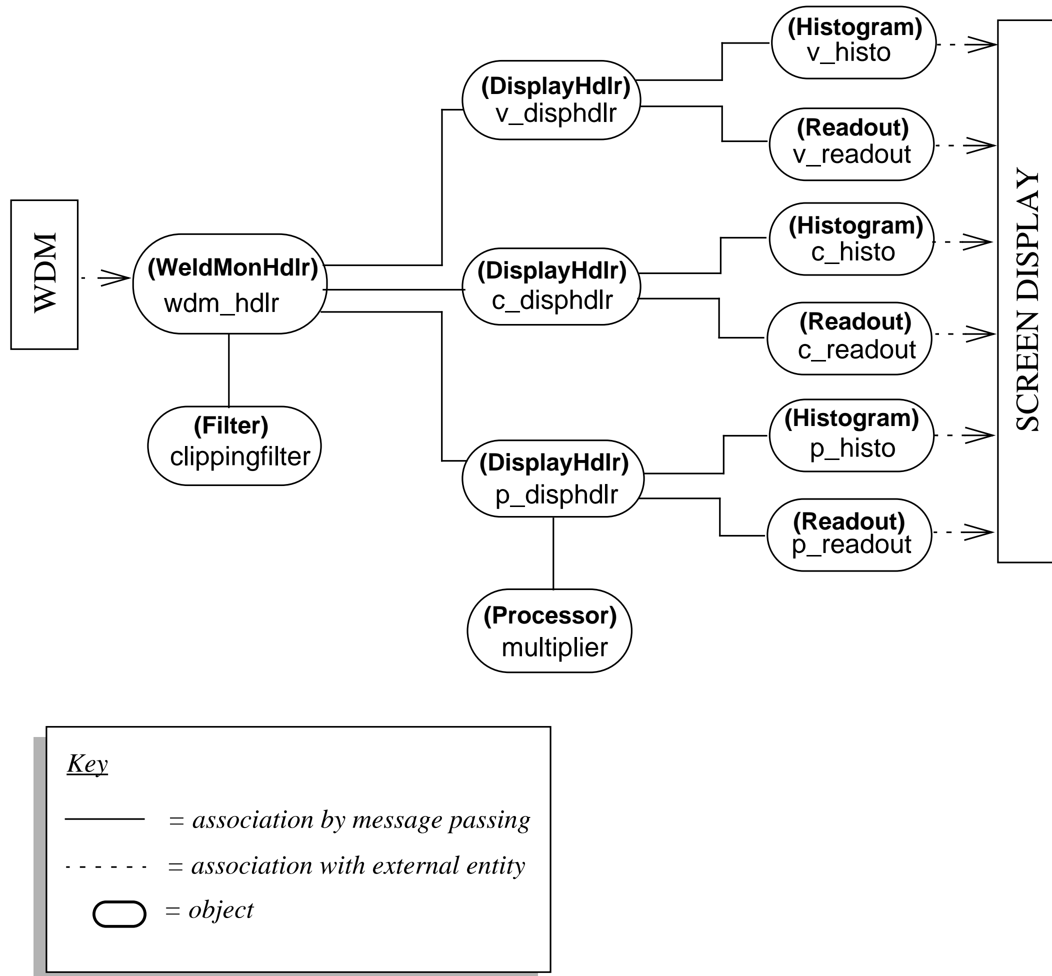


Figure 5-11 Sample object-instance graph for *DataManager* subsystem

### 5.2.4 Panel functionality

The Panel prototype shown in Figure 5-10 is capable of being fitted with all functions relating to setup that are currently available on the robot controller shown in Figure 4-2. The Panel provides additional functions for writing and debugging the robot programs that were not available on the robot controller. The Panel can successfully transfer files between the computer and the robot controller. The Panel prototype demonstrates how all the functional requirements for support of robotic welding could be implemented in a fully functional system.

The push buttons, e.g. “Run”, issue commands to the robot controller. The toggle buttons - those with indicators, e.g. “Arc” - both control and display the status of the robot by means of a polling function, which regularly requests the robot’s status from the robot controller. The “File” menu provides program file management, the “Run” menu provides *RobotManager* commands, such as “Download”, “Upload”, “Run” programs and “Teach” to set the robot to Teach mode. *DataManager* buttons “Start

Watch" and "End Watch" control the data presentation session, and a "Watch:Logfile" menu option specifies a file for incoming data to be stored. The "Live/Replay" toggle button allows data to be replayed after the welding session.

Existing functions have been changed, from a long series of menu selections and key presses on the robot controller, to a few buttons and menu items on the Panel. Table 5-1 compares the Yasnac controller and the Panel in terms of the number of steps taken to perform the same functions using both interfaces. Although this comparison does not address functional complexity, the reduced number of steps intuitively illustrates the advantages of the Panel interface.

**Table 5-1 Interface comparison: Yasnac ERC vs Panel, by no. of steps**

| *Function* | *Yasnac controller* | *No. of Steps* | *The Panel* | *No. of Steps* |
|---|---|---|---|---|
| New job | [Teach]<br>[F3] new job<br>[F4] job<br>[F1] alphabet<br>cursor to locate letter<br>[Enter]<br>cursor to locate letter<br>[Enter]...<br>{etc.}<br>[F5] exit<br>[Enter] | 7 | menu: File, New<br>menu: File, Save as...<br>type name<br>click OK | 4 |
| Load job | [Teach]<br>[F4] other job<br>[F4] page down.<br>cursor to job<br>[Enter] | 4-5 | menu: File, Open<br>double click dir...{opt}<br>double click job | 2-3 |
| Edit line | cursor to line...<br>[Edit]<br>[F4] line edit<br>[F1] data change<br>type new numbers<br>[Enter]<br>[Enter] | 7 | cursor/scroll to line....<br>insert cursor<br>[Delete]<br>type new numbers/letters<br>menu: File, Save | 5 |
| Delete job | [Teach]<br>[F4] other job<br>[Edit]<br>[F3] job delete<br>[F4} page down...{opt}<br>cursor to job<br>[Enter]<br>[F5] execute | 7-8 or more | menu: File, Delete<br>double click dir...{opt}<br>double click job<br>click OK | 3-4 |

### 5.2.5 A typical scenario - the new environment

This section compares the new setup scenario with the old scenario described in section 4.3.3. The proposed new setup procedure using the Panel also follows the planning/replanning procedure described in Figure 1-1.

Planning can begin with the operator searching/browsing for a previous job that is similar to the welding job at hand, rather than referring only to published data. This existing job will provide a template (starting point) for parameter specification. The operator modifies the new job program using the text editor, based on published data and experience.

