# SPECIFICATION OF USER INTERFACES FOR SAFETY-CRITICAL SYSTEMS

Matthew C. Elder

Computer Science Report No. CS-95-30
July 7, 1995

# SPECIFICATION OF USER INTERFACES FOR SAFETY-CRITICAL SYSTEMS

*Matthew C. Elder*

Department of Computer Science
University of Virginia
Charlottesville, VA 22903
elder@Virginia.edu

## Abstract

Safe operation of a safety-critical computer system depends on appropriate human-computer interaction, effected through the user interface. Thus, specification of the user interface is a key task in the development of such a system. This thesis presents a comprehensive, structured approach to formally specifying user interfaces for safety-critical systems. Based on a view of an interface as comprising three levels, the approach decomposes the user interface into multiple components: semantic, syntactic, and lexical, respectively addressing application function, dialogue control, and presentation. Each component utilizes a formal notation appropriate to its level, then these component specifications are integrated systematically. This approach promotes a modularization that enables prototyping and change at each level and validation of user input, as well as enabling a correspondence between specification and implementation architectures that facilitates software development and verification of the implementation. Feasibility of this specification method was demonstrated using two case studies involving safety-critical systems: the Magnetic Stereotaxis System (MSS) and the University of Virginia Reactor (UVAR).[1]

1. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. This work was also supported in part by the National Science Foundation under grant CCR-9213427, and in part by NASA under grant number NAG1-1123-FDP. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

# Table of Contents

## List of Figures

# 1 Introduction

Computers are introduced into many applications for the advantages they provide: potential advantages can include greater flexibility, faster operation, enhanced functionality, and decreased cost. Computer-based systems can be found in many application domains. Often though, the introduction of computers into these systems, in addition to providing additional capabilities, increases the complexity and the risk of system failure [29].

*Safety-critical* systems are those computer-based systems for which the consequences of failure are extreme. Consequences of failure in these systems could result in death, injury, loss of property, or environmental damage [29]. Safety-critical systems exist in a variety of application domains. For example, many computer-based medical applications, including those involved in robotic activity and surgical assistance, must be considered safety-critical because they have the property that human life can be endangered by computer failure. In the area of nuclear power, computer-based digital control and shutdown systems are safety-critical due to the risk posed by system failure to both humans and the environment. Other examples include fly-by-wire flight control systems in avionics and many military and aerospace systems.

Despite the variety of application domains, safety-critical computer systems have some common characteristics. In these systems, the computer must monitor and/or control complex, time-critical physical processes or mechanical devices [29]. The computer has primary, if not sole, control of the devices that are capable of causing hazards or catastrophic accidents. The software is often embedded and must perform under real-time constraints.

One other important characteristic of safety-critical systems is that they are often under the overall control of a human "operator." The human operator makes a number of decisions involving system operating policy based on relevant information, including that supplied by the computer system. The operator must also provide the proper input responses to the computer in order to effect control of the entire system. The computer, having centralized authority over devices, sensors, and other system components, is the primary means by which the human operator can control the system. Safe operation of the system, therefore, depends on appropriate interactions between the human and the computer system. These interactions take place through what is usually referred to as the *user interface*.

The user interface of a computer system enables communication between a human user and a computer system to perform some set of tasks [22]. At the most basic level, the user interface effects a dialogue between the human and the computer, where the human communicates via input devices and the computer communicates via its output devices, often a graphical display. This dialogue has certain rules that dictate legal and illegal sequences of user requests and computer responses.

In developing the software for any computer system, specification is an important activity. Specification is the stage in software development where *what* the software must accomplish is determined; *how* the software is to accomplish certain functionality is considered later in the development cycle. It is important to determine exactly what the software must accomplish before becoming concerned with the details of designing and implementing the software. Specification allows users, application engineers, and software developers to reason at a higher level about the software. For example, a precise specification allows users to determine whether all the necessary functionality of a system can in fact be used as desired. Specification also aids in planning before actual implementation, and sets forth requirements by which to evaluate the final, implemented software product.

Specifying the entire software system is also important because incorrect specifications are a major source of defects in computer systems [39]. Many software errors can be traced to incorrect, incomplete, or ambiguous requirements in the specification. Safety-critical systems are a class of applications where errors must be prevented to the extent possible since the cost of failure is very high. Thus, the entire system must be specified correctly, completely, and precisely to avoid and eliminate defects.

Why worry about the specification of user interfaces? In general, the development of user interfaces is a difficult and important problem [22, 4, 11]. As safety-critical applications rely more heavily on increasingly complex user interfaces, the need to specify the user interfaces of such systems precisely and correctly to ensure freedom from defects becomes essential. User interface errors can occur in displays; for example, data might be presented incorrectly or an object on a display might be mislabeled. If this occurs, accidents may occur because the operator acted on incorrect information. Similarly, user interface errors can also occur in user input; the operator may request an invalid function to be performed or perform activities in an incorrect order. If this occurs, accidents may occur because the system performed an action that it should have prevented. A critical aspect of user interface development to prevent such errors involves specifying what the user interface is to do. This is an extremely complicated problem that is a constant source of errors in computer systems [20]. This work addresses the issue of specifying user interfaces for safety-critical applications.

Errors that were at least in part attributable to defects in user interfaces have already been reported in safety-critical systems. One example is the case of the Therac-25, a computerized radiation therapy machine which between June 1985 and January 1987 caused six known accidents of massive overdoses resulting in serious injuries and deaths [30]. As in most accidents in safety-critical systems, the cause of the accident is a set of complex interactions between system components and operating procedure. In the case of the Therac-25, however, a user interface error, in which modification of an incorrectly-typed value on the screen did not modify internal values, contributed to some of the overdose accidents.

Specification is a critical stage in the construction of user interfaces, not only for the elimination of defects but also for validation of the user interface. Validation is the activity in software development concerned with determining whether the system to be built is really the one that the users need [3]. In other words, validation ensures that the requirements specified are actually the correct requirements. User interfaces are subject to a great need for user evaluation, and user interfaces must change frequently as user requirements are better understood. The act of specifying a user interface forces the developer to consider carefully the functions that the system must provide to the user, the dialogue between the human and the computer, and appearance of the user interface. Specification of the user interface aids in validating that the requirements are correct.

It is important to understand that specifying a user interface involves much more than merely stating what the screen(s) should look like. Although the graphics are important, it is crucial to specify the details of what the various graphic items are used for and how they change with the circumstances of the application. A critical aspect of this is modelling the dialogue of the user interface, as well as the functions that the user interface provides to the user.

This work presents a highly-structured, comprehensive approach to user interface specification. The user interface is viewed as a dialogue with three distinct portions or levels, each specified separately with appropriate techniques and notations for each. The application functionality provided by the user interface is a distinct component at the highest level of the specification; the interaction language used for communication between the user and computer is specified as a separate level; the presentation, typically the graphical display, is specified at the lowest level of the interface model. For integration of these specification components, a method for systematic interaction between the different levels of the specification is provided.

Evaluation of the proposed specification approach occurred on two case study applications involving safety-critical systems. The experiment undertaken in this work was to demonstrate the feasibility of the proposed specification approach. Specifications were composed for user interfaces of both case study safety-critical systems using the proposed method in order to evaluate the method's feasibility; further, a user interface implementation was constructed from the case study specifications for one of the safety-critical systems to gain additional insight into the utility of the proposed specification approach.

The following chapter introduces and examines the general structure of a user interface. In addition, the rationale for viewing a user interface as a dialogue with three levels is presented in Chapter 2.

Chapter 3 explores the existing literature on user interface specification. There is a surprisingly large body of literature pertaining to user interface specification; that work is presented here and organized according to the user interface structure introduced in Chapter 2.

A specification approach for user interfaces is proposed in Chapter 4. This chapter describes what needs to be specified, proposes a structure for the specification, and explores possible notations and techniques for specifying the various portions of a user interface. The proposed approach draws upon the extensive body of literature on user interface specification, surveyed in the previous chapter.

Chapter 5 provides a brief overview of two case study safety-critical applications, the Magnetic Stereotaxis System (MSS) and the University of Virginia Reactor Facility (UVAR), used in the evaluation of the proposed specification method. The Magnetic Stereotaxis System is a safety-critical medical application for human neurosurgery; the consequences of failure in the MSS are possible patient injury or even death. The University of

Virginia Reactor is a 2-megawatt research reactor, operated by the Department of Nuclear Engineering for research and industrial experiments. Possible consequences of failure for the UVAR include dangerous release of radiation associated with core meltdown, posing threats to both reactor operators and the surrounding community.

In Chapter 6, the specifications for the user interfaces of both the MSS and the UVAR are presented, describing the specifications produced from application of the proposed method.

Evaluation of the proposed method follows in Chapter 7. The advantages and limitations of the specification method are assessed, and experiences applying the specification method to the user interfaces for both the MSS and the UVAR are presented to support those assessments. As a part of the evaluation, an implementation of the user interface for the MSS was constructed from the specification; Chapter 7 also explains the observed benefits of this method with respect to implementation.

The final chapter, Chapter 8, presents the conclusions of this research. Future work is also discussed.

Finally, Appendices containing the complete specifications of user interfaces for both the MSS and UVAR are included.

# 2 User Interface Structure

Understanding the structure of a user interface aids in developing an appropriate specification structure. As early as 1974, Foley and Wallace introduced the notion of the user interface as a conversation, or dialogue [12]. Observing an interactive graphic system, i.e. a user interface, as a conversation between man and machine, Foley applied language and psychological principles to justify and guide design techniques for such systems. In the conversation between human and computer, there exists a language for communication between the two parties; however, it is explained that this language, rather than using spoken or written words, is composed of graphical objects displayed and manipulated by the computer and actions relating to those objects performed by the human. This work viewed three levels of closure for human action or response, lexical, syntactic, and semantic, roughly corresponding to the complexity of user actions [12]. Later work by Foley and Van Dam [13] expanded on the notion of three levels of user action; this work presented a model of the human-computer dialogue having three components, once again, a semantic, a syntactic, and a lexical, relating to the type of information dealt with at each level.

A similar three-level view of the user interface, called the Seeheim interface model, developed out of work on user interface management systems. The Seeheim model has a component relating to the application that presents the overall computer application in terms of the user interface. Another component is responsible for dialogue control, governing sequencing of events within the user interface. Finally, the Seeheim model contains a presentation component that deals with the appearance of the display [21].

Viewing the user interface as a conversation whose language of interaction consists of these three levels is a key structuring mechanism that will be utilized in the proposed specification architecture. The distinction between these portions of the user interface is described in more detail in this chapter.

## 2.1 Semantic Level

The semantic level of the user interface contains the high-level, system functions that are available to the user: application functionality provided by the user interface. Literally, the semantics defines the *meaning* of the actions that the user interface performs or provides, not the sequence or form of those actions [13]. The corresponding component of the Seeheim model relates to problem-data management, or the "application [10]."

Given a conventional word processing program as the application, an example of the semantic level would be editing functions, such as "cut" or "paste." These functions could be part of a dialogue between the computer and the user, and the semantic level is concerned with providing that functionality and determining the information necessary to effect those actions.

## 2.2 Syntactic Level

The syntactic level of the user interface is the structure of the human-computer dialogue. A dialogue can be viewed as a language with well-defined *syntax* or grammar: rules to determine legal sentences in that language. Sentences in a grammar are composed of words, called *tokens*. Tokens are units of meaning, which cannot be further decomposed without losing their meaning [13], just as words in conventional language are composed of letters and a word loses its meaning when decomposed into individual letters. The grammar in the syntactic level of the user interface defines legal sequences of these tokens, i.e. sentences. The corresponding level in the Seeheim model is called "dialogue control [10]."

In the case of a user interface, the language corresponds to the interaction between human and computer, so the syntax of that language defines valid sequences of user input and computer output [13]. Tokens in the human-computer interaction language are user input, typically commands and information the user provides, and computer output that provide responses to the user through screen manipulation.

For the word processor example, the syntactic level of a user interface would include the rules that govern the usage of such functions as "cut" or "paste." A possible rule defining a legal sequence of user input would be that the "paste" operation must follow a "cut" operation. There would also be rules for computer output, such as the appropriate computer action after a "cut" request is to remove the text from the screen.

## 2.3 Lexical Level

The lexical level of the user interface is concerned with exactly how the user interface effects its dialogue. To continue the language analogy, the lexical level is concerned with token definition, or defining precisely how tokens are formed from the available input and output capabilities of the interface [13]. For user input tokens, the lexical level binds particular user commands to specific input devices and graphical items. For computer responses to user requests, the lexical level defines exactly how computer responses are conveyed to the user through manipulation or alteration of the graphical interface. The Seeheim model defines this level as "presentation [10]."

Once again, in the case of the word processor, the lexical level would determine how functions such as "cut" or "paste" are presented to the user. A possible lexical issue would be whether to present the "cut" function as a button or a menu option. An example lexical issue for computer response would be how the application indicates to the user that text is highlighted: choices include whether to change the color of the text or use reverse-video.

## 2.4 Conclusion

The user interface is shown to have an inherent three-level structure based on the type of information in a dialogue. A specification method for user interfaces must utilize this information to provide a natural structuring mechanism for the specification architecture. The importance of this view of the user interface will be shown in the proposed specification approach.

# 3 Literature Survey

Given the view of the user interface presented in Chapter 2, existing specification techniques and notations in the literature can be explored and evaluated according to which level(s) of the user interface are best described by the various methods. Some methods address more than one level of specification, of course; this chapter classifies methods according to what portion of the user interface they best specify. In addition, some systems for user interface construction address more than specification of the user interface. In particular, user interface management systems (UIMS's) comprise the entire development of an interactive software system's human-computer interface, including specification, design, prototyping, implementation, evaluation, and maintenance [21]. There is an extensive body of literature on user interface specification and construction, and this chapter cannot summarize all previous work, but rather surveys a large subset, including the major advances and techniques.

## 3.1 Lexical Specification

As mentioned previously, the lexical level of the user interface is concerned with how the user interface effects its dialogue. The lexical level describes how input tokens are formed from the various input devices and their actions upon graphical objects. This level also describes how output responses are formed from output capabilities, primarily manipulation or alteration of on-screen graphics. In other words, how the screen looks and changes with respect to application operation is described in the lexical level of the specification. Various methods and notations for specifying this are described in this section.

### *3.1.1 Screen Prototyping Tools*

The appearance of the screen has been the focus of much work in user interface specification and development. For various reasons, great importance is placed on the on-screen graphics. The appearance of the screen display is important to the users [20] as a significant portion of usability concerns. Some usability questions, such as ease of use, can be addressed at the level of the appearance or arrangement of the display. In addition, there is a need for the users to visualize the user interface, and, especially for systems with many different users, it is difficult to design an interface acceptable to all parties [20].

A popular technique which addresses the appearance of the display is the use of screen prototyping tools. Prior to the availability of these tools, developers would produce paper drawings of screens to convey ideas to users; this was a costly and ineffective technique [20]. Screen prototyping tools allow the graphics of the user interface to be constructed quickly and efficiently.

Prototyping tools offer many advantages for software system construction, in general, and user interface construction, in particular. As mentioned previously, communication between the designer and user is facilitated and made possible earlier in the development effort. This rapid feedback from users aids in the development effort and can help to reduce costs. In addition, validation of the user interface is assisted by prototyping tools; prototyping allows visualization of the user interface in order to evaluate, in this case, the specification of the presentation [31].

There are two major categories of screen prototyping tools: interactive screen editors and high-level notations for screen definition [20]. Each of these techniques will be described in detail in this subsection, in addition to an extended capability supported by some screen prototyping tools for handling scenarios.

### *Interactive Screen Editors*

Interactive screen editors enable the user interface builder to construct mock-ups of the graphical display. Typically, the developer can choose predefined graphical objects and arrange these on the screen to construct the display. For text-based interfaces, these tools provide a framework for arranging text on the screen where the interface is a form or series of forms. Screen editors for graphical user interfaces enable graphical objects to be created and positioned on the screen interactively. These tools must also provide a method for attaching functionality to these graphical objects. In some tools, source code can be generated for the constructed display, or the displays can be integrated into part of the larger application without additional coding.

Many screen editors have been proposed and developed for various applications and purposes [33, 47, 6, 1]. Work done by both Mittermeir [33] and Van Hoeve [47] enables prototyping of the user interface through easy specification of a form template layout, in the domains of business data processing applications and interactive information systems, respectively. Buxton et al. developed a menu-based dialogue system, called Menulay, which generates C code [6]. Aaram's work relates to interactive information systems, presenting a system that allows a hierarchy of menus to be defined for the user interface [1].

Screen editors for graphical user interfaces include HyperNews, a UIMS capable of specifying and generating interfaces in a rapid prototyping environment [41]. HyperNews, being a UIMS, has the capability to specify and implement all portions of the user interface, i.e., the presentation, dialogue control, and application interface. HyperNews strength, however, is as a screen generator, by which the developer may use or modify preexisting graphical objects, in addition to creating new objects, and arrange them to generate the screen. Functionality is then attached to these objects and dialogue control can be included [41].

An example of a domain-specific screen generator is LabVIEW, by National Instruments Corporation [36]. The domain of this screen generator is scientific and engineering instrumentation; the application provides facilities to construct a user interface that mimics actual hardware instrumentation. LabVIEW provides facilities for integrated data acquisition, instrument control, analysis, and presentation. With respect to presentation, LabVIEW provides predefined, modifiable, graphic controls and indicators, such as strip charts and knobs; the functionality of these applications, actual control and monitoring of hardware instrumentation, can then be attached to the graphical objects [36].

#### *High-Level Notations*

An alternative to interactive construction of the graphical display is the usage of high-level notations for screen definition. Hekmatpour and Ince cite two manners in which these notations can be implemented: either using an interpreter that produces a screen prototype from the high-level notation, or providing a collection of library routines accessible from a programming language [20].

Christensen and Kreplin present a high-level notation for prototyping user interfaces in which a specification framework is provided that can be interpreted to get textual, form displays [8]. A more recent interpreted notation for user interfaces is Tcl [37]. Tcl is a scripting language for developing graphical user interfaces; it comes with an interpreter to generate displays rapidly. Tk is a toolkit for X Windows that provides an extension to Tcl so the developer can write Tcl scripts instead of C code. Both Tcl and Tk are written in C, so the notations can be used as C libraries and embedded in code, in addition to being interpreted.

Borland ObjectWindows is a library of C++ classes provided for graphical user interface construction [5]. Although not usually considered a specification language, a specification of the display can be written by merely inheriting off of these library classes. The class definitions of the interface can be considered an object-oriented specification, higher-level and structured more modularly than typical graphics code would be.

#### *Scenarios*

A capability that some screen prototyping tools support is the generation of *scenarios*. A scenario is an operational example that simulates the sequence of events a user would experience in performing the tasks that constitute the operation of a system [23]. Scenarios can be supported by either interactive screen generators or high-level notation prototyping tools.

Hooper and Hsia advocated scenarios for prototyping of data processing applications; however, they did not propose a method for generation of the screens in those scenarios [23]. Mason and Carey took the scenario-based approach further, building user interface specifications through scenarios [32]. Their work included a tool to produce scenario prototypes rapidly. Little or no application logic is included within a scenario, so the sequence of events are predetermined, but scenarios enable the user to explore a system prototype without the developer committing extensive resources [20].

### *3.1.2 User Action Notation*

An alternative to describing screens and their graphical construction is to specify user interaction from the behavioral view of the user. User Action Notation (UAN), devised by Hartson, Siochi, and Hix, is a detailed specification language which describes the physical behavior of the user and the interface together [22]. UAN is implemented with scenarios, sequencing screens, and textual representations of user actions, such as pressing a button or moving the cursor to an icon. The textual representations of user actions reflect physical operations on hardware devices manipulating screen objects; these specifications become very detailed descriptions of input tokens and output responses.

## 3.2 Syntactic Specification

Chapter 2 presented the syntactic level of the user interface as the structure of the dialogue between the user and the computer. This dialogue is part of a language composed of rules that determine legal sentences in that language. The syntactic portion of the specification has been described in the literature many times, using various notations explored in this section.

### *3.2.1 State Transition Diagrams*

Several existing techniques rely on state transition diagrams as the primary notation for specification. As early as 1969, Parnas suggested the use of state transition diagrams for specification of command language interfaces [38]. Extensions to the transition diagrams and textual representations for these diagrams are often proposed, as in the work of Casey and Dasarathy [7]. Many specification methods based on state transition diagrams have been proposed for different interface types in different domains.

In the domain of interactive information systems, Wasserman developed a methodology, called User Software Engineering (USE), for specification and implementation of such systems. Wasserman's system used extended state transition diagrams for specification of the human-computer dialogue [48]; USE included automated tools for specification of those transition diagrams, as well as the capability to execute the dialogue specified. Because the functionality of the information system was encapsulated primarily in database queries, the focus of Wasserman's work was specification of the dialogue and generation of the prototype interface [49]. USE supported textual interfaces, providing facilities to position text on the screen and receive character input.

Work very similar to Wasserman's is that of Jacob, who developed the State Diagram Specification Interpreter [25]. Jacob also uses state transition diagrams for specification of the dialogue, and provides extensions to generate prototypes of the interface. Initial work pertained to command language interfaces, but later work applied state transition diagrams to specifications for direct-manipulation (graphical) interfaces [26].

### *3.2.2 Statecharts*

An extension of state transition diagrams that can be used for syntactic specification is statecharts [19]. Statecharts address some of the limitations of state transition diagrams: state transition diagrams tend to become complicated and unwieldy for large interfaces, concurrency is not easily represented, and modularity using subgraphs is not always possible [43]. Statecharts allow grouping of states in what is called a *roundtangle*, which in user interfaces allows transitions to the same destination state from multiple states to be factored into the surrounding roundtangle [50].

### 3.2.3 Context-Free Grammars

An alternative notation to state transition diagrams explored extensively in the literature is a context-free grammar. Context-free grammars, often used to describe programming languages, seem a natural notation for specification of the syntactic level of the user interface, the interaction language.

Reisner proposed the use of a context-free grammar to specify the language denoted by a user interface [40]. The input tokens of the grammar are abstract actions, such as pushing a button or selecting a color, that permit any style of interface to be specified; unfortunately, only the input language is included in the grammar. It should be stated, however, that the focus of Reisner's work was evaluating grammar specifications as a predictive tool for the relative performance capabilities of competing user interfaces.

Shneiderman proposed an extension to context-free grammars, called multiparty grammars, to clearly differentiate between user input tokens and computer output responses [42]. Multiparty grammars include an identifier in front of each nonterminal to indicate which party produces the string, typically either the computer or the user.

## 3.3 Semantic Specification

The semantic level of the user interface describes the application functionality provided by the user interface. Less work has focused on specifying the functionality of the user interface than on specification of the other levels. Some of the aforementioned work, while not focusing on semantic specification, did provide or account for this level of specification. For example, Wasserman's USE encapsulated application functionality in database calls, a benefit of the interactive information systems domain [48]. HyperNews also mentioned separation of graphical specification from interface functionality, but no explicit notations were mentioned for semantic specification [41].

Work by Foley et al. deals directly and primarily with the semantics of a user interface. The User-Interface Design Environment focuses on a higher-level representation of the interface, rather than dealing with presentation and interaction language [15]. A representation, called Interface Definition Language, is provided for describing the objects that make up the semantics of the interface [14]. Algorithms for transforming the specification and a method for executing the specification are explored in Foley's work. Foley also cites other work which deals with semantic specification, such as Green's method using pre- and post-conditions [18].

The use of formal methods in human-computer interaction also deals with semantic specification. An example is the work of Sufrin and He, who use the formal specification language Z to model the states and objects of an interactive system [44]. The display and input devices are also modelled in Z, but the focus of the work is modelling how user operations affect the state of the system.

## 3.4 General Specification Approaches

Many other user interface specification approaches exist besides the ones presented in the previous sections. Other methods for specification may use a different model of the interface or focus on other goals in addition to specification. This section will explore some techniques that do not fall conveniently into the structure of the user interface presented in Chapter 2.

One technique with a different emphasis than previously presented methods is Moran's Command Language Grammar (CLG) [34]. CLG is a specification method for command language interfaces that describes four levels of the user interface: task, semantic, syntactic, and interaction. The additional level, task, describes the operations the user wants to perform, but not in terms of objects and manipulations found in the semantic level. CLG has an additional purpose to specification: this method enables description of the interface from the perspective of the user's mental model as well as for interaction evaluation and design.

Abowd and Dix use formal specification languages to describe the user interface in terms of status and events [2], rather than the three levels discussed at length. Status refers to persistent information, while events are atomic, non-persistent occurrences. Abowd and Dix use formal specification to integrate the two types of information for a more natural expression of user interface behavior, with a focus on multi-user systems. The work explores specifying these two types of information, not dealing explicitly with semantic or syntactic issues.

Ege and Stary explore task-oriented specification, focusing on user action rather than system components and their structure [10]. Their work integrates the Seeheim model of the interface with object-oriented mechanisms for implementation. The specification notation used is the Interaction Management Network, which models tasks, subtasks, and sequencing information.

Finally, there is an entire class of applications, User Interface Management Systems (UIMS's), devoted to the entire development life cycle of user interfaces. The focus of UIMS's is to improve interface developer productivity and ensure that the interface developer can be a non-programmer. These are not issues in construction of user interfaces for safety-critical systems: dependability is the focus of any work relating to safety-critical systems. Some of the specification systems surveyed above are considered UIMS's; traditionally, UIMS's have tended to focus on the appearance of the user interface, but more recent work has dealt with complicated interaction dialogues and devices [21]. UIMS's are a very important and rapidly growing area of user interface work that must be monitored for contributions to specification and development techniques.

## 3.5 Analysis

A great deal of work has occurred regarding user interface specification. Of the work surveyed, most of the techniques only addressed a particular level of the user interface structure. The focus of some of the techniques was not specification so much as prototyping or usability, which, while important, is not the focus of this work dealing with the concerns of safety-critical software development. A *comprehensive* approach to user interface specification must be developed to address all the components of the user interface structure. The body of work surveyed here can be utilized in the development of a comprehensive approach to specification.

# 4 User Interface Specification Approach

A user interface is a complex entity; specifying such an entity is correspondingly complex. The interface is far more than the graphics or the dialogue or even these two combined. As observed in Chapter 2, there are multiple levels to a user interface, each of which must be specified.

This chapter explores and proposes a comprehensive approach to user interface specification. First, characteristics that a specification approach should possess are described. Next, the components that must be present in an interface specification are discussed, and appropriate notations for specifying these components are proposed. Finally, a structure for the specification approach is presented and explored.

## 4.1 Desirable Specification Characteristics

Based on knowledge of the structure of a user interface and general specification principles, it is possible to determine desirable characteristics for a comprehensive approach to user interface specification. These characteristics are explored in this section.

### *4.1.1 Separation of Concerns*

Separation of concerns means that portions of the user interface that address different concerns are specified, and implemented, in distinct components. Because the human-computer dialogue has three distinct levels, the specification should enforce this logical separation, and the implementation can, and should, correspond. Separation of concerns involves specifying the user interface in distinct components, so that each component concerned with different levels is independent of the others. For example, graphic specification should be separated from dialogue specification except for the necessary interfaces that the model requires.

A major advantage of this separation is that it enables multiple specification solutions to be developed for a particular level while maintaining the remainder of the user interface specification. For example, the separation of levels into distinct components permits the entire graphic element of the user interface to be changed, allowing vastly different graphic interface displays to be specified for the *same* application if desired. Exploring multiple specification solutions facilitates validation and usability testing.

This separation also aids prototyping of levels; having separate specifications allows the implementor to prototype one level at a time, rather than having to worry about the entire user interface. Typically, the choice of low-level implementation of the graphics can be put off, while *what* the user interface must accomplish can be concentrated on. In fact, once the semantics of the user interface are determined, multiple specifications of the syntax, or dialogue, can be developed, so long as they provide the necessary functionality. Similarly, multiple lexical specifications, or graphical displays, can be explored so long as they provide the tokens needed to effect the syntactic dialogue.

Finally, separation of concerns enables testing of the user interface implementation at each level, in addition to comprehensive user interface testing. It is possible to generate faults at each level of the user interface to determine how the receiving level copes with the faults.

### *4.1.2 Systematic Interaction Between Components*

Given separate specification components, another desirable characteristic of a specification approach is support for necessary and appropriate interactions between the various components of the specification. For example, the syntax of the interaction language contains a set of tokens, received from both the user and the computer system; the graphic specification must define the user actions that effect those input tokens and the interface to the application program must define the composition of its computer response tokens.

### *4.1.3 Formal Notations*

Formal notations are mathematically-based, semantically-rigorous languages for specification. In software engineering, it is often the case that specifications are written in natural language. Natural language is notoriously ambiguous and not readily amenable to analysis [24]. Formal notations address the limitations of natural language: because formal notations are mathematically-based the meaning of a given statement is unambiguous. It is sometimes possible to mechanically generate an implementation from a specification in a formal notation. In addition, specifications written in a formal notation can be analyzed using mechanical theorem provers. These advantages present formal notations as desirable for specification.

### *4.1.4 Executable Notations*

Some formal notations have the added benefit that they can be executed, as well as providing the benefits of precision. Known as executable specification languages, the ability to be executed directly enables rapid prototyping.

One drawback of executable specifications, however, is that in order to effect an implementation design details sometimes must be incorporated into the specification. The tradeoff involved in exposing design decisions must also be evaluated.

### *4.1.5 Analysis*

An extremely desirable characteristic of any specification method, including one for user interfaces, is that it enables various types of analysis to be performed on the specification. The types of analysis possible on a specification are dependent, in part, on the degree of formality of the notation.

Analysis during the specification stage of software development may catch design flaws or errors in the specification that would be more expensive to correct if caught at later stages [39], or may go undetected if such analysis is absent. Of course, in a safety-critical application, no such errors should escape detection.

With respect to user interfaces, some desirable user interface characteristics can be checked for in the specification given adequate notation. An example is a check for consistency; consistency means that similar operations in the user interface are performed in a similar manner. If the specification indicates an inconsistent user interface, that problem can be corrected in the specification before it propagates to later stages of development. Other general properties of user interfaces that can be subject to analysis in the specification are redundancy, where, undesirably, the same action may be performed in multiple ways, and incompleteness, where it is not possible to perform certain necessary functions in the given user interface. Reisner's work used analysis of the specification for evaluation of the user interface design [40].

