

The Front End of Software Development:  
Implementation of a Formally Specified User Interface

**A Thesis  
in TCC402**

Presented to

The Faculty of the  
School of Engineering and Applied Science  
University of Virginia

In Partial Fulfillment  
of the Requirements for the Degree  
Bachelor of Science in Computer Science

by

Meng Yin

**March 21, 1997**

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for Papers in Humanities Courses.

\_\_\_\_\_  
(Full signature)

Approved \_\_\_\_\_  
(Technical Advisor) (Type Name) (Signature)

Approved \_\_\_\_\_  
(TCC Advisor) (Type Name) (Signature)

## Preface and Acknowledgment

As a fourth-year computer science student here at University of Virginia, I was fortunate to have had the opportunity to be part of a small research group, led by Professor John Knight. I had a chance to get hands-on experience in an exciting research project. I was able to apply what I have learned while learning new things everyday. Over the course of the project, I was able to both expand my knowledge on theory and sharpen my programming skill, something I dearly enjoy.

My initial motivation for joining the research was to fulfill the requirement of this technical report. However, this project has offered me more than I had ever hoped for. Some of the benefits I didn't even expect. Besides the new technical knowledge I have obtained from the research activities, it helped me develop an effective communication skill as well as a frame of mind – a serious attitude and respect – towards software development. What I have learned from this project will go a long way in preparing me for a career outside the University. Perhaps the most satisfaction I derive from knowing that I have contributed in some way to this research.

Sincere thanks go to all of the people with whom I had the pleasure of associating. I have learned something important from everyone I worked with. Special thanks go to Professor John Knight, whose encouragement and guidance is deeply appreciated.

# Table of Contents

<b>ABSTRACT.....</b>	<b>1</b>
<b>CHAPTER 1: INTRODUCTION .....</b>	<b>2</b>
<b>CHAPTER 2: PROJECT OVERVIEW .....</b>	<b>4</b>
2.1 PROJECT BACKGROUND.....	4
2.1.2 <i>state-of-the-art of user interface implementation</i> .....	5
2.1.3 <i>project environment and platform</i> .....	6
2.2 IMPLEMENTATION OVERVIEW.....	7
2.2.1 <i>system architecture</i> .....	7
2.2.2 <i>user-interface architecture</i> .....	7
<b>CHAPTER 3: IMPLEMENTATION PROCEDURE .....</b>	<b>10</b>
3.1 GRAPHICAL IMPLEMENTATION.....	10
3.1.1 <i>graphical elements in the user interface</i> .....	10
3.1.2 <i>OWL implementation of graphical elements</i> .....	11
3.2 SYNTACTIC IMPLEMENTATION.....	12
3.2.1 <i>callback functions</i> .....	13
3.2.2 <i>token generation</i> .....	13
3.2.3 <i>the token queue</i> .....	13
3.2.4 <i>associated windows programming</i> .....	14
3.2.4.1 <i>windows threads programming</i> .....	14
3.2.4.2 <i>windows sockets programming</i> .....	14
3.3 SEMANTIC IMPLEMENTATION – GENERATE PARSERS USING YACC .....	15
<b>CHAPTER 4: ANALYSIS AND DISCUSSION .....</b>	<b>17</b>
4.1 DIFFICULTIES IN IMPLEMENTATION .....	17
4.2 DISCUSSION.....	17
<b>CHAPTER 5: CONCLUSION.....</b>	<b>19</b>
5.1 FUTURE IMPROVEMENT AND RECOMMENDATION.....	19
5.2 CONCLUSION OF PROJECT ACTIVITIES.....	19
<b>APPENDIX A: BIBLIOGRAPHY.....</b>	<b>21</b>
<b>APPENDIX B: REACTOR OVERVIEW .....</b>	<b>23</b>
<b>APPENDIX C: SOURCE CODE LISTING .....</b>	<b>26</b>

# Table of Figures

FIGURE 2-1 THE SPECIFICATION STRUCTURE OF THE USER INTERFACE .....	5
FIGURE 2-2 THE OVERALL SYSTEM ARCHITECTURE .....	7
FIGURE 2-3 THE USER INTERFACE ARCHITECTURE .....	8
FIGURE 3-1 LIGHTS .....	11
FIGURE 3-2 GAUGES .....	11
FIGURE 3-3 STATIC PANEL .....	12
FIGURE 3-4 SHIM ROD PANEL .....	12
FIGURE 3-5 STRIP-CHARTS .....	12
FIGURE 3-6 AN EXAMPLE SCHEMA IN THE FORMAL SPECIFICATION .....	15
FIGURE B-1 UNIVERSITY OF VIRGINIA REACTOR (UVAR) .....	25

## Abstract

This technical report outlines the project and chronicles the research activities conducted by the author in the software implementation of a formally specified user interface.

As software applications diffuse into every facet of our contemporary society, controlling from the Wall Street Stock Exchange to our personal security, the magnitude of their consequence is becoming more apparent. The correct engineering approach and attitude are required for software development to avoid a billion-dollar damage by a switch-board crash or the loss of a single life. To address this concern, a study is undertaken here at University of Virginia to study software survivability. This study especially targets the safety critical application domain as it uses a small nuclear research facility as the case study tool.

The goal of this study is to develop a reliable prototype control system via the formal specification approach. This research is currently in the user-interface development stage. As chapter 2 will illustrate, the formal specification for the user interface plays a crucial role in the overall user-interface architecture.

The role of the author in this research project is confined to this user-interface development – implementing the graphical user interface according to the architecture discussed in chapter 2. Chapter 3 will describe in detail the procedure followed in this implementation process. The levels of implementation discussed in this chapter include the lexical and the syntactic implementation. The third level, semantic implementation is currently on-going and therefore is not included.

The discussion section in chapter 4 of this technical report will show that the framework of the user interface is in place. The lexical and the syntactic implementation have addressed many of the risks associated with this development. Chapter 4 also includes a listing of any difficulties encountered in the process.