Next comes the teaching procedure to specify a motion path. Although the robot can be placed in teach mode from the Panel, the teach pendant must still be used to teach the robot arm its motion path. The teach pendant sends coordinates to the controller, which are later embedded by the operator in the current job program.

Now the replanning cycle begins with the execution of the robot program.

Before running the program, the operator can set limit alarms on various parameter data streams. When the run command is invoked, all relevant files for the job will be automatically downloaded to the robot.

During the trial run the operator observes the process: both directly, by watching and listening to the robot, and indirectly, by watching the dynamic display of data during welding. The limit alarms free the operator from having to watch for obvious fault conditions. Direct observation is still necessary to detect conditions such as poor penetration or spatter, but the operator no longer has to wait till the end of the run to analyse the process data. During the run, if any obvious fault conditions were observed or detected by the Panel, the program may be stopped and corrected immediately. For this reason, detection of such faults is likely to be much faster than in the old scenario.

Otherwise, at end of the trial run, the operator can play the process data back at normal speed or slower, either to confirm or detect a fault condition. This facility was not available in the old scenario. If the process data indicate no faults, the welded joint itself can be tested, either destructively or non–destructively.

If necessary, the operator makes corrections to the program before starting the next cycle of testing, which continues until the welded joint is acceptable. The process data log is readily available to document the joint quality for post-weld analysis or for further research.

### 5.2.6 Demonstration

The Panel was demonstrated to evaluation panel[2] in two stages:

1. The *RobotManager* subsystem was used to transfer a file from the computer to the controller and back again. This verified that remote computer control of the robot would be possible, since file transfer is the most complex operation in the remote protocol.
2. The *DataManager* subsystem was tested with the WDM to ensure that the Panel could successfully request, obtain, and display multiple sources of data.

*DataManager* was demonstrated by having it retrieve a pre-recorded session from the Welding Data Monitor. The session was then replayed, enabling the evaluation panel to see how the live session would have appeared. Note that the objective of the evaluation was to assess the overall usability of the system, without reference to any particular welding procedures. In this regard, the demonstrations were a success.

The evaluation panel considered that the data presentation capabilities provided by *DataManager* were a vast improvement over that provided by the existing system and that these enhanced capabilities could be expected to impact positively on the decision making processes required in both setup and quality control. Determining the extent of this anticipated benefit would have required extensive trials using a range of welding procedures and was deemed to be beyond the scope of this project.

Demonstration of this prototype shows that it would be feasible to implement the information presentation framework as a cooperative software environment.

## 5.3 Evaluation of the framework

This section evaluates the framework in terms of how well it provides a reusable design for cooperative environments. This is looking at the framework from the toolkit developer's point of view. The implementation effort of developing the toolkit extension to fit the framework design is also examined.

### 5.3.1 Framework design

There are several aspects of the cooperative framework which are an advantage to a toolkit/application developer. One is the provision of a ready-made design, another is the potential for reusability of the design for different toolkits and different applications, and a third is the extensibility of the application based on this framework.

2. Members of the Welding Group, CSIRO DMT, Adelaide.

**Design of a subsystem** - the basic information presentation facilities of a CPS software environment are provided by the framework design. This design specifies not only the types of component that need to be built, but the ways in which they are linked together. However, the design is flexible, in that the numbers and combination of each component can be varied, while Filter and Processor components are optional. To build an application, specific components will need to be implemented, but this does not require the design to be changed. The application developer's effort is reduced because there is a specific architecture to implement, so decisions about structure have already been made.

**Generic design** - a developer should be able to apply this framework to any object-oriented GUI toolkit, although the implementation of individual components might be different. For example: the DataHandler class has the benefit of inheriting port handling and timer functions from the IOHandler; Displayers inherit a compositional display protocol from MonoGlyph. However, the framework takes advantage of typical GUI toolkit features to specify a generic cooperative problem solving toolkit. For example: common widgets such as buttons and menus can be used to implement static control functions; graphical drawing primitives allow specialised Displayer components to be created. In addition, the framework design should be applicable to a wide range of reactive applications, based on the premise that operators in these domains need on-line presentation of information.

**Expansion of the design** - the framework can be easily expanded around the existing design. The prototype developed under the guidance of this framework is not a complete cooperative interface, but it serves to demonstrate how a real cooperative software environment could be built without needing to change anything about the design, only by adding more of the same components.

### 5.3.2 Framework implementation effort

Extending InterViews to implement the framework design proved to be a straightforward exercise. Nine new classes were built: DataHandler, WeldMonHandler, DisplayHandler, Displayer, ClippingFilter, Multiplier, Histogram, Alarm, and Readout. The development of these classes required approximately 600 un-commented lines of code (ULOC). Implementing Filter and Processor abstract classes would add very little to the total effort required; for a class description of Filter and Processor, see Appendix A, and for a proposed ClippingFilter and Multiplier, see Appendix B.

## 5.4 Evaluation of the toolkit extension

This section looks at how well the toolkit extension facilitated the development of a cooperative software environment. Four of the CPS techniques from the taxonomy in

section 2.1.1 have been encapsulated in the toolkit extension. The effort of applying the toolkit components to the prototype cooperative software environment is also examined. The toolkit extension is evaluated from the application developer's point of view.

### 5.4.1 Cooperative software techniques

The toolkit extension classes employ four cooperative software techniques based on the taxonomy described in 2.1.1:

- visualisation;
- filtering;
- summation;
- highlighting.

**Visualisation** converts a stream of numeric data into a visual representation of a physical quantity or quality. The toolkit extension provides the means for any numeric data source to be represented visually using a Displayer component. The existing InterViews graphical drawing elements such as lines and text labels allow customised Displayers to be created relatively easily. Once a library of Displayer components has been developed, they can be used in different cooperative environments.

**Filtering** removes unnecessary data values in order to reduce the total amount of data that users have to process. The DataHandler allows filters (implemented as C functions) to be attached to any incoming data stream. A simple form of filtering (clipping filters) has been implemented to demonstrate how it is possible to attach more sophisticated filters to data streams.

**Summation** can take several forms, but generally combines data from different sources in such a way as to increase the amount of information. The toolkit extension provides the means for separate parameters to be viewed simultaneously for direct comparison. The ability to process and merge data streams provides summation by combination. For example, current and voltage can be merged and displayed as power, then compared to current and voltage.

**Highlighting** draws the user's attention to particular information; this is provided by deriving specialised Displayer components. A subclass of Displayer can be attached to any data stream and used to alert the operator of a particular condition being met in the data. For example, the Alarm used in the Panel detects values outside a range of values. The toolkit extension allows similar highlighting components to be created and placed on the screen in the same manner as other Displayer components.

