FORMAL SPECIFICATION: A SYSTEMATIC EVALUATION

Colleen DeJong
Matthew Gibble
John Knight
Luís Nakano

Department of Computer Science, University of Virginia

Computer Science Report No. CS-97-09
May 1997

FORMAL SPECIFICATION: A SYSTEMATIC EVALUATION

School of Engineering and Applied Science
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

Industrial practitioners require constant improvements in the software development process and the quality of the resulting product in order to satisfactorily build larger and more complex software systems. Academia praises formal specification techniques as a means to achieve these goals, yet formal specification has not been widely adopted by industry. The focus of this research is to study the disparity between industry and academia in their experience with formal specification methods.

During the specification of a significant software system, a control system for a nuclear reactor, it became clear that the use of formal specification methods had potential benefits, but there were practical requirements that were not being met. Previous evaluations of formal specification failed to identify many of these flaws and a new comprehensive approach based on the requirements of the current software development process is needed.

A comprehensive approach to evaluation was developed as part of this research. The evaluation method presented here does not examine theoretical qualities of language form and structure, rather it examines basic but vital practical issues involving the notation, tools, and methods for using them.

There were two objectives for this research:

- to identify these practical requirements and create a list of criteria for formal specification methods
- to evaluate several formal specification methods based on these criteria.

The criteria were systematically derived from current software development practice. This derivation links the criteria with specific activities in the software development

process and supports their inclusion in the evaluation. Using this set of criteria, an evaluation of three formal specification methods, Z, PVS, and statecharts, was conducted by developing and examining specifications for a preliminary version of the reactor control system.

Acknowledgments

It is a pleasure to acknowledge those who patiently described the requirements of the UVAR control system, including Tom Doyle, Bo Hosticka, Don Krause, and Bob Mulder. The participants of this study also deserve recognition: Roger Rydin, Reed Johnson, William Dixon, Jim Kelly, Bob Mulder, Tom Doyle, Emily West, Dale Newfield, Russ Haddleton, Craig Chaney, Cassie Trontoski, Meng Yin, Chenxi Wang.

This work was funded in part by the National Science Foundation under grant number CCR-9213427, in part by NASA under grant number NAG1-1123-FDP, and in part by the U.S. Nuclear Regulatory Commission under grant number NRC-04-94-093. This work was performed under the auspices of the U.S. Nuclear Regulatory Commission. The views expressed are those of the authors and do not necessarily reflect the position or policy of the U.S. Nuclear Regulatory Commission.

Table of Contents

1 Introduction

Industry is continuously building larger and more complex software systems. Despite the fact that a vast amount of software is currently in use and much more is being built, the processes used to build software and the quality of the results are generally poor. The cost and time needed to build software are quite unpredictable and usually high. The development of large systems often runs behind schedule, is over budget, and is either never completed or is completed unsatisfactorily. In order to address these problems, improvements in both the software process and the product quality are needed. These are the practical goals of industrial software practitioners:

- *Improve the process of software development*
  The software process needs to be well-defined, predictable, and faster. It should be broken into carefully delimited steps that will take predictable amounts of time. Rework of the specifications, design, and code should be minimized. As much of the process as possible should be automated.
- *Increase the quality of software produced*
  Software products need to be maintainable, dependable, testable, and verifiable.

1.1 Software Specification

Software specification is a critical element of the software development process. According to Clarke and Wing:

> *"The process of specification is the act of writing things down precisely. The main benefit in doing so is intangible--gaining a deeper understanding of*

> *the system being specified. It is through this specification process that developers uncover design flaws, inconsistencies, ambiguities, and incompleteness. A tangible by-product of this process, however, is an artifact, which itself can be formally analyzed, e.g., checked to be internally consistent or used to derive other properties of the specified system. The specification is a useful communication device between customer and designer, between designer and implementor, and between implementor and tester. It serves as a companion document to the system's source code, but at a high level of description* [CW96]."

A specification is where the requirements of the system are documented. This vision of the system is abstract, like an outline of a paper. However, it must be complete and specific enough that any system that satisfies these requirements is acceptable to the client. Specifications serve as a vehicle by which the desires of the clients are conveyed to the developers of the software system; they act as an informal contract. Therefore the specification must be used and understood by every person involved in the development of the system. Most errors in software are present already in the specification of the system. These errors, if found later in the development, cause rework, which extends development time and makes it unpredictable. If errors are not found, they may cause the system to fail during operation. From these findings, it may be inferred that a specification containing fewer errors would greatly improve both software process and software quality. In pursuit of this goal, academics have studied the specification phase to determine where improvements can be made.

Natural Language Specification

Currently, specifications are written largely in natural language. Natural language is understandable by clients, specifiers, and implementors. Everyone is accustomed to reading and writing natural language documents since that is taught at every level of education. The organization is familiar: a table of contents, chapters, sections, a glossary, and an index. The editing tools are mature; they support modification, searching, spell-checking, printing, and importing and export different formats. Natural language specification fits well into current development methods. However, natural language is informal, so it can have many interpretations. Natural language specifications are also prone to incompleteness and inconsistency because of the inability to do automated checking.

Formal Specification

Formal specification methods use mathematics-based principles to reason about computer hardware and software systems. The use of formal notations for specification combats the problem of varying interpretation by having formally defined syntax and semantics. They alleviate the ambiguity present in natural language specifications and curtail errors due to misunderstandings. They are usually based on basic discrete mathematics and, besides providing exact meanings for specifications written in the formal notation, make the specification amenable to automated checking and theorem proving. One benefit of using formal specification methods might be decreased work due to early identification of problems in the system while they are still inexpensive to correct. This improvement would result in a more predictable process that produces software with less defects.

However, formal notations are not useful if they cannot be understood. Different formal notations have differing degrees of understandability, but at least one study [Ard96] found them all relatively easy to learn. Since formal specification appears to be a promising route to obtaining better specifications, many specification notations and related tools have been introduced. The notations take on many different forms, including tabular, graphical, mathematical, and pseudo-code. Tools such as editors, animators, and verifiers have been built to manipulate these notations.