Some analyses on user interfaces might be particular to the given application, or a given class of applications, such as safety-critical applications. An example of an application-specific property would be that for a safety-critical application there must always be a means on the user interface to bring the application to a safe state. It would be useful to be able to prove in the specification that the means by which to achieve this safe state is always active; i.e. the user interface is never trapped in a state where it is not immediately possible for the user to cause a transition to some safe state.

### *4.1.6 Support for Implementation and Verification*

A specification approach is only useful insofar as it facilitates implementation: specification is not done for its own sake; it is only in the context of the entire software development process, where the goal is a final executable product, that specification is important. Based on the structure of the specification, an implementation can be constructed with a corresponding architecture. The clear separation of user interface levels into specification components enables a clean, highly-structured implementation.

A specification approach should also support verification. Verification asks the question "are we building the product right [3]?" In other words, verification is the act of ensuring that the implementation is built correctly, i.e., that it implements *all* of the requirements put forth in the specification.

The structural correspondence possible between the specification and implementation aids verification. The correspondence enables mapping the portions of the specification to the portions of the implementation; therefore, the verification can be broken down into more manageable pieces, verifying each portion of the user interface separately.

The use of formal notations also supports implementation and verification. When using formal notations for specification, often the underlying formal semantics enables a refinement process to be applied in order to verify incrementally that the implementation matches the specification [53]. The process begins with a formal specification (i.e. a specification in a formal notation), then a series of mechanical steps can be applied to generate the design, the detailed design, and the implementation. At each stage the product is verified to match the previous stage, so by transitivity the implementation is verified against the specification. The use of executable specification notations works similarly.

### *4.1.7 Error Processing*

Another desirable characteristic of a user interface specification approach is provision for the detection and handling, in a meaningful manner, of *all* illegal sequences of user and computer actions. This entails determination of legality each token generated by either the user or computer at the particular point in the grammar the user interface is in upon reception of the token.

An example of an error handling mechanism is the prevention of illegal commands being entered by the user through "graying out" particular graphical items, i.e., disabling portions of the interface that are not to be available at a given point in the dialogue. The syntax defined for the user interface should facilitate determination of the legal and illegal commands.

### *4.1.8 Direct-manipulation Interface Capabilities*

A specification approach for user interfaces must support direct-manipulation interfaces. Much of the work surveyed in Chapter 3 pertained to only command language or menu-based interfaces; the interfaces of today are much more complex and require specification techniques to manage that complexity.

## 4.2 Specification Components

For a user interface specification to be, in any sense, complete, multiple components must be described. The components correspond roughly to the levels of the user interface: the lexical, syntactic, and semantic, as described in Chapter 2. In addition, interfaces between the levels must be described. This section enumerates the components that must be specified for a user interface, and presents appropriate notations for each component in the proposed specification approach.

### *4.2.1 Presentation Specification*

A specification of the presentation describes the layout of the screen, meaning the appearance of the display and the arrangement of graphical objects on the display. The presentation component of the specification corresponds to part of the lexical level of the user interface.

The proposed approach employs Borland's ObjectWindows for specification of the graphical displays. This notation enables a display to be easily described by inheriting off of classes of graphical objects in the ObjectWindows class library. An additional advantage of this notation is that it can be compiled and executed to produce a mock-up of the display.

A high-level notation was chosen to describe this component rather than an interactive screen editor or set of scenarios for various reasons. Typical presentation specification is done with some sort of visualization of the graphical display; specification using ObjectWindows provides this capability through compilation and execution. The mock-up created from the ObjectWindows specification lacks all application functionality and dialogue control; it is solely presentation. Specification using the textual notation of Borland ObjectWindows provides an opportunity for formal analysis that cannot be provided for merely generating screens. In addition, inclusion of the Borland ObjectWindows Library documentation defines precisely the base classes used in the specification and the meaning of actions manipulating those graphical objects; for example, what in particular constitutes a button press can be determined from that documentation.

Finally, the particular high-level notation chosen, ObjectWindows, compares favorably in its ability to generate screens rapidly for evaluation and execution, the primary advantage of screen editors. Being a reuse-oriented solution, generation of displays is convenient by inheriting off the preexisting graphical classes.

### *4.2.2 Input Token Definition*

The syntactic level of the user interface contains user input tokens and computer response tokens which comprise the language of interaction. A component of the user interface specification must be input token definition; this involves specifying precisely what user manipulations of the interface cause what aspects of the dialogue to be recognized. For example, the question of whether a particular function is effected by a button or menu item must be specified for *all* user interface inputs.

This specification component employs a table to relate the enumeration of input tokens from the syntactic level of the specification with how each token is effected. Using the presentation specification, each input token is associated with the particular graphical object or operation that generates that token.

An enumeration of computer response tokens would complete a specification of token definition. Because the generation of computer response tokens is in the application program, external to the user interface, an enumeration of those tokens is sufficient.

### *4.2.3 Presentation Command Interpretation*

The commands issued from the semantic level to the presentation of the user interface must also be defined. Certain actions resulting from valid combinations of tokens necessitate some manipulation of the graphical display. For these presentation commands, this involves specifying precisely what manipulations of graphical objects and other output devices (e.g. sound, mechanical devices) are associated with various system activities.

This specification component utilizes a table to associate the enumeration of presentation commands to the graphical objects, specified in the presentation component, that effect those commands.

Commands issued from the semantic specification might prompt action from the application program; once again, the commands that the application program provides must be enumerated to complete command specification, but the system-level actions associated with these commands are an application program concern.

### *4.2.4 Interaction Language Syntax*

The interaction language syntax describes the valid sequences of user and computer actions, i.e. the grammar defining the rules of the interaction language. This component of the specification corresponds to the syntactic level of the user interface.

A context-free grammar is proposed for this component of the specification. A grammar is a concise notation, amenable to analysis, that describes legal sequences of events naturally. State transition diagrams and statecharts would work also, but the use of these notations requires additional technology for convenient manipulation and analysis. In addition, for large dialogue descriptions, the textual notation of context-free grammars are more manageable than state transition diagrams. The use of a context-free grammar also enables leveraging off of compiler technology and tools that are readily available. For example, there are parser generators, such as YACC [27], to construct parsers for context-free grammars.

There is a context-sensitive component and a context-free component to any interaction language. The context-free component is modelled using the grammar; the context-sensitive component is addressed in the semantic specification.

### *4.2.5 Semantic Specification*

The semantic specification describes the system-level functionality that the user interface provides. A substantial amount of system-level functionality is in the form of system operations provided by the application program; for that reason an important portion of the semantic specification is describing the structure of the interaction between the user interface and the application software. In addition, the semantic component includes any state information necessary to effect system-level operations. Often, of course, the application and the user interface are merged into a single software entity, but even in these situations it is possible, and helpful, to view a logical separation between the application and user interface. The semantic level of the user interface is described by this component of the specification.

The "state" of the user interface that is modelled in the semantic specification consists of persistent data, or information that must be kept throughout operation of the user interface. Included in this state information may be data to help model context-sensitive aspects of the dialogue. As mentioned previously, the context-free component of the interaction language is modelled in the syntactic portion of the user interface specification; context-sensitive issues are dependent upon data obtained from operations, so context-sensitive interaction is included in the semantic specification along with that data.

The high-level operations that are modelled in the semantic specification are comprised of three things. The first is the information that is input to the operation; this is information that arrives with the token and is passed when the token is recognized as part of a legal sentence. The other two aspects of high-level operations that must be modelled are interfaces with the application program and presentation. The effect of a high-level operation will be some action involving either the application program or the presentation. Therefore, the functions that the application program provides the user interface and any information necessary to perform those functions must be described. Similarly, the methods and informa-

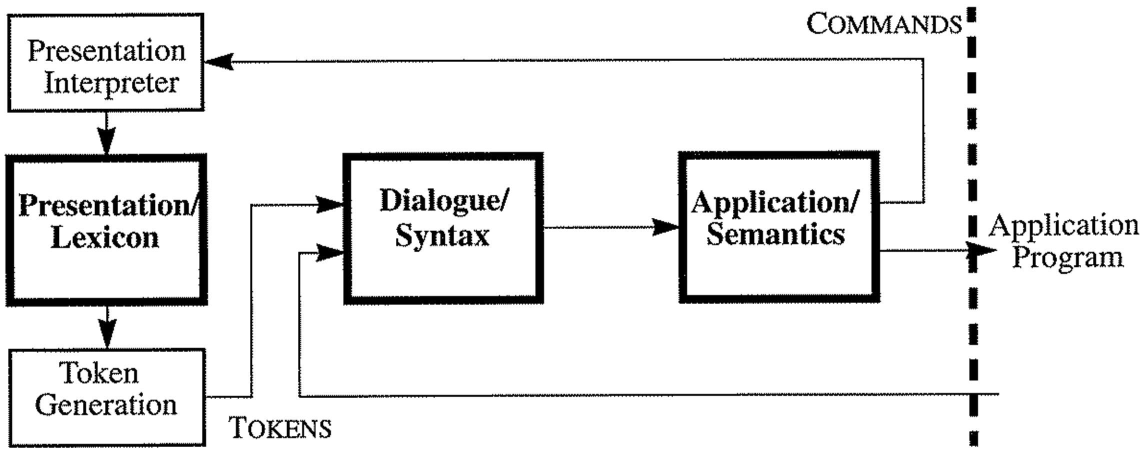

Fig. 1. User Interface Specification Structure.

tion involved in the interaction with the presentation must be described; this interface is specified in the presentation interpreter. It should be noted that what actions these methods actually perform are not specified in this portion of the document; only their existence is utilized.

The formal specification language Z is used for the semantic specification component. Z is a mathematically-rigorous notation with formal semantics, based on set theory and first- order predicate calculus [9]. Z models the state of a system and operations that act upon the state of the system, which is precisely what is necessary for the semantic level of the user interface.

## 4.3 Specification Structure

In a comprehensive approach to user-interface specification, all of the above components need to be addressed, and the specification technique must deal with each aspect completely and consistently.

Given the view of the user interface dialogue as three levels and the necessary specification components presented in the previous section, a corresponding three-part structure for the structure of the user interface specification is presented. Fig. 1 shows the structure in which each level can be specified independently, using different notations best-suited to what is being specified. Interactions between levels of the specification are also be modelled systematically.

At the center of the user interface is the dialogue syntax specifying the legal sequences of actions. The syntactic specification is a grammar defining the rules of the language, built on both input tokens coming from the user and computer response tokens from the application program.

The lexical specification defines the input and output capabilities of the interface, including how the graphical display looks and what objects are present on the display. Fig.

1 shows how the lexical component of the interface communicates with the syntactic component, generating the user input tokens of the grammar.

The semantic component of the specification is a high-level abstraction of the application functionality provided by the user interface. The semantic level's relation to the syntactic level is that the semantic level defines the operations of the interface and the information necessary to perform those operations. The dialogue syntax specifies a particular ordering of those operations. In addition, the input tokens received by the syntactic level contain state information that must be used by the semantic level to perform the operations.

A portion of the semantic specification is the interface to the application program. Many of the interface operations result in system functions being performed; the semantic level models these "commands" sent to the application program and the information provided by the user necessary for execution of these system functions. The application program might also return information to the user interface; these computer responses are received by the user interface in the form of tokens, which are processed by the grammar and checked for legality.

The exchange of tokens between the syntactic and lexical levels yields systematic communication between those components of the user interface. The semantic and syntactic levels of the user interface are related through the operations that the user interface provides; the semantic level models the operations and their effect on the state of the user interface, while the syntactic level specifies legal sequencing for those operations. The semantic specification requests commands to be performed with respect to the presentation, but this communication is handled completely through the presentation interpreter. Finally, the application program is isolated from the user interface specification structure and relates through generation of tokens and acceptance of commands.

This proposed specification approach will be evaluated in two case study applications that are introduced in the following chapter.

# 5 Case Study Applications

Given a proposed approach to specifying user interfaces for safety-critical systems, work must be done to evaluate the approach. The specification technique introduced in the previous chapter has been evaluated using two case study applications, the Magnetic Stereotaxis System (MSS) and the University of Virginia Reactor (UVAR), both of which are described in this chapter†.

## 5.1 Magnetic Stereotaxis System

Evaluation of the user interface specification technique occurred on a safety-critical medical application called *The Magnetic Stereotaxis System* (MSS) [52]. The MSS is an investigational medical device for performing human neurosurgery, being developed in a joint effort between the Department of Physics at the University of Virginia and the Department of Neurosurgery at the University of Iowa [16]. Stereotaxis is a neurosurgical technique for directing an instrument to a specific location in the brain for treatment of neurological disorders. Conventional stereotaxis techniques require a direct path to the location of treatment; often, though, a direct path is blocked by critically important or easily damaged brain tissue. Magnetic stereotaxis overcomes the limitations of conventional stereotaxis by providing an indirect path to the location of the brain requiring treatment [28].

The MSS operates by manipulating a small permanent magnet (known as a "seed") within the brain using an externally applied magnetic field, as shown in Fig. 2. By varying the magnitude and gradient of the external field, the seed can be moved along a non-linear path and positioned at a site requiring therapy. Envisioned therapy includes chemotherapy by using the seed to deliver drugs to a site within the brain and induction of hyperthermia for treatment of inoperable brain tumors by radio-frequency heating of the seed from an external source. The state of the MSS is that the concept is fully defined, the majority of the basic research in physics is complete, and a fully-functional prototype is nearing completion for demonstration and evaluation. Preliminary animal trials using the prototype will be conducted soon, and human clinical trials are planned at the Barnes Hospital of Washington University in St. Louis in the future [51].

### *5.1.1 System Overview*

The MSS hardware system that effects and monitors movement of the magnetic seed within the patient's brain is shown in Fig. 3. The patient is positioned at the center of six superconducting electromagnetic coils; power supplies and current controllers regulate the amount of electric current in the electromagnets, which produces a magnetic field that acts

†. Thank you to Kevin Wika for permission to use the figures in this chapter from his dissertation [51].

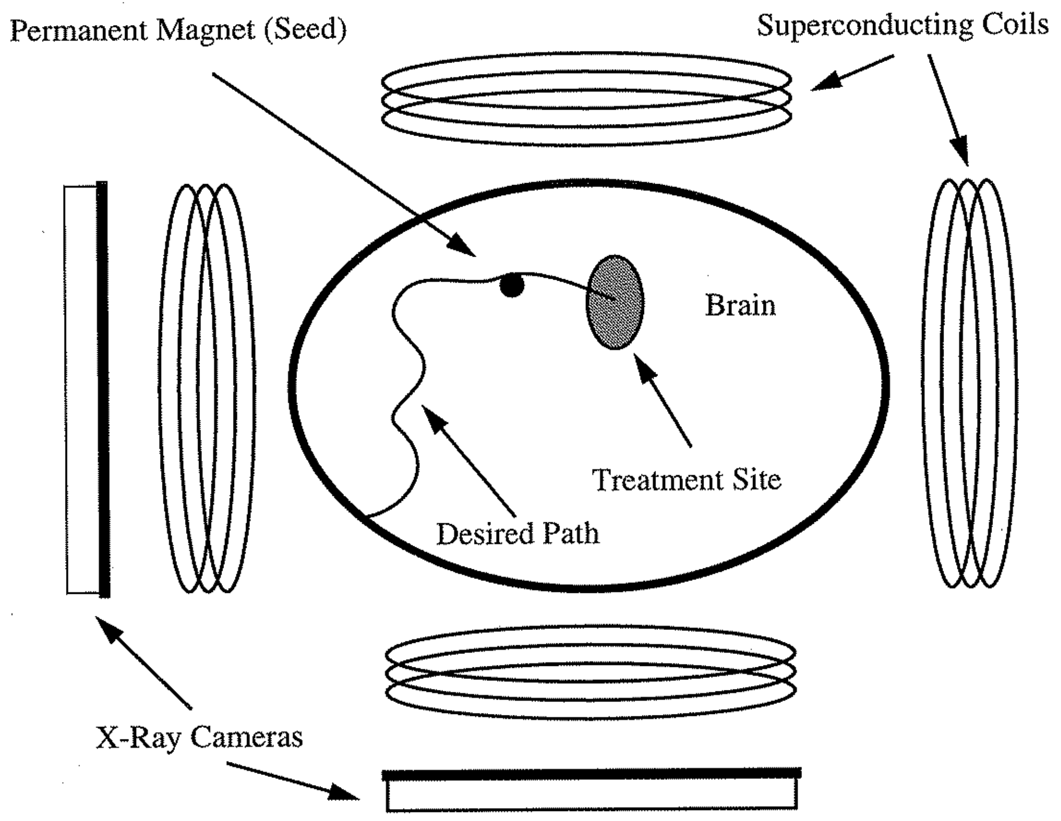

Fig. 2. Seed Guidance by the MSS.

on the seed and induces its movement. Along each axis perpendicular to the patient's body, an X-Ray source and camera produce fluoroscopic images for tracking the seed. Markers, affixed to the patient's skull and visible on the X-Ray images, enable the position of the seed to be determined [51].

Computer control is necessary in order to provide all of the functionality present in the MSS. During an operation with the MSS, a neurosurgeon directs the movement of the seed from a console that displays preoperative Magnetic Resonance (MR) images. The computer takes movement requests and computes the electromagnet current values required to produce the desired seed movement. A computer vision system analyzes the X-Ray images to locate the markers affixed to the patient's skull. Visible on both the MR and X-Ray images, the markers enable the position of the seed to be transformed into the MR frame of reference and subsequently superimposed on the preoperative MR images [51].The computer control system that has been developed for the prototype MSS includes multiple graphical displays. The main control module of the MSS is called the *Control Program*. The Control Program monitors all physical devices, accepts requests from and distributes information to all displays, and maintains state information on the entire system. The primary display, or user interface, is the *Operator Display*, through which an operator controls calibration and surgical procedures. Other optional displays include the *Field Display*, which displays a visualization of the magnetic field produced by the coils, and the *Engineering Display*, which presents items of engineering data.

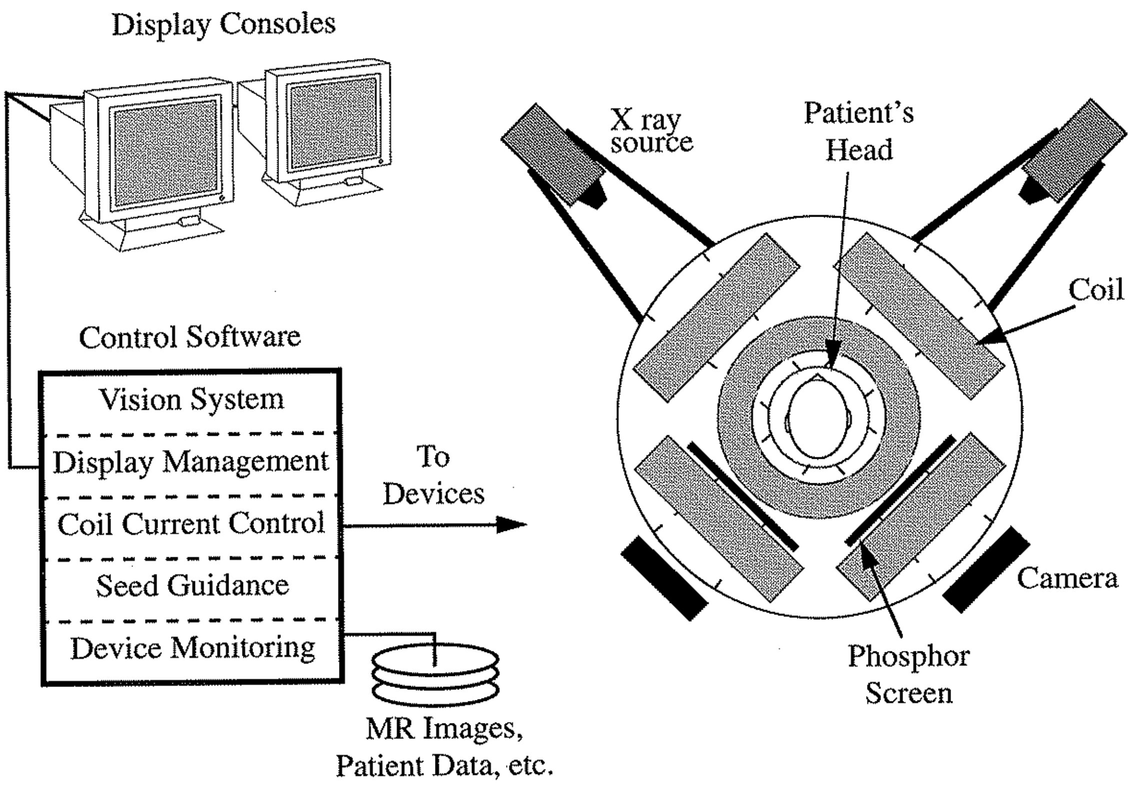

Fig. 3. Magnetic Stereotaxis System.

### *5.1.2 User Interface Overview*

The Operator Display, the primary user interface to the MSS, was the subject of case study investigation. As part of the prototype computer system for the MSS, a prototype user interface existed prior to the case study specification of the Operator Display. A picture of the prototype Operator Display is shown in Fig. 4.

The Operator Display provides the majority of the functionality of the MSS. The Operator Display contains mechanisms for system calibration prior to a surgical operation. Calibration functions primarily involve determination of vision system parameters necessary to accurately locate the markers and seed on the X-Ray images.

The Operator Display also provides patient data functionality, such as choosing a particular patient and loading that patient's preoperative MR images. In order to display the seed position on the MR images during an operation, the operator must identify the marker positions on both the MR images and the X-Ray images; the Operator Display includes capabilities for this identification to be accomplished.

Finally, mechanisms for performing surgical procedures are provided by the Operator Display. The user interface supplies a method for choosing the desired direction and distance of seed movement, displays the calculated movement from those values, and provides

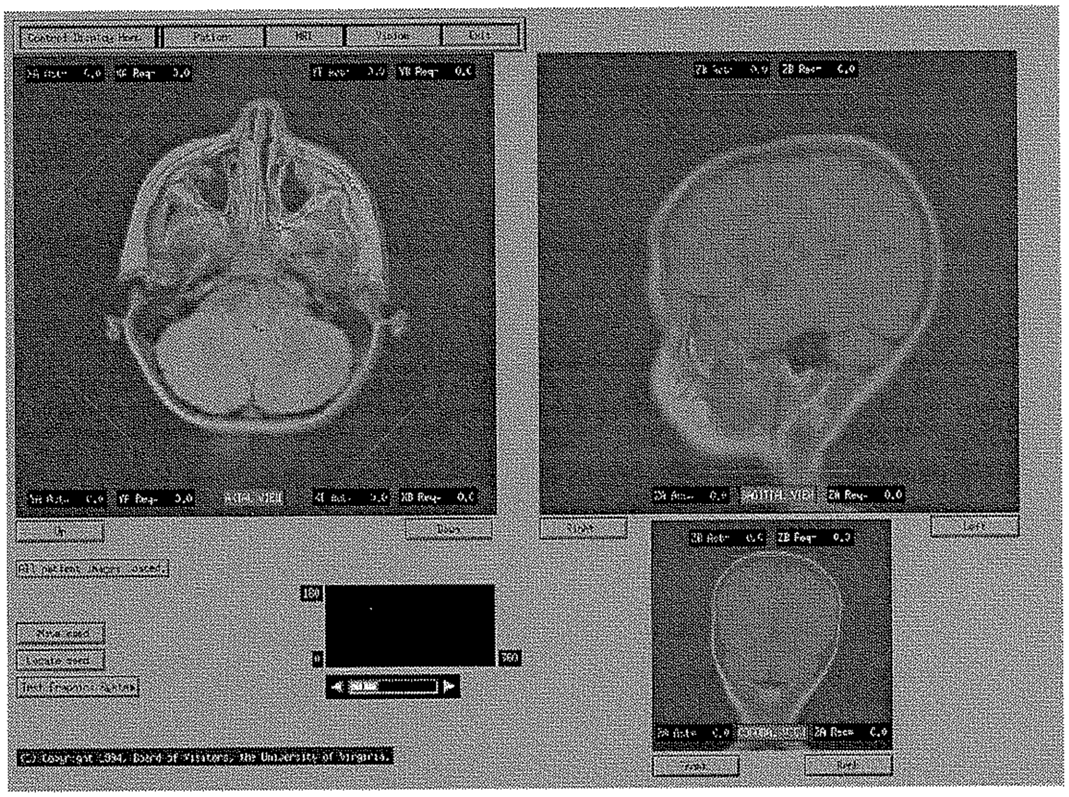

Fig. 4. MSS Operator Display.

a means of going ahead with that seed movement. The seed location within the patient's skull is then tracked and displayed on the preoperative MR images.

## 5.2 University of Virginia Reactor

The second case study application used for evaluation of the specification technique is the *University of Virginia Reactor* (UVAR). The 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 [45]. 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.

### *5.2.1 System Overview*

A diagram of the primary components of the UVAR system is shown in Fig. 5. As the figure shows, the UVAR is a light-water cooled, moderated, and shielded "pool" reactor [45]. The primary component of the reactor is the *reactor core*, an assembly that contains fuel elements, control rod elements, graphite reflector elements, and possible in-

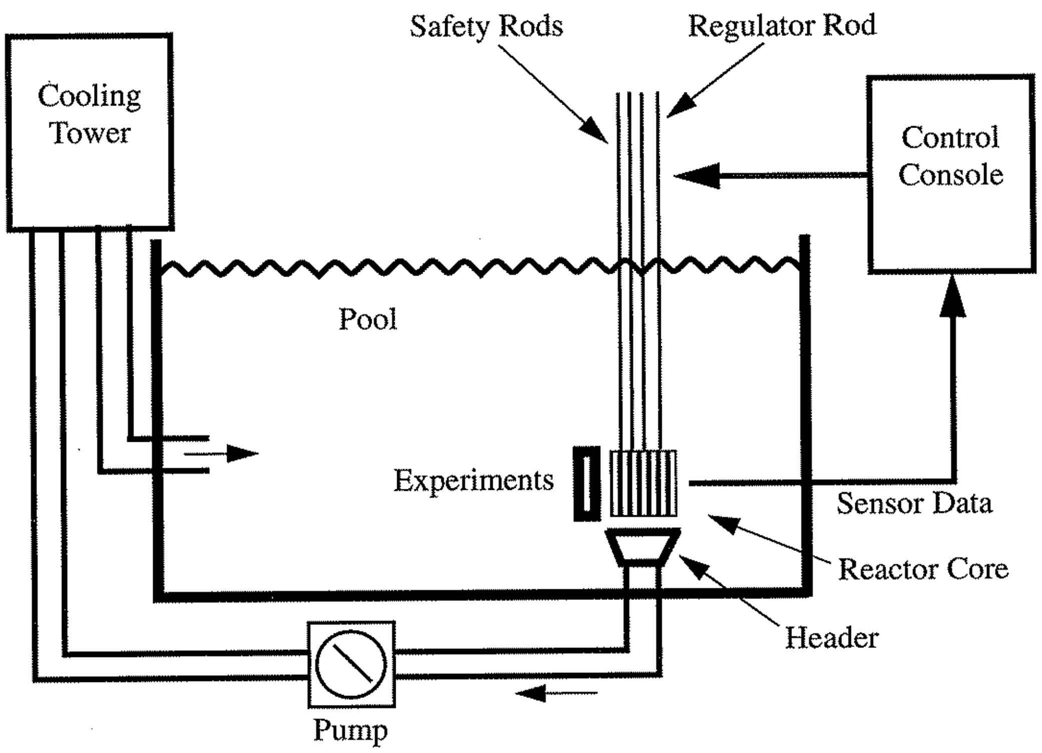

Fig. 5. University of Virginia 2.0 MW Research Reactor.

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 control rods, 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 that 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 [46].

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 that 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 [46].

The current control system is primarily analog instrumentation to monitor and regulate operating parameters over all ranges of operation, from start-up to full power. A digital computer control system is being designed for the UVAR and is currently in the specification stage.

### *5.2.2 User Interface Overview*

The current user interface for the UVAR is the control console. A first-generation prototype user interface for a digital computer control system will replicate the functionality of the current control console. The majority of that functionality is the display of process variables, including but not limited to gross output, neutron flux and period, differential temperature about the core, control and regulating rod positions, primary system flow, and pool level [51]. The control console also provides input to the reactor system, including control of the regulating and safety rods, a means to test instrumentation, and responses to unsafe conditions.

## 5.3 Evaluation Technique

Evaluation of the proposed specification approach occurred on the two case study safety-critical systems presented above. The experiment undertaken in this work was to demonstrate the feasibility of the proposed specification approach. Specifications were composed for user interfaces of both case study safety-critical systems using the proposed method in order to evaluate the method's feasibility; further, a user interface implementation was constructed from the case study specifications for one of the safety-critical systems to gain additional insight into the utility of the proposed specification approach.

A concern in software engineering research is how generally applicable are the methods and techniques explored in any work. It is often not possible to reason across all software systems about proposed software engineering methods, so evaluation occurs on case studies. The applicability of the case study must be considered when surveying research in software engineering.

This work benefits from evaluation on two very complex case study applications. Both systems are real safety-critical systems, the MSS in an advanced experimental stage and the UVAR operational in the current analog control system configuration. Both systems require complicated user interfaces. In addition, the two safety-critical systems are in different application domains and this aids assessment of general applicability. Finally, the user interfaces of the two applications have fundamental differences that further test the utility of the proposed specification approach.

Of course, it must be acknowledged that two data points still offers simply anecdotal evidence of a method's quality. In this work, the goal of the experiment is to prove either the feasibility or infeasibility of the proposed approach to specifying user interfaces for safety-critical systems. Further, the characteristics of the case study applications can be assessed in order to reason about the advantages and disadvantages of the specification technique. The validity of the two applications used in this application lends credence to the conclusions drawn through experience working with the systems.

# 6 Case Study User Interface Specifications

Given the approach to user interface specification proposed in Chapter 4, specifications were written employing that method for the user interfaces of both safety-critical case study applications, the Magnetic Stereotaxis System (MSS) and the University of Virginia Reactor (UVAR). Both user interfaces employ the same structure, as presented in Chapter 4, for their specifications. That structure is shown in Fig. 6, including the notations utilized for the primary components; the same notations were used for both case studies. Descriptions of both specifications are presented in this chapter; the complete specifications for both user interfaces can be found in the appendices.†

## 6.1 Magnetic Stereotaxis System

The user interface for the MSS, the Operator Display, was the subject of the first case study. An overview of the Operator Display specification is presented in the following subsections; the entire specification is provided in Appendix A.