For future research in the same area, chapter 5 will offer recommendations on improving the implementation techniques.

## Chapter 1: Introduction

This technical report demonstrates, via a case study, the feasibility of *formal specification* as applied to user interface implementation. This case study involves designing and implementing a software user interface for a small research nuclear reactor control system. The issue of concern addressed by the formal specification lies in the *semantics* of the user interface, which defines the exact consequence of an arbitrary user action.

As a *demonstration of concept*, or *feasibility*, a comprehensive representation of the nuclear reactor is not essential as long as the core mechanism, or model, is captured. Based on this idea, the user interface can be a simplified representation, a prototype, of the real reactor control panel. Only the features essential to the demonstration of this formal specification concept are implemented. Despite its reduced complexity, this case study shows the practical advantages a formal specification offers in facilitating systematic and reliable user interface design and implementation.

A formal specification is an extension, or application, of discrete math in the development of a software model. This specification is written in *formal languages*, and hence the name. Formal languages are sets of discrete math symbols and notations specialized in representing and encapsulating relations. As applied to this user interface specification, the formal languages define the semantics associated with the controls on the user interface. For example, if the pressing of a button activates a temperature display, then the semantic specification will associate the button with the display. In a similar manner, this formal specification defines the relational model between the objects on the user interface.

The concept of formal and rigorous specifications for software has been around for decades; the existence of safety-critical applications even longer. Research which applies formal specification to software safety-critical systems, however, has been very insufficient. The reason is that hard-wired logic can much more easily implement safety than can software. As a result, the lack of research in this field has led to a critical lack of understanding in software safety in the industry. Today, many existing software-driven safety-critical systems did not undergo during their development the rigorous process demanded by their potential consequences. Air-traffic control systems and medical software systems used in operations are prime examples. This project attempts to fill in the gaps, and make a clear connection, between formal specification and safety critical applications.

The activities associated with this technical report are part of a larger effort undertaken at the University of Virginia, focused on software survivability and reliability. This project is funded primarily by the National Science Foundation(NSF), and is led by Prof. John Knight of the Computer Science Dept. at UVa and Prof. Sue Brilliant of

Virginia Commonwealth University. This research also involves cooperation from the Nuclear Engineering Dept., which provides access to the research reactor facilities as well as domain expertise in the nuclear engineering field. The successful completion of this project will not only provide a great deal of knowledge for future research, but also contribute to the establishment of a model by which other safety-critical systems can be developed. Therefore, the result of this research can have a strong impact in the industrial sector as well as the academic. However, at present, there is no commercial implication or inclination associated with this project.

This technical report serves as a documentation and evaluation of the activities conducted by the author throughout this research project. It also provides an in-depth discussion and analysis of the implementation methodology undertaken. It can assist similar research project in the future as a reference. The following sections of this technical report will describe the procedure followed in the implementation process(Chapter 3), discuss the implication of the implementation decisions(Chapter 3,4), analyze the results(Chapter 4), and assess the overall project regarding what it accomplished(Chapter 4,5). Since this research investigates specifically in the safety-critical application domain, any conclusion drawn from the research may also be specific to this domain. However, as we shall see, some underlying guidelines and principles are universal to all disciplines.

## Chapter 2: Project Overview

The first section in this chapter describes the background and motivation of the research project, the concerns this project addresses, and the questions it attempts to answer. The later sections will explain the large-scale design decisions.

### **2.1 Project Background**

It was only half a century ago in wartime when man first learned to harness the power of the atoms. It was merely a few decades ago when nuclear power production as a peacetime energy source first became feasible. Then began man's endeavor in the nuclear era. However, this short journey was not without impediments. As we shall see in this section, tragedies in the short history of nuclear power production have led to serious reflections in the engineering of safety-critical applications.

#### **2.1.1 project motivation**

Before the spring of 1979, nuclear reactor meltdown was merely a concern and *potential* threat. On March 28 of that year, that threat became very real. At the Three Mile Island nuclear station near Harrisburg, Pennsylvania, a meltdown in Unit 2 exposed the reactor core and destroyed half of it. The investigation by the Nuclear Regulatory Commission ended with the conclusion of *human error*. Or is it? Would the same accident have occurred if the user interface was better designed so that the operator couldn't possibly have misread the loss-of-coolant warning?

While the NRC was still trying to recover public trust in nuclear power in the U.S., another disaster struck in a distant part of the world – the infamous Chernobyl accident. On April 26, 1986, unit 4 of the nuclear power station at Chernobyl in the Soviet Union underwent a violent explosion that blew the reactor open, exposing the reactor core. Without any type of containment structure, the radioactive materials were spewed freely into the atmosphere and the surrounding environment. The contamination didn't stop there. The circulation in the upper atmosphere carried the radiation far into the western Soviet Union and some Eastern European countries, causing billions of dollars in damages. Like the Three Mile Island accident, the Chernobyl disaster was a major setback in the short history of nuclear technology. The voice of public support of nuclear power in the international community went silent. The investigation succeeding the accident again concluded that the cause was human error: the operator disengaged the safety system and then failed to recognize that the controls had malfunctioned until it was too late. Again we have to ask ourselves: is it really human error? Or is it a deficiency, a deadly flaw, in the user interface?

Since that tragedy, the nuclear reactors in the U.S., as well as abroad, have maintained a steady state of safe operation with reliable analog control systems. With the approach of the 21<sup>st</sup> century, the overwhelming tide of technology will inevitably present us a new challenge – the transformation of technology from *analog* to *digital*, from

*hardware to software*. For example, such a transformation in a nuclear power plant would involve replacing the hardwired analog circuitry with digital logic that is programmed entirely in software. As a matter of fact, this transformation has already taken place in most facets of our society. Nuclear stations, however, have lagged considerably behind, for good reasons.