### 5.4.2 Toolkit extension application effort

Once they are defined, the toolkit extension classes WeldMonHandler, ClippingFilter, Multiplier, DisplayHandler, Histogram, Alarm, and Readout are used in the same way as existing InterViews toolkit components. For visible interface components this happens in two stages:

- generating and linking components - to other components or to functions;
- arranging components on the screen.

Creating the *DataManager* subsystem, which configures all the information presentation components, follows this pattern. Figure 5-6 describes the generation and linkage of all components required to produce the Panel shown in Figure 5-10. Additional lines of code are required to arrange the visible Displayers into self-contained glyph hierarchies such as the Display Sets shown in Figure 5-9. Overall, creating the *DataManager* subsystem took approximately 200 ULOC.

The effort required to build *InterfaceManager*, which arranges all control widgets and Display Sets on the control panel, was approximately 400 ULOC. Therefore, using the given toolkit extension classes in the prototype application took no more effort than using the existing InterViews interface classes.

In addition to the *InterfaceManager* and *DataManager* subsystems, the *RobotManager* subsystem - most of which consisted of the Yasnac communication protocol - required approximately 700 ULOC to implement. The total effort required to implement the Panel was 1200 ULOC.

The existing InterViews classes that were used in the Panel worked well with the extension classes. The new interface components such as the Histogram, Readout, and Alarm were easily incorporated into the user interface, because they inherited the glyph protocol that enables composition into a glyph hierarchy.

## 5.5 Evaluation of the cooperative software environment

This section looks at how well the robotic welding software environment supports the operator during setup, in comparison with the existing Yasnac controller environment. The list of requirements in section 4.4 explained why each function is useful in a robotic welding environment, while this section describes how each functional requirement was met; it evaluates the framework from the operator's point of view.

### 5.5.1 Evaluation against requirements

This section evaluates the Panel against the requirements for a cooperative software environment for robotic welding described in section 4.4:

- robot programming and program management;
- control and status of robot systems;
- a graphical interface that mimics real button controls;
- presentation of feedback information from multiple sources;
- highlighting the most important information;
- storage of information for review;
- integration of the robot with other tools;
- abstracted functionality;
- repository of previous job descriptions and their program solutions;
- extensibility.

**Robot programming and management** using the Panel is faster for the operator than using the robot controller environment, since the program editor can be full screen and mouse driven and can be configured to run the user's preferred editor. Dialogue boxes that allow browsing are provided for file management, which reduces the amount of typing and navigating through menus.

**Robot control and status** - the Panel duplicates all the status indicators that were available on the controller, using toggle buttons that are both indicators and actuators.

**Graphical interface** - the Panel's interface is graphical and mouse driven, with single buttons and shallow menus to activate each function of setup. Wherever possible, graphical icons are used to control the presentation of information to minimise key presses, for example, the histogram's vertical scale is adjustable using arrow buttons.

**Data retrieval and presentation** - the Panel provides a flexible environment for observing welding process parameters during welding. Parameters can be combined and processed, then displayed in the most relevant format or in several formats for comparison. Parameters can also be compared to each other in adjoining displayers. The operator can examine the information according to which aspects of the welding process are important and see information displayed in an appropriate format, to make comprehension and hence diagnosis faster and easier.

**Highlighting** - the Panel allows visual alarms to be set on incoming data streams to alert the operator to obvious welding fault conditions. In this prototype, simple limit alarms have been implemented, to demonstrate how this feature could be used for more sophisticated detection algorithms.

**Data storage** - the data logging facility of the *DataManager* subsystem provides a simple and convenient tool for post-weld analysis. The data in the log file is the raw data from the monitoring sources. The log file can be analysed either by replaying the display session or by copying the data to the screen or to paper.

**Integration of tools** - the Panel provides a single development environment for robot programming from which the entire setup procedure can be controlled. Coding, executing and debugging of robot programs can all be performed within the Panel. Parameters can be observed and recorded during execution, then replayed later at normal speed. Program correction is faster and easier because programs can be edited, downloaded, and executed in a single environment.

**Abstracted functionality** - instead of needing to remember complex menu and function key pathways, the operator has all the required functions readily available on the panel. The operator can concentrate on determining whether the trial weld is acceptable or not. The low-level knowledge required to construct robot programs is still necessary, due to the restrictions imposed by the robot system. For example, the operator is still required to know the syntax of the robot programming language. However the interface and the available tools to support programming have been greatly improved, thereby giving the operator a clearer view of the task.

**Repository of cases** - the InterViews FileChooser widget (Figure 2-1) provides simple graphical file manipulation for saving, loading, and deleting files and for retrieving them based on keywords. Working programs can be stored as examples for a particular set of welding conditions, indexed by the appropriate keywords so they can be used as templates (starting points) for similar jobs.

**Extensibility** - the Panel's design is object-oriented, so it can easily be extended by an application developer. The robot-specific command protocol is encapsulated in the *RobotManager* subsystem and can be replaced with a subsystem that encapsulates a different protocol without affecting the rest of the system. The type of functions that the controller provides are likely to be available on any robot that allows remote control. Similarly, the *DataManager* subsystem contains an interchangeable WeldMonHandler that can be replaced or extended with other device-specific components. Data can be displayed using a library of components to which new components can be

readily added. *DataManager* can incorporate new display types, processing functions, and filters.

## 5.6 Summary

This chapter has demonstrated that the information presentation framework can be implemented as a toolkit extension and used as part of a prototype cooperative software environment. The InterViews extension components provide four of the cooperative techniques defined in the taxonomy in 2.1.1. Using these components along with InterViews widgets, the framework design was successfully implemented as an information presentation subsystem and used to build the Panel, a prototype cooperative software environment for robotic welding. The Panel is capable of being fitted with all the main functions relating to setup that are currently available on the robot controller to support both static and dynamic decision-making. Demonstrations of the Panel enabled welding experts to confirm that a fully functional system containing these features would be useful in supporting robotic welding. The toolkit extension classes are used in the same way as existing InterViews toolkit components, and require no more implementation effort to use. The next chapter will discuss contributions and enhancements arising from this research project.

# Chapter 6. Future Work and Conclusions

This chapter discusses the work presented in the thesis with respect to contributions, enhancements, and future work. Enhancements to both the Panel cooperative software environment and to the toolkit extension are outlined. Future work may involve the inclusion of high-level CPS techniques in the framework, evaluation of the framework in other testbeds, and the use of the framework to conduct research into problem solving. Contributions include a cooperative problem solving taxonomy, an information presentation framework that addresses some of these techniques, and a toolkit extension that helps validate that framework.