### *6.1.1 Semantic Specification*

The semantic specification, a specification of the high-level functionality of the user interface, is written in the formal specification language Z. For an overview of formal specification using Z, the reader is referred to Diller [9].

The specification, found in Section A.1 of the Appendix, consists of eight sections, described in detail below. The first two sections of the Z specification are the Axiomatic Descriptions and Set Definitions. Z is an extensible language, and these two sections augment the given types of this language with types particular to the Operator Display and MSS. The next two sections, the State Description and Initialization Schema, describe the state of Operator Display using a *state schema* and an *initialization schema*. A *schema* is a mechanism in Z for grouping relevant information of a state description, which is merely a collection of sets and objects and some predicates on the state [9]; the state schema models the state of the user interface and the initialization schema models the initial state of the system. The final four sections specify the operations available in the user interface, grouped roughly according to the phases of MSS functionality: Setup, Patient Data, Identifications, and Surgical Procedures. These sections model the operations of the Operator Display using schemas that specify the effect of each activity on the state of the user interface.

†. Special thanks to Charles Odell for his assistance with the presentation specifications written in Borland ObjectWindows. Mr. Odell generated mock-ups for the MSS Operator Display from the previous Operator Display prototype constructed by the author; he generated a mock-up for the UVAR user interface from sketches designed by the author and nuclear engineering representative. The code for Mr. Odell's ObjectWindows mock-ups is presented as specification in the appendices.

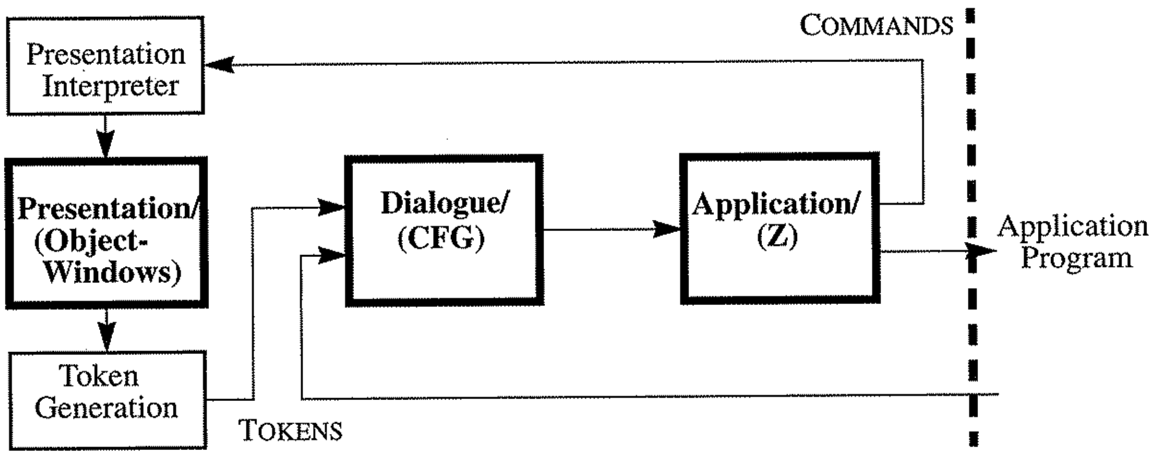

Fig. 6. Specification Structure with Notations for the MSS and UVAR.

### *Axiomatic Descriptions*

An axiomatic description in Z is a declaration that introduces one or more variables combined with an optional predicate to constrain the values of the variable(s) in the declaration [9]. For example, in the MSS, there are six electromagnetic coils which control the motion of the seed; the Operator Display must display the values of the currents through each of the six coils, therefore a type *CoilCurrents* is provided using an axiomatic description that consists of six real-number values.

### *Set Definitions*

Set definitions in Z can be either given sets or enumerated sets. Given sets are real-world, user-defined types used in the specification but not described any further, i.e. primitive types; enumerated sets are collections of elements where all the elements of the set are listed [9]. An example of a given set used in the specification of the Operator Display is [*Message*]; the Operator Display communicates with an application program interface and the interface to the graphical display, the presentation interpreter, using messages, but the composition of a *Message* is not detailed any further in this specification component. An example of an enumeration is the *View* set, which presents the three views of the patient's skull in the Operator Display's images, the Axial, Sagittal, and Coronal.

### *State Description*

The following section of the specification is the state description. Although the application functionality of the MSS is contained in a separate program, the Control Program, the Operator Display contains its own data values that it maintains and acts upon during operation. Much of this information is for determining valid sequencing of operations, i.e. context-sensitive language specification; other variables contained in the state schema contain intermediate values for high-level operations that must be stored until the operation can be effected through a call to the Control Program. The proposition section of this schema presents the invariants of the Operator Display; these invariants all pertain to context-sensitive sequencing of the state the Operator Display can be in.

### *Initialization Schema*

The initialization schema in the next section specifies initial values for those variables of the state schema that require them. The system mode is set to *InitSetup* and all variables pertaining to patient images and identification procedures, which have not occurred yet, are set to 0. All context-sensitive variables that indicate whether or not particular events have occurred are set to false.

### *Setup Operations*

Setup operations for the MSS currently consist of activities to calibrate the vision system; the Operator Display provides the means by which to perform these operations. Example setup operations are calibration of cameras and sources on each axis. The action associated with most of these operations is to send a message to the control program. There is extensive context-sensitive modelling in this section: all of these operations must be performed in order to proceed to further MSS functionality, but there are multiple, distinct sequences in which these operations can be performed. The grammar of the Syntactic Specification specifies the possible ordering, while this specification checks that all calibration operations are performed.

### *Patient Data Operations*

Patient data operations are the loading and closing of patient data, and the Control Program response to a load request. The action associated with loading patient data is simply to send a message to the Control Program. The Control Program responds by sending data; this action is modelled in the *CPLoadPatientData* schema, which models reception of data from the Control Program and the sending of a message to the presentation interpreter with that data. Finally, closing patient data consists of re-initializing some of the data elements of the Operator Display state schema and sending a message to the presentation interpreter.

### *Identification Operations*

There are two types of identifications necessary to register the position of the seed on pre-operative MR images: identification of objects on the x-rays and identification of markers on the MR images. The x-ray identifications must be performed for both axes, and the MRI identifications must be performed for all three views. There are schemas for ensuring that all necessary identifications were performed, in addition to schemas to choose which marker or object is being identified and schemas to set the position of that marker or object.

### *Surgical Procedure Operations*

The surgical procedure operations break down into two groups: schemas associated with user actions and schemas associated with computer response to those actions. For example, possible user operations that were modelled include locating the seed, moving the seed, and manipulating the slice displayed in any of the three views. The Control Program sends back messages in response to user actions; these responses usually involve data that must be passed on to the presentation interpreter in the form of messages for manipulation of the graphical display.

### *6.1.2 Syntactic Specification*

The syntactic specification, Section A.2 of the Appendix, presents the interaction language of the user interface; legal sequences of user actions and computer responses are specified. A context-free grammar is used for the specification of the interaction language.

The grammar for the Operator Display syntax consists of 84 productions and 42 tokens. The grammar was put in a format by which YACC could generate a parser for the grammar and this process revealed that the grammar has no shift/reduce or reduce/reduce conflicts. The YACC-generated parser contains 118 states.

Relative to grammars used for describing programming languages, the language describing the Operator Display is rather primitive: the tokens of the grammar have most of the actions associated with them, rather than reductions of rules. Tokens are generated by both the user through graphical actions and the computer through messages from the Control Program.

The syntactic specification maps conveniently into the semantic specification due to the simplicity of the language. Tokens that terminate rules in the grammar have semantic actions associated with them; these actions are specified by the operation schemas in the Z specification, and the schemas in Section A.1 of the Appendix are mapped to the rules to which they correspond.

### *6.1.3 Presentation Specification*

The presentation component of the Operator Display is specified using Borland ObjectWindows. For a complete specification of the graphical display, in addition to the description in Section A.3 of the Appendix, the documentation for the Borland ObjectWindows Library must be included. The definitions of base objects used in the specification may be found in the ObjectWindows documentation, in addition to definitions for the semantics of actions, such as button presses.

In addition to the textual Borland ObjectWindows specification, a visual presentation of the interface may be constructed by compilation with the ObjectWindows Library and subsequent execution. The specification is for presentation only; therefore no application functionality or dialogue control is included, and compilation would create simply a mock-up of the display. The mock-up defined by this specification looks very much like the previous prototype user interface, shown in Fig. 4 of Chapter 5.

The specification of the Operator Display consists of five sections, corresponding to the five files needed to describe the entire visual representation. The first section presents `med-sys.rc`, which is a resource file automatically generated from a graphical mock-up of a portion of the display. Graphical information for the definition of the menus and temporary button panels of the Operator Display is defined in this file.

The second section presents the `medsys.h` file. This file contains the primary class of the Operator Display, MedSysWindow. The MedSysWindow class contains declarations for the buttons, slider, three canvasses (one for each view), and other graphical objects of the Operator Display.

The next section of the presentation specification is for the `medsys.cpp` file. In this section is defined the constructor for the MedSysWindow class, which in turn instantiates the graphical objects declared in the class definition found in the previous section. In addition, the Paint function for this class displays a status string for the Operator Display and a box for input of seed direction values.

The fourth section contains the file `bmpview.h`, which defines the TCanvas class for displaying bit-mapped images. The bit-mapped images can display either the pre-operative MRI images or x-ray images.

The final section presents the `bmpview.cpp` file, which contains the constructor, destructor, and Paint function for the TCanvas class. The Paint function creates lines on the images for representation of the electromagnetic coil positions, a label identifying each view, and positions to display the actual coil current values.

### *6.1.4 Specification of Token Generation*

Tokens can be generated by either the presentation or the application program. The tokens that the presentation produces map that portion of the specification to the context-free grammar. Typically, for tokens generated by the presentation it is the graphical object's callback function which generates the token. An enumeration of the tokens and the objects that generate each of those tokens specifies the token generation.

A table in Section A.4 of the appendix relates each of the 42 tokens of the context-free grammar to either the application program or objects from the presentation specification. The callback function of the graphical object is assumed to generate presentation tokens; in cases where this is not the case, the particular method is specified with the object. Many tokens are generated by multiple graphical objects; for example, there is a single token for CALIB_CAMERA but two objects listed because camera calibration must be performed for both axes. The tokens denoted "Control Program" signify that they are generated by the application program.

### *6.1.5 Specification of Command Interpretation*

There is an enumerated set of messages specified in the semantic component whose destinations are the application program or the presentation interpreter. The messages to the presentation interpreter instruct the interpreter to alter the on-screen graphical display in some way. Each of these messages can be mapped to the particular graphical object they manipulate and the method(s) called on that object.

The messages to the application program are enumerated in section A.5 of the appendix; this is simply an extraction from the semantic specification to make explicit this interface. The messages to the presentation interpreter are related to the graphical objects or methods of the presentation that they affect in Table 3 of Appendix A.

An important piece of command interpretation is "graying out" those graphical objects that are not legal at a particular point in the dialogue. Given the context-free grammar for the syntactic specification, it is possible to determine for every step in the dialogue the set of tokens that could legally be generated by the user. When this information is passed to the presentation interpreter, that component could invalidate all other graphical objects through "graying out" or some equivalent function. *This makes it impossible for the user to generate illegal tokens.* Each graphical object in the presentation has a means by which it can be "grayed out," but this is not a part of the specification in the appendix.

## 6.2 University of Virginia Reactor

The user interface for the University of Virginia Reactor (UVAR) was the subject of the second case study. An overview of the UVAR user interface specification is presented in this section; the entire specification is provided in Appendix B. The structure of the specification is the same as that of the MSS Operator Display, including the notations used for each component, shown in Fig. 6.

As a case study for safety-critical research in general, the UVAR is far less mature than the MSS. Currently, the UVAR possesses an analog control system; design of a computer-controlled digital control system for the UVAR is in the specification stage. The initial version of the prototype software for a digital control system will simply replicate the present functionality; therefore, the current control panel was studied to determine the functionality required of the UVAR user interface.

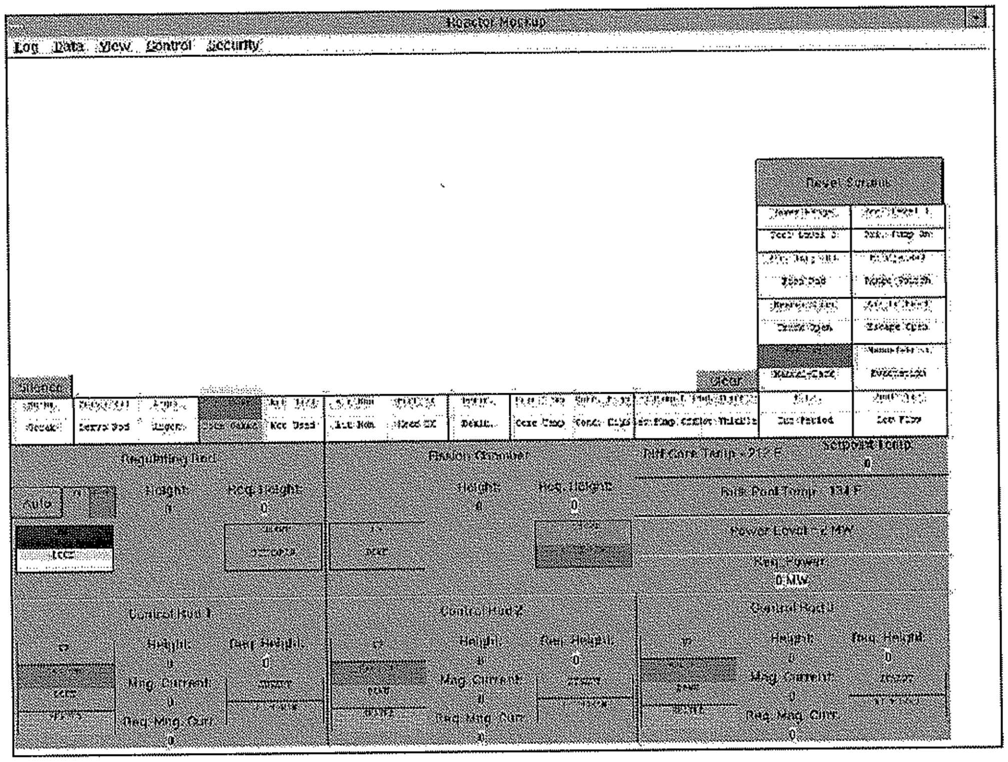

Fig. 7. Mock-up of the UVAR User Interface.

The specification for the UVAR user interface pertains to the reactor running in normal operating mode. There is an elaborate, pre-operation safety procedure that requires operator interaction with the user interface not included in the specification; however, this procedure follows a set checklist, so specifying this interaction would merely involve replicating the current checklist procedures. The user interface specification presented in this work contains the majority of operator inputs required for the user interface running under normal operation, and the associated computer responses to those inputs. The mock-up for the specified display is shown in Fig. 7.

### *6.2.1 Semantic Specification*

The semantic specification of the UVAR user interface, like that of the MSS Operator Display, is written in Z. This specification, found in Appendix Section B.1, is composed of five sections, the first four of which are the same as those found in the MSS Operator Display Z specification: the sections for Axiomatic Descriptions and Set Definitions introduce types pertaining to the UVAR, and the following two sections for the State Description and Initialization Schema describe the model of the UVAR user interface, including the state information maintained by the user interface during reactor operation and the initial values of those data items. Finally, the fifth section of the semantic specification contains the operations provided by the UVAR user interface and their effects upon the state schema.

The five sections of the semantic specification are described in further detail below.

#### *Axiomatic Descriptions*

The axiomatic descriptions for the UVAR user interface include some types that are similar to those found in the MSS Operator Display specification; for example, there is a message type to an application program and one to a presentation interpreter as in the MSS Operator Display. Other axiomatic descriptions are derived from the UVAR system details, such as the *SafetyRodHeights* and *MagneticCurrents*. The UVAR has three safety shim rods, each of which has a height and magnetic current that can be set and displayed, hence the three real-number values for each of those types.

#### *Set Definitions*

The only given set in the UVAR user interface specification provides the message type, just as introduced in the MSS Operator Display specification. The first enumerated set presents the three modes the reactor system may be in: *StartUp*, *Operating*, and *ShutDown*. Other enumerated sets include the possible alarms flagged by the reactor system and the two possible modes pertaining to the regulating rod, *Auto* and *Manual*. The possible messages that can be sent to the application program and the presentation interpreter are also enumerated.

#### *State Description*

The state description section contains the state schema for the UVAR user interface. The information contained in the state schema is that which the operator can manipulate through operations provided by the user interface; for example, the safety rod heights are stored in the state schema and there are operation schemas that alter those values. The user interface displays much more information than is modelled in the state schema, but it is not necessary that the user interface maintain those values. The state schema also contains an invariant, which is that set of alarms still sounding and the set of current alarms are always subsets of those alarms that have not been acknowledged.

#### *Initialization Schema*

The initialization schema in the next section sets all real-numbered values to 0.0 before operation, as required by the reactor system. In addition, it initializes the regulating rod control to *Manual,* the power level setting to the lower of the two, *200KW,* and all alarm sets to empty sets. Finally, the reactor system is modelled to begin operation in *StartUp* mode.

#### *Operations*

Most of the operations modelled in the semantic specification pertain to inputs to the system that the user may provide. For example, there are schemas for setting the heights of the safety shim rods, putting the regulating rod in automatic control mode, and setting a target power value to maintain while in automatic regulating mode. There are also operations associated with system alarms: there is a schema that models reception of the current set of alarms from the application program, and there are schemas to silence the sound associated with an alarm and to clear acknowledged alarms.

### *6.2.2 Syntactic Specification*

A context-free grammar is used to specify the legal sequences of events in the UVAR user interface. The grammar for this user interface, in Section B.2 of the Appendix, consists of 40 productions and 21 tokens. The relative brevity of this component compared to the MSS Operator Display syntax specification is due to the nature of the display: the MSS Operator Display is more of an interactive user interface, while the UVAR requires primarily a vehicle for displaying sensor values.

The grammar for the UVAR user interface is also in a form by which YACC could process it. Analysis with YACC revealed the grammar has no shift/reduce or reduce/reduce conflicts. A YACC-generated parser would contain 55 states, which is considerably smaller than the MSS Operator Display parser.

### *6.2.3 Presentation Specification*

The presentation of the UVAR user interface is specified using Borland ObjectWindows and can be found in Appendix Section B.3. The specification describes, and generates, a display mock-up lacking all functionality; the mock-up is shown in Fig. 6.

This user interface specification component consists of five sections. The first section is the `reactor.rc` file, which contains graphical information for the definition of the user interface menus.

The second section presents the `reactor.h` file. The primary class of the UVAR user interface, MainReactorWindow, is defined in this section. This class contains declarations of all the other graphical objects on the display: visual enumerations of scram and alarm conditions, buttons for alarm response, and classes for control rod and fission chamber information and manipulation. These classes pertaining to the control rods and fission chamber are also defined in this file.

The next section contains the file `reactor.cpp`, which has the constructors, destructors, and paint functions for each of the classes defined in the previous section. The constructor for the MainReactorWindow instantiates the graphical objects on the display.

The fourth section presents the file `light.h`, which defines the Light class. The Light class is used to display status information in each of the control rod and fission chamber objects; it replicates light indicators on the analog control panel, being a text box that turns a different color to indicate status on.

The final section of the presentation specification contains `light.cpp`; the constructor, destructor, and functions to change color for the Light class are presented in this file.

### *6.2.4 Specification of Token Generation*

The table in Section B.4 of the appendix relates each of the 21 tokens of the context-free grammar to either the application program or objects from the presentation specification. The callback function of the graphical object is assumed to generate presentation tokens; in cases where this is not the case, the particular method is specified with the object. There is a single token denoted "Application Program," which is a token generated by the application program involving the current alarm conditions.

### *6.2.5 Specification of Command Interpretation*

The messages to the application program are enumerated in section B.5 of the appendix; this is simply an extraction from the semantic specification to make explicit this interface. The messages to the presentation interpreter are related to the graphical objects or methods of the presentation that they affect in Table 3 of Appendix B.

# 7 Evaluation

Having composed specifications for both case study safety-critical applications, assessment of the experiences must occur. One measure of a specification method is how well it supports construction of an implementation. As mentioned previously, specification occurs in the context of software development, where the end goal is an executable program. For one of the case studies, the MSS Operator Display, an implementation was constructed using the specification; the first section of this chapter examines the experiences implementing the Operator Display based on the specification composed using the proposed approach. The effects of the specification method upon implementation will be explored.

The second section of this chapter evaluates the two case study specifications with respect to the desirable specification characteristics presented in Section 4.1. The specification approach will be evaluated according to how well these goals were supported, drawing on experiences from the two case studies.

## 7.1 MSS Operator Display Implementation

Prior to specification of the MSS Operator Display, a fully-operational prototype software system existed for the MSS, including a user interface. This version of the user interface was implemented in C++, using an in-house library of graphical objects to provide an object-oriented interface to X/Windows running Motif. This library provides classes to inherit off of for basic graphical objects, such as buttons and menus. There was no specification for this prototype user interface.

The Operator Display was specified using the proposed specification approach, as described in the previous chapter, and then a new implementation constructed. More precisely, some components from the original prototype user interface were reused or modified and restructured within a new implementation architecture based on the specification. Other components were written from the specification or generated directly off of the specification. The new, resulting implementation is described below.

### *7.1.1 Description of Implementation*

The structure of the revised Operator Display implementation is shown in Fig. 8. The implementation of the Operator Display is written primarily in C++ and uses the same in-house class library to interface with X/Windows. The dotted lines in Fig. 8 enclose separately-executing processes. The Control Program is the application program for the MSS; communication between the Operator Display and Control Program is via a socket connection. A parser generated by YACC is a separate process from the Operator Display proper but is logically a part of the Operator Display.

At the lowest level is the code for the graphics. The graphical object library used to implement the graphics of the Operator Display works in a similar manner as Borland ObjectWindows. A graphical object can be created and placed on the display, then to add functionality to that object, its callback function must be defined.

All of the callback functions of the graphical interface generate tokens, which are sent to the parser; further, *the callback functions do nothing more than generate tokens*. It is important to note that this is very different from most graphical user interfaces employing the traditional callback structure.

Central to the implementation is the parser. A parser was generated directly from the grammar of the syntactic specification using YACC, a widely-known parser generator. The parser receives tokens from both the graphical objects and the application program interface, known as the net_cd.

Because the parser was generated by YACC, it most naturally functions as a stand-alone process. This necessitates an interface to the parser from the Operator Display; an object in the Operator Display serves as an interface to the YACC-generated parser, passing tokens and any necessary state information to the parser via a pipe. The parser consumes the token if its legal, saves any state information received with the token, and returns to the C++ parser interface object via a pipe the validity of the token, any necessary state information, and a set of graphical objects to be grayed out. The C++ parser interface object then performs the actions associated with the token if it was legal; this involves sending commands to either the net_cd or presentation interpreter or both. This includes passing the gray-out set to the presentation interpreter.

The net_cd is the interface to the MSS application program, the Control Program. The net_cd sends messages to the Control Program to effect certain system operations and receives messages from the Control Program to display the results of system operations.

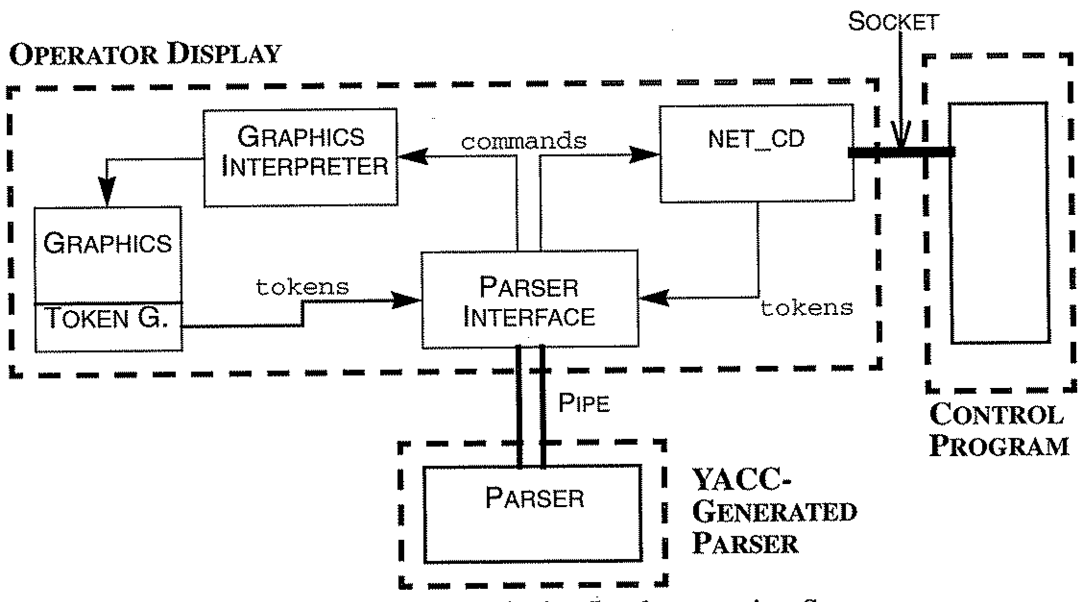

Fig. 8. MSS Operator Display Implementation Structure.

The messages the net_cd sends are the results of tokens consumed by the parser; all requests to send messages come from the parser. Similarly, all messages received by the net_cd, including the information from the Control Program, are passed to the parser in the form of a token. The parser treats tokens from the Control Program just as it does those from the user produced by callback functions: their validity at that point in the dialogue is checked and commands issued only if they are legal.

The presentation interpreter receives action requests associated with legal operations from the parser. This object manipulates the graphical objects on the screen, including graying out those objects whose callbacks are not valid at a particular point in operation.

### *7.1.2 Evaluation of Implementation*

The implementation of the Operator Display based on the specification presented in the previous chapter proved to be well-organized and cleanly-structured in such a manner as to facilitate understanding of the interface. The characteristics of the specification that contributed to such a clean, well-structured implementation are explored in this subsection.

The previous prototype of the Operator Display was implemented in a manner similar to many user interfaces: graphical objects possess callback functions, which effect the functionality of the objects. This structure requires that each graphical object also possess the state information necessary to perform its operation, which may involve information from other graphical objects. In addition, any syntactic checking of whether or not the operation is legal at that point in the dialogue requires syntactic information to be possessed by each graphical object as well. Frequently, syntactic checking is omitted entirely. The result of this structure in the prototype user interface was the intermingling of semantic information and syntactic checking in graphics code; the inclusion of syntactic and semantic information necessitated a great deal of duplicate code throughout the interface that increased the complexity of the implementation. Unfortunately, this complexity is unavoidable in situations where dialogue control is not centralized. This amalgamation of concerns led to an apparently disorganized and overly complex implementation.

The revised implementation constructed from the new Operator Display specification possesses a structure similar to that of the specification. The YACC-generated parser performs all syntactic checking in a centralized piece of software, and all state information is removed to that portion of the user interface as well. Graphic code remains separate from syntax enforcement and semantic actions. In addition, because the sole effect of all callbacks is to generate tokens, it is ensured that no action is performed without first checking its legality in the context of the current dialogue, i.e. *no syntactic checking is omitted.* This improved implementation architecture is a direct result of the structure imposed by the user interface specification.

A similar improvement in the new Operator Display implementation involves centralization of all code to manipulate the graphical display. The effect of some actions that must be performed in a user interface is to alter the presentation in some way. In the previous implementation, because all the actions performed were contained in the callback functions, those functions had to possess the means by which to alter the presentation of the graphical objects its functionality affected. Therefore, access to all of the graphical objects that were affected by the operation had to be provided, in addition to all the additional syntactic and semantic information mentioned previously.

The use of the presentation interpreter in the revised Operator Display implementation eliminates the need for graphical objects to access each other. The presentation interpreter possesses access to any graphical objects that are affected throughout the course of the dialogue; this centralizes alteration of the display. In addition, because alteration of the display is basically a semantic action initiated by the parser, the presentation interpreter provides a well-defined interface between the parser and the graphics code.

One final improvement in the revised implementation concerns error handling. With respect to handling of syntactic errors, i.e. the user requesting invalid commands, in the previous implementation a large portion of syntactic checking was omitted due to the complexity of distributed syntax checking discussed earlier. This means that errors were often not detected, much less handled properly.

In the new implementation, because the interaction language is modelled using a context-free grammar, the legal user input tokens can be determined for any particular point in the dialogue. This information enables user input syntax errors to be completely prevented through the use of "graying out" those graphical objects whose input tokens are not valid. An additional benefit was that no error recovery was necessary in the parser that enforced the grammar, because no invalid tokens could be generated. Once again, this is possible because the specification models the interaction language explicitly.

## 7.2 Specification Properties

Desirable specification properties were explored in Chapter 4. A specification approach was introduced that, if successful, would produce specifications with those desirable characteristics. Given that the specification approach has been applied to the two safety-critical case study applications, it is possible to determine for those two data points whether or not the approach successfully introduced those properties into the resultant specifications. Further, the advantages and disadvantages brought about by the given properties can be explored, drawing on experience from the two case studies.

### *7.2.1 Separation of Concerns*

Separation of concerns means that portions of the user interface that address different concerns are specified in distinct pieces. In the case of the Operator Display, this separation of concerns further enabled different concerns to be implemented in distinct pieces of code. With respect to the specification, the human-computer dialogue has three distinct levels. The specification enforces this logical separation, and, as mentioned above, the implementation corresponds. Separation of concerns involves specifying the user interface in three distinct levels, so that each level is independent of the other two.

A major advantage of this separation is that it enables multiple specification solutions to be developed for a particular level while maintaining the remainder of the user interface specification. In the MSS case study, this separation of the graphics makes it possible to specify either a two-dimensional or three-dimensional graphic display for digital imagery without altering *any* of the remainder of the specification. In fact, this approach permits entirely different display devices to be incorporated into the specification with only minimal change.

Specifying the user interface in three levels also enables prototyping of levels. Prototyping is a technique where an implementation is built very quickly to help determine what the correct requirements are. In the UVAR case study, having three separate specifications allowed specification and prototyping of one level at a time, rather than having to worry about the entire user interface. The presentation specification underwent extensive prototyping, receiving feedback from a nuclear engineer, until a mock-up was arrived at which addressed all his concerns.