1.2 Research Focus

For many years academics have claimed that the addition of formal specification methods to the lifecycle will meet industrial goals of generating a better software process and increasing software quality, yet formal specification methods are still not widely used by commercial software companies. Industrial authors have expressed frustration in trying to incorporate formal technologies into practical software development for reasons such as the perception that they add lengthy stages to the process, require extensive personnel training, or are incompatible with other software packages.

The focus of this research is the disparity between academia and industry in their experience with formal specification. The goal is to determine what is needed to increase the benefits realized by industry from formal specification. The initial hypothesis for the lack of use of formal specification by industrial practitioners is that they were reluctant to change their current methods and overlooked the benefits that formal specification could provide. However, upon attempting to apply several formal specification methods to a significant application, a nuclear reactor control system, shortcomings were discovered quickly in the formal specification methods that impeded progress dramatically. Some examples of the difficulties faced were: (1) that the notations are not suited to describe all parts of the system; and (2) that tools are not available, too slow, or not compatible with other hardware or software used in the development.

Based on this experience, the following new hypothesis was formulated:

> *Formal specification techniques offer significant advantages over natural language but there are practical hurdles limiting their routine application. They must overcome these practical hurdles before their benefits can be realized.*

While the hurdles to which we refer are mundane, they are nevertheless vital to the success of formal specification in an industrial setting. Based on this new hypothesis, the primary goal of this research is to enumerate these practical hurdles.

Evaluations of formal specification have previously appeared in the literature; they are largely written by researchers and tend to praise formal specification methods. However further investigation found these studies lacking. While the criteria used for evaluation included important attributes, the terms were vague and ambiguous. They were often derived from the author’s experience with a particular project, with little substantiation that the list of criteria was complete or applicable to a range of projects. In addition to defects in the criteria themselves, evaluation was often entirely the opinion of the author. Evaluations of formal specification methods are important because they are used by software engineers in choosing an appropriate formal specification method for their project and by inventors of formal specification methods in the design of new notations, new tools, and in the improvement of existing ones. Previous evaluations of formal specification methods failed to find significant flaws, yet industrial experience and even use in a one sample application revealed shortcomings. This indicates the need for a new approach to evaluation.

1.3 Approach

The approach that was followed in this research project was to evaluate the hypothesis by experiment. The experiment consisted of applying formal methods to a single safety-critical system, and observing the benefits realized and difficulties encountered. In order to ensure that the observations captured the necessary information, a framework for evaluation was developed. More specifically, the experiment was as follows:

- A comprehensive set of criteria for evaluation was developed based on the entire software development lifecycle.
- A safety-critical application was studied and a set of requirements for part of a digital control system was developed.
- Formal specifications were written in three separate notations for the requirements.
- Based on the three specifications, the criteria were applied and conclusions drawn.

The safety-critical application that was studied in this research project was a simple control system for a research nuclear reactor. The research reactor is owned and operated by the University of Virginia and is a two megawatt pool reactor. The control system requirements that were used included the reactor system emergency shutdown mechanisms, the reactor alarm system, and the process whereby the reactor is started and brought up to operating power.

It is important to understand that the preparation of complete and accurate specifications from which high-quality implementations could be built was *not* a goal of this research. The goal was to evaluate formal specification. Although accuracy and completeness were of concern, no special effort was made to verify the specifications. Thus the specifications contained in the appendices do not necessarily document functionally complete or appropriate systems.

Many of the evaluation criteria that were used are subjective. This is inevitable because so much of the use of a specification involves people reading it. To ensure that the subjective assessments that were used were representative, the specifications were evaluated by both computer experts and domain experts.

Evaluation Criteria

The shortcomings in previous evaluations inspired the current objective: to evaluate formal specification in a systematic manner from industrial requirements. Although evaluation is the ultimate goal, the derivation of the criteria is as important as the criteria themselves. While a list of seemingly relevant criteria might appear useful, it is essential that the reason for the inclusion of the criteria in the list be documented. Without this, some important questions remain—question such as the following:

- *Why are these criteria important?*
- *Where did these criteria come from?*
- *Is this list complete?*

A defendable list of criteria can only be obtained from a clearly defined basis for evaluation. The proposed approach is to substantiate the criteria by deriving them from current practice.

The aim is to expose what is needed to put formal specification into industrial practice, so the criteria will not be concerned with theoretical issues like the orthogonality of features of the notation. There are three aspects to formal specification: (1) the notation itself; (2) the available tools; and (3) the method used to create a specification. Research often evaluates only the notation, as exemplified in the following quote by Hall, “At first, the productivity was lower, but this was attributed to learning to use various non-user-friendly tools and was not connected with the formal method itself” [Hal90]. However, a toolset that lacks usability can prohibit use of the notation. Similarly, a completed formal specification may provide many benefits, but writing it requires a development method. In this study, we have evaluated all three of these aspects of formal specification.

In order to be incorporated into industrial practice, formal specification methods must match current accomplishments. They must be consistent with current methods and compatible with current tools. While the methods used in industry are not formally based, they are reasonably well developed and understood. However, matching the accomplishments is not enough. The second aspect of the evaluation is to examine how formal specification will augment the current development practice of industry to build high quality software in a cost-effective manner.

Current practice breaks the development into lifecycle phases. Such a division focuses the developers’ attention on the tasks that must be completed. The specification should participate in every stage of the software lifecycle. Writing the specification is only one activity that involves the specification. While the process of composing the specification itself facilitates a better understanding of the system, the usefulness of the specification does not end there.

The specification is the primary vehicle of communication about the system between the many people involved in the software development and maintenance, such as the software engineers, the client, safety engineers, system engineers, and implementors. Many people will study it to understand the behavior of the system it is modeling, to check that it meets regulations, to implement the system, or to assess the impact of a potential modification. Each activity in the lifecycle will place different demands on the specification technology. In order to develop a complete set of criteria to evaluate specification technologies, the requirements of each person and activity must be considered. But the lifecycle alone is not sufficient to describe the current process of building software because the development is guided by management activities, such as scheduling and quality control. The lifecycle phases together with management activities characterizes current practice, therefore these will provide the basis for evaluation.