## 6.1 Enhancements

This section discusses areas for improvement to the work presented in this thesis. Possible enhancements to the Panel include full object-oriented implementation of the Panel, advanced robot program debugging, and a knowledge base of previous cases. Enhancements to the toolkit extension include expert parameter limit alarms, multimedia presentation components, and user configuration of the *DataManager* subsystem.

### 6.1.1 Object-oriented implementation

Not all parts of the Panel were implemented in an object-oriented language: the *RobotManager* subsystem was written in C, and the program editor is a standard text editor invoked from the operating system.

*RobotManager* (as described in section 5.2.2) was largely written in C for convenience, but could be better integrated with the toolkit if it was written in an object-oriented language such as C++. For instance, the protocol needed to interact with the robot controller could utilise the InterViews IOHandler component. An object-oriented design and implementation would also make the components of the *RobotManager* subsystem available for reuse in other machine-specific subsystems. Likewise, a library of communication modules could be created for different vendors' robot controllers.

The Panel provides a text editor for job program modification by invoking an external editor via the operating system. This has the advantage of allowing users to specify their editor of choice. However, a specialised editor for robot programming would improve the coherence of the cooperative software environment, in terms of both function and appearance. Such an editor could easily be constructed using InterViews, perhaps based on Textedit, a sample document editor that is provided with the InterViews installation.

### 6.1.2 Advanced program debugging

The inclusion of timing information with the log file would enable sensory data to be synchronised and directly compared with the steps in the program, perhaps using a step/watch function similar to that of a development environment for computer programming. This feature, combined with the presentation of real-time video and audio would enable more comprehensive analysis of the welding process using a play-back feature. Video, audio, and other sensory data could be compared with the job sequence at various speeds. With more dynamic control of the presented information, the operator would have a better chance of detecting problems in the program.

### 6.1.3 Knowledge base of cases

The ability to recall the details of previous jobs - case-based reasoning - is a common starting point for human problem solving. The operator should be able to search a repository for cases with similar characteristics to the job at hand. The Panel demonstrates a facility for searching through a collection of data files using keywords. However, a better solution would be a knowledge base of cases, organised according to the characteristics of the job. Such a knowledge base could provide sets of parameters for initial specification of similar jobs. In addition, the process of categorising cases according to their characteristics (such as by workpiece metal, joint type, or welding technique) may lead to a better understanding of the problem domain.

### 6.1.4 Parameter limit alerting

The Panel allows visual alarms to be set on incoming data streams to alert the operator to obvious welding fault conditions. A rule base of these conditions could be provided in terms of measured parameters, allowing the operator to retrieve prepared alarms and install them with the appropriate input data streams. In addition, a self-learning, diagnosing expert system could detect more subtle anomalies in the incoming data, such as those relating to closely coupled parameters.

### 6.1.5 Multimedia data presentation

A CPS framework should allow different forms of information to be presented to the operator. This may include multimedia, such as video and audio, which can be particularly useful in situations where the operator is not able to observe the monitored system directly. Some user interface toolkits already cater for multimedia and it is likely that they will routinely do so in the future [5].

How should multimedia be incorporated into CPS toolkits? The cooperative framework has been designed for continuous streams of serial data, but could be adapted to parallel streams of video data. The mechanism of DataHandlers, Filters, DisplayHandlers, Processors, and Displayers should form a sound basis for specialised objects that

work well for these more demanding forms of data. Multimedia presentation is a difficult problem; for example, video and audio data must be synchronised. But given good solutions to these specific problems, this framework can help the developer put the components together to build multimedia CPS systems.

### 6.1.6 User–configured information presentation

In the current toolkit, components are designed to be assembled by the application developer into an information presentation system. The operator is presented with a fixed set of display facilities by the developer. This can result in a mismatch between the operator's task and the monitoring tools available. For example, the operator may only need to examine two parameters to solve a particular problem - the other parameters are irrelevant, so they use unnecessary screen space and may be distracting.

A possible enhancement to the toolkit extension would allow the operator to customise the information presentation system. A dialogue box containing a list of available data streams, filters, processors, and displayers could be used by the operator to specify and configure various views of the data. This feature would not require the current framework to be changed; rather it need only be extended to a higher level to provide the mechanism for dynamic, user configuration of the information presentation system. Instead of information presentation being defined by the application developer within the program, it could be defined by a Constructor class; for example, a configuration file (used for persistent configuration) could be used in conjunction with an interface dialogue box for user manipulation.

The operator would then be able to choose the most appropriate tools from a palette. The view of data could be changed to focus on the current activity, thereby allowing the operator to assess and act on the situation more quickly. The drawback is the need to spend extra time configuring the display elements rather than solving the problem. This could be allayed by having default configurations for the most common activities and by saving frequently used configurations. The operator should have choices, but not be forced to configure every display.

The prototype cooperative software environment could be improved by writing all modules in C++; for example, the *RobotManager* subsystem could be built from a library of communication modules for different vendor's robot controllers. The software environment could incorporate a full knowledge base of solutions to previous cases, organised according to the characteristics of the problem. Another useful source of automated expertise would be a rule base of acceptable parameter ranges, to be used as ready-made limit alarms. The toolkit extension should be expanded to allow the user to configure the information presentation components and to allow the presentation

and synchronisation of multimedia feedback. The next chapter outlines some future work arising from these contributions.

## 6.2 Future Work

Some future work arises out of the framework described in this thesis and from the software prototypes developed. This section discusses the inclusion of high-level CPS techniques in the framework, evaluation of the framework in other testbeds, and the use of the framework to conduct research into problem solving.

### 6.2.1 High-level cooperative problem solving techniques

This section considers how to incorporate the higher-level CPS techniques (as described in section 2.1.1: multiple views, advising, and problem structuring) into the framework.

Advisory modules could be defined as components of the information presentation architecture, able to be associated with particular data streams in the same way as filters and displayers. Although such components would require access to domain knowledge, a generic component could be defined that displays advice to users based on the assessment made by a separate, specialised decision-making module. An advisory module need not be any more sophisticated than a textual display.

Multiple views of the data could be provided by utilising dynamic configuration of the *DataManager* subsystem, as described in section 3.3. This could be realised by adding a higher level *View* component to the framework. A view would be composed of a complete set of information presentation components for a particular purpose, for example a zoom view that shows many different aspects of one parameter and hides all other parameters. Instead of the user configuring the components on an individual component basis, the user switches between these predefined views. The addition of multiple views doesn't require any changes to the framework's basic design, it merely extends it to a hierarchical structure.

Adding problem structuring to the framework will probably prove more complicated. Like multiple views, problem structuring requires a hierarchical structure to be imposed on the components of the information presentation. To follow the model of a cooperative design software environment described by Fisher, there would need to be a critiquing module and a palette of tools (in this case a palette of information presentation components, as described above). The critiquer could then oversee the actions of the operator and make suggestions and corrections based on the system state.