Finally, separation of concerns enables testing of the user interface implementation at each level, in addition to comprehensive user interface testing. It is possible to generate faults at each level of the user interface to determine how the receiving level copes with the faults.

Experience in testing of the MSS shows the advantages of this specification approach, with its corresponding implementation structure. Previous to this specification strategy, a test harness was built for the system that generated user actions by simulating graphical actions. This necessitated access to all of the graphic functions although these were not readily available. The resulting complication in the test harness led to mistakes and likely inaccuracies in testing. Using the new user interface structure, the test harness for system testing became merely a generator of tokens to the syntax analyzer because the effect of each graphic function now was solely to generate a token. This simplified the design of the test harness and increased confidence in the testing procedures.

### *7.2.2 Systematic Interaction Between Components*

Interaction between components of the specification is quite systematic, as alluded to in the previous chapter. At the highest level is the semantic specification, written in Z. The operations of the user interface are modelled using schemas, and these schemas map almost one-for-one to the tokens of the context-free grammar. The reason for this is that the user actions are all produced by the callback functions, which solely generate tokens in the current implementation. The simplicity of the language is the reason that the composition of tokens does not equate to more rules with semantic actions.

The interaction with the presentation is systematized by interface components for communication both to and from the presentation component. The graphics generate the tokens of the context-free grammar, mapping from the presentation to the syntax. In the opposite direction, the graphics must be acted upon in response to certain tokens received from the application program. The presentation interpreter enumerates the possible manipulation requests upon the graphics and identifies the associated graphic objects and methods that effect those operations.

### *7.2.3 Formal Notations*

Formal notations were employed for each component of the user interface specification in both case study applications. The formal specification language Z was used to describe the semantic component of the user interface. Z enabled the state of the user interface to be modelled, in addition to providing the capabilities to make a precise statement of the operations and their effect on the state. Part of the state described in Z was information used for context-sensitive sequencing of operations that could not be easily modelled by the particular notation used for the syntactic component of the user interface. The formal semantics associated with Z offers the possibility of analysis being performed on this specification, such as type-checking.

The formal notation employed for describing the rules of the interaction language was a context-free grammar. This type of formal grammar, used for the context-free portion of the interaction language, allows a clear, unambiguous statement of possible ordering of operations. The notation is also easily accessible to the application engineer, specifier, and implementor, so communication regarding legal and illegal command sequences is facilitated. For both case studies, the context-free grammar was presented to the application engineer and feedback regarding its correctness was received. In the case of the UVAR, a nuclear engineer unfamiliar with this notation was presented this portion of the specification and a beneficial exchange occurred between this nuclear engineer and the specifier regarding possible operator input requests.

Another advantage of context-free grammars is that an abundance of tools exist for processing and analysis, due to the utility of this notation in compilers for programming languages. An automated tool was used to generate the parser from the grammar of the specification for this portion of the Operator Display implementation. Other analysis is discussed in a later subsection.

Finally, the formal notation utilized for presentation specification was Borland ObjectWindows. Typical methods for specifying the presentation aspect of the user interface are display mock-ups or scenarios, which are basically collections of pictures or display mock-ups. ObjectWindows provides this capability to visualize the display, but because it is a textual notation it offers advantages over using only a visualization. Pictures provide an operational definition of the interface, which an ObjectWindows specification also provides when compiled with its class libraries and executed. This operational description allows rapid learning through experimentation. It is possible, however, to reason about a formal notation in ways not possible with pictures. A textual notation is amenable to analysis; there is a formal semantics associated with the classes of the ObjectWindows library, and the semantics of object-oriented specification, including concepts such as inheritance, are well-understood. Even when screen generating systems for presentation specification produce code, a notation amenable to analysis, there is no guarantee it will be readable. This approach offers a framework for formal reasoning.

### *7.2.4 Executable Notations*

The advantage of using a notation such as Borland ObjectWindows for presentation specification is that it can provide a visualization of the interface in addition to providing all the advantages mentioned above as a formal notation. The ability to compile and execute the specification written in Borland ObjectWindows is what provides this mock-up capability.

One of the primary purposes of executable specifications is rapid prototyping, in this case of display mock-ups. ObjectWindows provides this capability: the mock-ups of the Operator Display were produced in a matter of hours. In the case of the UVAR user interface, multiple display mock-ups were rapidly generated in order to assist validation of the presentation requirements. With a library of classes to inherit off of for most graphical objects, this reuse-based solution makes construction of display mock-up sufficiently fast.

### *7.2.5 Analysis*

The use a parser generator to mechanically generate a parser for the grammar provides automatic analysis performed by the generating tool. For example, using YACC on the grammar for the Operator Display identified shift/reduce conflicts; this analysis enabled the elimination of those conflicts and simplified the grammar. In addition, YACC will generate an enumeration of all the tokens used in the grammar; this enumeration can be used to ensure that the presentation has a means to generate all of those tokens.

Analysis of the grammar itself is necessary in order to ensure consistency, completeness, and the absence of redundancy in the dialogue. The specification must also be analyzed to ensure that all necessary operations of the user interface offered in the semantic level are included in the grammar.

Finally, it should be possible to mechanically generate the set of syntactically valid tokens at any particular point in the grammar, using the action table produced for the parser and looking at the entries with valid entries for the state the grammar is in. Unfortunately, the action table used by YACC is encoded, which puts it in a format not readily read and used for mechanical generation of these sets. For the grammar of the Operator Display, it was possible to perform human analysis on the grammar to generate these sets of syntactically valid tokens. This formal inspection of the grammar for the Operator Display revealed instances where tokens were not valid in the grammar specified that needed to be.

There are also tokens that are syntactically valid but not semantically meaningful, based on context-sensitive information. Inspection of the grammar revealed tokens that were supposedly valid in the dialogue because the grammar modelled only the context-free interaction; the "gray out" set was expanded to include semantically invalid tokens as well.

### *7.2.6 Support for Implementation and Verification*

Based on the structure of the specification, an implementation can be constructed with a corresponding architecture. The clear separation of user interface levels in the specification enables a clean, highly-structured implementation. This was demonstrated for the case study involving the Operator Display, and those experiences were explored earlier in the chapter.

The structural correspondence between the specification and implementation also aids verification. Verification is the act of ensuring that the implementation is built correctly, i.e., that it correctly implements *all* of the requirements put forth in the specification. Because the specification structure and implementation structure correspond, verification of the entire user interface can be divided into manageable pieces, verifying each portion of the user interface separately.

The verification that the grammar is implemented correctly is trivial. The use of a parser generator to implement the grammar ensures that the parser correctly implements the grammar, assuming the particular tool used, YACC, is correct. YACC is a widely-used parser generator that has been in existence for many years, providing informal assurance that it operates correctly. The importance of this trivial verification must be stressed: for the Operator Display, the interaction language consists of 85 productions, describing a non-trivial language. It is unreasonable to believe that the syntax enforcement of such a complicated language could be performed under a traditional user interface structure, where enforcement is distributed amongst the various graphical objects. The language is sufficiently complicated that programming a parser, for example a recursive-descent parser, based on the grammar would be a significant undertaking and it would be difficult to verify this hand-generated parser. The use of a parser generator guarantees syntax enforcement is implemented correctly in the user interface.

Verification of the presentation should also be simple because the same executable notation, Borland ObjectWindows, could be used for both the specification and the implementation; the only modifications that would have to be made to the presentation specification would be to add a token generation call to each graphical callback, which are not defined for the display mock-up. Currently in the Operator Display, the graphic implementation uses a different notation because the MSS runs on the Sun platform and Borland ObjectWindows runs under Microsoft Windows on IBM personal computers and compatibles. This could be resolved by porting the entire Operator Display to Windows; this is discussed as future work in the final chapter.

### *7.2.7 Error Processing*

A complicated issue in user interfaces is the error processing, detection of errors and meaningful handling of those errors. The proposed specification approach provides the means by which invalid sequences of user input can be completely prevented through the use of "gray out" sets. As mentioned previously, the set of valid tokens can be determined at any point in the dialogue through analysis of the grammar; the graphical objects that produce tokens not in this set comprise the "gray out" set.

The Operator Display demonstrated the feasibility of this approach. It is not possible for the user to request an invalid operation in the Operator Display because all but the valid operations are grayed out on the display.

# 8 Conclusion

This report presents a comprehensive, highly-structured approach to specification of user interfaces for safety-critical systems. The user interface is observed to be comprised of three distinct levels, the semantic, the syntactic, and the lexical, which relate to application functionality, dialogue control, and presentation, respectively. Drawing on this view of a user interface structure, a specification approach with the following characteristics is described:

- multiple components corresponding to the various levels of the user interface.
- each component utilizing appropriate formal notations for that which is being described.
- systematic integration of specification components.
- separation of concerns that facilitates verification, testing, prototyping, and change at each level.
- validation of user input through "gray out" sets of invalid functions, generated from the syntactic specification, to prevent user input syntax errors.

The specification approach provides numerous advantages with respect to implementation as well, demonstrated on the Magnetic Stereotaxis System case study. These advantages include the following:

- strong correspondence between specification and implementation, which aids verification.
- isolation of presentation-related code from all syntax-enforcement and semantic functionality, due to similar architecture in the specification and implementation.
- automatic generation of a parser, using existing compiler technology, to enforce the grammar describing the syntax of the user interface.
- trivial verification of automatically generated component, the parser.
- centralized dialogue enforcement in a distinct portion of the user interface.
- generation of tokens, and no other functionality, by graphical object callback functions.
- well-defined interface between the presentation and dialogue control, due to the generation of tokens by graphical objects.
- assurance that all user commands are checked for syntactic validity.

Feasibility of this specification method was demonstrated using two case studies involving safety-critical systems, the Magnetic Stereotaxis System (MSS) and the University of Virginia Reactor (UVAR). A specification was composed for the Operator Display, the primary user interface for the MSS, and this specification possesses many of the desirable properties the proposed approach intended to deliver. In addition, a successful implementation for the Operator Display was constructed using the specification, demonstrating definitively the feasibility of the specification approach on this case study. A specification was also composed for the user interface to the UVAR, validating the feasibility of the approach on a second case study.

Future work includes construction of an implementation based on the specification for the UVAR user interface. Construction of a well-organized implementation will conclusively demonstrate the success of the specification approach for the second case study. This work will proceed as development of the entire software system progresses.

Future work on the Operator Display includes utilizing the specification of the presentation for the implementation. The presentation specification is written using Borland ObjectWindows; the specification can be compiled with the Borland ObjectWindows library and executed in order to provide display mock-ups. Because the specification is concerned only with presentation, none of the callback functions for the graphical objects are defined. The only extension necessary for the presentation specification to serve as the implementation is each callback must be defined to generate a token, which is sent to the parser.

The larger obstacle in using Borland ObjectWindows code for both specification and implementation is that ObjectWindows runs under Microsoft Windows and the existing MSS software runs under SunOS. The Operator Display communicates to the rest of the MSS software via a socket connection, though, so only the Operator Display would need to be ported over to Microsoft Windows and it could then communicate with the remainder of the existing software still running under SunOS.

Finally, a great deal of future work remains in the area of analysis and tool support. The semantic specification, written in Z, could have various forms of analysis performed on it. There exist tools for analysis of Z specifications, for example, type checkers; the possibility of converting the Z specification presented in this work into a form able to be processed by existing tools should be studied.

With respect to the syntactic specification, an obvious target for tool support involves the generation of valid token sets for the dialogue. For a context-free grammar, these sets can be determined from the action table of the parser. The tool used for generation of a parser, YACC, provides its action tables but encodes them. Understanding the format of these action tables in order to build a tool for generation of the gray out sets is a definite area for future study.

Lastly, automatic analysis could be performed with respect to interfaces between components in either the specification or implementation. Regarding the relationship between the presentation code and the parser, there should be a mechanical check performed that all of the tokens of the grammar are generated by some graphical object callback. Also, a mechanical check could be performed relating all messages defined in the semantic specification to the interfaces with the application program and the presentation interpreter; all messages should be mapped to capabilities provided by each of those entities.

# References

1. Aaram, J., "The BOP Prototyping Concept," *Approaches to Prototyping*, eds. R. Budde *et al.*, Springer-Verlag, 1984, pp. 179-187.

2. Abowd, G. and Dix, A., "Integrating Status and Event Phenomena in Formal Specifications of Interactive Systems," *ACM SIGSOFT*, Vol. (Dec. 1994), pp. 44-52.

3. Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981, p. 37.

4. Borenstein, N., *Programming as if People Mattered*, Princeton University Press, 1991.

5. Borland International, *Borland ObjectWindows for C++ Programmer's Guide*, 1993.

6. Buxton, W., Lamb, M., Sherman, D., and Smith, K., "Towards a Comprehensive User Interface Management System," *Computer Graphics*, Vol. 17-3 (July 1983), pp. 35-42.

7. Casey, B. and Dasarathy, B., "Modelling and Validating the Man-Machine Interface," *Software-Practice and Experience*, Vol. 12-6 (June 1982), pp. 557-569.

8. Christensen, N., and Kreplin, K., "Prototyping of User-Interfaces," *Approaches to Prototyping*, eds. R. Budde *et al.*, Springer-Verlag, 1984, pp. 58-67.

9. Diller, A., Z: *An Introduction to Formal Methods*, John Wiley and Sons, Inc., 1990.

10. Ege, R. and Stary, C., "Designing Maintainable, Reusable Interfaces," *IEEE Software*, Vol. 9-6 (Nov. 1992), pp. 24-32.

11. Engler, N., "Turning GUI Into Gold," *Unix World's Open Computing*, Vol. 11-3 (March 1994), pp. 68-74.

12. Foley, J. and Wallace, V., "The Art of Natural Graphic Man-Machine Conversation," *Proceedings of the IEEE*, Vol. 62-4 (April 1974), pp. 462-471.

13. Foley, J. and Van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 1982, pp. 217-242.

14. Foley, J., Kim, W., and Gibbs, C., "Algorithms to Transform the Formal Specification of a User-Computer Interface," in *Proceedings of Second IFIP Conference Human-Computer Interaction*, IFIP, Geneva, 1987, pp. 1001-1006.

15. Foley, J., Kim, W., Kovacevic, S., and Murray, K., "Defining Interfaces at a High Level of Abstraction," *IEEE Software*, Vol. 6-1 (Jan. 1989), pp. 25-32.

16. Gillies, G. T. et al, "Magnetic Manipulation Instrumentation for Medical Physics Research," *Review of Scientific Instruments*, Vol. 65-3 (March 1994), pp. 533 - 562.

17. Grady, M. S. et al, "Preliminary Experimental Investigation of *in vivo* Magnetic Manipulation: Results and Potential Application in Hyperthermia," *Medical Physics*, Vol. 16-2 (March/April 1989), pp. 263 - 272.

18. Green, M., "The Design of Graphical User Interfaces," Tech. Report CSRI-170, Computer Systems Research Institute, University of Toronto, 1985.

19. Harel, D., "On Visual Formalisms," *CACM*, Vol. 31-5 (May 1988), pp. 514-530.

20. Hekmatpour, S. and Ince, D., *Software Prototyping, Formal Methods, and VDM*, Addison-Wesley Publishing Company, 1988, pp. 26-35.

21. Hix, D., "Generations of User-Interface Management Systems," *IEEE Software*, Vol. 7-5 (Sept. 1990), pp. 77-87.

22. Hix, D. and Hartson, R., *Developing User Interfaces: Ensuring Usability Through Product and Process*, John Wiley and Sons, Inc., 1993.

23. Hooper, J. and Hsia, P., "Scenario-Based Prototyping for Requirements Identification," *ACM SIGSOFT*, Vol. 7-5 (Dec. 1982), pp. 88-93.

24. Ince, D., *An Introduction to Discrete Mathematics and Formal System Specification*, Clarendon Press, 1988.

25. Jacob, R., "Using Formal Specifications in the Design of a Human-Computer Interface," CACM, Vol. 26-4 (April 1983), pp. 259-264.

26. Jacob, R., "A Specification Language for Direct-Manipulation User Interfaces," *ACM Transaction on Graphics*, Vol. 5-4 (Oct. 1986), pp. 283-317.

27. Johnson, S., "YACC - Yet Another Compiler Compiler," Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

28. Kienzle, D., "Software Safety: A Formal Approach," M.S. Thesis, University of Virginia, Charlottesville, VA, May 1992.

29. Leveson, N., "Software Safety: Why, What, and How," *Computing Surveys*, Vol. 18-2 (June 1986), pp. 125-163.

30. Leveson, N.G., and Turner, C.S., "An Investigation of the Therac 25 Accidents," *IEEE Computer*, Vol. 26-7 (July 1993), pp. 18-41.

31. Luqi, and Royce, W., "Status Report: Computer-Aided Prototyping," *IEEE Software*, Vol. 6-6 (Nov. 1989), pp. 77-81.

32. Mason, R. and Carey, T., "Prototyping Interactive Information Systems," *CACM*, Vol. 26-5 (May 1983), pp. 347-354.

33. Mittermeir, R., "HIBOL: A Language for Fast Prototyping in Data Processing Envi-

ronments," *ACM SIGSOFT*, Vol. 7-5 (Dec. 1982), pp. 133-140.

34. Moran, T., "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," *International Journal Man-Machine Studies*, Vol. 15 (1981), pp. 3-50.

35. Myers, B., "User-Interface Tools: Introduction and Survey," *IEEE Software*, Vol. 6-1 (Jan. 1989), pp. 15-23.

36. National Instruments Corporation, "LabVIEW 2: The Complete Instrumentation Software System," *IEEE-488 and VXIbus Control, Data Acquisition, and Analysis*, pp. 1:5-1:14.

37. Ousterhout, J., *Tcl and the Tk Toolkit*, Addison-Wesley Publishing Company, 1994.

38. Parnas, D., "On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System," in *Proceedings of the 24th National Conference of the ACM*, 1969, pp. 379-385.

39. Potter, B. et al, *An Introduction to Formal Specification and Z*, Prentice Hall, 1991.

40. Reisner, P., "Formal Grammar and Human Factors Design of an Interactive Graphics System," *IEEE Transactions on Software Engineering*, Vol. 7-2 (March 1981), pp. 229-240.

41. Rudolf, J. and Waite, C., "Completing the Job of Interface Design," IEEE Software, Vol. 9-6 (Nov. 1992), pp. 11-22.

42. Shneiderman, B., "Multiparty Grammars and Related Features for Defining Interactive Systems," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 12-2 (March/April 1982), pp. 148-154.

43. Shneiderman, B., *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley Publishing Company, 1992, pp. 506-517.

44. Sufrin, B. and He, J., "Specification, Analysis, and Refinement of Interactive Processes," Formal Methods in Human-Computer Interaction, eds. M. Harrison and H. Thimbleby, Cambridge University Press, 1990, pp. 153-200.

45. University of Virginia Reactor, *The University of Virginia Nuclear Reactor Facility Tour Information Booklet*, `http://minerva.acc.virginia.edu/~reactor`.

46. University of Virginia Reactor Safety Committee, *University of Virginia Reactor Safety Analysis Report*, `http://minerva.acc.virginia.edu/~reactor`.

47. Van Hoeve, F. and Engmann, R., "The TUBA-Project: A Set of Tools for Application Development and Prototyping," *Approaches to Prototyping*, eds. R. Budde *et al.*, Springer-Verlag, 1984, pp. 202-213.

48. Wasserman, A., "Extending State Transition Diagrams for the Specification of Human-Computer Interaction," *IEEE Transactions on Software Engineering*, Vol.

11-8 (Aug. 1985), pp. 699-713.

49. Wasserman, A. and Shewmake, D., "Rapid Prototyping of Interactive Information Systems," *ACM SIGSOFT Software Engineering Notes*, Vol. 7-5 (Dec. 1982), pp. 171-180.

50. Wellner, P., "Statemaster: A UIMS Based on Statecharts for Prototyping and Target Implementation," in *Proceedings CHI '89 Conference - Human Factors in Computer Systems,* ACM, New York, 1989, pp. 177-182.

51. Wika, K., "Safety Kernel Enforcement of Software Safety Policies," Ph.D. Dissertation, University of Virginia, Charlottesville, VA, May 1995.

52. Wika, K. and Knight, J., "Software Safety in a Medical Application," in *Proceedings of the First International Symposium on Medical Robotics and Computer Assisted Surgery*, Pittsburgh, PA, 1994.

53. Wing, J., "A Specifier's Introduction to Formal Methods," *IEEE Computer*, Vol. 23-9 (Sept. 1990), pp. 8-24.

# Appendix A

# MSS Operator Display Specification

This is the specification of the Operator Display, the user interface to the Magnetic Stereotaxis System. The specification of the Operator Display consists of various component specifications which comprise different aspects of the user interface. The sections of the Operator Display specification with their associated specification notations are shown in Table 1.

**Table 1: Operator Display Specification Components**

| Section | Component | Technology |
|---|---|---|
| A.1 | Semantics | Z |
| A.2 | Syntax | Context-Free Grammar |
| A.3 | Presentation | Borland OWL |
| A.4 | Token Generation | Tables |
| A.5 | Command Interpretation | Tables |

## A.1 Semantic Specification

The functionality of the Operator Display is specified in the formal specification language Z. For an overview of formal specification using Z, the reader is referred to Diller.

The semantic specification consists of the following sections:

- Axiomatic Descriptions
- Set Definitions
- State Description
- Initialization Schema
- Setup Operations
- Patient Data Operations
- Operations for X-Ray Object and MRI Marker Identification
- Surgical Procedure Operations

The first four sections define the basic types used in the specification and describe the state of the Operator Display as modelled in this specification. The latter four sections model the operations that the user interface provides and their effect upon the state of the system. These four sections group operations roughly according to phases in the operation of the MSS; numbers above the operation schemas refer to the production numbers in the grammar of Section A.2. Most schemas map to either a token or a rule reduction in the context-free grammar.

## Axiomatic Descriptions

$$\begin{array}{|l} \mathit{ImageViewSeq} : \operatorname{seq} \mathit{ImageSeq} \\ \hline \#\mathit{ImageViewSeq} = 3 \end{array}$$

$$\begin{array}{|l} \mathit{ImageSeq} : \operatorname{seq} \mathit{Slices} \\ \hline \#\mathit{ImageSeq} = 13 \end{array}$$

$$\begin{array}{|l} \mathit{Slices} : \operatorname{seq} \mathit{Pixel} \end{array}$$

$$\begin{array}{|l} \mathit{XrayImage} : \operatorname{seq} \mathit{Pixel} \end{array}$$

$$\begin{array}{|l} \mathit{Position} : \operatorname{seq} \mathbb{R} \\ \hline \#\mathit{Position} = 2 \end{array}$$

$$\begin{array}{|l} \mathit{Coordinate} : \operatorname{seq} \mathbb{R} \\ \hline \#\mathit{Coordinate} = 3 \end{array}$$

$$\begin{array}{|l} \mathit{CoilCurrents} : \operatorname{seq} \mathbb{R} \\ \hline \#\mathit{CoilCurrents} = 6 \end{array}$$

$$String : \mathrm{seq}\, Char$$

$$MessageToCP : MethodsToCP \times \mathbb{N} \times \mathbb{R} \times String \times Position \pfun Message$$

$$MessageToGI : MethodsToGI \times \mathbb{N} \times \mathbb{R} \times Axis \times View \times Xray \times String \times Position \times Coordinate \times ImageViewSeq \pfun Message$$

## Set Definitions

[*Message, Pixel*]

| | |
|---|---|
| *Char* | == {*a..z*, 0..9} |
| *Modes* | == {*InitSetup, PatientData, Identifications, Surgery*} |
| *Axis* | == {*Xaxis, Yaxis*} |
| *View* | == {*Axial, Sagittal, Coronal*} |
| *MethodsToCP* | == {*AcquireXrayBackX, AcquireXrayBackY, CalibrateCameraX, CalibrateCameraY, CalibrateSourceX, CalibrateSourceY, LoadPatientData, InitiateXrayObjIdX, InitiateXrayObjIdX, SetInitObjPositsX, SetInitObjPositsY, CancelXrayObjIdX, CancelXrayObjIdY, LocateSeed, MoveSeed, SetMotion, Quit*} |
| *MethodsToGI* | == {*LoadPatient, IDXrayImageObjs, CreateArrow, ReorientArrow, DisplaySlices, CreateCircle, MoveFeature, ClosePatient, DeleteVisPanel, DeleteMRIPanel, CreateMRIPanel, DisplayCurrentXray, DismissXray, ShutDown, ViewUp, ViewDown, ViewRight, ViewLeft, ViewFront, ViewBack, SliderVal, SetReqCoil, ChangeMRIButtonColor, ChangeVisButtonColor*} |

## State Description

*OperatorDisplay*

| | | |
|---|---|---|
| *mode* | : | *Modes* |
| *axelems, sagelems, corelems* | : | $\mathbb{N}$ |
| *axslice, sagslice, corslice* | : | $\mathbb{N}$ |
| *sliderval* | : | $\mathbb{R}$ |
| *curview* | : | *View* |
| *curaxis* | : | *Axis* |
| *numids* | : | $\mathbb{N}$ |
| *idposits* | : | seq *Position* |
| *idflag* | : | seq *Boolean* |
| *markslice* | : | seq $\mathbb{N}$ |
| *idinprog* | : | $\mathbb{N}$ |
| *reqcurrs* | : | *CoilCurrents* |
| *backx, backy, camx, srcx, camy, srcy* | : | *Boolean* |
| *patientloaded* | : | *Boolean* |
| *donexrayid, xaxisdone, yaxisdone* | : | *Boolean* |
| *donemriid, axdone, sagdone, cordone* | : | *Boolean* |

| | |
|---|---|
| $(mode = PatientData)$ | $\Rightarrow (backx \wedge backy \wedge camx \wedge srcx \wedge camy \wedge srcy)$ |
| $(mode = Identifications)$ | $\Rightarrow patientloaded$ |
| $(mode = Surgery)$ | $\Rightarrow (donexrayobjid \wedge donemrimarkid)$ |
| $donexrayobjid$ | $\Rightarrow (xaxisdone \wedge yaxisdone)$ |
| $donemrimarkid$ | $\Rightarrow (axdone \wedge sagdone \wedge cordone)$ |

## Initialization Schema

*InitOperatorDisplay*

*OperatorDisplay'*

$$
\begin{array}{lllllllllllll}
mode' & = & InitSetup \\
axelems' & = & sagelems' & = & corelems' & = & 0 \\
axslice' & = & sagslice' & = & corslice' & = & 0 \\
sliderval' & = & 0.0 \\
numids' & = & 0 \\
idposits' & = & \langle\rangle \\
idflag' & = & \langle\rangle \\
markslice' & = & \langle\rangle \\
idinprog' & = & 0 \\
reqcurrs' & = & \langle 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 \rangle
\end{array}
$$

$$backx' = backy' = camx' = srcx' = camy' = srcy' = false$$

$$
\begin{array}{lllllllll}
patientloaded' & = & false \\
donexrayobjid' & = & xaxisdone' & = & yaxisdone' & = & false \\
donemrimarkid' & = & axdone' & = & sagdone' & = & cordone' & = & false
\end{array}
$$

## Setup Operations

- 2: vis_params

$$\begin{array}{|l}
\textit{VisionParams} \\
\Xi \textit{OperatorDisplay} \\
\hline
\textit{mode} = \textit{InitSetup} \\
\hline
\end{array}$$

- 4: backgrnd

$$\begin{array}{|lcl}
\multicolumn{3}{|l}{\textit{AcquireXrayBackground}} \\
\multicolumn{3}{|l}{\Delta \textit{OperatorDisplay}} \\
\textit{msgtocp}! & : & \textit{MessageToCP} \\
\textit{whichaxis}? & : & \textit{Axis} \\
\hline
\multicolumn{3}{|l}{\textit{mode} = \textit{InitSetup}} \\
\multicolumn{3}{|l}{(((\textit{whichaxis}? = \textit{Xaxis}) \Rightarrow (\textit{backx}' \wedge \textit{msgtocp}!(\textit{AcquireXrayBackX}) ))} \\
\multicolumn{3}{|l}{\qquad\qquad\vee} \\
\multicolumn{3}{|l}{((\textit{whichaxis}? = \textit{Yaxis}) \Rightarrow (\textit{backy}' \wedge \textit{msgtocp}!(\textit{AcquireXrayBackY}) )))} \\
\hline
\end{array}$$

- 6: cam_calib

$$\begin{array}{|lcl}
\multicolumn{3}{|l}{\textit{CalibrateCamera}} \\
\multicolumn{3}{|l}{\Delta \textit{OperatorDisplay}} \\
\textit{msgtocp}! & : & \textit{MessageToCP} \\
\textit{whichaxis}? & : & \textit{Axis} \\
\hline
\multicolumn{3}{|l}{\textit{mode} = \textit{InitSetup}} \\
\multicolumn{3}{|l}{(((\textit{whichaxis}? = \textit{Xaxis}) \Rightarrow (\textit{camx}' \wedge \textit{msgtocp}!(\textit{CalibrateCameraX}) ))} \\
\multicolumn{3}{|l}{\qquad\qquad\vee} \\
\multicolumn{3}{|l}{((\textit{whichaxis}? = \textit{Yaxis}) \Rightarrow (\textit{camy}' \wedge \textit{msgtocp}!(\textit{CalibrateCameraY}) )))} \\
\hline
\end{array}$$

• 7: src_calib

```
┌── CalibrateSource ──────────────────────
│ ΔOperatorDisplay
│ msgtocp!      :   MessageToCP
│ whichaxis?    :   Axis
├──────────────────
│ mode  =  InitSetup
│ (((whichaxis? = Xaxis) ⇒ (srcx' ∧ msgtocp!(CalibrateSourceX) ))
│                 ∨
│  ((whichaxis? = Yaxis) ⇒ (srcy' ∧ msgtocp!(CalibrateSourceY) )))
└─────────────────────────────────────────
```

• 3: check_calib, 5: check_axis

```
┌── DoneVisionCalib ──────────────────────
│ ΔOperatorDisplay
├──────────────────
│ mode  =  InitSetup
│ (backx ∧ backy ∧ camx ∧ srcx ∧ camy ∧ srcy) ⇒ (mode' = PatientData)
└─────────────────────────────────────────
```