Such an examination of the demands of the software lifecycle identifies specific areas in which formal specification methods are lacking as well as areas in which formal specification can provide improvements over the current method. The intention here is to go beyond vague terms such as *readability* that have appeared in other studies to more precise criteria, such as:

> *Criterion: The ease with which a computer scientist can obtain answers to questions about implementation from a specification written by someone else in a formal notation*

Such a criterion is derived from a demonstrated need for such a person to perform such a task during software development. The primary benefit to formulating criteria this way is that the criteria are associated with specific lifecycle activities. This demonstrates that the criteria are relevant and allows practitioners to choose the method that meets their needs.

It is also important to consider the fact that projects have diverse goals. For some speed to market is most important, while for others dependability is the utmost concern. Development environments also vary. The criteria generated here were not particular to a certain set of needs, rather they addressed all aspects of improving software. However, when evaluating the usefulness of a formal specification method for a specific project, the goals of that project affect the importance of the criteria. Therefore, although one formal specification method might not meet certain criteria, those criteria may be unimportant to the goals of the project, so the method would still be a good choice.

It is unlikely that one formal specification method is best for every project, so the purpose of this research is not to name a winner. Rather it is to provide an approach for generating a comprehensive set of criteria that can be refined to fit a particular project or used to identify flaws in formal specification methods.

Evaluation Process

Once the criteria were formulated, three specification methods, Z, statecharts, and PVS, were evaluated. Specifications for a preliminary version of the nuclear reactor control system were developed and examined. Due to resource constraints, this evaluation is not ideal, however there are strong indications that formal specification can provide improvements in the software development process and resulting product once practical requirements are met.

1.4 Contents Summary

In chapter two, the nuclear reactor control system is summarized. An overview of each of the notations evaluated in this study is given in chapter three. Research and industrial projects involving formal specification methods are described in chapter four. Chapter five describes in detail the basis from which the criteria that will be used for evaluating formal specification methods will be derived. Since the basis for evaluation is current practice, in chapter six, the demands placed on the specification notation, toolset, and method for writing a specification in this notation during each activity in the software development process are explored. The list of criteria derived from this in-depth examination of the software lifecycle are enumerated in chapter seven. The method used in this study to evaluate three formal specification methods based on these criteria is described in chapter eight and the results of this evaluation are recorded in chapter nine. In the final chapter, conclusions are presented.

2 Application Summary: University of Virginia Reactor

The safety-critical system that was the subject of the experiment performed in this research is described in this section. This description is informal, and it is intended to provide a general understanding of what the reactor system is like.

2.1 System Overview

The Department of Mechanical, Aerospace, and Nuclear Engineering of the University of Virginia operates a research nuclear reactor. The reactor is described in “The Nuclear Reactor Facility Tour Information Booklet”, as follows with word changes for brevity:

> *“The University of Virginia Reactor (UVAR) is a nuclear research reactor, operated by the Department of Mechanical, Aerospace, and Nuclear Engineering. It 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 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”*[UVAR].

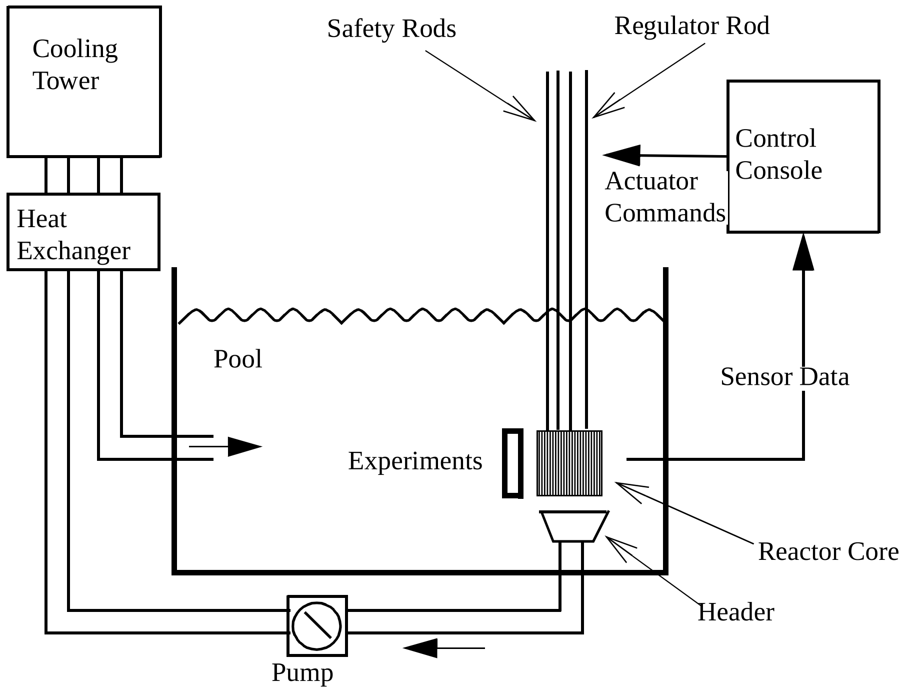


Figure 1: The University of Virginia reactor system.

Despite being a small research reactor and not a commercial power reactor, the UVAR is a complex system facing many of the same issues as a full-scale reactor.