The main problem facing software engineers is that the reliability of such a software system is yet to be demonstrated convincingly. In an industry like nuclear power production where public trust is the sole determination of its destiny, reluctance to accept an unproven system is perfectly understandable, and expected. One proposed solution to this dilemma involves the development of a rigorous *specification* of the control system in which the user interface is a major component. The rigid syntax and the precise semantics of this specification allows it to be analyzed, understood, and agreed upon by both the software engineers and the nuclear engineers. As a result, this specification will serve as a solid foundation for the implementation phase that follows.

Such is the circumstances and motivation for this software survivability study. This technical report will focus on the implementation process for the user interface. It will chronicle the activities undertaken and the design and implementation decisions made along the way.

### 2.1.2 state-of-the-art of user interface implementation

The user-interface design of a safety-critical system, whether it is the cockpit of an airliner or the control panel of a nuclear reactor, can be decomposed into three levels: the semantic, the syntactic, and the lexical level as shown in Figure 2-1. Each level of design is specified in a different way using a different formal language. The semantic specification of the user interface defines the meaning of every possible user action using the formal language Z(pronounced “zed”, [2], [5]); the syntactic specification defines the hierarchy and the sequence of permissible user actions, as well as the token generated from each action, using BNF([7]), another formal language; the lexical specification defines the graphical representation of the elements on screen using the Object Windows Library(OWL) of Borland C++. The goal of formally specifying the user interface is to create an exact specification which has no ambiguity regarding the operation of the user interface. Such a precise specification also facilitates the implementation process.

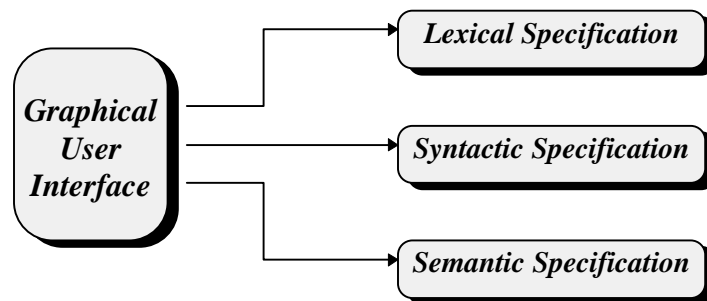


Figure 2-1 The specification structure of the user interface

The importance of a well designed user interface in a safety-critical system is apparent. Every action in the communication between the operator and the system begins and ends in the user interface. The *usability* and *reliability* of the user interface is crucial to the “daily safe operations of the system”([12]). In fact, they are perhaps the two most critical criteria in judging a user interface. The interface of a nuclear reactor control system must convey to the operator the *correct* information in the clearest and the most efficient manner. We often attribute mistakes to the incompetence of the operator while design flaws – or even serious errors – of the user interface go undetected. History often suggests that bad designs are more detrimental to operation safety than “incompetent” users. Hence, a good design of the user interface is unquestionably an effective – maybe the most effective – measure to reduce the probability of the so called “human error”.

### 2.1.3 project environment and platform

The following decisions/constraints specify the environment in which this project is conducted. Some of these decisions are dictated by the current state-of-the-art of software technology, and are subject to change as new developments – new operating systems and new software packages, for example – unravels.

- *Programming language* – C++  
C++ is the choice for this project because it is a powerful, *object oriented* programming language.
- *Target platform* – Windows 95, Windows NT 4.0  
The current platform is Windows 95, which supports multitasking. The source code is expected to be ported to Windows NT 4.0 – which is a true 32-bit multitasking operating system and has much more robust security features – in the near future.
- *Software package*: Borland C++ 5.0, OWL graphical library  
The Borland C++ package has a very rich graphical library and capabilities to help design and implement the user interface. It also has a good compiler. However, the Microsoft Visual C++ and its MFC graphical library may be the choice for future development of this project.
- *Hardware*: Pentium 166 MHz PCs, running Windows 95  
These computers have sufficient resources and power for this development.
- *Communications(for simulation)*: standard Internet socket communication  
A networked simulation uses standard Windows sockets for communication. The computers have direct Internet access.

## 2.2 Implementation Overview

On the outset of this project, certain large-scale design decisions were made regarding the subsequent implementation approach. Among them are the overall system architecture and the user-interface architecture. Throughout the implementation phase, issues on a smaller scale led to many more finer decisions, including the implementation of parallel execution and the processing of token stream. The following sections describe the most important of these implementation decision as they are crucial to the understanding of the underlying methodology.

### 2.2.1 system architecture

A reliable control system demands a sound *system architecture*. So what constitutes a sound architecture? First, it should optimize implementability. Such an architecture should give clear structure to the implementation and reduce its complexity. Second, it should be robust; i.e., it should facilitate, rather than hinder, modifications to the overall system. Third, it should optimize security. In a safety-critical application such as this, failure(or breach of security) in one part of the system should have a minimal effect on another.

The architecture implemented in this study attempts to decompose the system and isolate its different components, as depicted in Figure 2-2, by having each component reside on a different machine. The individual components in this three-tiered architecture communicate through reliable network socket connections. This divide-and-conquer approach to system architecture design simplifies the task of ensuring reliability into smaller constituents. Furthermore, modification to one component does not propagate to other components as long as the interface structure between them remains the same.