## Patient Data Operations

- 11: load_pat

$$
\begin{array}{|l}
\textit{LoadPatientData} \\
\hline
\Xi\, \textit{OperatorDisplay} \\
\begin{array}{lcl}
\textit{msgtocp!} & : & \textit{MessageToCP} \\
\textit{filename?} & : & \textit{String}
\end{array} \\
\hline
\begin{array}{lcl}
\textit{mode} & = & \textit{PatientData}
\end{array} \\
\textit{msgtocp!}(\textit{LoadPatientData}, \textit{filename?}) \\
\hline
\end{array}
$$

- 10: pat_data

$$
\begin{array}{|l}
\textit{CPLoadPatientData} \\
\hline
\Delta\, \textit{OperatorDisplay} \\
\begin{array}{lcl}
\textit{setofimages?} & : & \textit{ImageViewSeq} \\
\textit{msgtogi!} & : & \textit{MessageToGI}
\end{array} \\
\hline
\begin{array}{lclclcl}
\textit{mode} & = & \textit{PatientData} \\
\textit{axelems'} & = & \#(\textit{setofimages?}(1)) \\
\textit{sagelems'} & = & \#(\textit{setofimages?}(2)) \\
\textit{corelems'} & = & \#(\textit{setofimages?}(3)) \\
\textit{axslice'} & = & \textit{sagslice'} & = & \textit{corslice'} & = & 1
\end{array} \\
\textit{patientloaded'} \\
\begin{array}{lcl}
\textit{mode'} & = & \textit{Identifications}
\end{array} \\
\textit{msgtogi!}(\textit{LoadPatient}, \textit{setofimages?}) \\
\hline
\end{array}
$$

- 82: close_pat

$$
\begin{array}{|l}
\textit{ClosePatient} \\
\hline
\Delta\, \textit{OperatorDisplay} \\
\begin{array}{lcl}
\textit{msgtogi!} & : & \textit{MessageToGI}
\end{array} \\
\hline
\begin{array}{lclclcl}
\textit{mode} & = & \textit{Surgery} \\
\textit{axelems'} & = & \textit{sagelems'} & = & \textit{corelems'} & = & 0 \\
\textit{axslice'} & = & \textit{sagslice'} & = & \textit{corslice'} & = & 0
\end{array} \\
\neg \textit{patientloaded'} \\
\textit{msgtogi!}(\textit{ClosePatient}) \\
\begin{array}{lcl}
\textit{mode'} & = & \textit{PatientData}
\end{array} \\
\hline
\end{array}
$$

## Operations for X-Ray Object and MRI Marker Identification

$$Identifications \;\widehat{=}\; (DoneVisID \fatsemi CheckBothIDsDone) \vee (DoneMRIID \fatsemi CheckBothIDsDone)$$

- 12: check_both_ids

*CheckBothIDsDone*

$\Delta OperatorDisplay$

$mode = Identifications$
$(donexrayobjid \wedge donemrimarkid) \Rightarrow (mode' = Surgery)$

- 14: 42: vis_done

*DoneVisID*

$\Delta OperatorDisplay$

$(mode = Identifications) \vee (mode = Surgery)$
$(xaxisdone \wedge yaxisdone) \Rightarrow donexrayobjid'$

- 15: 43: mark_done

*DoneMRIID*

$\Delta OperatorDisplay$

$(mode = Identifications) \vee (mode = Surgery)$
$(axdone \wedge sagdone \wedge cordone) \Rightarrow donemrimarkid'$

*IDCleanup*

$\Delta OperatorDisplay$

$(mode = Identifications) \vee (mode = Surgery)$
$numids' = 0$
$idinprog' = 0$
$idposits' = \langle\rangle$
$idflag' = \langle\rangle$

- 19: init_obj

$$
\begin{array}{|l}
\textit{InitiateXrayObjectID} \\
\Xi\, \textit{OperatorDisplay} \\
\textit{msgtocp}! \quad : \quad \textit{MessageToCP} \\
\textit{whichaxis}? \quad : \quad \textit{Axis} \\
\hline
(\textit{mode} = \textit{Identifications}) \vee (\textit{mode} = \textit{Surgery}) \\
(((\textit{whichaxis}? = \textit{Xaxis}) \Rightarrow \textit{msgtocp}!(\textit{InitiateXrayObjIdX})\,) \\
\qquad\qquad \vee \\
((\textit{whichaxis}? = \textit{Yaxis}) \Rightarrow \textit{msgtocp}!(\textit{InitiateXrayObjIdY})\,)) \\
\hline
\end{array}
$$

- 18: cp_init_o

$$
\begin{array}{|l}
\textit{CPIdentifyXrayObjects} \\
\Delta\, \textit{OperatorDisplay} \\
\textit{whichaxis}? \quad : \quad \textit{Axis} \\
\textit{animage}? \quad : \quad \textit{XrayImage} \\
\textit{numobjects}? \quad : \quad \mathbb{N} \\
\textit{names}? \quad : \quad \mathrm{seq}\ \textit{String} \\
\textit{posits}? \quad : \quad \mathrm{seq}\ \textit{Position} \\
\textit{objidd}? \quad : \quad \mathrm{seq}\ \textit{Boolean} \\
\textit{msgtogi}! \quad : \quad \textit{MessageToGI} \\
\hline
(\textit{mode} = \textit{Identifications}) \vee (\textit{mode} = \textit{Surgery}) \\
\textit{curaxis}' \quad = \quad \textit{whichaxis}? \\
\textit{numids}' \quad = \quad \textit{numobjects}? \\
\forall\ \textit{posit} : \textit{Position} \mid \textit{posit} \in \textit{posits}? \bullet \textit{idposits}' = \textit{idposits}' \frown \langle \textit{posit} \rangle \\
\forall\ \textit{flag} : \textit{Boolean} \mid \textit{flag} \in \textit{objidd}? \bullet \textit{idflag}' = \textit{idflag}' \frown \langle \textit{flag} \rangle \\
\textit{msgtogi}!(\textit{IDXrayImageObjs}, \textit{whichaxis}?, \textit{animage}?, \textit{numobjects}?, \textit{names}?, \\
\qquad\qquad \textit{objidd}?) \\
\hline
\end{array}
$$

- 25: 26: select_o

$$
\begin{array}{|l}
\textit{SelectObject} \\
\Delta\, \textit{OperatorDisplay} \\
\textit{whichobj}?, \textit{prevobj}? \quad : \quad \mathbb{N} \\
\textit{msgtogi}! \quad : \quad \textit{MessageToGI} \\
\hline
(\textit{mode} = \textit{Identifications}) \vee (\textit{mode} = \textit{Surgery}) \\
\textit{idinprog}' \quad = \quad \textit{whichobj}? \\
\textit{msgtogi}!(\textit{ChangeVisButtonColor}, \textit{idinprog}', \textit{prevobj}?, \textit{idflag}(\textit{prevobj}?)) \\
\hline
\end{array}
$$

- 22: obj_posit

*GetObjectPosition*

$\Delta \mathit{OperatorDisplay}$
$\mathit{objectposit?} : \mathit{Position}$
$\mathit{msgtogi!} : \mathit{MessageToGI}$

$(\mathit{mode} = \mathit{Identifications}) \vee (\mathit{mode} = \mathit{Surgery})$
$\neg(\mathit{idinprog} = 0)$
$\mathit{idposits}(\mathit{idinprog})' = \mathit{objectposit?}$
$\mathit{idflag}(\mathit{idinprog})'$
$\mathit{idinprog}' = 0$
$\mathit{msgtogi!}(\mathit{ChangeVisButtonColor}, \mathit{idinprog})$

$\mathit{DoneAxis} \mathrel{\widehat=} \mathit{SetInitialObjectPositions} \wedge \mathit{IDCleanup}$
$\mathit{CancelAxis} \mathrel{\widehat=} \mathit{CancelInitialObjectPositions} \wedge \mathit{IDCleanup}$

- 23: done_obj

*SetInitialObjectPositions*

$\Delta \mathit{OperatorDisplay}$
$\mathit{msgtocp!} : \mathit{MessageToCP}$
$\mathit{msgtogi!} : \mathit{MessageToGI}$

$(\mathit{mode} = \mathit{Identifications}) \vee (\mathit{mode} = \mathit{Surgery})$
$((\forall\ \mathit{objidd} : \mathit{Boolean} \mid \mathit{objidd} \in \mathit{idflag} \bullet \mathit{objidd}) \Rightarrow$
$(\mathit{msgtogi!}(\mathit{DeleteVisPanel})$
$\wedge$
$(((\mathit{curaxis} = \mathit{Xaxis}) \Rightarrow$
$(\mathit{xaxisdone}' \wedge \mathit{msgtocp!}(\mathit{SetInitObjPositsX}, \mathit{idposits}, \mathit{numids})))$
$\vee$
$((\mathit{curaxis} = \mathit{Yaxis}) \Rightarrow$
$(\mathit{yaxisdone}' \wedge \mathit{msgtocp!}(\mathit{SetInitObjPositsY}, \mathit{idposits}, \mathit{numids})))$

- 24: cancel_obj

*CancelInitialObjectPositions*

$\Delta OperatorDisplay$
$msgtocp! : MessageToCP$
$msgtogi! : MessageToGI$

---

$(mode = Identifications) \vee (mode = Surgery)$
$msgtogi!(DeleteVisPanel)$
$(((curaxis = Xaxis) \Rightarrow msgtocp!(CancelXrayObjIdX))$
$\vee$
$((curaxis = Yaxis) \Rightarrow msgtocp!(CancelXrayObjIdY)))$

- 29: init_mark

*InitiateMRIMarkerID*

$\Delta OperatorDisplay$
$whichview? : View$
$nummarks? : \mathbb{N}$
$names? : \mathrm{seq}\ String$
$posits? : \mathrm{seq}\ Position$
$markidd? : \mathrm{seq}\ Boolean$
$msgtogi! : MessageToGI$

---

$(mode = Identifications) \vee (mode = Surgery)$
$curview' = whichview?$
$numids' = nummarks?$
$\forall\, marks : \mathbb{N} \mid marks \in markslice \bullet markslice' = markslice' \frown \langle 0 \rangle$
$\forall\, posit : Position \mid posit \in posits? \bullet idposits' = idposits' \frown \langle posit \rangle$
$\forall\, flag : Boolean \mid flag \in markidd? \bullet idflag' = idflag' \frown \langle flag \rangle$
$msgtogi!(CreateMRIPanel)$

- 35: 36: slct_mark

*SelectMarker*

$\Delta OperatorDisplay$
$whichmark?, prevmark? : \mathbb{N}$
$msgtogi! : MessageToGI$

---

$(mode = Identifications) \vee (mode = Surgery)$
$idinprog' = whichmark?$
$msgtogi!(ChangeMRIButtonColor, idinprog', prevmark?, idflag(prevmark?))$

- 32: mrk_posit

```
── GetMarkerPosition ──────────────────────
│ ΔOperatorDisplay
│ markerposit?  :  Position
│ slicenum?     :  ℕ
│ msgtogi!      :  MessageToGI
├──────────────────────
│ (mode = Identifications) ∨ (mode = Surgery)
│ ¬(idinprog = 0)
│ idposits(idinprog)'   =  markerposit?
│ markslice(idinprog)'  =  slicenum?
│ idflag(idinprog)'
│ idinprog'   =   0
│ msgtogi!(ChangeMRIButtonColor, idinprog)
└──────────────────────────────────────────
```

$DoneView \;\widehat{=}\; SetMarkerPosits \wedge IDCleanup$
$CancelView \;\widehat{=}\; CancelMarkerPosits \wedge IDCleanup$

- 33: done_mark

```
── DoneView ───────────────────────────────
│ ΔOperatorDisplay
│ msgtogi!      :  MessageToGI
├──────────────────────
│ (mode = Identifications) ∨ (mode = Surgery)
│ ((∀ markidd : Boolean | markidd ∈ idflag' • markidd) ⇒
│  msgtogi!(DeleteMRIPanel) ∧
│  (markslice' = ⟨⟩) ∧
│  (((curview = Axial) ⇒ axdone') ∨
│   ((curview = Coronal) ⇒ cordone') ∨
│   ((curview = Sagittal) ⇒ sagdone'))))
└──────────────────────────────────────────
```

- 34: cancel_mark

```
── CancelView ─────────────────────────────
│ ΔOperatorDisplay
│ msgtogi!      :  MessageToGI
│
├──────────────────────
│ (mode = Identifications) ∨ (mode = Surgery)
│ markslice'   =   ⟨⟩
│ msgtogi!(DeleteMRIPanel)
│
└──────────────────────────────────────────
```

## Surgical Procedure Operations

- 41: locate_seed

*LocateSeed*

$\Xi \mathit{OperatorDisplay}$
$\mathit{msgtocp}! : \mathit{MessageToCP}$

---

$\mathit{mode} = \mathit{Surgery}$
$\mathit{msgtocp}!(\mathit{LocateSeed})$

- 60: move_seed

*MoveSeed*

$\Xi \mathit{OperatorDisplay}$
$\mathit{msgtocp}! : \mathit{MessageToCP}$

---

$\mathit{mode} = \mathit{Surgery}$
$\mathit{msgtocp}!(\mathit{MoveSeed})$

- 50: 51: set_slider_val

*SetSlider*

$\Delta \mathit{OperatorDisplay}$
$\mathit{distance}? : \mathbb{R}$
$\mathit{msgtogi}! : \mathit{MessageToGI}$

---

$\mathit{mode} = \mathit{Surgery}$
$\mathit{sliderval}' = \mathit{distance}?$
$\mathit{msgtogi}!(\mathit{SliderVal}, \mathit{distance}?)$

- 49: direction

*SetSendDirection*

$\Xi \mathit{OperatorDisplay}$
$\mathit{msgtocp}! : \mathit{MessageToCP}$
$\mathit{phi}?, \mathit{theta}? : \mathbb{R}$

---

$\mathit{mode} = \mathit{Surgery}$
$\mathit{msgtocp}!(\mathit{SetMotion}, \mathit{phi}?, \mathit{theta}?, \mathit{sliderval})$

$$
\begin{array}{lcl}
\mathit{ViewUpOp} & \widehat{=} & \mathit{ViewUp} \land \mathit{ViewUpFail} \\
\mathit{ViewDownOp} & \widehat{=} & \mathit{ViewDown} \land \mathit{ViewDownFail} \\
\mathit{ViewRightOp} & \widehat{=} & \mathit{ViewRight} \land \mathit{ViewRightFail} \\
\mathit{ViewLeftOp} & \widehat{=} & \mathit{ViewLeft} \land \mathit{ViewLeftFail} \\
\mathit{ViewFrontOp} & \widehat{=} & \mathit{ViewFront} \land \mathit{ViewFrontFail} \\
\mathit{ViewBackOp} & \widehat{=} & \mathit{ViewBack} \land \mathit{ViewBackFail}
\end{array}
$$

- 69: view_upp

*ViewUp*

$$
\begin{array}{|lcl}
\multicolumn{3}{|l}{\Delta \mathit{OperatorDisplay}} \\
\mathit{msgtogi!} & : & \mathit{MessageToGI} \\
\hline
\multicolumn{3}{|l}{(\mathit{mode} = \mathit{Identifications}) \lor (\mathit{mode} = \mathit{Surgery})} \\
\mathit{axslice} & < & \mathit{axelems} \\
\mathit{axslice'} & = & \mathit{axslice} + 1 \\
\multicolumn{3}{|l}{\mathit{msgtogi!}(\mathit{ViewUp})} \\
\hline
\end{array}
$$

- 70: view_down

*ViewDown*

$$
\begin{array}{|lcl}
\multicolumn{3}{|l}{\Delta \mathit{OperatorDisplay}} \\
\mathit{msgtogi!} & : & \mathit{MessageToGI} \\
\hline
\multicolumn{3}{|l}{(\mathit{mode} = \mathit{Identifications}) \lor (\mathit{mode} = \mathit{Surgery})} \\
\mathit{axslice} & > & 1 \\
\mathit{axslice'} & = & \mathit{axslice} - 1 \\
\multicolumn{3}{|l}{\mathit{msgtogi!}(\mathit{ViewDown})} \\
\hline
\end{array}
$$

- 71: view_right

*ViewRight*

$$
\begin{array}{|lcl}
\multicolumn{3}{|l}{\Delta \mathit{OperatorDisplay}} \\
\mathit{msgtogi!} & : & \mathit{MessageToGI} \\
\hline
\multicolumn{3}{|l}{(\mathit{mode} = \mathit{Identifications}) \lor (\mathit{mode} = \mathit{Surgery})} \\
\mathit{sagslice} & < & \mathit{sagelems} \\
\mathit{sagslice'} & = & \mathit{sagslice} + 1 \\
\multicolumn{3}{|l}{\mathit{msgtogi!}(\mathit{ViewRight})} \\
\hline
\end{array}
$$

• 72: view_left

*ViewLeft*

$\Delta \mathit{OperatorDisplay}$
$\mathit{msgtogi!} : \mathit{MessageToGI}$

---

$(\mathit{mode} = \mathit{Identifications}) \vee (\mathit{mode} = \mathit{Surgery})$
$\mathit{sagslice} > 1$
$\mathit{sagslice}' = \mathit{sagslice} - 1$
$\mathit{msgtogi!}(\mathit{ViewLeft})$

• 73: view_front

*ViewFront*

$\Delta \mathit{OperatorDisplay}$
$\mathit{msgtogi!} : \mathit{MessageToGI}$

---

$(\mathit{mode} = \mathit{Identifications}) \vee (\mathit{mode} = \mathit{Surgery})$
$\mathit{corslice} < \mathit{corelems}$
$\mathit{corslice}' = \mathit{corslice} + 1$
$\mathit{msgtogi!}(\mathit{ViewFront})$

• 74: view_back

*ViewBack*

$\Delta \mathit{OperatorDisplay}$
$\mathit{msgtogi!} : \mathit{MessageToGI}$

---

$(\mathit{mode} = \mathit{Identifications}) \vee (\mathit{mode} = \mathit{Surgery})$
$\mathit{corslice} > 1$
$\mathit{corslice}' = \mathit{corslice} - 1$
$\mathit{msgtogi!}(\mathit{ViewBack})$

*ViewUpFail*

$\Xi \mathit{OperatorDisplay}$

---

$\mathit{axslice} >= \mathit{axelems}$

*ViewDownFail*

$\Xi \mathit{OperatorDisplay}$

---

$\mathit{axslice} <= 1$

*ViewRightFail*

$\Xi OperatorDisplay$

---

$sagslice \;\; >= \;\; sagelems$

*ViewLeftFail*

$\Xi OperatorDisplay$

---

$sagslice \;\; <= \;\; 1$

*ViewFrontFail*

$\Xi OperatorDisplay$

---

$corslice \;\; >= \;\; corelems$

*ViewBackFail*

$\Xi OperatorDisplay$

---

$corslice \;\; <= \;\; 1$

- 52: 54: cp_c_arr

*CPCreateArrow*

$\Xi OperatorDisplay$

$featureid?, zval?, stat?, vis?, color?, width? : \mathbb{N}$

$lentoimg? : \mathbb{R}$

$UCSPos?, direction? : Position$

$msgtogi! : MessageToGI$

---

$mode = Surgery$

$msgtogi!(CreateArrow, featureid?, zval?, stat?, vis?, color?, width?, lentoimg?,$
$\quad UCSPos?, direction?)$

- 53: 55: cp_r_arr

*CPReorientArrow*

$\Xi OperatorDisplay$

$featureid? : \mathbb{N}$

$direction? : Position$

$msgtogi! : MessageToGI$

---

$mode = Surgery$

$msgtogi!(ReorientArrow, featureid?, direction?)$

- 61: 64: cp_slices

*CPDisplaySlices*

$\Xi OperatorDisplay$

$UCSposition? : Position$

$msgtogi! : MessageToGI$

---

$mode = Surgery$

$msgtogi!(DisplaySlices, UCSposition?)$

- 62: 65: cp_c_circ

*CPCreateCircle*

$\Xi OperatorDisplay$

$newid?, az?, status?, vis? : \mathbb{N}$

$color?, radius?, width?, fill? : \mathbb{N}$

$UCSpos? : Position$

$msgtogi! : MessageToGI$

---

$mode = Surgery$

$msgtogi!(CreateCircle, newid?, az?, status?, vis?, color?, radius?, width?,$
$\quad fill?, UCSpos?)$

- 63: 66: cp_m_feat

*CPMoveFeature*

$\Xi\, OperatorDisplay$
$featureid? : \mathbb{N}$
$UCSpos? : Position$
$msgtogi! : MessageToGI$

$mode = Surgery$
$msgtogi!(MoveFeature, featureid?, UCSpos?)$

- 77: inc_coil

*IncrementCoil*

$\Delta\, OperatorDisplay$
$whichcoil? : \mathbb{N}$
$msgtogi! : MessageToGI$

$mode = Surgery$
$reqcurrs(whichcoil?) = reqcurrs(whichcoil?) + 1$
$msgtogi!(SetReqCoil, reqcurrs(whichcoil?))$

- 78: dec_coil

*DecrementCoil*

$\Delta\, OperatorDisplay$
$whichcoil? : \mathbb{N}$
$msgtogi! : MessageToGI$

$mode = Surgery$
$reqcurrs(whichcoil?) = reqcurrs(whichcoil?) - 1$
$msgtogi!(SetReqCoil, reqcurrs(whichcoil?))$

- 79: set_coil

*SetCoil*

$\Delta\, OperatorDisplay$
$whichcoil? : \mathbb{N}$
$newval? : \mathbb{R}$
$msgtogi! : MessageToGI$

$mode = Surgery$
$reqcurrs(whichcoil?) = newval?$
$msgtogi!(SetReqCoil, newval?)$

- 81: curr_xray

```
── CurrentXray ──────────────────────
│ ΞOperatorDisplay
│ newxray?      :   XrayImage
│ axis?         :   Axis
│ msgtogi!      :   MessageToGI
├──────────────────────
│ mode = Surgery
│ msgtogi!(DisplayCurrentXray, axis?, newxray?)
│
└──────────────────────────────────────
```

- 80: bye_xray

```
── DismissXray ──────────────────────
│ ΞOperatorDisplay
│ msgtogi!      :   MessageToGI
├──────────────────────
│ mode = Surgery
│ msgtogi!(DismissXray)
└──────────────────────────────────────
```

- 83: quit

```
── QuitOD ──────────────────────
│ ΞOperatorDisplay
│ msgtocp!      :   MessageToCP
│ msgtogi!      :   MessageToGI
├──────────────────────
│ msgtocp!(Quit)
│ msgtogi!(ShutDown)
└──────────────────────────────────────
```

## A.2 Syntactic Specification

A context-free grammar (Backus-Naur Form) is used to specify the syntax of the user interface.

```
1:   mss            := set_up operations quit_seq
2:   set_up         := vis_calibrate VIS_PARAMETERS
3:   vis_calibrate  := acq_backgr acq_backgr calib_axis calib_axis
4:   acq_backgr     := ACQ_BACKGR
5:   calib_axis     := calib_cam calib_src
6:   calib_cam      := CALIB_CAMERA
7:   calib_src      := CALIB_SOURCE
8:   operations     := operations load_pat idents surg_proc close_pat
9:                   | load_pat idents surg_proc close_pat
10:  load_pat       := pat_data CP_PAT_DATA
11:  pat_data       := LOAD_PAT_DATA
12:  idents         := idents an_id
13:                  | an_id
14:  an_id          := x_obj_id VIS_ID_DONE
15:                  | m_mark_id MRI_ID_DONE
16:  x_obj_id       := x_obj_id init_obj_id id_objs
17:                  | init_obj_id id_objs
18:  init_obj_id    := init_token CP_ID_XRAY_OBJS
19:  init_token     := INIT_OBJ_ID
20:  id_objs        := id_objs obj_token
21:                  | obj_token
22:  obj_token      := select_obj OBJECT_POSIT
23:                  | DONE_AXIS_ID
24:                  | CANCEL_AXIS_ID
25:  select_obj     := select_obj SELECT_OBJ
26:                  | SELECT_OBJ
27:  m_mark_id      := m_mark_id init_mark_id id_marks
28:                  | init_mark_id id_marks
29:  init_mark_id   := INIT_MARK_ID
30:  id_marks       := id_marks mark_token
31:                  | mark_token
```

```
mark_token      := select_mark MARKER_POSIT
                |  DONE_VIEW_ID
                |  CANCEL_VIEW_ID
select_mark     := select_mark SELECT_MARK
                |  SELECT_MARK
surg_proc       := move_seed surg_actions
surg_actions    := surg_actions surg_act
                |  surg_act
surg_act        := set_dist_dir move_seed
                |  LOCATE_SEED_REQ
                |  x_obj_id VIS_ID_DONE
                |  m_mark_id MRI_ID_DONE
                |  disp_token
                |  view_xray
set_dist_dir    := set_dist_dir dist_dir_seq
                |  dist_dir_seq
dist_dir_seq    := set_slider dir_token cp_dist_resp opt_adjust
dir_token       := DIRECTION_INFO
set_slider      := set_slider SLIDER_VALUE
                |  SLIDER_VALUE
cp_dist_resp    := cp_dist_resp CP_CREAT_ARROW
                |  cp_dist_resp CP_REORNT_ARROW
                |  CP_CREAT_ARROW
                |  CP_REORNT_ARROW
opt_adjust      := adj_display
                |  adj_currents
                |
move_seed       := move_token cp_move_resp
move_token      := MOVE_SEED_REQ
cp_move_resp    := cp_move_resp CP_DISP_SLICES
                |  cp_move_resp CP_CREAT_CIRCLE
                |  cp_move_resp CP_MOV_FEATURE
                |  CP_DISP_SLICES
                |  CP_CREAT_CIRCLE
```

```
66:                  | CP_MOV_FEATURE
67:  adj_display     := adj_display disp_token
68:                  | disp_token
69:  disp_token      := VIEW_UPP
70:                  | VIEW_DWN
71:                  | VIEW_RT
72:                  | VIEW_LT
73:                  | VIEW_FRT
74:                  | VIEW_BCK
75:  adj_currents    := adj_currents curr_token
76:                  | curr_token
77:  curr_token      := INCR_COIL
78:                  | DECR_COIL
79:                  | RESET_COIL
80:  view_xray       := disp_xray BYE_XRAY
81:  disp_xray       := DISP_CUR_XRAY
82:  close_pat       := CLOSE_PAT_DATA
83:  quit_seq        := quit_confirm QUIT_O_D
84:  quit_confirm    := QUIT_CONFIRM
```

## A.3 Presentation Specification

The presentation component of the Operator Display is specified using Borland ObjectWindows. For a complete specification of the graphical display, in addition to the description found in this section, the documentation for the Borland ObjectWindows Library must be included. The definitions of base objects used in the specification may be found in the ObjectWindows documentation, in addition to definitions for the semantics of actions, such as button presses.

In addition to the textual Borland ObjectWindows specification, a visual presentation of the interface may be constructed by compilation with the ObjectWindows Library and subsequent execution. The specification is for presentation only; therefore no application functionality or dialogue control is included, and compilation would create simply a mock-up of the display.

The specification of the Operator Display consists of five sections, corresponding to the five files needed to describe the entire visual representation. The first section presents `med-sys.rc`, which is a resource file automatically generated from a graphical mock-up of a portion of the display. Graphical information for the definition of the menus and temporary button panels of the Operator Display is defined in this file.

The second section presents the `medsys.h` file. This file contains the primary class of the Operator Display, MedSysWindow. The MedSysWindow class contains declarations for the buttons, slider, three canvas (one for each view), and other graphical objects of the Operator Display.

The next section of the presentation specification is for the `medsys.cpp` file. In this section is defined the constructor for the MedSysWindow class, which in turn instantiates the graphical objects declared in the class definition found in the previous section. In addition, the Paint function for this class displays a status string for the Operator Display and a box for input of seed direction values.

The fourth section contains the file `bmpview.h`, which defines the TCanvas class for displaying bit-mapped images. The bit-mapped images can display either the pre-operative MRI images or x-ray images.

The final section presents the `bmpview.cpp` file, which contains the constructor, destructor, and Paint function for the TCanvas class. The Paint function creates lines on the images for representation of the electromagnetic coil positions, a label identifying each view, and positions to display the actual coil current values.