The UVAR is a light-water cooled, moderated, and shielded “pool” reactor. A diagram of the primary components of the UVAR system is shown in Fig. 1. At the center of the reactor is the *reactor core*, an assembly which contains fuel elements, control rod fuel elements, graphite reflector elements, and possibly in-core experiments. The reactor core is suspended from the top of the reactor pool and rests on an 8x8 grid-plate under approximately 22 feet of water. 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 coarse control and safety. Shim rods are suspended magnetically by electromagnets coupled to their drive mechanisms. In case the reactor has to be turned off immediately either by the operator or by the reactor protection system, the electromagnets are powered down and the shim rods drop into the core due to gravity, thus shutting down the reactor. This usually occurs in less than one second. This shutdown process is referred to as a *scram*. The fourth rod, designated as *regulating rod*, is fixed to its drive mechanism, and thus does not participate on a scram, but is used for fine-grain power control of the reactor to compensate for small changes in reactivity associated with normal operations [UvarSC].

The power level reported for this class of reactor corresponds to thermal power production. Power level is proportional to the neutron population. 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 across 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 cooling loop. The header can be lowered or raised, to allow the reactor to dissipate heat in natural convection mode (header lowered) or to direct water flow through the core (header raised to the grid plate). An air line allows the operator to raise the header by injection of compressed air into the header, thus displacing water and increasing the buoyancy of the header. This air line also has valves that allow the operator to bring the air pressure on that air line to the atmospheric pressure and to close the line to prevent air inside it to leave. When the pressure in the air line is equal to the atmospheric and the header is up, water flow through the core keeps the header in place. If the flow of water through the core is reduced below a certain threshold, the header will fall by gravity. If the valve on the air line is closed when the header falls down an increase in air pressure occurs on the air line. In these circumstances, a pressure sensor in this air line signals the pressure increase and is used to determine that the header has fallen.

Since this reactor uses light-water (as opposed to heavy-water used on the primary cooling loop of some power reactors), and this water is always kept at a temperature far from the boiling point, there is no need for a pressurized vessel to prevent radiation leakage. Water can be added to the pool as natural evaporation requires, and this water is merely demineralized tap water. A cooling tower located on the roof of the facility

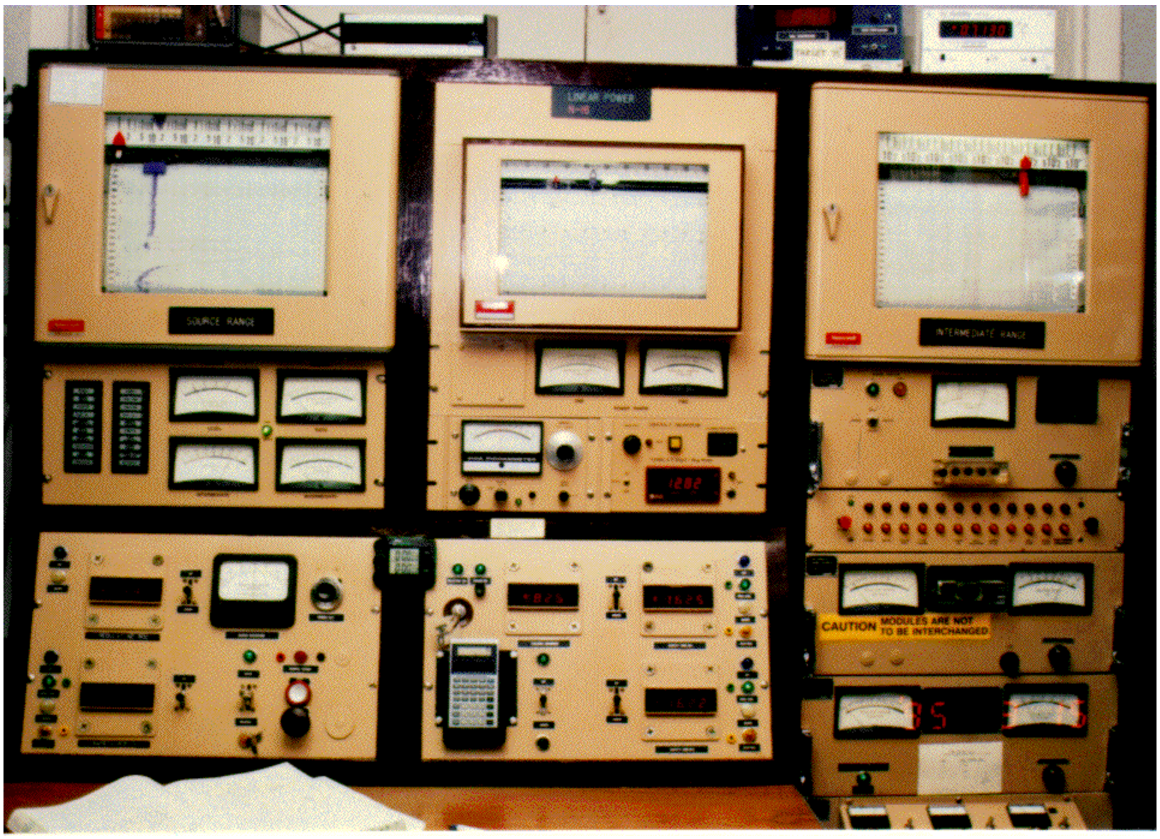


Figure 2: Partial view of the control pannel of the UVAR.

exhausts the heat and the cooled primary water is returned to the pool [UVAR].

Control System

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 with all electronic displays is being designed for the UVAR and is currently in the specification stage. Fig. 2 shows an overview of part of the current control pannel.

This nuclear reactor control system can be subdivided into smaller subsystems, for the sake of understanding. The main subsystems are: the scram logic, responsible for generating the signal that scrams the reactor, alarms that will call attention from the operator, and interlocks that prevent the shim rods to be moved if certain start-up conditions are not met.