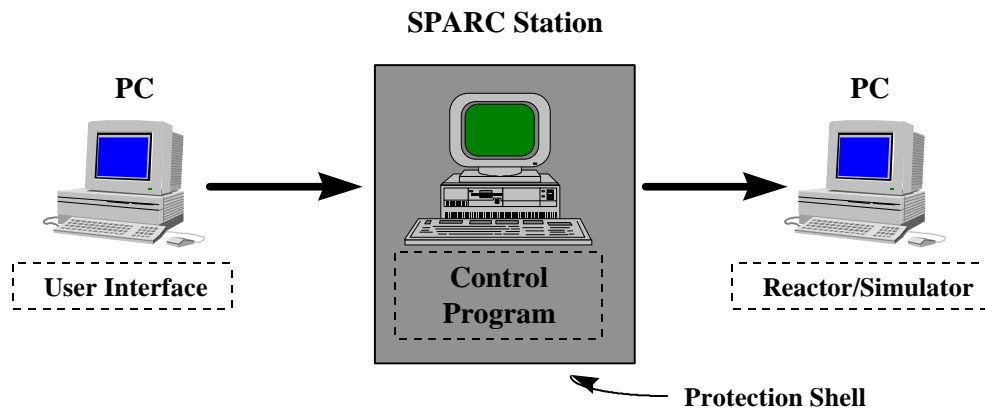


Figure 2-2 The overall system architecture

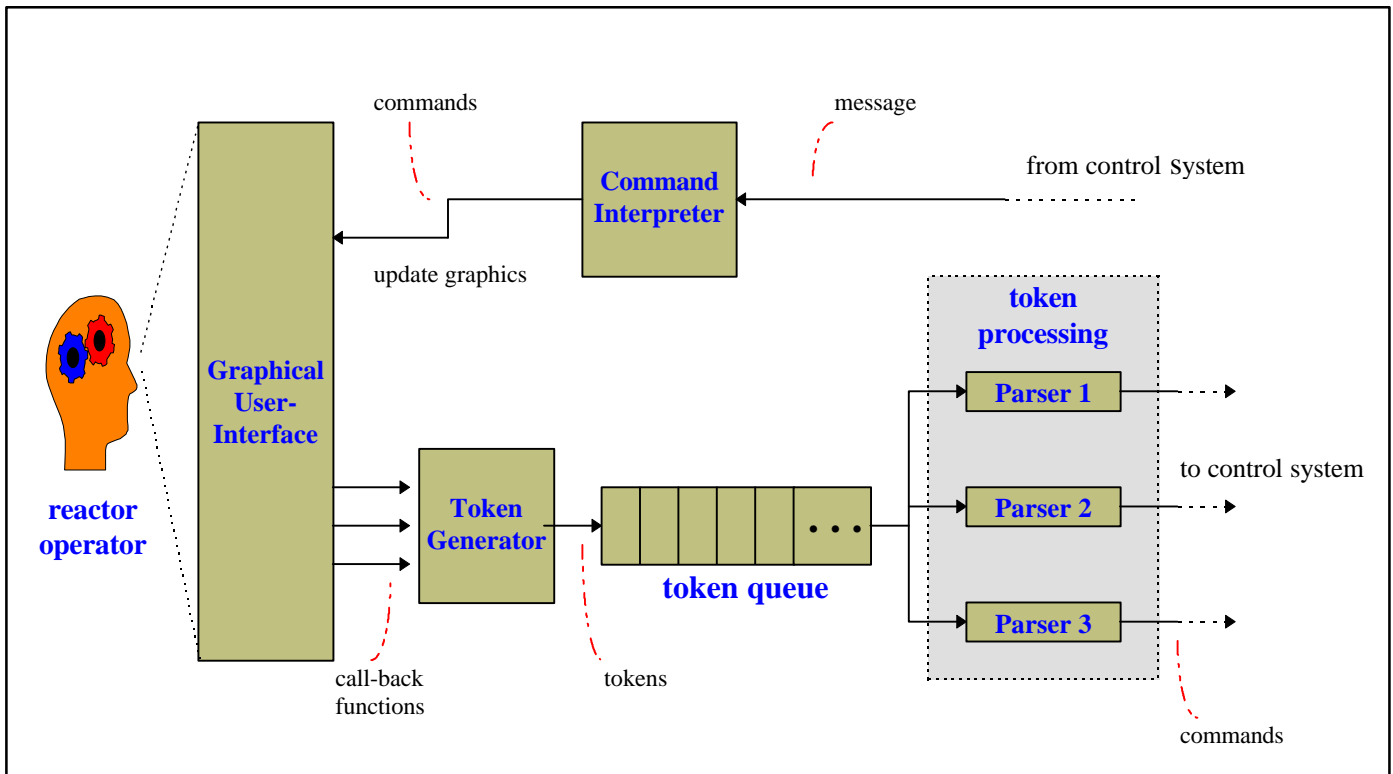
### 2.2.2 user-interface architecture

Figure 2-3 illustrates the architecture of the user interface. In this architecture, the graphical component of the user interface is separated from the syntax-analyzing parsers.

The two modules are linked by a token stream. The tokens are generated by the callback functions of the graphical user interface. These tokens are stored in a queue from which they can be retrieved and processed by the parsers. The parsers execute in parallel and each is responsible for a certain set of tokens. They convert valid tokens into commands to the control system. So, for example, if the operator presses the ‘up’ button of shim rod #1, and this action generates a valid token, then a legal command will be sent to the control system to raise shim rod #1.

The graphical user interface, the module that interact with the user, has in itself an *event-driven* architecture. The callback functions associated with each valid event are part of the syntactic implementation. They define the tokens – the syntax of information – that should be processed.

Information coming from the control system will be processed by the *command interpreter* module. The messages are interpreted and transformed into commands to update various displays in the graphical interface, including gauges, alarm lights, and digital output.



**Figure 2-3 The user interface architecture**

The most important component in this structure is the parser module. Several issues are of concern in particular. First, these parsers are generated automatically from the formal specification. The procedure includes implementing the specification in YACC, a syntax-analyzing language, and then translating the YACC source code into a C++

function via the YACC compiler. The resulting C++ function is then integrated into the user-interface architecture.

A second issue concerns parallel execution among the parsers. The execution of each parser is independent from those of others as they process different set of tokens. This conceptual independence suggests that they should be implemented as such. This task would involve multithread programming in C++.

The parser module serves as a firewall against the sending of invalid and illegal commands to the control system.

## Chapter 3: Implementation Procedure

The user interface implementation can be divided into three categories: *graphical* implementation, *syntactic* implementation, and *semantic* implementation.