## medsys.rc

```
/***************************************************************************


medsys.rc

produced by Borland Resource Workshop


***************************************************************************/

#include "medsys.rh"

MEDSYS_MENU MENU
{
  POPUP "Patient"
  {
    MENUITEM "Load Patient Data",                CM_PATIENT_LOADDATA
    MENUITEM "Close Patient Data",               CM_PATIENT_CLOSEDATA
  }

  POPUP "MRI"
  {
    MENUITEM "Axial",                            CM_MRI_AXIAL
    MENUITEM "Sagittal",                         CM_MRI_SAGITTAL
    MENUITEM "Coronal",                          CM_MRI_CORONAL
    MENUITEM "Done",                             CM_MRI_DONE
  }

  POPUP "Vision"
  {
    MENUITEM "Acquire X Background",             CM_VISION_XBACKGROUND
    MENUITEM "Acquire Y Background",             CM_VISION_YBACKGROUND
    MENUITEM SEPARATOR
    MENUITEM "X Camera",                         CM_VISION_XCAMERA
    MENUITEM "Y Camera",                         CM_VISION_YCAMERA
    MENUITEM SEPARATOR
    MENUITEM "X Source",                         CM_VISION_XSOURCE
    MENUITEM "Y Source",                         CM_VISION_YSOURCE
    MENUITEM SEPARATOR
    MENUITEM "X Objects",                        CM_VISION_XOBJECTS
    MENUITEM "Y Objects",                        CM_VISION_YOBJECTS
    MENUITEM SEPARATOR
    MENUITEM "Current X",                        CM_VISION_CURRENTX
    MENUITEM "Current Y",                        CM_VISION_CURRENTY
    MENUITEM SEPARATOR
    MENUITEM "Image Parameters",                 CM_VISION_IMAGEPARAMETERS
    MENUITEM "Done",                             CM_VISION_DONE
  }

  POPUP "Quit"
  {
    MENUITEM "Quit",                             CM_QUIT
  }
}

OBJECTS_DIALOG DIALOG                            OBJ_DIALOG_X,
                                                 OBJ_DIALOG_Y,
                                                 OBJ_DIALOG_WIDTH,
                                                 OBJ_DIALOG_HEIGHT
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Objects on X Object Axis"
FONT 8, "MS Sans Serif"
{
  DEFPUSHBUTTON "OK",                            IDOK,
                                                 OK_BUTTON_X,
                                                 OK_BUTTON_Y,
                                                 OKCANCEL_BUTTON_WIDTH,
                                                 OKCANCEL_BUTTON_HEIGHT

  PUSHBUTTON "Cancel",                           IDCANCEL,
                                                 CANCEL_BUTTON_X,
                                                 CANCEL_BUTTON_Y,
                                                 OKCANCEL_BUTTON_WIDTH,
                                                 OKCANCEL_BUTTON_HEIGHT

  PUSHBUTTON "Seed",                             IDC_SEED,
                                                 BUTTON_X,
                                                 SEED_BUTTON_Y,
                                                 DG_BUTTON_WIDTH,
                                                 DG_BUTTON_HEIGHT

  PUSHBUTTON "Left Temporal",                    IDC_LEFT_TEMPORAL,
                                                 BUTTON_X,
                                                 LEFT_TEMP_BUTTON_Y,
                                                 DG_BUTTON_WIDTH,
                                                 DG_BUTTON_HEIGHT

  PUSHBUTTON "Coronal",                          IDC_CORONAL,
                                                 BUTTON_X,
                                                 CORONAL_BUTTON_Y,
                                                 DG_BUTTON_WIDTH,
                                                 DG_BUTTON_HEIGHT

  PUSHBUTTON "Right Temporal",                   IDC_RIGHT_TEMPORAL,
                                                 BUTTON_X,
                                                 RIGHT_TEMP_BUTTON_Y,
                                                 DG_BUTTON_WIDTH,
                                                 DG_BUTTON_HEIGHT

  PUSHBUTTON "New Image",                        IDC_NEW_IMAGE,
                                                 BUTTON_X,
                                                 NEW_IMAGE_BUTTON_Y,
                                                 DG_BUTTON_WIDTH,
                                                 DG_BUTTON_HEIGHT

  GROUPBOX "",                                   IDC_GROUP,
                                                 GROUP_X,
                                                 GROUP_Y,
                                                 GROUP_WIDTH,
                                                 GROUP_HEIGHT,
                                                 BS_GROUPBOX

  CONTROL "Use Identified Positions",            IDC_USE_ID_OBJECTS,
                                                 "BUTTON",
                                                 BS_AUTORADIOBUTTON,
                                                  RADIO_X,
                                                  OBJ_RADIO_Y,
                                                  RADIO_WIDTH,
                                                  RADIO_HEIGHT

  CONTROL "Locate Objects",                       IDC_LOC_OBJECTS,
                                                  "BUTTON",
                                                  BS_AUTORADIOBUTTON,
                                                  RADIO_X,
                                                  LOC_RADIO_Y,
                                                  RADIO_WIDTH,
                                                  RADIO_HEIGHT
}

MARKERS_DIALOG DIALOG                             OBJ_DIALOG_X,
                                                  OBJ_DIALOG_Y,
                                                  OBJ_DIALOG_WIDTH,
                                                  OBJ_DIALOG_HEIGHT
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Objects on X Object Axis"
FONT 8, "MS Sans Serif"
{
  DEFPUSHBUTTON "OK",                             IDOK,
                                                  OK_BUTTON_X,
                                                  OK_BUTTON_Y,
                                                  OKCANCEL_BUTTON_WIDTH,
                                                  OKCANCEL_BUTTON_HEIGHT

  PUSHBUTTON "Cancel",                            IDCANCEL,
                                                  CANCEL_BUTTON_X,
                                                  CANCEL_BUTTON_Y,
                                                  OKCANCEL_BUTTON_WIDTH,
                                                  OKCANCEL_BUTTON_HEIGHT

  PUSHBUTTON "Left Temporal",                     IDC_MARK_LEFT_TEMPORAL,
                                                  BUTTON_X,
                                                  LEFT_TEMP_BUTTON_Y,
                                                  DG_BUTTON_WIDTH,
                                                  DG_BUTTON_HEIGHT

  PUSHBUTTON "Coronal",                           IDC_MARK_CORONAL,
                                                  BUTTON_X,
                                                  CORONAL_BUTTON_Y,
                                                  DG_BUTTON_WIDTH,
                                                  DG_BUTTON_HEIGHT

  PUSHBUTTON "Right Temporal",                    IDC_MARK_RIGHT_TEMPORAL,
                                                  BUTTON_X,
                                                  RIGHT_TEMP_BUTTON_Y,
                                                  DG_BUTTON_WIDTH,
                                                  DG_BUTTON_HEIGHT

  GROUPBOX "",                                    IDC_MARK_GROUP,
                                                  GROUP_X,
                                                  GROUP_Y,
                                                  GROUP_WIDTH,
                                                  GROUP_HEIGHT,
                                                  BS_GROUPBOX
  CONTROL "Use Identified Positions",             IDC_MARK_USE_ID_OBJECTS,
                                                  "BUTTON",
                                                  BS_AUTORADIOBUTTON,
                                                  RADIO_X,
                                                  OBJ_RADIO_Y,
                                                  RADIO_WIDTH,
                                                  RADIO_HEIGHT

  CONTROL "Locate Objects",                       IDC_MARK_LOC_OBJECTS,
                                                  "BUTTON",
                                                  BS_AUTORADIOBUTTON,
                                                  RADIO_X,
                                                  LOC_RADIO_Y,
                                                  RADIO_WIDTH,
                                                  RADIO_HEIGHT
}

QUIT_DIALOG DIALOG                                QUIT_DIALOG_X,
                                                  QUIT_DIALOG_Y,
                                                  QUIT_DIALOG_WIDTH,
                                                  QUIT_DIALOG_HEIGHT
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Quit?"
FONT 8, "MS Sans Serif"
{
  DEFPUSHBUTTON "Yes",                            IDOK,
                                                  YES_BUTTON_X,
                                                  YES_BUTTON_Y,
                                                  QUIT_BUTTON_WIDTH,
                                                  QUIT_BUTTON_HEIGHT
  PUSHBUTTON "No",                                IDCANCEL,
                                                  NO_BUTTON_X,
                                                  NO_BUTTON_Y,
                                                  QUIT_BUTTON_WIDTH,
                                                  QUIT_BUTTON_HEIGHT
  LTEXT "Are you sure you want to quit?",         -1,
                                                  TEXT_X,
                                                  TEXT_Y,
                                                  TEXT_WIDTH,
                                                  TEXT_HEIGHT
}
```

## medsys.h

```
#include <owl/applicat.h>
#include <owl/owlpch.h>
#include <owl/framewin.h>

#include "medsys.rh"
#include <owl/static.h>
#include "bmpview.h"


const int ID_AXIAL_UP                = 801;
const int ID_AXIAL_DOWN              = 802;
const int ID_SAGITTAL_RIGHT          = 803;
const int ID_SAGITTAL_LEFT           = 804;
const int ID_CORONAL_FRONT           = 805;
const int ID_CORONAL_BACK            = 806;

const int ID_MOVE_SEED               = 823;
const int ID_LOC_SEED                = 824;
const int ID_HOZSLIDER               = 825;
const int ID_DISMISS_XRAY            = 826;


const int NUM_AXIAL_FILES            = 2;
const int NUM_SAGITTAL_FILES         = 2;
const int NUM_CORONAL_FILES          = 2;

const int BIG_PANE_SIZE              = 512;
const int LITTLE_PANE_SIZE           = 256;
const int MEDSYS_ORIGIN              = 0;

const int AXIAL_VIEW_LEFT            = MEDSYS_ORIGIN;
const int AXIAL_VIEW_TOP             = MEDSYS_ORIGIN;

const int SAGITTAL_VIEW_LEFT         = 600;
const int SAGITTAL_VIEW_TOP          = MEDSYS_ORIGIN;

const int CORONAL_VIEW_LEFT          = MEDSYS_ORIGIN;
const int CORONAL_VIEW_TOP           = 550;

const int ANGLEBOX_LEFT              = 400;
const int ANGLEBOX_TOP               = 600;
const int ANGLEBOX_RIGHT             = 760;
const int ANGLEBOX_BOTTOM            = 780;

const int BIG_PANE_BUTTON_WIDTH      = BIG_PANE_SIZE / 2;
const int LITTLE_PANE_BUTTON_WIDTH   = LITTLE_PANE_SIZE / 2;
const int VIEW_BUTTON_HEIGHT         = CORONAL_VIEW_TOP - BIG_PANE_SIZE;

const int AX_UP_BUTTON_LEFT          = MEDSYS_ORIGIN;
const int AX_UP_BUTTON_TOP           = BIG_PANE_SIZE;
const int AX_DN_BUTTON_LEFT          = BIG_PANE_BUTTON_WIDTH;
const int AX_DN_BUTTON_TOP           = BIG_PANE_SIZE;
const int SG_RT_BUTTON_LEFT          = SAGITTAL_VIEW_LEFT;
const int SG_RT_BUTTON_TOP           = BIG_PANE_SIZE;
const int SG_LT_BUTTON_LEFT          = SAGITTAL_VIEW_LEFT +
                                       BIG_PANE_BUTTON_WIDTH;
```

```
const int SG_LT_BUTTON_TOP          = BIG_PANE_SIZE;
const int CO_FT_BUTTON_LEFT         = MEDSYS_ORIGIN;
const int CO_FT_BUTTON_TOP          = LITTLE_PANE_SIZE + CORONAL_VIEW_TOP;
const int CO_BK_BUTTON_LEFT         = LITTLE_PANE_BUTTON_WIDTH;
const int CO_BK_BUTTON_TOP          = LITTLE_PANE_SIZE + CORONAL_VIEW_TOP;

const int MOVE_SEED_LEFT            = 790;
const int MOVE_SEED_TOP             = 600;
const int SEED_OFFSET               = 10;
const int SEED_BUTTON_WIDTH         = 180;
const int SEED_BUTTON_HEIGHT        = 30;
const int LOC_SEED_LEFT             = MOVE_SEED_LEFT;
const int LOC_SEED_TOP              = MOVE_SEED_TOP + SEED_BUTTON_HEIGHT +
                                      SEED_OFFSET;

const int STATUS_LABEL_LEFT         = MOVE_SEED_LEFT;
const int STATUS_LABEL_TOP          = LOC_SEED_TOP + SEED_BUTTON_HEIGHT +
                                      SEED_OFFSET;
const int STATUS_LABEL_RIGHT        = MOVE_SEED_LEFT + 500;
const int STATUS_LABEL_BOTTOM       = STATUS_LABEL_TOP + 30;

const int SLIDER_LEFT               = ANGLEBOX_LEFT;
const int SLIDER_TOP                = ANGLEBOX_BOTTOM + 30;
const int SLIDER_WIDTH              = ANGLEBOX_RIGHT - ANGLEBOX_LEFT;
const int SLIDER_HEIGHT             = 40;

const int COPYRIGHT_LEFT            = 0;
const int COPYRIGHT_TOP             = SLIDER_TOP + SLIDER_HEIGHT + 20;
const int COPYRIGHT_WIDTH           = 900;
const int COPYRIGHT_HEIGHT          = 30;

const int SLIDER_MIN                = 0;
const int SLIDER_MAX                = 400;
const int SLIDER_PERCENTILE         = 5;
const int SLIDER_POS                = 0;

const int BOX_LABEL_WIDTH           = 30;
const int BOX_LABEL_HEIGHT          = 20;


class MedSysWindow : public TWindow
{
  public:
     MedSysWindow();
     ~MedSysWindow();
     void          Paint(TDC&, BOOL, TRect&);
     void          CmVisionXObjects();
     void          CmVisionCurrentX();
     void          CmVisionYObjects();
     void          CmQuit();
     void          DismissXRay();
  private:
     TCanvas*      axial_view;
     TCanvas*      sagittal_view;
     TCanvas*      coronal_view;
     TRect*        angle_box;

     TButton*      axial_up_button;
     TButton*      axial_down_button;
```

```
    TButton*      sagittal_right_button;
    TButton*      sagittal_left_button;
    TButton*      coronal_front_button;
    TButton*      coronal_back_button;
    TButton*      move_seed_button;
    TButton*      locate_seed_button;
    TButton*      dismiss_Xray_button;

    THSlider*     hoz_slider;
    TStatic*      copyright_label;
    MagCoil       magcoil;

    TDib**        axial_files;
    TDib**        sagittal_files;
    TDib**        coronal_files;


  DECLARE_RESPONSE_TABLE(MedSysWindow);
};

class MedSysApp : public TApplication
{
  public:
     MedSysApp() : TApplication() {}
     void InitMainWindow();
};
```

## medsys.cpp

```
#include "medsys.h"

DEFINE_RESPONSE_TABLE1(MedSysWindow, TWindow)
  EV_WM_PAINT,
  EV_COMMAND(CM_VISION_XOBJECTS, CmVisionXObjects),
  EV_COMMAND(CM_VISION_CURRENTX, CmVisionCurrentX),
  EV_COMMAND(CM_VISION_YOBJECTS, CmVisionYObjects),
  EV_COMMAND(CM_QUIT, CmQuit),
  EV_COMMAND(ID_DISMISS_XRAY, DismissXRay),
END_RESPONSE_TABLE;

MedSysWindow::MedSysWindow() :
  TWindow(0,0,0)
{
  Attr.Style              |= WS_BORDER | WS_MAXIMIZE;

  magcoil.XA              = 0.0;
  magcoil.XB              = 0.0;
  magcoil.YA              = 0.0;
  magcoil.YB              = 0.0;
  magcoil.ZA              = 0.0;
  magcoil.ZB              = 0.0;

  axial_files             = new TDib*[NUM_AXIAL_FILES];
  axial_files[0]          = new TDib("j:an024.bmp");
  sagittal_files          = new TDib*[NUM_SAGITTAL_FILES];
  sagittal_files[0]       = new TDib("j:sn075.bmp");
  coronal_files           = new TDib*[NUM_CORONAL_FILES];
  coronal_files[0]        = new TDib("j:cn019.bmp");

  axial_view              = new TCanvas(this, AXIAL, AXIAL_VIEW_LEFT,
                                        AXIAL_VIEW_TOP, BIG_PANE_SIZE,
                                        BIG_PANE_SIZE, &magcoil,
                                        axial_files[0]);


  sagittal_view           = new TCanvas(this, SAGITTAL, SAGITTAL_VIEW_LEFT,
                                        SAGITTAL_VIEW_TOP, BIG_PANE_SIZE,
                                        BIG_PANE_SIZE, &magcoil,
                                        sagittal_files[0]);

  coronal_view            = new TCanvas(this, CORONAL, CORONAL_VIEW_LEFT,
                                        CORONAL_VIEW_TOP, LITTLE_PANE_SIZE,
                                        LITTLE_PANE_SIZE, &magcoil,
                                        coronal_files[0]);

  angle_box               = new TRect(ANGLEBOX_LEFT,
                                      ANGLEBOX_TOP,
                                      ANGLEBOX_RIGHT,
                                      ANGLEBOX_BOTTOM);


  axial_up_button         = new TButton(this, ID_AXIAL_UP, "Up",
                                        AX_UP_BUTTON_LEFT,
                                        AX_UP_BUTTON_TOP,
                                        BIG_PANE_BUTTON_WIDTH,
                                   VIEW_BUTTON_HEIGHT);

axial_down_button          = new TButton(this, ID_AXIAL_DOWN, "Down",
                                   AX_DN_BUTTON_LEFT,
                                   AX_DN_BUTTON_TOP,
                                   BIG_PANE_BUTTON_WIDTH,
                                   VIEW_BUTTON_HEIGHT);

sagittal_right_button      = new TButton(this, ID_SAGITTAL_RIGHT, "Right",
                                   SG_RT_BUTTON_LEFT,
                                   SG_RT_BUTTON_TOP,
                                   BIG_PANE_BUTTON_WIDTH,
                                   VIEW_BUTTON_HEIGHT);

sagittal_left_button       = new TButton(this, ID_SAGITTAL_LEFT, "Left",
                                   SG_LT_BUTTON_LEFT,
                                   SG_LT_BUTTON_TOP,
                                   BIG_PANE_BUTTON_WIDTH,
                                   VIEW_BUTTON_HEIGHT);

coronal_front_button       = new TButton(this, ID_CORONAL_FRONT, "Front",
                                   CO_FT_BUTTON_LEFT,
                                   CO_FT_BUTTON_TOP,
                                   LITTLE_PANE_BUTTON_WIDTH,
                                   VIEW_BUTTON_HEIGHT);

coronal_back_button        = new TButton(this, ID_CORONAL_BACK, "Back",
                                   CO_BK_BUTTON_LEFT,
                                   CO_BK_BUTTON_TOP,
                                   LITTLE_PANE_BUTTON_WIDTH,
                                   VIEW_BUTTON_HEIGHT);


move_seed_button           = new TButton(this, ID_MOVE_SEED, "Move Seed",
                                   MOVE_SEED_LEFT,
                                   MOVE_SEED_TOP,
                                   SEED_BUTTON_WIDTH,
                                   SEED_BUTTON_HEIGHT);

locate_seed_button         = new TButton(this, ID_LOC_SEED, "Locate Seed",
                                   LOC_SEED_LEFT,
                                   LOC_SEED_TOP,
                                   SEED_BUTTON_WIDTH,
                                   SEED_BUTTON_HEIGHT);

hoz_slider                 = new THSlider(this, ID_HOZSLIDER,
                                    SLIDER_LEFT,
                                    SLIDER_TOP,
                                    SLIDER_WIDTH,
                                    SLIDER_HEIGHT);

hoz_slider->SetRange(SLIDER_MIN, SLIDER_MAX);
hoz_slider->SetRuler(SLIDER_PERCENTILE, FALSE);
hoz_slider->SetPosition(SLIDER_POS);

copyright_label            = new TStatic(this, -1,
                           "(C) 1995 University of Virginia Board of Visitors",
                                    COPYRIGHT_LEFT,
                                    COPYRIGHT_TOP,
                                              COPYRIGHT_WIDTH,
                                              COPYRIGHT_HEIGHT);
}

MedSysWindow::~MedSysWindow()
{
  delete axial_view;
  delete sagittal_view;
  delete coronal_view;
  delete angle_box;
  delete axial_up_button;
  delete axial_down_button;
  delete sagittal_right_button;
  delete sagittal_left_button;
  delete coronal_front_button;
  delete coronal_back_button;
  delete move_seed_button;
  delete locate_seed_button;
  delete hoz_slider;
}

void MedSysWindow::Paint(TDC& dc, BOOL, TRect&)
{
  dc.FillRect(*angle_box, TBrush(TColor::LtRed));
  TRect text_region(0,0,0,0);

  dc.SelectObject(TPen(TColor::Black));
  text_region.Set(STATUS_LABEL_LEFT,
                  STATUS_LABEL_TOP,
                  STATUS_LABEL_RIGHT,
                  STATUS_LABEL_BOTTOM);
  dc.DrawText("Operator Display Ready", -1, text_region, DT_LEFT);
  text_region.Set(ANGLEBOX_LEFT - BOX_LABEL_WIDTH,
                  ANGLEBOX_TOP,
                  ANGLEBOX_LEFT,
                  ANGLEBOX_TOP + BOX_LABEL_HEIGHT);
  dc.DrawText("180",-1,text_region, DT_CENTER);
  text_region.Set(ANGLEBOX_LEFT - BOX_LABEL_WIDTH,
                  ANGLEBOX_BOTTOM - BOX_LABEL_HEIGHT,
                  ANGLEBOX_LEFT,
                  ANGLEBOX_BOTTOM);
  dc.DrawText("0",-1,text_region, DT_CENTER);
  text_region.Set(ANGLEBOX_RIGHT,
                  ANGLEBOX_BOTTOM - BOX_LABEL_HEIGHT,
                  ANGLEBOX_RIGHT + BOX_LABEL_WIDTH,
                  ANGLEBOX_BOTTOM);
  dc.DrawText("360",-1,text_region, DT_CENTER);
}


void MedSysWindow::CmVisionXObjects()
{
  TDialog* object_dialog = new TDialog(this, OBJECTS_DIALOG);
  object_dialog->Create();
}

void MedSysWindow::CmVisionYObjects()
{
  TDialog* marker_dialog = new TDialog(this, MARKERS_DIALOG);
  marker_dialog->Create();
}

void MedSysWindow::CmVisionCurrentX()
{
  dismiss_Xray_button = new TButton(this, ID_DISMISS_XRAY,
                                    "Dismiss",
                                    BIG_PANE_SIZE,
                                    BIG_PANE_SIZE,
                                    SAGITTAL_LEFT - BIG_PANE_SIZE,
                                    VIEW_BUTTON_HEIGHT);
  dismiss_Xray_button->Create();
}

void MedSysWindow::DismissXRay()
{
  delete dismiss_Xray_button;
}

void MedSysWindow::CmQuit()
{
  TDialog* quit_dialog = new TDialog(this, QUIT_DIALOG);
  int retval = quit_dialog->Execute();

  if (retval == IDOK) {
    CloseWindow();
  }
}


void MedSysApp::InitMainWindow()
{
  MainWindow = new TFrameWindow(0,"Operator Display",new MedSysWindow());
  MainWindow->AssignMenu(MEDSYS_MENU);
}

int OwlMain(int, char* [])
{
  MedSysApp MyMedSysApp;
  return MyMedSysApp.Run();
}
```

## bmpview.h

```
#ifndef _BMPVIEW
#define _BMPVIEW

#include <owl\owlpch.h>
#include <owl\applicat.h>
#include <owl\dc.h>
#include <owl\gdiobjec.h>
#include <owl\chooseco.h>
#include <owl\choosefo.h>
#include <owl\opensave.h>
#include <owl\editfile.rh>
#include <owl\point.h>
#include <stdio.h>
#include <dir.h>


const int LABEL_HEIGHT                          = 20;
const int LABEL_WIDTH                           = 100;
const int LABEL_OFFSET                          = 5;
const int AREA_OFFSET                           = 10;

const int XA_COIL_TOP                           = 0,
          XA_COIL_BOTTOM                        = 1,
          YA_COIL_TOP                           = 2,
          YA_COIL_BOTTOM                        = 3,
          YB_COIL_TOP                           = 4,
          YB_COIL_BOTTOM                        = 5,
          XB_COIL_TOP                           = 6,
          XB_COIL_BOTTOM                        = 7,
          ZA_COIL_LEFT                          = 8,
          ZA_COIL_RIGHT                         = 9,
          ZB_COIL_LEFT                          = 10,
          ZB_COIL_RIGHT                         = 11;

const int NUM_COILS                             = 6;
const int NUM_REGION_POINTS                     = 6;
const int FONT_SIZE                             = 15;

typedef enum ViewType {AXIAL, SAGITTAL, CORONAL};

struct MagCoil {
  float XA;
  float XB;
  float YA;
  float YB;
  float ZA;
  float ZB;
};

class TCanvas : public TWindow
{
  public:
    TCanvas(TWindow* parent, ViewType new_type,
            int X, int Y, int W, int H,
            MagCoil* magcoil, TDib* this_file);
    ~TCanvas();
  protected:
    void          Paint(TDC&, BOOL, TRect&);
    void          EvSize(UINT, TSize&);
    BOOL          EvEraseBkgnd(HDC hDC);
    void          EvMouseMove(UINT modKeys, TPoint& point);
    void          EvSysColorChange();

  private:
    BOOL          Drawing;
    TBrush*       BkBrush;
    ViewType      view_type;
    TDib*         current_file;
    TPoint*       endpoints;
    MagCoil*      my_mag_coil;

  DECLARE_RESPONSE_TABLE(TCanvas);
};

#endif
```

## bmpview.cpp

```
//--------------------------------------------------------------------------
// ObjectWindows - (C) Copyright 1991, 1993 by Borland International
//   Example paint program
//--------------------------------------------------------------------------

#include "bmpview.h"



DEFINE_RESPONSE_TABLE1(TCanvas, TWindow)
  EV_WM_ERASEBKGND,
  EV_WM_SIZE,
  EV_WM_DROPFILES,
  EV_WM_SYSCOLORCHANGE,
END_RESPONSE_TABLE;


TCanvas::TCanvas(TWindow* parent,
                 ViewType new_view,
                 int X, int Y, int W, int H,
                 MagCoil* magcoil,
                 TDib* this_file) :
  TWindow(parent,0,0),
  Drawing(FALSE)
{
  BkBrush = 0;
  EvSysColorChange();

  view_type                   =  new_view;
  current_file                =  this_file;
  my_mag_coil                 =  magcoil;
  Attr.X                      =  X;
  Attr.Y                      =  Y;
  Attr.W                      =  W;
  Attr.H                      =  H;

  endpoints                   =  new TPoint[NUM_COILS*2];
  endpoints[XA_COIL_TOP]      =  TPoint(Attr.W/3,Attr.H/8);
  endpoints[XA_COIL_BOTTOM]   =  TPoint(Attr.W/8,Attr.H/3);
  endpoints[YA_COIL_TOP]      =  TPoint(2*Attr.W/3, Attr.H/8);
  endpoints[YA_COIL_BOTTOM]   =  TPoint(7*Attr.W/8, Attr.H/3);
  endpoints[YB_COIL_TOP]      =  TPoint(Attr.W/8, 2*Attr.H/3);
  endpoints[YB_COIL_BOTTOM]   =  TPoint(Attr.W/3, 7*Attr.H/8);
  endpoints[XB_COIL_TOP]      =  TPoint(7*Attr.W/8, 2*Attr.H/3);
  endpoints[XB_COIL_BOTTOM]   =  TPoint(2*Attr.W/3, 7*Attr.H/8);
  endpoints[ZA_COIL_LEFT]     =  TPoint(Attr.W/3, Attr.H/8);
  endpoints[ZA_COIL_RIGHT]    =  TPoint(2*Attr.W/3, Attr.H/8);
  endpoints[ZB_COIL_LEFT]     =  TPoint(Attr.W/3, 7*Attr.H/8);
  endpoints[ZB_COIL_RIGHT]    =  TPoint(2*Attr.W/3, 7*Attr.H/8);
}

TCanvas::~TCanvas()
{
  delete BkBrush;
  delete endpoints;
}
```

```
void TCanvas::EvSize(UINT SizeType, TSize& Size)
{
  TWindow::EvSize(SizeType, Size);
}

BOOL TCanvas::EvEraseBkgnd(HDC)
{
  return TRUE;
}

void TCanvas::EvSysColorChange()
{
  delete BkBrush;
  BkBrush = new TBrush(::GetSysColor(COLOR_APPWORKSPACE)); //COLOR_WINDOW
}


void TCanvas::Paint(TDC& dc, BOOL, TRect&)
{
  TRect drect(TPoint(0,0), current_file->Size());
  TRect crect = GetClientRect();

  TPalette* current_palette = new TPalette(*current_file);
  dc.SelectObject(*current_palette);
  dc.RealizePalette();

  dc.SetDIBitsToDevice(drect, TPoint(0,0), *current_file);

  dc.SelectObject(*BkBrush);
  dc.PatBlt(TRect(drect.right, 0, crect.right, crect.bottom));
  dc.PatBlt(TRect(0, drect.bottom, crect.right, crect.bottom));

  dc.SelectObject(TPen(TColor::LtRed));
  delete current_palette;

  if (view_type == AXIAL) {
    dc.MoveTo(endpoints[XA_COIL_TOP]);
    dc.LineTo(endpoints[XA_COIL_BOTTOM]);
    dc.MoveTo(endpoints[YA_COIL_TOP]);
    dc.LineTo(endpoints[YA_COIL_BOTTOM]);
    dc.MoveTo(endpoints[YB_COIL_TOP]);
    dc.LineTo(endpoints[YB_COIL_BOTTOM]);
    dc.MoveTo(endpoints[XB_COIL_TOP]);
    dc.LineTo(endpoints[XB_COIL_BOTTOM]);
  } else {
    dc.MoveTo(endpoints[ZA_COIL_LEFT]);
    dc.LineTo(endpoints[ZA_COIL_RIGHT]);
    dc.MoveTo(endpoints[ZB_COIL_LEFT]);
    dc.LineTo(endpoints[ZB_COIL_RIGHT]);
  }

  dc.SetBkColor(TColor::White);
  dc.SelectObject(TFont("Courier New",FONT_SIZE));
  TRect textreg(0,0,0,0);

  char* my_buffer = new char[20];
  if (view_type == AXIAL) {
    textreg.Set(0, 0, LABEL_WIDTH, LABEL_HEIGHT);
  sprintf(my_buffer, "XA Act = %g", my_mag_coil->XA);
  dc.DrawText(my_buffer,-1,textreg, DT_CENTER);

  textreg.Set(LABEL_WIDTH, 0, 2*LABEL_WIDTH, LABEL_HEIGHT);
  sprintf(my_buffer, "XA Req = %g", my_mag_coil->XA);
  dc.DrawText(my_buffer,-1,textreg, DT_CENTER);

  textreg.Set(Attr.W - 2*LABEL_WIDTH, 0, Attr.W - LABEL_WIDTH, LABEL_HEIGHT);
  sprintf(my_buffer, "YA Act = %g", my_mag_coil->YA);
  dc.DrawText(my_buffer,-1,textreg, DT_CENTER);

  textreg.Set(Attr.W - LABEL_WIDTH, 0, Attr.W , LABEL_HEIGHT);
  sprintf(my_buffer, "YA Req = %g", my_mag_coil->YA);
  dc.DrawText(my_buffer,-1,textreg, DT_CENTER);

  textreg.Set(0, Attr.H - LABEL_HEIGHT, LABEL_WIDTH, Attr.H);
  sprintf(my_buffer, "YB Act = %g", my_mag_coil->YB);
  dc.DrawText(my_buffer,-1,textreg, DT_CENTER);

  textreg.Set(LABEL_WIDTH, Attr.H - LABEL_HEIGHT, 2*LABEL_WIDTH, Attr.H);
  sprintf(my_buffer, "YB Req = %g", my_mag_coil->YB);
  dc.DrawText(my_buffer,-1,textreg, DT_CENTER);

  textreg.Set(Attr.W - 2*LABEL_WIDTH, Attr.H - LABEL_HEIGHT,
              Attr.W - LABEL_WIDTH, Attr.H);
  sprintf(my_buffer, "XB Act = %g", my_mag_coil->XB);
  dc.DrawText(my_buffer,-1,textreg, DT_CENTER);

  textreg.Set(Attr.W - LABEL_WIDTH, Attr.H - LABEL_HEIGHT, Attr.W, Attr.H);
  sprintf(my_buffer, "XB Req = %g", my_mag_coil->XB);
  dc.DrawText(my_buffer,-1,textreg, DT_CENTER);

  delete my_buffer;

} else {

  textreg.Set(0, 0, LABEL_WIDTH, LABEL_HEIGHT);
  sprintf(my_buffer, "ZA Act = %g", my_mag_coil->ZA);
  dc.DrawText(my_buffer,-1,textreg, DT_CENTER);

  textreg.Set(Attr.W - LABEL_WIDTH, 0, Attr.W, LABEL_HEIGHT);
  sprintf(my_buffer, "ZA Req = %g", my_mag_coil->ZA);
  dc.DrawText(my_buffer,-1,textreg, DT_CENTER);

  textreg.Set(0, Attr.H - LABEL_HEIGHT, LABEL_WIDTH, Attr.H);
  sprintf(my_buffer, "ZB Act = %g", my_mag_coil->ZB);
  dc.DrawText(my_buffer,-1,textreg, DT_CENTER);

  textreg.Set(Attr.W - LABEL_WIDTH, Attr.H - LABEL_HEIGHT, Attr.W, Attr.H);
  sprintf(my_buffer, "ZB Req = %g", my_mag_coil->ZB);
  dc.DrawText(my_buffer,-1,textreg, DT_CENTER);

  delete my_buffer;
}

textreg.Set(Attr.W/2 - LABEL_WIDTH/2, Attr.H - LABEL_HEIGHT,
            Attr.W/2 + LABEL_WIDTH/2, Attr.H);

switch (view_type) {
   case AXIAL:
     dc.DrawText("AXIAL VIEW",-1,textreg,DT_CENTER);
     break;
   case SAGITTAL:
     dc.DrawText("SAGITTAL VIEW", -1, textreg, DT_CENTER);
     break;
   case CORONAL:
     dc.DrawText("CORONAL VIEW", -1, textreg, DT_CENTER);
     break;
 }

 dc.RestoreObjects();
}
```

## A.4 Specification of Token Generation

Tokens can be generated by either the presentation or the application program. The tokens that the presentation produces map that portion of the specification to the context-free grammar. Typically, for tokens generated by the presentation it is the graphical object's callback function which generates the token. An enumeration of the tokens and the objects which generate each of those tokens specifies the token generation.