Instrument	Analog/ Boolean	Quantity	Units
Pool Water-Temperature Monitor	analog	pool water temperature	°F
Pool Water-Level Monitor (two sensors)	analog	height of the water in the pool	’”
	boolean	above/below or at 19’3”	
Power-Level Sensor (two identical sensors)	analog	power output	MW
Pool-Water Conductivity	analog	water conductivity in demineralizer room	mhos/cm
Reactor Period (two channels)	analog	reactor period	s
Gamma-Radiation Monitor	analog	gamma radiation in core	mR/h
Constant Air Monitor	analog	radiation level in the reactor room	mR/h
Airborne Effluents/Duct Monitor	analog	radiation from airborne effluents	mR/h
Area-Radiation Monitor	analog	radiation levels	mR/h
Core Temperature Differential (two sensors)	analog	temperature differential between the water leaving the core and the water entering the core	°F
			°C
Differential-Pressure Across Orifice	analog	indirect measure of water flow across the core	atm
Air To Header	boolean	pressure on the airline is above/below or at 2 psi above the atmospheric pressure	

Table 1: Sensor signals provided to the control system

Core Sensor Signals

Several sensors are available to the control system. The main sensor signals, corresponding quantities that are measured and types are described in table 1. In this table, boolean sensors are the ones that provide only two possible values for a condition, with the analog sensors indicating values over a range of continuous values.

Units are described by their abbreviation: °F for degrees Fahrenheit,’” for feet and inches, MW for megawatts, mhos/cm for mhos per centimeter (1mhos=1 Ampere/Volt, the inverse of 1 Ohms, indicating electrical conductivity instead of electrical resistance), s for seconds, mR/h for miliroetgens per hour (radiation unit used to measure gamma and X-ray

Actuator	Description
Shim Rods	scrammable, magnetically suspended by its driver, provide coarse-grain control of the reactor power level
Regulating Rod	unscramble, physically connected to its driver, provide fine-grain control of the reactor power level
Primary Pump Header	responsible for directing water flow through the core
Secondary Pump	produces water flow in secondary loop. If this pump if off, heat exchange efficiency is significantly decreased
Manual Scram Button	emergency button to generate a scram signal and stop reactor
Water Cleanup System	responsible for removing minerals from water to keep it adequate for operation
Start-up Interlock	interlocking mechanism that prevents reactor start-up if a minimum of two neutron counts per second is not available

Table 2: Actuators present in the system

radiations), °C for degrees Celsius and atm for atmospheres (pressure unit).

A few of the measures deserve a closer look and further explanation. Power output corresponds to the thermal power produced by the reactor, and the reactor period is a quantity that indicates the period of time that is required for the neutron population to double. The differential pressure across orifice is an indirect way to provide a estimate for water flow inside the core, based on fluid dynamics equations.

Actuators

Some of the actuators present on the system are described on table 2 Although these are the most relevant actuators, they are not the only ones. Some of them are connected to special sensors, used to determine their position. In particular, it is important to have a precise description of the position of the shim rods, since they are the basic mechanism preventing the core to reach too high a power level. They are also used to prevent the reactor from being started and can only be deployed if the start-up interlock conditions are satisfied.

Shim Rods

There are three shim rods that are raised and lowered by using their drivers. Lowering the rods decreases the speed of the reaction, while raising the rods will increase the speed of the reaction. The drivers contain electromagnets that when in contact with the rods and electrically powered can lift and lower the shim rods in and out of the core. When a scram occurs, the power to the magnets is automatically shut off and the rods drop to their lowest position in the core. A set of four lamps per rod indicate possible positions for the rods and their driver mechanism. The following lamps indicate the state of a rod and its driver:

- Up - the driver is at its highest position (with or without the rod)
- Down - the driver is at its lowest position
- Seated - the rod is at its lowest position (the driver need not be down)
- Magnetically engaged - the driver is in physical contact with the rod (the magnet does not have to be on for the driver to be magnetically engaged)

2.2 Protection System

Scram Signal Generation Logic

The UVAR has an automatic system to shut down neutron production if undesired conditions occur. This mechanism is implemented by solid state circuits and works by verifying 12 different conditions simultaneously. If any of the conditions does not hold, a scram signal is generated and the safety rods are inserted into the core, not only stopping neutron production but also reducing the neutron population to near zero in a short period of time.

This scram signal generation logic is one of our targets in this specification effort. Although it is not extremely complex, it does provide an interesting non-trivial example from the real world. The term *scram the reactor* will represent the generation of the scram signal responsible for turning off the reactor. Also, when the reactor is scrammed, it will not come back to operating state without the operator pressing the reset scram button and the conditions that caused the scram disappearing.

If any of the following conditions is met, the reactor is scrammed:

- power level is above 250 kW and the reactor is operating in natural convection mode.
- power level is above 2.5 MW and the reactor is operating in forced convection mode
- during forced convection operation, the pressure in the air line that raises the flow header goes 2 psi above the atmospheric pressure
- flow across the core is below 960 gal/min and the reactor is in forced convection mode
- pressure in the air line that raises the primary pump head is 2 psi above the atmospheric and the range switch #2 is switched from 0.2 MW to 2MW position
- start button for the primary pump is pressed
- primary pump voltage goes from on to off
- header is down and the primary pump is turned on
- radiation level measured on bridge above the pool is higher than 30 mR/h
- radiation level at ground level is higher than 2 mR/h
- pool water level is at or below 19’3”
- pool water temperature is above 108 ^{O}F
- reactor period is shorter than 3.3 s
- truck door is opened
- escape hatch door is opened
- key switch at the control panel is removed
- scram button by the back door is pressed
- scram button by the room door is pressed
- scram button on the control panel is pressed
- any of the four evacuation alarms is pressed
- reactor was already in scram condition, keep it on scram condition until the scram reset button is pressed.

Alarms

The UVAR has also a set of alarms that go off when attention is required from the

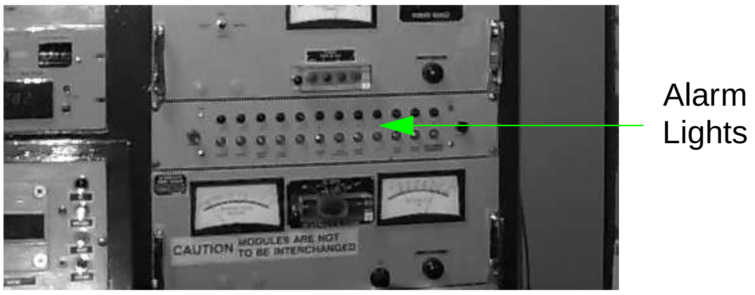


Figure 3: Lateral panel with alarm lights.

operator to verify some condition. The states related to the alarms are not dangerous enough to justify a scram, but they require the operator to perform some action.