Graphical implementation refers to the visual programming of user-interface elements, such as gauges and buttons. The scope of graphical implementation is limited to the appearance of the controls – e.g., shape and form – but not function.

Syntactic implementation refers to implementing the internal operation mechanism of the user interface. It includes programming the callback functions of the graphical controls(i.e. events), token generation, communications between modules, messages, etc. – elements in the user interface architecture that are not visible to the user but essential to its correct operation.

Semantic implementation refers to the implementation of the syntax-checking parsers, which define the semantics associated with each legal user action. The following sections will discuss each in detail.

### 3.1 Graphical Implementation

This implementation step involves programming with the Object Windows Library(OWL) of Borland C++. OWL is a graphical library designed to facilitate Graphical User Interface(GUI) implementation. It provides many base classes as well as specialized classes that are essential for a window-based application.

#### 3.1.1 graphical elements in the user interface

Many of the controls in the GUI are established by a previous version. Their purposes and behaviors are already defined though their appearance can change. Some of these controls are programmer-defined, such as gauges and lights, while others are provided by OWL, such as buttons and text. The following is a list of programmer-defined controls currently in the GUI along with a description of their purposes.

◆ *Lights*

- ⇒ controls that have a binary state
- ⇒ turns on or off corresponding to the presence or absence of a condition
- ⇒ output device

◆ *Gauges*

- ⇒ controls that have a continuous state
- ⇒ has marker indicating the current value if it is inside the predefined range
- ⇒ output device

◆ *Panels*

- ⇒ graphics that provide visual grouping of controls
- ⇒ has caption and 3-D borders
- ⇒ static display

◆ *Rods*

- ⇒ graphics indicating the current position of a control rod
- ⇒ simulates a rod in a tube with an analog display
- ⇒ output device

◆ *Strip-charts*

- ⇒ graphics to display a time-varying element
- ⇒ simulates a paper strip-chart recording device
- ⇒ output device

### 3.1.2 OWL implementation of graphical elements

The ObjectWindows Library is a collection of C++ classes designed to facilitate graphical user interface(GUI) implementation. In many ways, OWL is a formal language: it implements a clear hierarchical structure among classes; it defines a precise behavior, i.e., semantics, of each class. Therefore, it is a reasonable choice for the implementation of the lexical specification of user interface.

The following screen dumps are the graphical representations of the user-defined controls described in the previous section.

◆ *Lights*

- ⇒ class: Light
- ⇒ instantiation: specifies size, caption, and colors at run-time
- ⇒ notes: has on color and off color; off color is the default

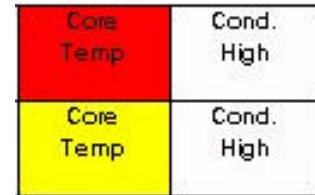


Figure 3- Lights

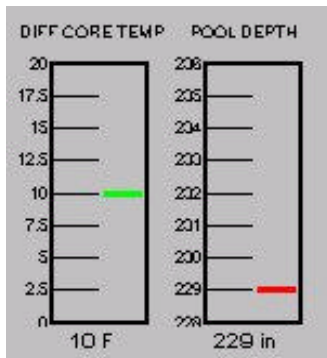


Figure 3-2

◆ *Gauges*

- ⇒ class: ComboGauge, LogComboGauge, PeriodComboGauge
- ⇒ instantiation: specifies size, caption, range and unit
- ⇒ notes: LogComboGauge and PeriodComboGauge are derived from ComboGauge

- ◆
  - ⇒ class
  - ⇒ instantiation: specifies size, caption(w/ color),
  - ⇒ notes: appearance refers to the appearance of

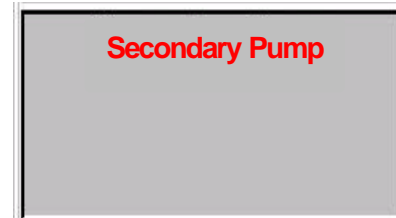


Figure 3- Static panel

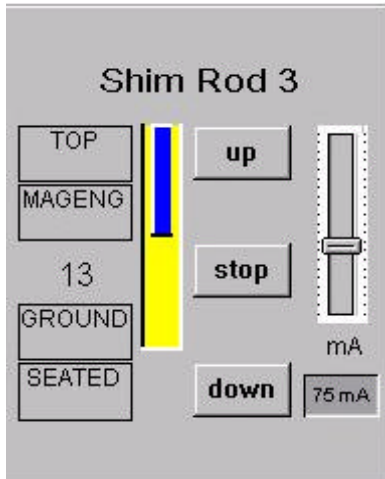


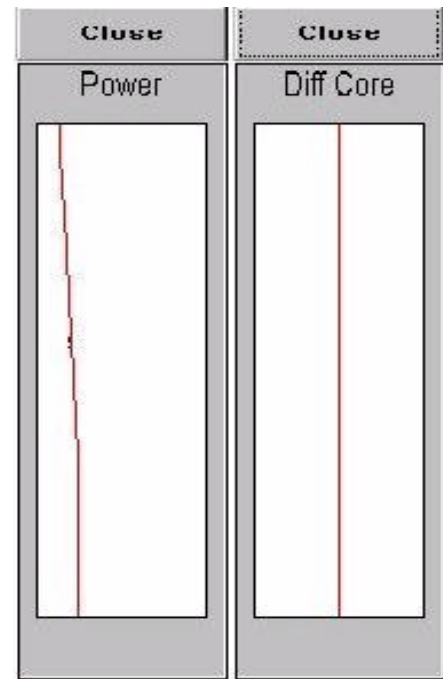
Figure 3- Shim rod panel

◆ *Rods*

class: ControlRod, RegulatingRod  
 instantiation: specifies size, rod number, and electrical current  
 notes: the regulating rod panel is slightly different from the shim rod panels

*Strip-charts*

- ⇒ : SimpleStripChart
- ⇒ range of the x-axis
- ⇒ visible is determined by the height of the charts and the rate of update



5 Strip-charts

A correct syntactic implementation is crucial in the long-term stability of a software application. It

and enforces cohesion among them. It glues the different pieces of the system together into one cohesive entity. The following sections will describe the different aspects of

### 3.2.1 callback functions

Callback functions are event-driven responses associated with a graphical control. For example, when a button is clicked, the operating system dispatcher sends a message to the appropriate window, which in turn process the message and invoke any function defined for that event.