### 6.2.2 Other testbeds

To validate the ideas presented in this framework, InterViews was chosen as a GUI toolkit from which to construct CPS extension classes, while robotic welding setup was chosen as a testbed CPS application. However, a useful framework should be applicable to other toolkits and to a range of similar domains.

To further validate the framework, it should be implemented in other GUI toolkits and applied to other reactive domains by deriving more toolkit components. More work needs to be done to ensure that the design is not biased towards InterViews or robotic welding and to come up with a broadly useful generic specification for CPS extensions.

### 6.2.3 Study of problem solving

A cooperative software environment would be a useful tool for research into machine learning. By having human domain experts working in a responsive and structured environment such as the Panel, it would be possible to record and analyse specific data about decisions that leads to a better understanding of expertise and problem solving. The Panel could be the basis for an intelligent cooperative software environment that learns directly from the decisions made by operators of the environment.

## 6.3 Conclusion

This thesis makes the following contributions to CPS:

1. definition of a cooperative problem solving taxonomy that helps identify:
   - low-level activities that need to be supported in a CPS system for reactive environments;
   - a subset of these activities needed for dynamic replanning of welding parameters, which has been endorsed by welding experts as being useful for robotic welding setup;
2. definition of a framework that supports these techniques to provide a reusable design for the information presentation activities in a cooperative software environment;
3. demonstration of a toolkit extension to InterViews that implements the framework design and is no more difficult to use than the existing user interface toolkit.

The following sub-sections will discuss the contributions made by the taxonomy, the framework, and the toolkit extension in more detail.

### 6.3.1 Cooperative problem solving taxonomy

This section summarises the contributions of the cooperative problem solving taxonomy described in section 2.1.1. Many researchers focus on the specific problems of one application domain, rather than the generic problems of CPS. Instead, this thesis examines the CPS work being done in the domain of ill-structured, reactive tasks, particularly in dynamic replanning of manufacturing processes. Rencken [34] has examined a wide range of adaptive aiding techniques in the literature and classified them according to the method of reducing the operator's workload:

- transformation;
- partitioning;
- allocation.

This thesis classifies a narrower range of techniques appropriate to reactive tasks into a new taxonomy of seven types, ordered from low to high levels of assistance provided to the operator:

- visualisation;
- multiple views;
- filtering;
- highlighting;
- summation;
- advising;
- problem structuring.

This taxonomy is designed to identify the most appropriate low-level techniques for incorporation in a generic cooperative problem solving framework.

### 6.3.2 Information presentation framework

This thesis proposes a framework - a set of cooperating classes for information presentation - which, when used as part of a user interface toolkit, become a CPS toolkit extension. The framework provides application developers with a reusable design for applying CPS techniques to reactive decision tasks.

The techniques described in this framework do not represent all the approaches that have been used in other CPS work. Based on the taxonomy of techniques described in section 2.1, the framework employs four low-level forms of assistance:

- visualisation;

- filtering;
- highlighting;
- summation.

These techniques can be implemented with little or no domain knowledge and are therefore more readily incorporated in a generic framework than, for example, advisory or problem structuring techniques. The framework incorporates these techniques into a design for an information presentation subsystem, which provides dynamic decision support in a cooperative problem solving system.

### 6.3.3 Toolkit extension

The toolkit extension implements the information presentation framework design. Low-level support for CPS can be readily packaged in the classes of an object-oriented toolkit, because the core classes specify the behaviour of all specialised subclasses derived from them. All the application developer needs to do is specialise enough classes to suit the application, then compose both generic and specialised classes into a system, rather than having to design and implement a complete CPS application.

The main advantage of extending a user interface toolkit for CPS is that many CPS requirements can already be met by a GUI toolkit. A CPS software environment needs to cover both static and dynamic aspects of a reactive task. Most static functions and some dynamic functions for control and monitoring are easily provided by various kinds of buttons and indicators. The toolkit extension was only required to implement classes to handle the flow and processing of data into information.

The development of the prototype software environment has validated the toolkit extension in terms of potential for practical applications. The software environment is capable of supporting the operator by presenting information and controls in a timely and appropriate manner. The interface enables the operator to focus on the task, rather than the procedure for performing the task. The toolkit is flexible enough to cater to a wide variety of applications, since it is based on simple components that can be composed in a variety of ways.

### 6.3.4 Summary

This thesis has investigated the problem of supporting cooperative problem solving in complex, reactive systems such as robotic welding. The low level activities which need to be supported if cooperative problem solving is to be deployed successfully in such systems were identified and classified according to a novel cooperative problem solving taxonomy. A framework, i.e. a reusable design containing cooperating components, was defined to support these low-level activities. The framework components were designed to work together in specific ways to present unprocessed sensory data

as information to the user in a timely and appropriate manner. The framework was implemented as an extension to the InterViews user interface toolkit.

In order to validate the framework, the extension was used to build a prototype cooperative software environment for robotic welding. The robotic welding software environment was successful in demonstrating the utility of the framework design in a real-world application.

# Bibliography


[1] Baxter, J.D., Oldland, R.B., Bottomley, R., “The use of expert knowledge in the selection of CIG welding consumables”, *Proceedings of the Australian Joint AI Conference*, pp 190-202, (1988).

[2] Bennett, K.B., “Representation aiding: complementary decision support for a complex, dynamic control task,” *Control Systems* **vol 12 no 4** pp 19-24, (1992).

[3] Bentley, A.E. and Marburger, S.J., “Arc welding penetration control using quantitative feedback theory,” *Welding Research Supplement to the Welding Journal* **vol 71 no 11** pp 397s-405s, (1992).

[4] Bergmann, N.W., Harris, D., Jarvis, D.H., Mudge, J.C., “Collaboration Technology Support for the Distributed Design Process”, *Technical Report MTA 310*, CSIRO Division of Manufacturing Technology, (1994).

[5] Blattner, M.M., Dannenberg, R.B., “Multimedia Interface Design”, ACM Press, (1992).

[6] Borland C++, Part No: BCP1245WW21772, Borland International Inc., 100 Borland Way, Scotts Valley, CA 95066-3249, (1994).

[7] Bruck, D.M., “Experiences of object-oriented programming development in C++ and InterViews”, *Proceedings of TOOLS’89 Technology of Object-Oriented Languages and Systems*, pp123-7, (1989).

[8] Calder, P.R., *Building user interfaces with lightweight objects*, Ph.D. Dissertation, Stanford University, (1993).

[9] Cassar, J.P. and Vanheeghe, P., “Programming Environment for robotized arc welding” pp 809-814 in *Proceedings of the 20th International Symposium on Industrial Robots*, (1989).