All alarms but the scram alarm are sounded for 2 minutes, after which time their sound goes off. The sound can also be silenced by the operator, by pressing a button. The scram alarm can only be silenced by the operator.

Visual indication of the alarms is provided by two rows of lights. The first row, composed of red lights, indicates the current status of each alarm, on or off. The second row, composed of yellow lights, keeps one light on for each alarm that has gone off until the operator resets the alarm. However, the yellow light does not go off when the operator resets the alarm if the corresponding red light is still on. Fig. 3 shows the lateral panel were the alarm lights are located.

The alarms are:

- Reactor is in scram condition.
- Automatic control of regulating rod is lost.
- Area radiation or argon monitor indicates high level.
- Gamma radiation measure is too high.
- Spare (not used)
- Constant air monitor indicates high level.
- Heat exchanger room door is open.
- Demineralizer room door is open.
- Core differential temperature is too high.
- Demineralizer room water conductivity measure is higher than 2 μmhos/cm.

- Secondary pump is off while the reactor is operating in high power mode.
- Hot thimble temperature.

2.3 Minimal Start-Up Sequence

A very specific procedure, hereafter referred to as the start-up sequence, has to be performed in order to bring the power level of the reactor from nearly zero to an operating condition without entering a dangerous state. Before the reactor is started for the first time, many tests, checks, and logging activities are performed. Most of these correspond to bookkeeping (registering values for certain variables in log books, verifying that a variable is within acceptable range, registering in the log book that this check has been completed, etc.). Extensive tests are performed to ensure that each scram condition, if satisfied, does indeed generate a scram. These tests involve turning on and off each piece of equipment. Such bookkeeping activities and equipment tests are tedious and will not be described in full. Instead, a token test sequence will be used:

1. Reset reactor scram.
2. Admit air to header until it raises to the grid plate.
3. Verify that a scram was generated; if not, stop the procedure and call the senior operator.
4. Bleed off air from the header mechanism, making pressure in the air line to the header equal to the atmospheric.
5. Close the valve on the air line to the header.
6. Reset reactor scram.
7. Start the primary pump.
8. Verify that a scram was generated; if not, stop the procedure and call the senior operator.
9. Reset the scram.
10. Turn off the pump.
11. Verify that a scram was generated; if not, stop the procedure and call the senior operator.
12. Reset the scram.

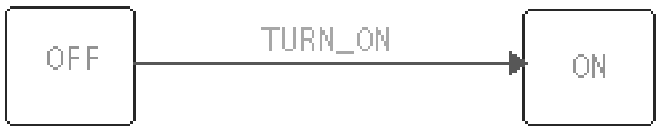
The start-up sequence described here details the steps needed bring the reactor into an operating condition after all the tests have been completed. There are two operating conditions, high power and low power. The steps that are necessary for bringing the reactor to high power, but not for low power, are indicated with an asterisk. These operations have to be performed in sequence, as they specify changes from one state to another. The sequence of events that is specified for start-up is:

1. Reset reactor scram.
2. *Admit air to header until it raises to the grid plate.
3. *Verify that a scram was generated; if not, stop the procedure and call the senior operator.
4. *Start the primary and secondary pumps.
5. *Bleed off air from header mechanism, making pressure in the air line to the header equal to the atmospheric.
6. *Close valve on the air line to the header.
7. *Reset reactor scram.
8. *Check that the header remains up.
9. Bring all the shim rod drivers to the lowest position.
10. Verify that the seated lamps are on for each individual rod; if not, stop the procedure and call the senior operator.
11. Verify that the magnetically engage lamp corresponding to each of them is on; if not, stop the procedure and call the senior operator.
12. Turn on the magnetic currents on the shim rod drivers.
13. Raise the shim rod drivers
14. Verify that the seated position indicator lamp and the rod down lamp indicator go off; if not, stop the procedure and call the senior operator.
15. Request power level from operator and start control algorithm for reactor.

3 Notation Summaries

3.1 The Statecharts Notation

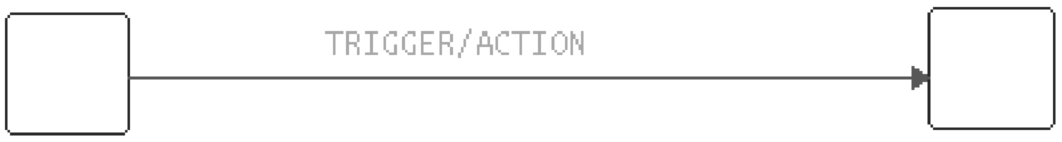
Statecharts is a graphical specification language introduced by David Harel in *Statecharts: A Visual Formalism for Complex Systems* [Har87]. The STATEMATE family of tools implements this notation and provides capabilities such as static checking and animation. More information about STATEMATE can be found in [STM]. Statecharts is based on the conventional state machine model in which systems are described naturally in state-transition diagrams. States are indicated by boxes and transitions between the states are indicated by arrows. The name of a state appears in its box; names are optional.



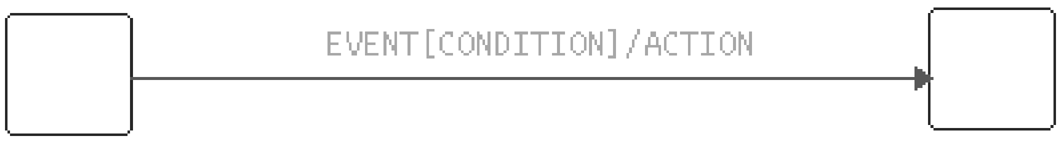
Conventional state machines do not scale well; the number of states grows uncontrollably and the diagram becomes unstructured and incomprehensible. Statecharts is an extension of state-transition diagrams that can deal with more complex systems. In particular it is intended to address a class of problems that is very difficult to specify, reactive systems. The complexity of reactive systems is handled by Statecharts through three principles: communication, concurrency, and hierarchy.