The OWL graphical library provides the framework for many common events, such as push-button click and radio-button selection. The names of these functions are already defined. A programmer only needs to implement these functions with the desired behaviors.

The callback functions defined for the graphical user interface can be categorized into two sets: display functions and operation functions. The *display* functions are those that only affect the appearance of the GUI. The opening and closing of the strip-charts, and the display of the socket communication status are a few examples. The *operation* functions are those that may affect the appearance of the GUI as well as the state of the reactor. For example, pressing the ‘up’ button in a control rod panel may cause the corresponding rod in the reactor to rise as well as the GUI display to change.

The operation callback functions are especially of concern in the syntactic implementation. The behavior of these functions are identical: generate a token identifying the user action.

### 3.2.2 token generation

The token-production function is invoked from within an operation callback function. It takes a single parameter which identifies the user action and generates a corresponding token. The generated token is then pushed onto the queue (Figure 2-3).

Each token is an object instantiated from class *Token*, which contains two integer data members. The first data member, *token\_id*, distinguishes one user action from another. A parser evaluates the legality of that user action based on this variable and the current state of the reactor. The second data member, *sequence\_num*, identifies the sequential order of the tokens. For example, if at the time token *x* is created the token queue has 4 items, then *x* will have a *sequence\_num* of 5; i.e., it is the 5<sup>th</sup> item in the queue. Therefore, a token with a higher sequence number represents a later action than a token with a lower sequence number.

In trying to maintain a sequential order of commands, the *sequence\_num* data member becomes important since in parallel execution, a token with a higher sequence number may conceivably get processed prior to a token with a lower sequence number. However, if both tokens were successfully transformed into commands, they shouldn’t be sent to the control system out of order. By checking the sequence number of each command prior to transmission, they can be sent to the control system *in order* even if they are processed out of order in real time.

### 3.2.3 the token queue

The token queue behaves the same way as a normal First In First Out(FIFO) data structure. Tokens are pushed onto the queue at the front and popped off the queue at the

rear. The size of the queue changes dynamically to accommodate all the tokens at any given time, and it determines the sequence number of the next token.

Conceptually, the token queue will not get very large since the rate of user action is slow compared to the speed of token processing by the parsers. Therefore, memory consumption by the dynamic allocation of the queue is a negligible issue; that is, the queue won't get big enough to consume all system memory and crash the application.

### 3.2.4 associated windows programming

Besides implementing the conceptual architecture of the GUI, the syntactic implementation presents a few programming challenges as well. Windows threads and sockets programming are especially interesting. To achieve real-time parallel execution in a Win32 environment such as Windows 95, multithread programming is the best solution. To implement reliable network communication between stand-alone computers, windows sockets provides the most robust programming interface.

#### 3.2.4.1 windows threads programming

The most straightforward way of programming multiple threads in an application makes direct use of the Win32 API's (Application Programming Interface), which is a set of functions that interact directly with the operating system kernels. These functions are designed to facilitate programming within the operating system.

Once created, a new thread has its own thread ID and stack space, but it shares the same address space with its peer threads. Threads are light weight compared to processes and require little context switching.

As applied to this application, each thread represents a single parser. It will continuously execute a C++ function, which implements that parser, as long as it exists. The creation and destruction of threads are handled by the *begin\_thread()* and *end\_thread()* functions, respectively, and threads are maintained by the run-time threads library.

#### 3.2.4.2 windows sockets programming

Borland C++ provides a clear, robust, and object-oriented interface for the underlying windows sockets (winsock) API's. The *TStreamSocket* class encapsulates all the functionality necessary to establish a stream-socket server-client session. This winsock interface also maintains the platform-independent property of socket communication – the property that a PC can communicate equally well with a UNIX machine through sockets as with another PC.

This application uses stream sockets, which is a subset of Internet sockets. Therefore, each machine in the local network must have its own unique Internet Protocol (IP) address.

### 3.3 Semantic Implementation – generate parsers using YACC

The first step in the semantic implementation process is to get a clear understanding of the workings of the controls. The establishment of this understanding would necessarily require the expertise of nuclear reactor engineers. In particular, Tom Doyle, the director of operations at UVAR, was exceptionally helpful as he answered questions during an on-site visit and in subsequent email correspondence. He helped clarify misconception on the part of the research group about the semantics of some of the controls, such as the up/down switches which control the shim rods. He also made suggestions with respect to the usability of the user interface from a reactor operator's point of view. His cooperation has given us a solid, albeit yet limited, understanding of the reactor.

The second step in the semantic implementation involves the application of Z. In this step, the working of the reactor – the semantics associated with each of the controls – are expressed in Z as states and relations. The resulting document is the formal specification of the user interface from which the parsers will be generated. An example state, or schema, from this document is shown below in Figure 3-6:

<i>InitAlarmSystem'</i>	
<i>AlarmSystem'</i>	
$\forall a : Alarms / AlarmConditions'(a)$	$= Lowered$
$\forall a : Alarms / AlarmIndicator'(a)$	$= Off$
$\forall a : Alarms / AlarmLatch'(a)$	$= Off$
<i>BuzzerState'</i>	$= Silent$

**Figure 3-6 An example schema in the formal specification**

This schema([12], p. 6) describes the initial state of the alarm system. This specification is currently being developed by Dr. Susan Brilliant at the Virginia Commonwealth University.