Table 2 relates each of the 42 tokens of the context-free grammar to either the application program or objects from the presentation specification. The callback function of the graphical object is assumed to generate presentation tokens; in cases where this is not the case, the particular method is specified with the object. The tokens denoted "Control Program" signify that they are generated by the application program.

**Table 2: Token Generation**

| Token | Object/Method |
|---|---|
| VIS_PARAMETERS | CM_VISION_IMAGEPARAMETERS |
| ACQ_BACKGR | CM_VISION_XBACKGROUND<br>CM_VISION_YBACKGROUND |
| CALIB_CAMERA | CM_VISION_XCAMERA<br>CM_VISION_YCAMERA |
| CALIB_SOURCE | CM_VISION_XSOURCE<br>CM_VISION_YSOURCE |
| LOAD_PAT_DATA | CM_PATIENT_LOADDATA |
| CP_PAT_DATA | Control Program |
| INIT_OBJ_ID | CM_VISION_XOBJECTS<br>CM_VISION_YOBJECTS |
| VIS_ID_DONE | CM_VISION_DONE |
| INIT_MARK_ID | CM_MRI_AXIAL<br>CM_MRI_SAGITTAL<br>CM_MRI_CORONAL |
| MRI_ID_DONE | CM_MRI_DONE |
| CP_ID_XRAY_OBJS | Control Program |
| SELECT_OBJ | IDC_SEED<br>IDC_LEFT_TEMPORAL<br>IDC_CORONAL<br>IDC_RIGHT_TEMPORAL |
| DONE_AXIS_ID | OBJECTS_DIALOG, IDOK |
| CANCEL_AXIS_ID | OBJECTS_DIALOG, IDCANCEL |

**Table 2: Token Generation**

| Token | Object/Method |
|---|---|
| OBJECT_POSIT | axial_view->EvLButtonDown<br>sagittal_view->EvLButtonDown<br>coronal_view->EvLButtonDown |
| SELECT_MARK | IDC_MARK_LEFT_TEMPORAL<br>IDC_MARK_CORONAL<br>IDC_MARK_RIGHT_TEMPORAL |
| DONE_VIEW_ID | MARKERS_DIALOG, IDOK |
| CANCEL_VIEW_ID | MARKERS_DIALOG, IDCANCEL |
| MARKER_POSIT | axial_view->EvLButtonDown<br>sagittal_view->EvLButtonDown<br>coronal_view->EvLButtonDown |
| LOCATE_SEED_REQ | locate_seed_button |
| SLIDER_VALUE | hoz_slider |
| DIRECTION_INFO | angle_box |
| CP_CREAT_ARROW | Control Program |
| CP_REORNT_ARROW | Control Program |
| MOVE_SEED_REQ | move_seed_button |
| CP_DISP_SLICES | Control Program |
| CP_CREAT_CIRCLE | Control Program |
| CP_MOV_FEATURE | Control Program |
| VIEW_UPP | axial_up_button |
| VIEW_DWN | axial_down_button |
| VIEW_RT | sagittal_right_button |
| VIEW_LT | sagittal_left_button |
| VIEW_FRT | coronal_front_button |
| VIEW_BCK | coronal_back_button |
| INCR_COIL | axial_view->EvLButtonDown<br>sagittal_view->EvLButtonDown<br>coronal_view->EvLButtonDown |
| DECR_COIL | axial_view->EvLButtonDown<br>sagittal_view->EvLButtonDown<br>coronal_view->EvLButtonDown |

**Table 2: Token Generation**

| Token | Object/Method |
|---|---|
| SET_COIL | axial_view->EvLButtonDown<br>sagittal_view->EvLButtonDown<br>coronal_view->EvLButtonDown |
| BYE_XRAY | dismiss_Xray_button |
| DISP_CUR_XRAY | CM_VISION_CURRENTX<br>CM_VISION_CURRENTY |
| CLOSE_PAT_DATA | CM_PATIENT_CLOSEDATA |
| QUIT_O_D | QUIT_DIALOG, IDOK |
| QUIT_CONFIRM | CM_QUIT |

## A.5 Specification of Command Interpretation

There is an enumerated set of messages specified in the semantic component whose destinations are the application program or the presentation interpreter. The messages to the presentation interpreter instruct the interpreter to alter the on-screen graphical display in some way. Each of these messages can be mapped to the particular graphical object they manipulate and the method(s) called on that object.

The messages to the application program are enumerated below; this is simply an extraction from the semantic specification to make explicit this interface. The messages to the presentation interpreter are related to the graphical objects or methods of the presentation that they affect in Table 3.

*MethodsToCP* == {*AcquireXrayBackX*, *AcquireXrayBackY*, *CalibrateCameraX*, *CalibrateCameraY*, *CalibrateSourceX*, *CalibrateSourceY*, *LoadPatientData*, *InitiateXrayObjIdX*, *InitiateXrayObjIdX*, *SetInitObjPositsX*, *SetInitObjPositsY*, *CancelXrayObjIdX*, *CancelXrayObjIdY*, *LocateSeed*, *MoveSeed*, *SetMotion*, *Quit*}

**Table 3: Presentation Interpreter**

| Command | Object/Method |
|---|---|
| *LoadPatient* | `axial_view->current_file`<br>`sagittal_view->current_file`<br>`coronal_view->current_file` |
| *IDXrayImageObjs* | `OBJECTS_DIALOG` |
| *CreateArrow* | `MoveTo(); LineTo();` |
| *ReorientArrow* | `MoveTo(); LineTo();` |
| *DisplaySlices* | `axial_view->current_file`<br>`sagittal_view->current_file`<br>`coronal_view->current_file` |
| *CreateCircle* | `Ellipse();` |
| *MoveFeature* | `Ellipse();` |
| *ClosePatient* | `axial_view->current_file`<br>`sagittal_view->current_file`<br>`coronal_view->current_file` |
| *DeleteVisPanel* | `OBJECTS_DIALOG` |
| *DeleteMRIPanel* | `MARKERS_DIALOG` |
| *CreateMRIPanel* | `MARKERS_DIALOG` |

**Table 3: Presentation Interpreter**

| Command | Object/Method |
|---|---|
| *DisplayCurrentXray* | `axial_view->current_file`<br>`sagittal_view->current_file` |
| *DismissXray* | `axial_view->current_file`<br>`sagittal_view->current_file` |
| *ShutDown* | `MedSysWindow` |
| *ViewUp* | `axial_view->current_file` |
| *ViewDown* | `axial_view->current_file` |
| *ViewRight* | `sagittal_view->current_file` |
| *ViewLeft* | `sagittal_view->current_file` |
| *ViewFront* | `coronal_view->current_file` |
| *ViewBack* | `coronal_view->current_file` |
| *SliderVal* | `hoz_slider` |
| *SetReqCoil* | `axial_view->my_mag_coil`<br>`sagittal_view->my_mag_coil`<br>`coronal_view->my_mag_coil` |
| *ChangeMRIButtonColor* | `MARKERS_DIALOG, IDC_MARK_LEFT_TEMPORAL`<br>`MARKERS_DIALOG, IDC_MARK_CORONAL`<br>`MARKERS_DIALOG, IDC_MARK_RIGHT_TEMPORAL` |
| *ChangeVisButtonColor* | `OBJECTS_DIALOG, IDC_SEED`<br>`OBJECTS_DIALOG, IDC_MARK_LEFT_TEMPORAL`<br>`OBJECTS_DIALOG, IDC_MARK_CORONAL`<br>`OBJECTS_DIALOG, IDC_MARK_RIGHT_TEMPORAL` |

## Appendix B

# UVAR User Interface Specification

This is the specification of the user interface for the University of Virginia Reactor (UVAR). The specification of the UVAR user interface consists of various component specifications which comprise different aspects of the user interface. The sections of the UVAR user interface specification with their associated specification notations are shown in Table 1.

**Table 1: UVAR User Interface Specification Components**

| Section | Component | Technology |
|---|---|---|
| B.1 | Semantics | Z |
| B.2 | Syntax | Context-Free Grammar |
| B.3 | Presentation | Borland OWL |
| B.4 | Token Generation | Tables |
| B.5 | Command Interpretation | Tables |

## B.1 Semantic Specification

The functionality of the UVAR user interface is specified in the formal specification language Z. For an overview of formal specification using Z, the reader is referred to Diller.

The semantic specification consists of the following sections:

- Axiomatic Descriptions
- Set Definitions
- State Description
- Initialization
- Operations

The first four sections define the basic types used in the specification and describe the state of the UVAR user interface as modelled in this specification. The last section models the operations that the user interface provides and their effect upon the state of the system; numbers above the operation schemas refer to the production numbers in the grammar of Section B.2. Most schemas map to either a token or a rule reduction in the context-free grammar.

## Axiomatic Descriptions

$$String : \mathrm{seq}\, Char$$

$$MagneticCurrents : \mathrm{seq}\, \mathbb{R}$$
$$\#MagneticCurrents = 3$$

$$SafetyRodHeights : \mathrm{seq}\, \mathbb{R}$$
$$\#SafetyRodHeights = 3$$

$$MessageToAP : MethodsToAP \times \mathbb{N} \times \mathbb{R} \times PowerSettings \rightarrowtail\!\!\!\!\!\rightarrow Message$$

$$MessageToPI : MethodsToPI \times \mathbb{P}\, AlarmSet \rightarrowtail\!\!\!\!\!\rightarrow Message$$

## Set Definitions

[*Message*]

| | |
|---|---|
| *Modes* | == {*StartUp, Operating, ShutDown*} |
| *Char* | == {*a..z*, 0..9} |
| *RegRodMode* | == {*Auto, Manual*} |
| *PowerSettings* | == {*200KW, 2MW*} |
| *AlarmSet* | == {*Scram, ServoRodCntrlLost, MonitorHigh, CoreGammaHigh, ConstAirMonitor, HeatExchRoomDoor, DeminRoomDoor, CoreDiffTempHigh, DeminCondHigh, SecondPumpOff, HotThimbleTemp*} |
| *MethodsToAP* | == {*NewPowerLevel, NewMagCurr, NewFissionHeight, SafetyRodHeight, NewTargetPower, SwitchAutoReg, NewTripPoint*} |
| *MethodsToPI* | == {*DisplayAlarms, SilenceAlarms, ClearAlarms*} |

## State Description

*UVARUserInterface*

| | | |
|---|---|---|
| $modes$ | : | $Modes$ |
| $targetpower$ | : | $\mathbb{R}$ |
| $alarmtrippt$ | : | $\mathbb{R}$ |
| $magcurrs$ | : | $MagneticCurrents$ |
| $safetyrodhts$ | : | $SafetyRodHeights$ |
| $fisschamht$ | : | $\mathbb{R}$ |
| $regrodht$ | : | $\mathbb{R}$ |
| $regrodcontrol$ | : | $RegRodMode$ |
| $powerlevel$ | : | $PowerSettings$ |
| $curralarms$ | : | $\mathbb{P}\ AlarmSet$ |
| $unackdalarms$ | : | $\mathbb{P}\ AlarmSet$ |
| $soundingalarms$ | : | $\mathbb{P}\ AlarmSet$ |

$$soundingalarms \subseteq unackdalarms$$
$$curralarms \subseteq unackdalarms$$

## Initialization

*InitUVARUserInterface*

*UVARUserInterface'*

$$\begin{aligned}
&\mathit{mode} &&= \mathit{StartUp}\\
&\mathit{targetpower}' &&= 0.0\\
&\mathit{alarmtrippt}' &&= 0.0\\
&\mathit{magcurrs}' &&= \langle 0.0, 0.0, 0.0\rangle\\
&\mathit{safetyrodhts}' &&= \langle 0.0, 0.0, 0.0\rangle\\
&\mathit{fisschamht}' &&= 0.0\\
&\mathit{regrodht}' &&= 0.0\\
&\mathit{regrodcontrol}' &&= \mathit{Manual}\\
&\mathit{powerlevel}' &&= \mathit{200KW}\\
&\mathit{curralarms}' &&= \varnothing\\
&\mathit{unackdalarms}' &&= \varnothing\\
&\mathit{soundingalarms}' &&= \varnothing
\end{aligned}$$

## Operations

2:

*SwitchPower*

$\Delta\, UVARUserInterface$
$msgtoap! : MessageToAP$
$newpower? : PowerSettings$

---

$((mode = StartUp) \vee (mode = ShutDown))$
$newpower? \neq powerlevel$
$powerlevel' = newpower?$
$msgtoap!(NewPowerLevel, newpower?)$

7: 8: 9:

*SetMagneticCurrent*

$\Delta\, UVARUserInterface$
$msgtoap! : MessageToAP$
$newcurrval? : \mathbb{R}$
$whichcurr? : \mathbb{N}$

---

$mode = StartUp$
$magcurrs(whichcurr?)' = newcurrval?$
$msgtoap!(NewMagCurr, whichcurr?, newcurrval?)$

4:

*FissionUp*

$\Delta\, UVARUserInterface$
$msgtoap! : MessageToAP$
$newheight? : \mathbb{R}$

---

$mode = StartUp$
$fisschamht' = newheight?$
$msgtoap!(NewFissionHeight, newheight?)$

12: 13: 14: 29: 31: 33:

*SafetyRodUp*

$\Delta\, UVARUserInterface$

$msgtoap! \quad : \quad MessageToAP$

$newheight? \quad : \quad \mathbb{R}$

$whichrod? \quad : \quad \mathbb{N}$

---

$((mode = StartUp) \vee ((mode = Operating) \wedge (regrodcontrol = Manual)))$

$newheight? \quad > \quad safetyrodhts(whichrod?)$

$safetyrodhts(whichrod?)' \quad = \quad newheight?$

$msgtoap!(SafetyRodHeight, whichrod?, newheight?)$

39: 40: 41: 30: 32: 34:

*SafetyRodDown*

$\Delta\, UVARUserInterface$

$msgtoap! \quad : \quad MessageToAP$

$newheight? \quad : \quad \mathbb{R}$

$whichrod? \quad : \quad \mathbb{N}$

---

$((mode = ShutDown) \vee ((mode = Operating) \wedge (regrodcontrol = Manual)))$

$newheight? \quad < \quad safetyrodhts(whichrod?)$

$safetyrodhts(whichrod?)' \quad = \quad newheight?$

$msgtoap!(SafetyRodHeight, whichrod?, newheight?)$

15:

*SetTargetPower*

$\Delta\, UVARUserInterface$

$msgtoap! \quad : \quad MessageToAP$

$newpower? \quad : \quad \mathbb{R}$

---

$((mode = StartUp) \vee (mode = Operating))$

$targetpower' \quad = \quad newpower?$

$msgtoap!(NewTargetPower, newpower?)$

15:

$$\begin{array}{|l}
\textit{AutoRegulationOn} \\
\Delta \textit{UVARUserInterface} \\
\textit{msgtoap}! : \textit{MessageToAP} \\
\hline
((\textit{mode} = \textit{StartUp}) \vee (\textit{mode} = \textit{Operating})) \\
\textit{regrodcontrol} \neq \textit{Auto} \\
\textit{regrodcontrol}' = \textit{Auto} \\
((\textit{mode} = \textit{StartUp}) \Rightarrow (\textit{mode}' = \textit{Operating})) \\
\textit{msgtoap}!(\textit{SwitchAutoReg}) \\
\hline
\end{array}$$

24:

$$\begin{array}{|l}
\textit{AutoRegulationOff} \\
\Delta \textit{UVARUserInterface} \\
\textit{msgtoap}! : \textit{MessageToAP} \\
\hline
((\textit{mode} = \textit{StartUp}) \vee (\textit{mode} = \textit{Operating})) \\
\textit{regrodcontrol} \neq \textit{Manual} \\
\textit{regrodcontrol}' = \textit{Manual} \\
\textit{msgtoap}!(\textit{SwitchAutoReg}) \\
\hline
\end{array}$$

19:

$$\begin{array}{|l}
\textit{SetTripPoint} \\
\Delta \textit{UVARUserInterface} \\
\textit{msgtoap}! : \textit{MessageToAP} \\
\textit{newtrippt}? : \mathbb{R} \\
\hline
\textit{mode} = \textit{Operating} \\
\textit{alarmtrippt}' = \textit{newtrippt}? \\
\textit{msgtoap}!(\textit{NewTripPoint}, \textit{newtrippt}?) \\
\hline
\end{array}$$

21:

$$
\begin{array}{l}
\textit{APReceiveAlarm} \\
\hline
\Delta\, \mathit{UVARUserInterface} \\
\mathit{msgtopi!} \quad : \quad \mathit{MessageTopi} \\
\mathit{alarmset?} \quad : \quad \mathbb{P}\ \mathit{AlarmSet} \\
\hline
\mathit{mode} = \mathit{Operating} \\
\mathit{curralarms'} = \mathit{alarmset?} \\
\mathit{unackdalarms'} = \mathit{unackdalarms} \cup (\mathit{alarmset?} \setminus \mathit{curralarms}) \\
\mathit{soundingalarms'} = \mathit{soundingalarms} \cup (\mathit{alarmset?} \setminus \mathit{curralarms}) \\
\mathit{msgtopi!}(\mathit{DisplayAlarms}, \mathit{alarmset?}) \\
\hline
\end{array}
$$

22:

$$
\begin{array}{l}
\textit{SilenceAlarm} \\
\hline
\Delta\, \mathit{UVARUserInterface} \\
\mathit{msgtopi!} \quad : \quad \mathit{MessageToPI} \\
\hline
\mathit{mode} = \mathit{Operating} \\
\mathit{soundingalarms'} = \varnothing \\
\mathit{msgtopi!}(\mathit{SilenceAlarms}) \\
\hline
\end{array}
$$

23:

$$
\begin{array}{l}
\textit{ClearAlarm} \\
\hline
\Delta\, \mathit{UVARUserInterface} \\
\mathit{msgtopi!} \quad : \quad \mathit{MessageToPI} \\
\hline
\mathit{mode} = \mathit{Operating} \\
\mathit{unackdalarms'} = \mathit{unackdalarms} \setminus \mathit{curralarms} \\
((\mathit{soundingalarms} \neq \varnothing) \Rightarrow ((\mathit{soundingalarms'} = \varnothing) \land \\
\qquad\qquad \mathit{msgtopi!}(\mathit{SilenceAlarms}))) \\
\mathit{msgtopi!}(\mathit{ClearAlarms}, \mathit{unackdalarms'}) \\
\hline
\end{array}
$$

35:

$$
\begin{array}{l}
\textit{ShutDown} \\
\hline
\Delta\, \mathit{UVARUserInterface} \\
\hline
((\mathit{mode} = \mathit{Operating}) \land (\mathit{regrodcontrol} = \mathit{Manual})) \\
\mathit{mode'} = \mathit{ShutDown} \\
\hline
\end{array}
$$

## B.2 Syntactic Specification

A context-free grammar (Backus-Naur Form) is used to specify the syntax of the user interface.

```
1:    uvar            := start_up operation shut_down
2:    start_up        := thru_fiss pull_rods SWITCH_PWR pull_rods auto_on
3:                    |  thru_fiss pull_rods auto_on
4:    thru_fiss       := mag_currs pull_rods FISS_UP
5:    mag_currs       := mag_currs mc_val
6:                    |  mc_val
7:    mc_val          := SET_MC1
8:                    |  SET_MC2
9:                    |  SET_MC3
10:   pull_rods       := pull_rods pull_one
11:                   |  pull_one
12:   pull_one        := SAFETY1_UP
13:                   |  SAFETY2_UP
14:                   |  SAFETY3_UP
15:   auto_on         := SET_POWER AUTO_REG_ON
16:   operation       := operation op_acts
17:                   |  op_acts
18:   op_acts         := mc_val
19:                   |  SET_TRIP
20:                   |  auto_on
21:                   |  AP_RCV_ALRM
22:                   |  SIL_ALRM
23:                   |  CLR_ALRM
24:                   |  AUTO_REG_OFF
25:                   |  manip_rods
26:   manip_rods      := sr1_act
27:                   |  sr2_act
28:                   |  sr3_act
29:   sr1_act         := SAFETY1_UP
30:                   |  SAFETY1_DWN
31:   sr2_act         := SAFETY2_UP
```

```
32:                      | SAFETY2_DWN
33:    sr3_act          := SAFETY3_UP
34:                      | SAFETY3_DWN
35:    shut_down        := SHUTDOWN insert_rods SWITCH_PWR insert_rods OFF
36:                      | SHUTDOWN insert_rods OFF
37:    insert_rods      := insert_rods insert_one
38:                      | insert_one
39:    insert_one       := SAFETY1_DWN
40:                      | SAFETY2_DWN
41:                      | SAFETY3_DWN
```

## B.3 Presentation Specification

The presentation of the UVAR user interface is specified using Borland ObjectWindows. For a complete specification of the graphical display, in addition to the description found in this section, the documentation for the Borland ObjectWindows Library must be included. The definitions of base objects used in the specification may be found in the ObjectWindows documentation, in addition to definitions for the semantics of actions, such as button presses.

In addition to the textual Borland ObjectWindows specification, a visual presentation of the interface may be constructed by compilation with the ObjectWindows Library and subsequent execution. The specification is for presentation only; therefore no application functionality or dialogue control is included, and compilation would create simply a mock-up of the display.

This user interface specification component consists of five sections. The first section is the `reactor.rc` file, which contains graphical information for the definition of the user interface menus.

The second section presents the `reactor.h` file. The primary class of the UVAR user interface, MainReactorWindow, is defined in this section. This class contains declarations of all the other graphical objects on the display: visual enumerations of scram and alarm conditions, buttons for alarm response, and classes for control rod and fission chamber information and manipulation. These classes pertaining to the control rods and fission chamber are also defined in this file.

The next section contains the file `reactor.cpp`, which has the constructors, destructors, and paint functions for each of the classes defined in the previous section. The constructor for the MainReactorWindow instantiates the graphical objects on the display.

The fourth section presents the file `light.h`, which defines the Light class. The Light class is used to display status information in each of the control rod and fission chamber objects; it replicates light indicators on the analog control panel, being a text box which turns a different color to indicate status on.

The final section of the presentation specfication contains `light.cpp`; the constructor, destructor, and functions to change color for the Light class are presented in this file.

## reactor.rc

```
/****************************************************************************


reactor.rc

produced by Borland Resource Workshop


*****************************************************************************/

#include "reactor.rh"

MAIN_REACTOR_MENU MENU
{
  POPUP "&Log"
  {
    MENUITEM "&Researcher",                   CM_LOG_RESEARCHER
    MENUITEM "&Operator",                     CM_LOG_OPERATOR
    MENUITEM "&Maintainer",                   CM_LOG_MAINTAINER
  }

  POPUP "&Data"
  {
    MENUITEM "&Histories",                    CM_DATA_HISTORIES
  }

  POPUP "&View"
  {
    MENUITEM "&Cameras",                      CM_VIEW_CAMERAS
  }

  POPUP "&Control"
  {
    MENUITEM "&Auto",                         CM_CONTROL_AUTO
    MENUITEM "&Manual",                       CM_CONTROL_MANUAL
    MENUITEM "&Switch Power",                 CM_SWITCH_POWER
  }

  POPUP "&Security"
  {
    MENUITEM "Set &User",                     CM_SECURITY_SETUSER
    MENUITEM "Change &Password",              CM_SECURITY_CHANGEPASS
  }
 }
```

## reactor.h

```
#include "reactor.rh"
#include <stdlib.h>
#include <stdio.h>
#include <iomanip.h>
#include <owl/owlpch.h>
#include <owl/buttonga.h>
#include <owl/gadgetwi.h>
#include <owl/applicat.h>
#include <owl/framewin.h>
#include <owl/point.h>
#include <owl/dc.h>
#include <owl/edit.h>
#include "light.h"


const TColor Yellow(255,255,0);
const TColor Violet(255,0,255);
const TColor Cyan(0,255,255);
const TColor DkGray(160,160,164);

const int ID_AUTO_CONTROL          = 540;
const int ID_CLEAR                 = 541;
const int ID_SILENCE               = 542;
const int ID_RESET                 = 543;

const int LIGHT_OFFSET             = 5;
const int TEXT_OFFSET              = 10;
const int LIGHT_HEIGHT             = 30;
const int LIGHT_WIDTH              = 120;
const int ALARM_HEIGHT             = 30;
const int ALARM_WIDTH              = 80;

const int CONTROL_ROD_TOP          = 700;
const int CONTROL_HEIGHT           = 200;
const int CONTROL_WIDTH            = 400;
const int CR1_LEFT                 = 0;
const int CR2_LEFT                 = 400;
const int CR3_LEFT                 = 800;
const int REG_ROD_TOP              = CONTROL_ROD_TOP - CONTROL_HEIGHT;
const int SCRAM_BOX_TOP            = REG_ROD_TOP - 10*LIGHT_HEIGHT;
const int SCRAM_BOX_LEFT           = 12*ALARM_WIDTH;
const int ALARMS_TOP               = REG_ROD_TOP - 2*ALARM_HEIGHT;
const int ALARMS_LEFT              = 0;
const int CLEAR_LEFT               = ALARMS_LEFT + 11*ALARM_WIDTH;
const int SILENCE_LEFT             = 0;
const int ALARM_BUTTON_HEIGHT      = ALARM_HEIGHT;
const int ALARM_BUTTON_WIDTH       = 80;
const int ALARM_BUTTON_TOP         = ALARMS_TOP - ALARM_BUTTON_HEIGHT;

const int RESET_LEFT               = SCRAM_BOX_LEFT;
const int RESET_HEIGHT             = 60;
const int RESET_TOP                = SCRAM_BOX_TOP - RESET_HEIGHT;
const int RESET_WIDTH              = 2*LIGHT_WIDTH;

const char* SCRAMS[20] = { "Power Range",
                           "Pool Level 1",
                                 "Pool Level 2",
                                 "Pri. Pump On",
                                 "Pri. Pump Off",
                                 "Bridge Rad",
                                 "Face Rad",
                                 "Range Switch",
                                 "Header Down",
                                 "Air to Head",
                                 "Truck Open",
                                 "Escape Open",
                                 "Min Irr",
                                 "Manual-Room",
                                 "Manual-Back",
                                 "Evacuation",
                                 "Fire",
                                 "Pool Temp",
                                 "Int Period",
                                 "Low Flow" };

const char* ALARMS[12] = { "Scram",
                                 "Servo Rod",
                                 "Argon",
                                 "Core Gamma",
                                 "Not Used",
                                 "Air Mon",
                                 "Heat EX",
                                 "Demin.",
                                 "Core Temp",
                                 "Cond. High",
                                 "Sec Pump Off",
                                 "Hot Thimble" };


class ControlBox : public TWindow
{
  public:
    ControlBox(TWindow*, char*, int, int, int, int);
    ~ControlBox();
    void Paint(TDC &, BOOL, TRect&);
    void Withdraw(TDC &,StatusType);
    void Insert(TDC &,StatusType);

  protected:
    char*            name;
    char*            buffer;
    int              width;
    int              height;
    float            rod_height;
    float            req_height;
    Light*           inserting_light;
    Light*           withdrawing_light;

  DECLARE_RESPONSE_TABLE(ControlBox);
};

class ControlRod : public ControlBox
{
  public:
    ControlRod(TWindow*, char*, int, int, int, int);
    ~ControlRod();
    void Paint(TDC &, BOOL, TRect &);

  protected:
    float            mag_current;
    Light*           up_light;
    Light*           down_light;
    Light*           seated_light;
    Light*           mag_eng_light;

  DECLARE_RESPONSE_TABLE(ControlRod);
};

class FissionChamber : public ControlBox
{
  public:
    FissionChamber(TWindow*, char*, int, int, int, int);
    ~FissionChamber();
    void Paint(TDC &, BOOL, TRect &);

  private:
    Light*           up_light;
    Light*           down_light;

  DECLARE_RESPONSE_TABLE(FissionChamber);
};

class RegulatingRod : public ControlBox
{
  public:
    RegulatingRod(TWindow*, char*, int, int, int, int);
    ~RegulatingRod();
    void Paint(TDC &, BOOL, TRect &);

  private:
    Light*           up_light;
    Light*           down_light;
    TButton*         auto_control;
    Light*           ac_on_light;
    Light*           ac_off_light;

  DECLARE_RESPONSE_TABLE(RegulatingRod);
};


class MainReactorWindow : public TWindow
{
  public:
    MainReactorWindow();
    ~MainReactorWindow();
    void Paint(TDC&, BOOL, TRect &);

  private:
    float            req_power;
    float            setpoint_temp;
    Light**          scram_box_array;
    Light**          alarm_array;
    TButton*         clear_button;
    TButton*         silence_button;
    TButton*          reset_button;
    RegulatingRod*    reg_rod;
    ControlRod*       control_rod_1;
    ControlRod*       control_rod_2;
    ControlRod*       control_rod_3;
    FissionChamber*   fission_chamber;

  DECLARE_RESPONSE_TABLE(MainReactorWindow);
};

class ReactorApp: public TApplication
{
  public:
    ReactorApp() : TApplication() {}
    void InitMainWindow();
};
```