Communication: Transitions

Transition labels consist of a trigger and an action, separated by a slash.



Both parts of the transition label are optional. A trigger is made up of events, which are instantaneous, and conditions enclosed in brackets, which are continuous. If the event is signaled and the condition is true, then the transition is taken and the action occurs.

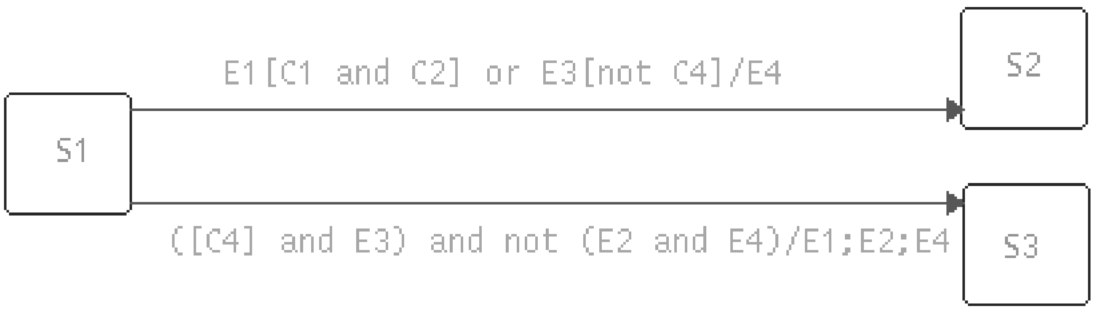


The trigger can consist of events and conditions connected with "`and`", "`or`", and "`not`". An action might be signaling an event or assigning a new value to a variable. Multiple actions are separated by semicolons. Events, conditions, and actions provide communication for the system because they are broadcast throughout.

The following example demonstrates the use of complex transition labels. Table 3 contains the current state of the system. The diagram shows the system itself.

Events	Signalled?	Conditions	True/False	States	Currently In
E1	Yes	C1	T	S1	Yes
E2	No	C2	F	S2	No
E3	Yes	C3	T	S3	No
E4	No	C4	F	--	--

Table 3: Initial state description of the system


E1[C1 and C2] or E3[not C4]/E4
S1
S2
S3
([C4] and E3) and not (E2 and E4)/E1;E2;E4

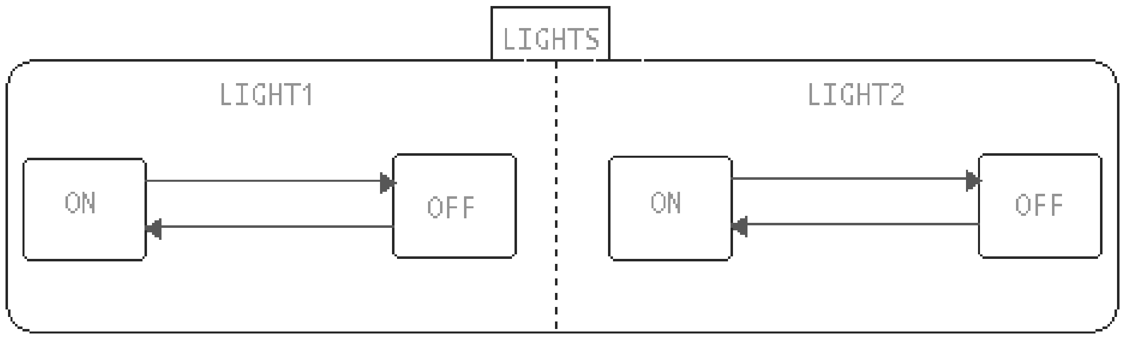
Because of the values that the events and conditions take, the state entered next will be `S2` and the state description of the system will be that shown in Table 4.

Events	Signalled?	Conditions	True/False	States	Currently In
E1	No	C1	T	S1	No
E2	No	C2	F	S2	Yes
E3	No	C3	T	S3	No
E4	Yes	C4	F	--	--

Table 4: Next state description of the system

Concurrency: AND/OR States

States `S1`, `S2`, and `S3` in the example above were OR states. The system must be in exactly one of these states at a time: `S1` or `S2` or `S3`. In Statecharts, there are also AND states which can be identified by the dotted line that partitions them. These indicate parallel or independent activities. For example, if there are two lights in the system, they can be described with an `AND` state.



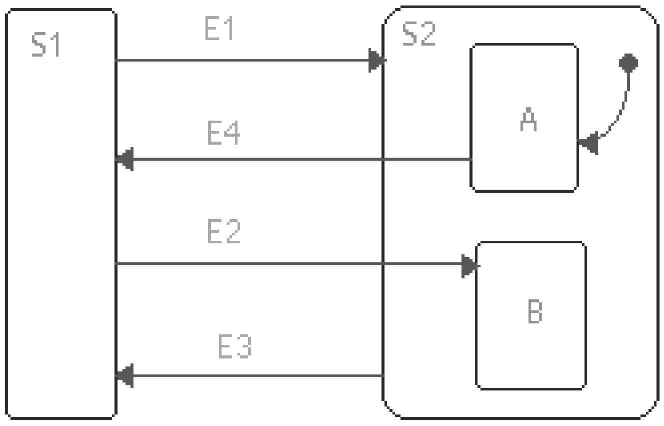
When the system is in the `Lights` state, it is in both `Light1` and `Light2`. The two lights can be turned `on` and `off` independently of each other. Representing this same relationship with OR states would require different states for `Light1 on` with `Light2 off`, `Light1 on` with `Light2 on`, etc. An AND state can contain two or more sections. AND and OR states can be combined freely within the statechart specification for a system.