[10] Clarke, A.A., Smyth, M.G.G., “A cooperative computer based on the principles of human cooperation.”, *International Journal of Man Machine Studies*, **vol 38**, pp3-22 (1993).

[11] Crawford, J.L., “The Intelligent Graphic Interface Project: operator interfaces for the year 2000”, *Proceedings of the Human Factors Society 36th Annual Meeting. Innovations for Interactions*, **vol.1**, pp 465-9, (1992).

[12] Ferguson, P., Brennan, D., *Volume 6B: Motif Reference Manual*, O’Reilly and Associates, (1993).

[13] Fisher, G., Lemke, C., Mastaglio, T., Morch, A.I.; “The role of Critiquing in Cooperative Problem Solving”, *ACM Transactions on Information Systems*, **vol 9, no 3**, April 1991, pp 123-151, (1991).

[14] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, (1994).

[15] Goldberg, D.E., *Genetic Algorithms in Search, Optimisation and Machine Learning*, Addison-Wesley, (1989).

[16] Guu, A.C. and Rokhlin, S.I., "Technique for simultaneous real-time measurements of weld pool surface geometry and arc force*", Welding Research Supplement to the Welding Journal*, **vol 71 no 12**, pp 473s-482s (1992.).

[17] Harbour, J.L. "Integrating HRA into decision support systems: a new frontier?", *Conference Record for 1992 IEEE Fifth Conference on Human Factors and Power Plants* (Cat. No.92CH3233-4), pp 467-70, IEEE, New York, NY, USA, (1992).

[18] Hemmerle J.S., Terk, M., Gursoz, E.L., Prinz, F.B., Doyle, T.E., "Next generation manufacturing task planner for robotic arc welding", *ISA Transactions*, **vol 31 no 2**, pp 97-113, (1992).

[19] Horowitz, P., "The Art of Electronics", second edition, Cambridge University Press, (1989).

[20] ILOG Rules, ILOG SA, 2 avenue Raspail, BP 7, 94251 Gentilly Cedex, France, (1994).

[21] ILOG Views, ILOG SA, 2 avenue Raspail, BP 7, 94251 Gentilly Cedex, France (1995).

[22] Jarvis, D.H., Seabrook, T.D., and Jarvis, J.H., "Decision Support Systems for Manufacturing", *Proceedings of the 1990 Pacific Conference on Manufacturing*, (1990).

[23] Kempf, K., "Intelligent interfaces for computer integrated manufacturing", *Proceedings of the Third International Conference. Expert Systems and the Leading Edge in Production and Operations Management*, pp 269-79, (1989).

[24] Knuth, D., *The TeXbook*, Addison Wesley, (1984).

[25] LabWindows, *LabWindows/CVI Demonstration Guide for Windows*, Part Number 350173A-01, National Instruments, 6504 Bridge Point Parkway, Austin, TX 78730-5039, USA, (1994).

[26] Linton, M.A., Vlissides, J.M., Calder, P.R. "Composing user interfaces with InterViews", *Computer*, **vol Feb 1989**, pp 8-22 (1989).

[27] Linton, M.A, Calder, P.R., Interrante, J.A., Tang, S., Vlissedes, J.M., *InterViews Reference Manual Version 3.1*, The Board of Trustees of the Leland Stanford Junior University, (1992).

[28] Madsen, O. and Holm, H., "Real time requirements to a CAD and sensor based control system for robotic multi-pass TIG welding," *IEEE International Workshop on Intelligent Motion Control*, **vol 1**, pp 347-54, (1990).

[29] Matlab Professional, Part No: ML402, The Math Works Inc., 21 Eliot Street, South Natick, MA 01760, (1994).

[30] Nann, S.R., Ray, A., and Kumara, S., "A decision support system for real-time monitoring and control of dynamical processes", *International Journal of Intelligent Systems,* **vol 6 no 7,** pp 739-58, (1991).

[31] Norrish, J., *Advanced Welding Processes*, Institute of Physics Publishing, (1989).

[32] O'Connor, L.J., Lan, M.S., Partridge,D.R., Lee,J.M.F., "A case-based approach to automated weld-process design", *Applied Artificial Intelligence*, **vol 6**, pp 315-30, (1992).

[33] Pree, W., *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, (1995).

[34] Rencken, W.D., Durrant-Whyte, H.F. "A quantitative model for adaptive task allocation in human-computer interfaces", *IEEE Transactions on Systems, Man and Cybernetics*, **vol 23 no 4**, pp 1072-90, (1993).

[35] Reilly, R., "Real-time weld quality monitor controls GMA Welding", *Welding Journal,* **vol 70 no 1,** pp 37-41, (1991).

[36] Rettig, Mark, "Cooperative Software," *Communications of the ACM*, (1993).

[37] Roseman, M, *Design of a real-time groupware toolkit*, Master's Dissertation, Department of Computer Science, University of Calgary, Alberta, (1993).

[38] Rumbaugh, J., Blaha, M., Premerlaini, W., Eddy, F., Lorensen, W., *Object-oriented Modeling and Design*, Prentice-Hall, (1991).

[39] Scheifler, Robert W., Gettys, Jim, "The X Windows System", *ACM Transactions on Graphics*, **vol 5 no 3**, Apr 1986, pp 79-109, (1986).

[40] Schneiderman, B., *Designing the User Interface*, Addison-Wesley, (1992).

[41] Schwuttke, U.M., Quan, A.G., Holder, B. "Intelligent data presentation for real-time spacecraft monitoring", *Proceedings of the SPIE - The International Society for Optical Engineering,* **vol 1963**, pp 80-9, (1993).

[42] Sicard, P. and Levine, M.D., "An approach to an expert robot welding system", *IEEE Transactions on Systems, Manufacturing and Cybernetics*, **vol 18 no 2,** pp 204-222, (1988).

[43] Spelt, P.F., Knee, H.E., Glover, C.W., "Hybrid artificial intelligence architecture for diagnosis and decision-making in manufacturing", *Journal of Intelligent Manufacturing*, **vol 2 no 5**, pp 261-8, (1991).

[44] Tsai, M.J., Shi-Da Lin, Meng-Chiun Chen, "Mathematical model for robot arc-welding off-line programming system", *International Journal of Computer Integrated Manufacturing*, **vol 5 no 4-5**, pp 300-9, (1992).

[45] Villanueva, H.E., Arena, S.N., Albertos, P., "Data filtering and presentation for decision support in power stations", *Expert System Application to Power Systems IV Proceedings*, pp 630-5, (1992).

[46] Waller, D.N., Foster, C.J., and Wagner, R., "Real-time imaging for arc welding", *International Journal of Computer Integrated Manufacturing*, **vol 3 no 3/4**, pp 249-60, (1990).