## reactor.cpp

```
#include "reactor.h"

DEFINE_RESPONSE_TABLE1(ControlBox, TWindow)
  EV_WM_PAINT,
END_RESPONSE_TABLE;

DEFINE_RESPONSE_TABLE1(ControlRod,ControlBox)
  EV_WM_PAINT,
END_RESPONSE_TABLE;

DEFINE_RESPONSE_TABLE1(FissionChamber,ControlBox)
  EV_WM_PAINT,
END_RESPONSE_TABLE;

DEFINE_RESPONSE_TABLE1(RegulatingRod, ControlBox)
  EV_WM_PAINT,
END_RESPONSE_TABLE;

DEFINE_RESPONSE_TABLE1(MainReactorWindow,TWindow)
  EV_WM_PAINT,
END_RESPONSE_TABLE;


ControlBox::ControlBox(TWindow* parent, char* title,
                       int X, int Y, int W, int H)
  : TWindow(parent, 0, 0)
{
  name                = title;
  buffer              = new char[20];
  rod_height          = 0.0;
  req_height          = 0;
  Attr.X              = X;
  Attr.Y              = Y;
  Attr.W              = W;
  Attr.H              = H;

  inserting_light   = new Light((2*Attr.W/3)+LIGHT_OFFSET,
                              Attr.H/2+LIGHT_OFFSET,
                              Attr.W-LIGHT_OFFSET,
                              Attr.H/2+LIGHT_HEIGHT+LIGHT_OFFSET,
                              "INSERT",
                              TColor::LtGray,
                              Cyan);
  withdrawing_light = new Light((2*Attr.W/3)+LIGHT_OFFSET,
                              Attr.H/2+LIGHT_HEIGHT+LIGHT_OFFSET,
                              Attr.W-LIGHT_OFFSET,
                              Attr.H/2+2*LIGHT_HEIGHT+LIGHT_OFFSET,
                              "WITHDRAW",
                              TColor::LtGray,
                              Violet);
}

ControlBox::~ControlBox()
{
  delete inserting_light;
  delete withdrawing_light;
```

```
}

void ControlBox::Withdraw(TDC & dc,StatusType status)
{
  if (status == ON)
    withdrawing_light->TurnOn(dc);
  else
    withdrawing_light->TurnOff(dc);
}

void ControlBox::Insert(TDC & dc,StatusType status)
{
  if (status == ON)
    inserting_light->TurnOn(dc);
  else
    inserting_light->TurnOff(dc);
}

void ControlBox::Paint(TDC& dc, BOOL, TRect&)
{
  TRect text_region(0,0,Attr.W,Attr.H);
  TBrush lt_gray_brush(TColor::LtGray);
  dc.FillRect(text_region,lt_gray_brush);
  dc.SetBkColor(TColor::LtGray);

  text_region.Set(0, TEXT_OFFSET ,
                  Attr.W , Attr.H/4 - TEXT_OFFSET);
  dc.DrawText(name, -1, text_region, DT_CENTER | DT_VCENTER);
  text_region.Set(Attr.W/3, Attr.H/4,
                  2*Attr.W/3, 3*Attr.H/8);
  dc.DrawText("Height:",-1, text_region, DT_CENTER | DT_VCENTER);
  text_region.Set(Attr.W/3, 3*Attr.H/8,
                  2*Attr.W/3, Attr.H/2);
  sprintf(buffer, "%g", rod_height);
  dc.DrawText(buffer, -1, text_region, DT_CENTER | DT_VCENTER);

  text_region.Set(2*Attr.W/3, Attr.H/4,
                  Attr.W - 20, 3*Attr.H/8);
  dc.DrawText("Req. Height:",-1, text_region, DT_CENTER | DT_VCENTER);
  text_region.Set(2*Attr.W/3, 3*Attr.H/8,
                  Attr.W - 20, Attr.H/2);
  dc.SetBkColor(TColor::White);
  dc.DrawText(buffer, -1, text_region, DT_CENTER | DT_VCENTER);

  delete buffer;
  dc.SelectObject(TPen(TColor::Black));

  dc.MoveTo(0,0);
  dc.LineTo(Attr.W, 0);
  dc.LineTo(Attr.W, Attr.H);
  dc.LineTo(0, Attr.H);
  dc.LineTo(0,0);

  dc.SelectObject(TPen(TColor::White));
  dc.MoveTo(1,Attr.H-1);
  dc.LineTo(1, 1);
  dc.LineTo(Attr.W-1, 1);
  dc.SelectObject(TPen(DkGray));
  dc.LineTo(Attr.W-1, Attr.H-1);
  dc.LineTo(1,Attr.H-1);

  withdrawing_light->DrawRect(dc);
  inserting_light->DrawRect(dc);
}

ControlRod::ControlRod(TWindow* parent, char* title,
                       int X, int Y, int W, int H)
  : ControlBox(parent, title,X,Y,W,H)
{
  mag_current   = 0.0;

  up_light      = new Light(5, Attr.H/4 + LIGHT_OFFSET,
                            Attr.W/3 - LIGHT_OFFSET,
                            Attr.H/4 + LIGHT_OFFSET + LIGHT_HEIGHT,
                            "UP",
                            TColor::LtGray,
                            TColor::LtBlue);

  mag_eng_light = new Light(LIGHT_OFFSET,
                            Attr.H/4 + LIGHT_OFFSET + LIGHT_HEIGHT,
                            Attr.W/3 - LIGHT_OFFSET,
                            Attr.H/4 + LIGHT_OFFSET + 2*LIGHT_HEIGHT,
                            "MAG ENG",
                            TColor::LtGray,
                            TColor::LtGreen);

  down_light    = new Light(LIGHT_OFFSET,
                            Attr.H/4 + LIGHT_OFFSET + 2*LIGHT_HEIGHT,
                            Attr.W/3 - LIGHT_OFFSET,
                            Attr.H/4 + LIGHT_OFFSET + 3*LIGHT_HEIGHT,
                            "DOWN",
                            TColor::LtGray,
                            TColor::White);

  seated_light  = new Light(LIGHT_OFFSET,
                            Attr.H/4 + LIGHT_OFFSET + 3*LIGHT_HEIGHT,
                            Attr.W/3 - LIGHT_OFFSET,
                            Attr.H/4 + LIGHT_OFFSET + 4*LIGHT_HEIGHT,
                            "SEATED",
                            TColor::LtGray,
                            Yellow);
}

ControlRod::~ControlRod()
{
  delete up_light;
  delete down_light;
  delete seated_light;
  delete mag_eng_light;
}

void ControlRod::Paint(TDC & dc, BOOL state, TRect& rect)
{
  ControlBox::Paint(dc, state, rect);

  dc.SetBkColor(TColor::LtGray);

  char* buffer = new char[20];
  TRect text_region(0,0,0,0);
  dc.RestoreObjects();
  text_region.Set(Attr.W/3, Attr.H/2,
                  2*Attr.W/3, 5*Attr.H/8);
  dc.DrawText("Mag. Current:",-1, text_region, DT_CENTER | DT_VCENTER);
  text_region.Set(Attr.W/3, 5*Attr.H/8,
                  2*Attr.W/3, 3*Attr.H/4);
  sprintf(buffer, "%g", mag_current);
  dc.DrawText(buffer, -1, text_region, DT_CENTER | DT_VCENTER);

  text_region.Set(Attr.W/3, 3*Attr.H/4,
                  2*Attr.W/3, 7*Attr.H/8);
  dc.DrawText("Req. Mag. Curr.",-1, text_region, DT_CENTER | DT_VCENTER);
  text_region.Set(Attr.W/3, 7*Attr.H/8,
                  2*Attr.W/3, Attr.H);
  dc.SetBkColor(TColor::White);
  dc.DrawText(buffer, -1, text_region, DT_CENTER | DT_VCENTER);

  delete buffer;

  up_light->DrawRect(dc);
  down_light->DrawRect(dc);
  seated_light->TurnOn(dc);
  mag_eng_light->TurnOn(dc);
}

FissionChamber::FissionChamber(TWindow* parent, char* title,
                               int X, int Y, int W, int H)
  : ControlBox(parent,title,X,Y,W,H)
{
  up_light   = new Light(LIGHT_OFFSET,
                         Attr.H/2 + LIGHT_OFFSET,
                         Attr.W/3 - LIGHT_OFFSET,
                         Attr.H/2 + LIGHT_OFFSET + LIGHT_HEIGHT,
                         "UP",
                         TColor::LtGray,
                         TColor::LtBlue);

  down_light = new Light(LIGHT_OFFSET,
                         Attr.H/2 + LIGHT_OFFSET + LIGHT_HEIGHT,
                         Attr.W/3 - LIGHT_OFFSET,
                         Attr.H/2 + LIGHT_OFFSET + 2*LIGHT_HEIGHT,
                         "DOWN",
                         TColor::LtGray,
                         TColor::White);
}

FissionChamber::~FissionChamber()
{
  delete up_light;
  delete down_light;
}


void FissionChamber::Paint(TDC & dc, BOOL state, TRect & rect)
{
  ControlBox::Paint(dc, state, rect);

  up_light->DrawRect(dc);
  down_light->DrawRect(dc);
  Insert(dc, ON);
  Withdraw(dc,ON);
}

RegulatingRod::RegulatingRod(TWindow* parent, char* title,
                             int X, int Y, int W, int H)
  : ControlBox(parent, title,X,Y,W,H)
{
  up_light     = new Light(LIGHT_OFFSET,
                           Attr.H/2 + LIGHT_OFFSET,
                           Attr.W/3 - LIGHT_OFFSET,
                           Attr.H/2 + LIGHT_OFFSET + LIGHT_HEIGHT,
                           "UP",
                           TColor::LtGray,
                           TColor::LtBlue);

  down_light   = new Light(LIGHT_OFFSET,
                           Attr.H/2 + LIGHT_OFFSET + LIGHT_HEIGHT,
                           Attr.W/3 - LIGHT_OFFSET,
                           Attr.H/2 + LIGHT_OFFSET + 2*LIGHT_HEIGHT,
                           "DOWN",
                           TColor::LtGray,
                           TColor::White);

  auto_control = new TButton(this, ID_AUTO_CONTROL, "Auto",
                             0, Attr.H/4 + 5,
                             Attr.W/6,
                             Attr.H/4 - 10);

  ac_on_light  = new Light(Attr.W/6,
                           Attr.H/4 + 5,
                           Attr.W/6 + Attr.W/12,
                           Attr.H/2 - 5,
                           "ON",
                           TColor::LtGray,
                           TColor::LtGreen);

  ac_off_light = new Light(Attr.W/6 + Attr.W/12,
                           Attr.H/4 + 5,
                           Attr.W/6 + 2*Attr.W/12,
                           Attr.H/2 - 5,
                           "OFF",
                           TColor::LtGray,
                           TColor::LtGreen);
}

RegulatingRod::~RegulatingRod()
{
  delete up_light;
  delete down_light;
  delete auto_control;
  delete ac_on_light;
  delete ac_off_light;
}

void RegulatingRod::Paint(TDC& dc, BOOL state, TRect & rect)
{
  ControlBox::Paint(dc, state, rect);
  up_light->TurnOn(dc);
  down_light->TurnOn(dc);
  ac_on_light->DrawRect(dc);
  ac_off_light->TurnOn(dc);
}

MainReactorWindow::MainReactorWindow()
  : TWindow(0, 0, 0)
{
  Attr.X          = 50;
  Attr.Y          = 50;
  Attr.W          = 1200;
  Attr.H          = 900;
  setpoint_temp   = 0.0;
  req_power       = 0.0;

  reg_rod         = new RegulatingRod(this, "Regulating Rod",
                                  CR1_LEFT, REG_ROD_TOP,
                                  CONTROL_WIDTH, CONTROL_HEIGHT);
  control_rod_1   = new ControlRod(this, "Control Rod 1",
                               CR1_LEFT, CONTROL_ROD_TOP,
                               CONTROL_WIDTH, CONTROL_HEIGHT);
  control_rod_2   = new ControlRod(this, "Control Rod 2",
                               CR2_LEFT, CONTROL_ROD_TOP,
                               CONTROL_WIDTH, CONTROL_HEIGHT);
  control_rod_3   = new ControlRod(this, "Control Rod 3",
                               CR3_LEFT, CONTROL_ROD_TOP,
                               CONTROL_WIDTH, CONTROL_HEIGHT);
  fission_chamber = new FissionChamber(this, "Fission Chamber",
                                   CR2_LEFT, REG_ROD_TOP,
                                   CONTROL_WIDTH, CONTROL_HEIGHT);

  scram_box_array = new Light*[20];
  for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 2; j++) {
      scram_box_array[(2*i)+j] = new Light(SCRAM_BOX_LEFT+j*LIGHT_WIDTH,
                                       SCRAM_BOX_TOP + (LIGHT_HEIGHT)*i,
                                       SCRAM_BOX_LEFT+(j+1)*LIGHT_WIDTH,
                                       SCRAM_BOX_TOP+(LIGHT_HEIGHT)*(i+1),
                                       (char*) SCRAMS[(2*i)+j],
                                       TColor::White,
                                       TColor::LtRed);
    }
  }
  alarm_array = new Light*[24];
  for (i = 0; i < 12; i++) {
    alarm_array[2*i]     = new Light(ALARMS_LEFT + i*ALARM_WIDTH,
                                 ALARMS_TOP,
                                 ALARMS_LEFT + (i+1)*ALARM_WIDTH,
                                 ALARMS_TOP + ALARM_HEIGHT,
                                 (char*) ALARMS[i],
                                 TColor::White,
                                 TColor::LtRed);

    alarm_array[(2*i)+1] = new Light(ALARMS_LEFT + i*ALARM_WIDTH,
                                 ALARMS_TOP + ALARM_HEIGHT,
                                 ALARMS_LEFT + (i+1)*ALARM_WIDTH,
                                 ALARMS_TOP + 2*ALARM_HEIGHT,
                                           (char*) ALARMS[i],
                                           TColor::White,
                                           Yellow);
  }
  clear_button   = new TButton(this, ID_CLEAR, "Clear", CLEAR_LEFT,
                              ALARM_BUTTON_TOP, ALARM_BUTTON_WIDTH,
                              ALARM_BUTTON_HEIGHT);
  silence_button = new TButton(this, ID_SILENCE, "Silence", SILENCE_LEFT,
                              ALARM_BUTTON_TOP, ALARM_BUTTON_WIDTH,
                              ALARM_BUTTON_HEIGHT);
  reset_button   = new TButton(this, ID_RESET, "Reset Scrams", RESET_LEFT,
                              RESET_TOP, RESET_WIDTH, RESET_HEIGHT);
}

MainReactorWindow::~MainReactorWindow()
{
  delete reg_rod;
  delete control_rod_1;
  delete control_rod_2;
  delete control_rod_3;
  delete fission_chamber;
  delete scram_box_array;
  delete alarm_array;
  delete clear_button;
  delete silence_button;
  delete reset_button;
}


void MainReactorWindow::Paint(TDC& paint_dc, BOOL, TRect &)
{
  TRect info_area(CR3_LEFT,REG_ROD_TOP, CR3_LEFT + CONTROL_WIDTH,
                  CONTROL_ROD_TOP);
  paint_dc.FillRect(info_area, TBrush(TColor::LtGray));
  paint_dc.SelectObject(TPen(TColor::Black));

  paint_dc.SetBkColor(TColor::LtGray);
  info_area.Set(CR3_LEFT, REG_ROD_TOP + 10,
              CR3_LEFT + (CONTROL_WIDTH/2),
              REG_ROD_TOP + (CONTROL_HEIGHT/4) - 10);
  paint_dc.DrawText("Diff Core Temp = 212 F", -1, info_area,
                    DT_CENTER | DT_VCENTER);
  info_area.Set(CR3_LEFT+(CONTROL_WIDTH/2), REG_ROD_TOP,
              CR3_LEFT + CONTROL_WIDTH, REG_ROD_TOP + (CONTROL_HEIGHT/8));
  paint_dc.DrawText("Setpoint Temp:",-1,info_area,DT_CENTER);
  info_area.Set(CR3_LEFT+(CONTROL_WIDTH/2), REG_ROD_TOP + (CONTROL_HEIGHT/8),
              CR3_LEFT + CONTROL_WIDTH, REG_ROD_TOP + (CONTROL_HEIGHT/4));
  char* temp_buffer = new char[20];
  paint_dc.SetBkColor(TColor::White);
  sprintf(temp_buffer, "%g", setpoint_temp);
  paint_dc.DrawText(temp_buffer, -1, info_area, DT_CENTER);
  paint_dc.SetBkColor(TColor::LtGray);
  delete temp_buffer;
  info_area.Set(CR3_LEFT, REG_ROD_TOP + CONTROL_HEIGHT/4 + 10,
              CR3_LEFT + CONTROL_WIDTH,
              REG_ROD_TOP + 2*(CONTROL_HEIGHT/4)-10);
  paint_dc.DrawText("Bulk Pool Temp = 134 F", - 1, info_area,
                    DT_CENTER | DT_VCENTER);
  info_area.Set(CR3_LEFT, REG_ROD_TOP + 2*(CONTROL_HEIGHT/4) + 10,
              CR3_LEFT + CONTROL_WIDTH,
              REG_ROD_TOP +  3*(CONTROL_HEIGHT/4)-10);
paint_dc.DrawText("Power Level = 2 MW", -1, info_area,
                  DT_CENTER | DT_VCENTER);
info_area.Set(CR3_LEFT, REG_ROD_TOP + 3*(CONTROL_HEIGHT/4),
              CR3_LEFT + CONTROL_WIDTH, REG_ROD_TOP+7*(CONTROL_HEIGHT/8));
paint_dc.DrawText("Req Power:", -1, info_area, DT_CENTER);
info_area.Set(CR3_LEFT, REG_ROD_TOP + 7*(CONTROL_HEIGHT/8),
              CR3_LEFT + CONTROL_WIDTH, CONTROL_ROD_TOP);
temp_buffer = new char[20];
sprintf(temp_buffer, "%g MW", req_power);
paint_dc.SetBkColor(TColor::White);
paint_dc.DrawText(temp_buffer, -1, info_area, DT_CENTER);
paint_dc.SetBkColor(TColor::LtGray);
delete temp_buffer;

scram_box_array[12]->TurnOn(paint_dc);
for (int i = 0; i < 10; i++) {
  for (int j = 0; j < 2; j++) {
     scram_box_array[(2*i)+j]->DrawRect(paint_dc);
  }
}

alarm_array[6]->TurnOn(paint_dc);
alarm_array[7]->TurnOn(paint_dc);
for (i = 0; i < 24; i++) {
  alarm_array[i]->DrawRect(paint_dc);
}

paint_dc.SelectObject(TPen(TColor::White));
paint_dc.MoveTo(CR3_LEFT + CONTROL_WIDTH - 1, REG_ROD_TOP);
paint_dc.LineTo((CR3_LEFT)+1, REG_ROD_TOP + 1);
paint_dc.LineTo((CR3_LEFT)+1, CONTROL_ROD_TOP - 1);
paint_dc.MoveTo(CR3_LEFT + CONTROL_WIDTH - 1,
                CONTROL_ROD_TOP - (CONTROL_HEIGHT/4) + 1);
paint_dc.LineTo((CR3_LEFT)+1, CONTROL_ROD_TOP - (CONTROL_HEIGHT/4) + 1);
paint_dc.MoveTo(CR3_LEFT + CONTROL_WIDTH - 1,
                CONTROL_ROD_TOP - (2*(CONTROL_HEIGHT/4)) + 1);
paint_dc.LineTo((CR3_LEFT)+1, CONTROL_ROD_TOP - (2*(CONTROL_HEIGHT/4)) + 1);
paint_dc.MoveTo(CR3_LEFT + CONTROL_WIDTH - 1,
                CONTROL_ROD_TOP - (3*(CONTROL_HEIGHT/4)) + 1);
paint_dc.LineTo((CR3_LEFT)+1, CONTROL_ROD_TOP - (3*(CONTROL_HEIGHT/4)) + 1);

paint_dc.SelectObject(TPen(DkGray));
paint_dc.MoveTo(CR3_LEFT + CONTROL_WIDTH - 1, CONTROL_ROD_TOP - 1);
paint_dc.LineTo((CR3_LEFT)+1, CONTROL_ROD_TOP - 1);
paint_dc.LineTo((CR3_LEFT)+1, REG_ROD_TOP + 1);
paint_dc.MoveTo(CR3_LEFT + CONTROL_WIDTH - 1,
                CONTROL_ROD_TOP - (CONTROL_HEIGHT/4) - 1);
paint_dc.LineTo((CR3_LEFT)+1, CONTROL_ROD_TOP - (CONTROL_HEIGHT/4) - 1);
paint_dc.MoveTo(CR3_LEFT + CONTROL_WIDTH - 1,
                CONTROL_ROD_TOP - (2*(CONTROL_HEIGHT/4)) - 1);
paint_dc.LineTo((CR3_LEFT)+1, CONTROL_ROD_TOP - 2*(CONTROL_HEIGHT/4) - 1);
paint_dc.MoveTo(CR3_LEFT + CONTROL_WIDTH - 1,
                CONTROL_ROD_TOP - (3*(CONTROL_HEIGHT/4)) - 1);
paint_dc.LineTo((CR3_LEFT)+1, CONTROL_ROD_TOP - 3*(CONTROL_HEIGHT/4) - 1);

paint_dc.SelectObject(TPen(TColor::Black));
paint_dc.MoveTo(CR3_LEFT + CONTROL_WIDTH, CONTROL_ROD_TOP);
```

```
  paint_dc.LineTo((CR3_LEFT), CONTROL_ROD_TOP);
  paint_dc.MoveTo((CR3_LEFT), CONTROL_ROD_TOP - CONTROL_HEIGHT);
  paint_dc.LineTo(CR3_LEFT + CONTROL_WIDTH, CONTROL_ROD_TOP - CONTROL_HEIGHT);

  paint_dc.MoveTo(CR3_LEFT + CONTROL_WIDTH,
                  CONTROL_ROD_TOP - (CONTROL_HEIGHT/4));
  paint_dc.LineTo((CR3_LEFT), CONTROL_ROD_TOP - (CONTROL_HEIGHT/4));
  paint_dc.MoveTo(CR3_LEFT + CONTROL_WIDTH,
                  CONTROL_ROD_TOP - (2*(CONTROL_HEIGHT/4)));
  paint_dc.LineTo((CR3_LEFT), CONTROL_ROD_TOP - 2*(CONTROL_HEIGHT/4));
  paint_dc.MoveTo(CR3_LEFT + CONTROL_WIDTH,
                  CONTROL_ROD_TOP - (3*(CONTROL_HEIGHT/4)));
  paint_dc.LineTo((CR3_LEFT), CONTROL_ROD_TOP - 3*(CONTROL_HEIGHT/4));
}

void ReactorApp::InitMainWindow()
{
  MainWindow = new TFrameWindow(0, "Reactor Mockup", new MainReactorWindow());
  MainWindow->AssignMenu(MAIN_REACTOR_MENU);
}

int OwlMain(int, char* [])
{
  return ReactorApp().Run();
}
```

## light.h

```
#include <owl/point.h>
#include <owl/dc.h>
#include <owl/color.h>
#include <owl/gdiobjec.h>


typedef enum StatusType  { ON, OFF };

class Light : public TRect
{
  public:
    Light(char* text,
          TColor off_color = TColor::LtBlue,
          TColor on_color = TColor::LtRed);
    Light(const RECT far&,
          char* text,
          TColor off_color = TColor::LtBlue,
          TColor on_color = TColor::LtRed);
    Light(int, int, int, int,
          char* text,
          TColor off_color = TColor::LtBlue,
          TColor on_color = TColor::LtRed);
    Light(const TPoint&, const TPoint&,
          char* text,
          TColor off_color = TColor::LtBlue,
          TColor on_color = TColor::LtRed);
    Light(const TPoint&, const TSize&,
          char* text,
          TColor off_color = TColor::LtBlue,
          TColor on_color = TColor::LtRed);
    ~Light();
    void TurnOn(TDC&);
    void TurnOff(TDC&);
    void DrawRect(TDC&);

  private:
    char* title;
    StatusType status;
    TColor off_color;
    TColor on_color;
    TBrush* on_brush;
    TBrush* off_brush;
};
```

## light.cpp

```
#include "light.h"

Light::Light(char* text, TColor OffColor,
         TColor OnColor)
  : TRect()
{
  off_color  = OffColor;
  on_color   = OnColor;
  status     = OFF;
  title      = text;
  off_brush  = new TBrush(off_color);
  on_brush   = new TBrush(on_color);
}

Light::Light(const RECT far& rect,
             char* text,
             TColor OffColor,
             TColor OnColor)
  : TRect(rect)
{
  off_color  = OffColor;
  on_color   = OnColor;
  title      = text;
  status     = OFF;
  off_brush  = new TBrush(off_color);
  on_brush   = new TBrush(on_color);
}

Light::Light(int left, int top, int right, int bottom,
             char* text,
             TColor OffColor,
             TColor OnColor)
  : TRect(left,top,right,bottom)
{
  off_color  = OffColor;
  on_color   = OnColor;
  title      = text;
  status     = OFF;
  off_brush  = new TBrush(off_color);
  on_brush   = new TBrush(on_color);
}

Light::Light(const TPoint& upLeft, const TPoint& loRight,
             char* text,
             TColor OffColor,
             TColor OnColor)
  : TRect(upLeft,loRight)
{
  off_color  = OffColor;
  on_color   = OnColor;
  title      = text;
  status     = OFF;
  off_brush  = new TBrush(off_color);
  on_brush   = new TBrush(on_color);
}
```

```
Light::Light(const TPoint& origin, const TSize& extent,
             char* text,
             TColor OffColor,
             TColor OnColor)
  : TRect(origin, extent)
{
  off_color  = OffColor;
  on_color   = OnColor;
  title      = text;
  status     = OFF;
  off_brush  = new TBrush(off_color);
  on_brush   = new TBrush(on_color);
}

Light::~Light() {
  //delete off_brush;
  //delete on_brush;
}

void Light::TurnOn(TDC& dc)
{
  status = ON;
  DrawRect(dc);
}

void Light::TurnOff(TDC& dc)
{
  status = OFF;
  DrawRect(dc);
}

void Light::DrawRect(TDC& dc)
{
  dc.SetBkColor(off_color);
  dc.SelectObject(TPen(TColor::Black));

  if (status == ON)  {
    dc.FillRect(*this,*on_brush);
    dc.SetBkColor(on_color);
  } else {
    dc.FillRect(*this,*off_brush);
    dc.SetBkColor(off_color);
  }
  dc.SelectObject(TFont("Courier New", 15));
  dc.DrawText(title, -1, *this, DT_CENTER);

  dc.MoveTo(left, top);
  dc.LineTo(right, top);
  dc.LineTo(right, bottom);
  dc.LineTo(left, bottom);
  dc.LineTo(left,top);
}
```

## B.4 Specification of Token Generation

Tokens can be generated by either the presentation or the application program. The tokens that the presentation produces map that portion of the specification to the context-free grammar. Typically, for tokens generated by the presentation it is the graphical object's callback function which generates the token. An enumeration of the tokens and the objects which generate each of those tokens specifies the token generation.

Table 2 relates each of the 21 tokens of the context-free grammar to either the application program or objects from the presentation specification. The callback function of the graphical object is assumed to generate presentation tokens; in cases where this is not the case, the particular method is specified with the object. There is a single token denoted "Application Program," which is a token generated by the application program involving the current alarm conditions.

**Table 2: Token Generation**

| Token | Object/Method |
|---|---|
| SWITCH_PWR | `CM_SWITCH_POWER` |
| FISS_UP | `fission_chamber.req_height` |
| SET_MC1 | `control_rod_1.mag_current` |
| SET_MC2 | `control_rod_2.mag_current` |
| SET_MC3 | `control_rod_3.mag_current` |
| SAFETY1_UP | `control_rod_1.req_height` |
| SAFETY2_UP | `control_rod_2.req_height` |
| SAFETY3_UP | `control_rod_3.req_height` |
| SET_TRIP | `setpoint_temp` |
| AUTO_REG_ON | `reg_rod->auto_control` |
| AUTO_REG_OFF | `reg_rod->auto_control` |
| SET_POWER | `req_power` |
| AP_RCV_ALRM | Application Program |
| SIL_ALRM | `silence_button` |
| CLR_ALRM | `clear_button` |
| SAFETY1_DWN | `control_rod_1.req_height` |
| SAFETY2_DWN | `control_rod_2.req_height` |

**Table 2: Token Generation**

| Token | Object/Method |
|---|---|
| SAFETY3_DWN | `control_rod_3.req_height` |
| SHUT_DOWN | `CM_DATA_HISTORIES` |
| KEY_OUT | `MainReactorWindow` |

## B.5 Specification of Command Interpretation

There is an enumerated set of messages specified in the semantic component whose destinations are the application program or the presentation interpreter. The messages to the presentation interpreter instruct the interpreter to alter the on-screen graphical display in some way. Each of these messages can be mapped to the particular graphical object they manipulate and the method(s) called on that object.

The messages to the application program are enumerated below; this is simply an extraction from the semantic specification to make explicit this interface. The messages to the presentation interpreter are related to the graphical objects or methods of the presentation that they affect in Table 3, with the exception of SilenceAlarms which pertains to sound capabilities not modelled in this specification.

$$\mathit{MethodsToAP} == \{\mathit{NewPowerLevel}, \mathit{NewMagCurr}, \mathit{NewFissionHeight}, \mathit{SafetyRodHeight}, \mathit{NewTargetPower}, \mathit{SwitchAutoReg}, \mathit{NewTripPoint}\}$$

**Table 3: Presentation Interpreter**

| Command | Object/Method |
|---|---|
| *DisplayAlarms* | `alarm_array` |
| *SilenceAlarms* | n/a |
| *ClearAlarms* | `alarm_array` |