The third and final step in the semantic implementation is to translate this Z specification, into YACC(Yet Another Compiler-Compiler). YACC is a powerful and efficient programming tool for syntax-analysis. It transforms(i.e., compiles) a YACC specification into a C++ function that implements the specification. This resulting C++ function can then be integrated into a larger program as an independent module and perform syntax checking(parser module, Figure 2-3). Of course, the semantic implementation is not complete after the completion of this step. Because of the magnitude of complexity associated with any such semantic specification, it requires many iterations of this process to continuously improve the overall quality of the formal specification. The semantic implementation is a persistent process throughout the development of the user interface.

At the time of this writing, the second step, the development of the formal specification, is nearing completion. Once finished, translation to YACC will begin.

## Chapter 4: Analysis and Discussion

### 4.1 Difficulties in the Implementation

Many of the programming tasks in the implementation phase were challenging as it involved a variety of programming disciplines. The major technical difficulties encountered during this phase are listed here.

- ◆ *OWL programming*: Object Windows tends to redraw at unpredictable intervals.
- ◆ *concurrent programming*: When programming multiple threads in Windows 95, the use of function pointers as parameters to another function was confusing at first. When programming multiple processes in UNIX, inter-process communication(IPC) was a challenge.
- ◆ *LEDA library reuse*: The application needs a queue data structure. The LEDA library provides robust implementation of various data structures, including the queue. Integrating the LEDA package into the application, however, was difficult due to name conflicts with existing Borland classes.
- ◆ *UNIX sockets programming*: Although the socket programming interface is similar in UNIX as in Windows, there are a few significant differences in the library functions. Also, it is difficult to implement *event-driven* architectures in UNIX.

### 4.2 Discussion

Unlike commercial software development which has deadlines and precise functionality requirements, this research at its current state is simply a proof of concept. The goal here is to demonstrate convincingly that formal specification is an effective approach and deserves more research interests. Although my project is only an intermediate step towards accomplishing this goal, what has been completed warrants a brief discussion regarding the feasibility of formal specification.

First of all, the user-interface architecture(Fig. 2-3) appears to be a sound design. The modular structure has reduced an overall very complex system into much simpler sub-systems. Over the course of implementation, as more components of this architecture were put in place, the architecture diagram became more realistic and logical. As of now, much of the user interface architecture is implemented(or at least has a framework of implementation). The missing piece – the parsers – is due to be implemented next. Besides the complexity associated with programming the parsers in YACC, which is an anticipated and manageable risk, there are few foreseeable difficulties for this task.

Overall, based on the implementation completed so far, we are reasonably confident about accomplishing this task in moderate amount of time. Once the parsers are

implemented, we will proceed to fill in the blanks and put the pieces together to complete the coherent structure illustrated in Figure 2-3.

## Chapter 5: Conclusion

### 5.1 *Future Improvements and Recommendations*

While most of the design decisions made along the way are proven correct, a few can be revised. The following is a short list of planned improvement to the current project:

- ◆ *replace OWL with MFC*: The Microsoft Foundation Classes(MFC) graphical library is similar to the Object Windows Library, but slightly better. It has a clearer and more intuitive class hierarchy as well as better implementation of comparable features.
- ◆ *implement more functionality*: The current user interface provides a very limited set of functionality. Future versions of the interface will expand to include more subtle, nevertheless important, features.
- ◆ *rearrange the GUI*: The graphical representation of some of the controls need to be modified to give a more realistic appearance. For example, changing the linear gauges to angular gauges when appropriate. Also, the arrangement of the controls on the GUI may need revision as to optimize usability, and therefore, safety.
- ◆ *use version control to manage baselines*: A version-control software can help tremendously in maintaining different baselines of implementation, and provide a clear chronological development timeline for reference and documentation. Future versions of the project will be managed using a version-control software.

### 5.2 *Conclusion of Project Activities*

Although the implementation of the user interface, as well as the overall research, is still ongoing, what the project has already accomplished demonstrates that formal specification is a feasible approach to implementation. Especially for safety-critical applications, a formal specification can help establish a solid foundation for the subsequent stages of development and make the difference between a correct and an incorrect implementation.

Regardless of the outcome, this research project is a significant step towards the realization of the underlying guidelines of developing software-driven safety-critical applications. Its importance cannot be overstated. As our understanding of the reactor improves, this procedure will be repeated and the implementation refined. Possible improvement on the procedure will be documented for future reference.

On a personal note, I have learned a great deal while working on this project, more than any class has ever taught me. Perhaps the most important lesson I learned out of this experience taught me how difficult software engineering is and how dedicated one must be in this profession. In the end, I hope I gained some personal wisdom from this rich experience and contributed in some way to the development of this field.

## Appendix A: Bibliography

## Bibliography

1. G. Leveson, "**Software Safety: Why, What, and How**", Computing Surveys, 1986.
2. Potter et al, *An Introduction to Formal Specification and Z*, Prentice Hall, Inc., New Jersey, 1991.
3. Hix and R. Hartson, *Developing User Interfaces: Ensuring Usability Through Product and Process*, John Wiley and Sons, Inc., New York, 1993.
4. Ince, *An Introduction to Discrete Mathematics and Formal System Specification*, Clarendon Press, 1988.
5. Diller, *Z: An Introduction to Formal Methods*, John Wiley and Sons, Inc., New York, 1990.
6. Elder, *Specification of User Interfaces for Safety-Critical Systems*, Technical Report CS-95-30, Department of Computer Science, University of Virginia, 1995.
7. Aboud et al, "**User Interface Languages: A Survey of Existing Methods**," Technical Report PRG-TR-5-89, Oxford University Computing Laboratory, 1989.
8. Harrison and H. Thimbleby, *Formal Methods in Human-Computer Interaction*, Cambridge University Press, 1990.
9. John R. Levine, Tony Mason and Doug Brown, *LEX and YACC*, O'Reilly & Associates, Inc., 1990.
10. Jones, *Software Development: A Rigorous Approach*, Prentice-Hall International, London, 1980.
11. Johnson, *YACC: Yet Another Compiler Compiler*, Computing Science Technical Report 32, Murray Hill, 1975.
12. Susan Brilliant and John Knight, "**Formal Specification of a User Interface**", Communications of the ACM, 1995.
13. Susan Brilliant and John Knight, "**Preliminary Evaluation of a Formal Approach To User Interface Specification**", Communications of the ACM, 1996.
14. Dept. of Computer Science, UVa, "**UVa Reactor Control System Software Pre-Specification**", University of Virginia, 1996.