[47] Wirfs-Brock, R., Wilkerson, B., Wiener, L., *Designing Object-Oriented Software*, Prentice Hall, (1990).

[48] Wybo, J.-L., Meunier, E., "Architecture of a decision support system for forest fire prevention and fighting", *Proceedings of the 1993 Simulationa Multiconference on the International Emergency Management and Engineering Conference*, pp 186-90, (1993).

[49] “Yasnac ERC Controller Communications”, 479236-17 Revision 1.0, Yaskawa Electric Corporation, 2-1, Kurosaki-shiroishi, Yahatanishi-ku, Kitakyushu 806, Japan.

# Appendix A. Framework Class Reference

This appendix describes the classes that make up the information presentation framework.

The use of italics in the names of member functions indicates that the function has been inherited from the parent class.

**NAME**

DataHandler (abstract class, derived from IOHandler)

**SYNOPSIS**

Link data streams representing measured parameters to DisplayHandlers; manage ports; pass data streams to DisplayHandlers.

**DESCRIPTION**

A DataHandler is associated with a file descriptor (typically an external port) through which it will receive and manage data streams. A data stream is a series of values arriving over a period of time representing a single measured quantity. The DataHandler can link a single data stream to a Display Handler and may also link a filter to a data stream. When a block of data arrives, DataHandler is notified by a Dispatcher (InterViews class). DataHandler reads the block of data, which represents the latest acquired values in each data stream from the port and interprets the data block as separate values according to the information stored in a subclass-defined data format file. DataHandler extracts the value in each stream and writes it to the appropriate DisplayHandler.

**PUBLIC MEMBER FUNCTIONS**

**DataHandler(char* portname)**

Accepts the name of the port from which the DataHandler will receive data.

***virtual int inputReady(int port)***

Notification that data is ready on the port. Reads and interprets data from the port, applies any Filters, then passes data streams to linked DisplayHandlers. Subclasses must redefine this function.

***virtual void timerExpired(long sec, long usec)***

Timer expiry notification. Resets timer and gets more data. Subclasses must define this function.

**void linkStream(const char* stream1, const char* stream2, DisplayHandler* disphdlr, Filter* filter = nil)**

Links one or two data streams to a DisplayHandler and optionally to a filter. The data streams are represented by string names, which correspond to names listed in a device-specific format file. The filter is passed as a pointer and initialised to “nil” to make it an optional parameter.

**virtual void startRetrieve(char* logfilename)**

Starts retrieving the data from a data acquisition device attached to the DataHandler's port. If a logfilename is specified, the incoming data will be stored in it, in unfiltered and unprocessed form. Subclasses must define this function.

**virtual void stopRetrieve(char* logfilename)**

Stops retrieving data from the port.

**void startReplay(char* logfilename)**

**void stopReplay(char* logfilename)**

Start and stop a logged data replay session.

**NAME**

Filter (abstract class)

**SYNOPSIS**

Removes or modifies floating point values from a data stream that match the given specification.

**DESCRIPTION**

Filter checks whether each received value matches an undefined filter specification. The returned value depends on whether or not the input value matches the filter specification.

**PUBLIC MEMBER FUNCTIONS**

**virtual void applyFilter(float* value)**

Compares the input value with the filter removal specification. If it matches, filter returns an acceptable substitute value (which may be zero). If it does not match, filter returns the input value. Subclasses must define this function.

**NAME**

DisplayHandler (concrete class)

**SYNOPSIS**

Link data streams to Processors and Displayers; send data values to Displayers via Processors.

**DESCRIPTION**

DisplayHandler maintains associations between each data stream that represents a parameter with one or more Displayers that will display the values in the data stream. DisplayHandler receives a data stream and possibly applies a Processor before writing the stream to the appropriate Displayers.

**PUBLIC MEMBER FUNCTIONS**

**linkStream(Displayer* displayer, Processor* processor)**

Create a link between this DisplayHandler's data stream (one or two) and a Displayer; optionally link a Processor to the data stream.

**void update(float* value)**

**void update(float* value1, float* value2)**

Notify the DisplayHandler to send the given value(s) to all the Displayers it maintains.

**NAME**

Processor (abstract class)

**SYNOPSIS**

Combine and/or process a single value or a series of values (data stream).

**DESCRIPTION**

Processor takes either one or two floating point values as inputs and performs a subclass-defined operation on them, then returns the resulting floating point value. Subclasses of Processor may be defined to accept only one value, or only two values, or either one or two values as input.

**PUBLIC MEMBER FUNCTIONS**

**virtual float Process(float* value)**

**virtual float Process(float* value1, float* value2)**

Perform a calculation on the input value(s) and return the result. Subclasses must define this function.

**NAME**

Displayer (abstract class, derived from MonoGlyph)

**SYNOPSIS**

Display a series of values from a data stream.

**DESCRIPTION**

Displayer is an abstract class that defines the interface through which data is passed to it; it does not define how the data is displayed.

**PUBLIC MEMBER FUNCTIONS**

**Displayer()**

Uses a Patch (InterViews class) to create a basic redrawable display element.

***virtual void draw(Canvas* canvas, const Allocation alloc)***

Draws the Displayer and its contents on the user screen.

**virtual void update(float* value)**

Updates the displayed value with a new value. Subclasses must define this function so that only the dynamic parts of the Displayer's appearance are changed. For example, a histogram's axis labels may be static while the bars that represent the values are dynamic.

**virtual void clear()**

Clears all displayed values from the Displayer. Subclasses must define this function.

# Appendix B. Panel Class Reference

This appendix describes the classes used to construct the Panel cooperative software environment, based on the information presentation framework design. ClipperFilter and Multiplier were implemented as C functions rather than classes for building the Panel. However, they are described here in class form to indicate the effort required to implement them as classes.

In this appendix, the use of italics in the names of member functions indicates that the function has been inherited from the parent class.

**NAME**

WeldMonHandler (concrete class, derived from DataHandler)

**SYNOPSIS**

Link floating point data streams to Filters and DisplayHandlers; manage the retrieval of data from Welding Data Monitor (WDM); pass data streams to linked Filters and DisplayHandlers.

**DESCRIPTION**

A WeldMonHandler is associated with a file descriptor (port) through which it will receive and manage data streams. WeldMonHandler manages the interaction with the WDM that acquires welding data. To start retrieval, WeldMonHandler polls the WDM for data until receiving the order to stop. As each block of data arrives, WeldMonHandler extracts the values, then writes the values to the appropriate DisplayHandler.

The incoming data can be stored in unprocessed form and replayed to appear as though it is being received live.

**PUBLIC MEMBER FUNCTIONS**

**WeldMonHandler(char* portname)**

The constructor stores the name of the port through which WeldMonHandler will retrieve data from the Welding Data Monitor.

***virtual int inputReady(int port)***

Notification that data is ready on the port. Reads and interprets data from the port, applies any Filters, then passes data streams to linked DisplayHandlers.

***virtual void timerExpired(long sec, long usec)***

Timer expiry notification. Resets timer and gets more data, either from the WDM or from the log file. Getting more data from the WDM involves sending it the “receive” command.

***void linkStream(const char* stream1, const char* stream2, DisplayHandler* disphdlr, Filter* filter = nil)***

Link one or two data streams to a DisplayHandler via optional filter(s). The Filter parameter is initialised to “nil” to make it an optional parameter. Data streams are represented by string names, which correspond to names listed in a WDM data format template file, as shown in Figure B-1. The template file describes which column of the WDM data file contains each named data stream, as shown in Figure B-2. The data file is an ASCII text file, in which

each column represents a data stream, and each line represents a set of values measured simultaneously by the WDM.

```
VOLTS   0
AMPS    1
```

Figure B-1 Sample WDM data format file

```
8.3     234     115     3091
11.6    362     113     4224
11.5    362     113     4179
11.8    364     115     4328
12.2    366     113     4492
12.4    366     113     4554
12.6    366     113     4642
12.7    366     113     4681
12.9    366     113     4733
13.0    364     113     4783
```

Figure B-2 Sample WDM data file

Note that in the InterViews toolkit extension implementation of the framework, the filter is defined as a C function, so it is passed as a C function pointer.

***void startRetrieve(char* logfilename)***

Opens the port to which the Welding Data Monitor is attached. Starts polling the Welding Data Monitor for data, using a timing loop. Reads data from the port as it arrives, when notified by the Dispatcher (InterViews class that performs interrupt driven data detection). If a logfilename is specified, the incoming data will be stored in it.

***void stopRetrieve(char* logfilename)***

Stops retrieving data from the port by stopping the timing loop and closing the port.

***void startReplay(char* logfilename)***

Opens the logfile and starts to replay the data by reading and interpreting the file in the same way as for the port.

***void stopReplay(char* logfilename)***

Stops replay of data and closes the logfile.

**NAME**

ClipperFilter (concrete class, derived from Filter)

**SYNOPSIS**

Remove floating point values from a data stream that fall below the threshold.

**DESCRIPTION**

ClipperFilter checks whether each received value matches its given threshold. A value is returned according to whether or not the value was below the threshold.

**PUBLIC MEMBER FUNCTIONS**

**void ClipperFilter(float threshold)**

Accepts a threshold value.

***void applyFilter(float* value)***

Compares the input value with the filter threshold. If it matches, filter returns the threshold value. If it falls above, filter returns the input value.

**NAME**

Multiplier (concrete class, derived from Processor)

**SYNOPSIS**

Multiply the floating point values from two data streams.

**DESCRIPTION**

Multiplier takes two floating point values as inputs and multiplies them together, then returns the resulting floating point value.

**PUBLIC MEMBER FUNCTIONS**

***float Process(float* value1, float* value2)***

Multiplies the input values and returns the result.

**NAME**

Readout (concrete class, derived from Displayer)

**SYNOPSIS**

Display a series of floating point data values in the form of a numeric gauge.

**DESCRIPTION**

Readout accepts one value at a time and displays each one immediately in numeric format, to a specified number of decimal places. The precision is specified in the Readout's constructor function.

**PUBLIC MEMBER FUNCTIONS**

**Readout(int precision, WidgetKit* widget)**

Creates itself using widget components. Stores the number of decimal places to which it must display incoming data values.

***virtual void draw(Canvas* canvas, const Allocation alloc)***

Draws the Readout and its contents on the user screen.

***void update(float* value)***

Replaces the current value with the new value to the specified number of decimal places. Redraws itself using the patch.

***void clear()***

Resets the displayed value to zero.

**NAME**

Alarm (concrete class, derived from Displayer)

**SYNOPSIS**

Light up a visual alarm indicator whenever a floating point value is higher than a user-specified limit.

**DESCRIPTION**

Alarm accepts one value at a time and checks whether a value is higher the current upper limit. If so, its indicator lights up; if not its indicator stays off. The indicator light takes the form of a toggle button that bears a light indicator with two states: dark means "Ok" while light means "Alarm". The limit is set by the user typing a numerical value into the input field.

**PUBLIC MEMBER FUNCTIONS**

**Alarm(LayoutKit* layout, WidgetKit* widget)**

Composes itself from layout and widget components.

***virtual void draw(Canvas* canvas, const Allocation alloc)***

Draws the Alarm and its contents on the user screen.

***void update(float* value)***

Compares the value to the current upper limit; if the value exceeds the limit, the indicator region lights up (becomes a light tone/colour). If the value falls below the limit, the indicator light goes out (becomes a dark tone/colour).

**void reset()**

Resets the indicator light to "Ok" (dark).

***void clear()***

Resets the current limit to zero.

**NAME**

Histogram (concrete class, derived from Displayer)

**SYNOPSIS**

Display a floating point data stream in the form of a dynamic vertical bar histogram.

**DESCRIPTION**

A Histogram accepts one value at a time and displays each one in the form of a vertical bar on the histogram. The size of the bar will represent the proportion of that value to the current scale range. The horizontal axis of the bar represents elapsed time, relative to the frequency of the arrival of values. When the horizontal axis has been filled with bars, the Histogram shifts the bars to the left to give the appearance of the viewscreen "panning" to the right along the time axis, so that the most recent values are always displayed.

The height of the histogram (i.e. length of the vertical axis) is specified in the constructor function. The user can adjust the vertical scale of the histogram (i.e. the scale relating to the size of the values) at any time by clicking on the up or down arrow buttons. The scale can be adjusted up or down by a factor of ten. The histogram can be cleared of all values at any time.

**PUBLIC MEMBER FUNCTIONS**

**Histogram(int height, int width, LayoutKit* layout, WidgetKit* widget)**

Composes itself from layout and widget components and places both its axis labels and viewscreen inside separate patches to allow them to be updated dynamically. Height specifies the maximum length for a bar representing a value. Width specifies the maximum number of values that can be displayed along the horizontal (time) axis, based on a fixed bar width.

***virtual void draw(Canvas* canvas, const Allocation alloc)***

Draws the Histogram and its contents on the user screen.

***void update(float* value)***

Adds a new bar to the right hand end of the horizontal axis. If the viewscreen is full along the time axis, shifts all bars once to the left so that the oldest bar disappears before adding the new bar on the right hand end. Redraws itself using the viewscreen patch.

***void clear()***

Removes all bars from its screen.