## Appendix B: Reactor Overview

## Reactor Overview

The following paragraphs and diagram are excerpts taken, with permission, from the *UVa Reactor Control-System Software Pre-specification*[14]. They are placed here to provide an entry-level understanding of the reactor facility modeled in this research as well as of reactors in general.

### **B.1 UVAR**

The University of Virginia Reactor(UVAR) is a nuclear research reactor, operated by the Department of Mechanical, Aerospace, and Nuclear Engineering, which began operation in 1960 at a power level of 1 MW using Highly Enriched Uranium (HEU) fuel elements. In 1971, its power level was upgraded to 2 MW, and in 1994 the reactor was converted to use Low Enriched Uranium (LEU) fuel elements. The reactor is used for the training of nuclear engineering students, service work in the areas of neutron activation analysis and radioisotope generation, neutron radiography, radiation damage studies, and other research. Despite being a research reactor and not a power reactor, the UVAR is a complex system facing many of the same issues as a full-scale reactor.

### **B.2 System Overview**

A diagram of the primary components of the UVAR system is shown in Fig. B-1. As the figure shows, the UVAR is a light-water cooled, moderated, and shielded “pool” reactor. At the center of the reactor is the reactor core, an assembly which contains fuel elements, control rod fuel elements, graphite reflector elements, and possible in-core experiments.

The reactor core is loaded under approximately 22 feet of water onto an 8x8 grid-plate that is suspended from the top of the reactor pool. The reactor core loading contains a variable number of fuel elements and in-core experiments; it always includes 4 control rod elements. Three of these, designated as Shim rods(or Safety rods) are designed for gross control and safety. Shim rods are magnetically coupled to their drive mechanisms and drop into the core by gravity on a scram signal, activated by either the operator or the reactor protection system; this shuts down the reactor in less than one second. The fourth rod is a regulating rod which is fixed to its drive mechanism and is therefore non-scramable. The regulating rod, only a weak absorber of neutrons, is used for fine control of the power to compensate for small changes in reactivity associated with normal operations.

The heat capacity of the pool is sufficient for steady-state operation at 200 kW with natural convection cooling. When the reactor is operated above 200 kW, however, the water in the pool must be pumped down through the core through a header located beneath the grid-plate to a heat exchanger, which transfers the heat generated in the water to a secondary system. A cooling tower located on the roof of the facility exhausts the heat and the cooled primary water is returned to the pool.

For a more detailed documentation, consult the full pre-specification.

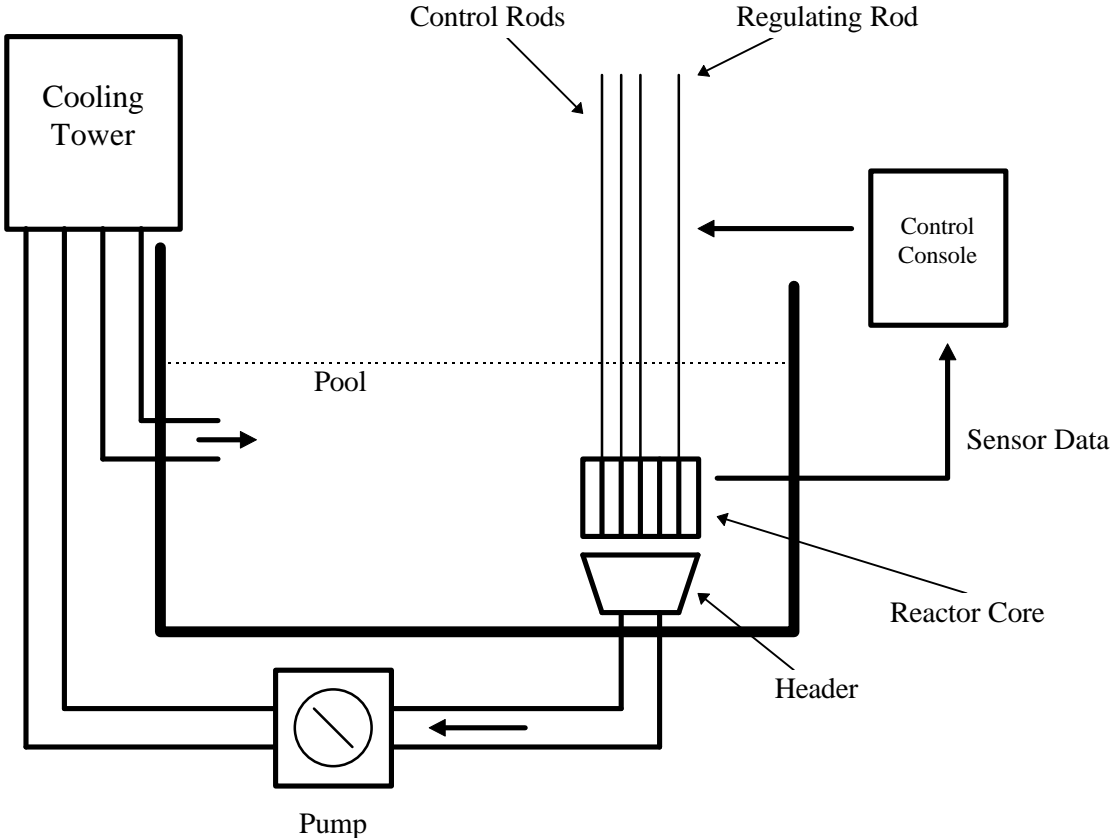


Figure B-1 University of Virginia Reactor (UVAR)

## Appendix C: Source Code Listing