Hierarchy: Levels of States

As the previous figure demonstrates, states can be nested within another state. This nesting creates a hierarchy of states that can be divided into levels. `Light1` and `Light2` are at a higher level than the `On` and `Off` states. It is not a problem that there are two states named `On` because they can be referred to as `Light1.On` and `Light2.On`. The same is true with the two `Off` states. Nesting can be arbitrarily deep and transitions can go between states of any level. When the system enters a state at one level, it also enters all the levels in its branch of the hierarchy. For example, when the state `Lights` becomes active, `Light1` and `Light2` are also active because `Lights` is an AND state. `Light1` has two substates, `On` and `Off`, and one of these must be active. `Light2` must also be either `On` or `Off`. If the state `Lights` were part of a larger state, perhaps `System`, it would also become active. Grouping states in a hierarchical manner provides structure and modularity in the model and allows it to be viewed at different levels of detail.

Default Entrances

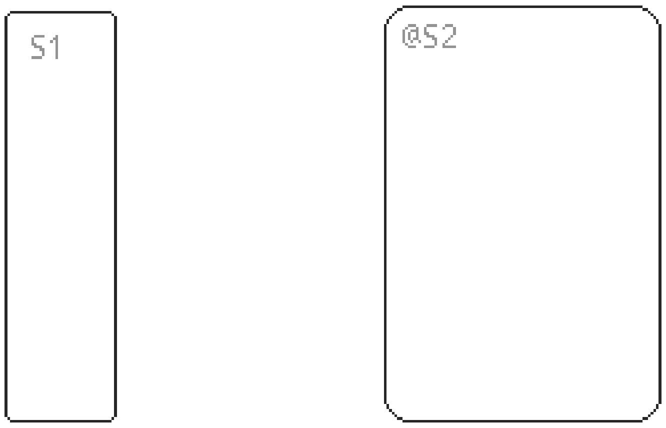
In the diagram below, `S1` and `S2` are at the same level and states `A` and `B` are nested within `S2`. From `S1`, transitions can go to a state at the same level, as the one labelled `E1` does, or to a different level, as the one labelled `E2` does.



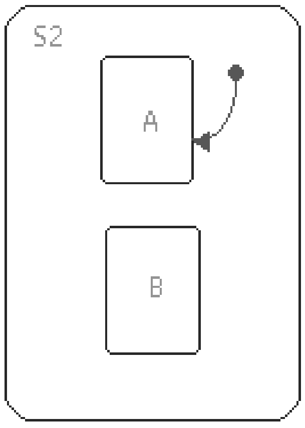
When the transition triggered by the event `E2` is taken, it is clear that `B` within `S2` is entered. However, when the transition triggered by the event `E1` is taken, `S2` is entered, but the arrow does not indicate which one of its substates will be entered (one of them must be). Which state will be entered is decided by the default transition, the arrow in the diagram originating from a dot and pointing to `A`. Therefore, upon the transition triggered by `E1`, the state `A` within `S2` will be entered. The transition labelled `E4` will only be taken if state `S2.A` is active when the event `E4` is signalled because the arrow originates from `A`. It will not be taken if `S2.B` is active when `E4` is signalled. The transition labelled by `E3` will be taken when `S2` is active and `E3` is signalled, regardless of whether `A` or `B` is active. In the case that either transition is taken, `S2` and all of its substates will be exited (become inactive).

Decluttering

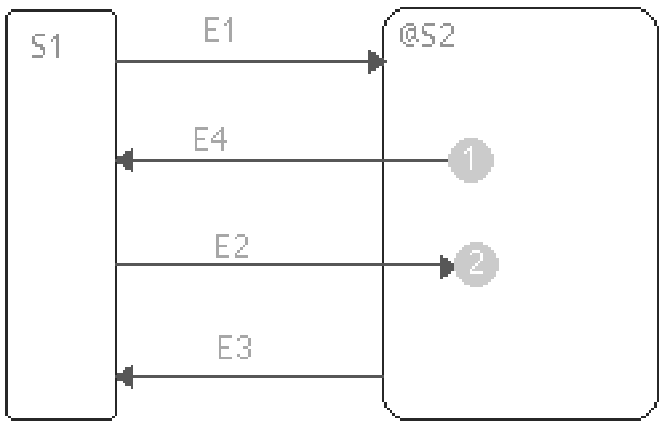
The statechart describing a system can become quite large and deeply nested. The notation allows the chart to be stored in multiple files through a method called decluttering. In decluttering all the substates of a selected state are put in another file. To designate that its contents are in another file, the name of the state has an "`@`" in front of it. For example, in the previous figure, `S2` might be chosen for decluttering, so `A` and `B` would be put in a new file named `S2` and the name of the state would now be "`@S2`." Here is the new figure, with the arrows omitted for the moment.



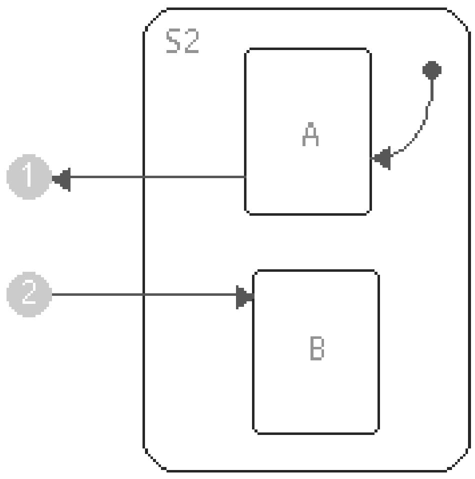
In file `S2` are the contents of the state `S2`:



Now the arrows must be added. The transitions labelled `E1` and `E3` are no problem since they connect `S2` and `S2`, however the transitions labelled `E2` and `E4` connect states which are now in two different files. To deal with this diagram connectors are used. They are shaded ovals containing a label. One or more arrows may point to a connector in one file and away from one with the same name in another file. This is demonstrated by adding the transitions to the previous figures:



And in file `S2`:



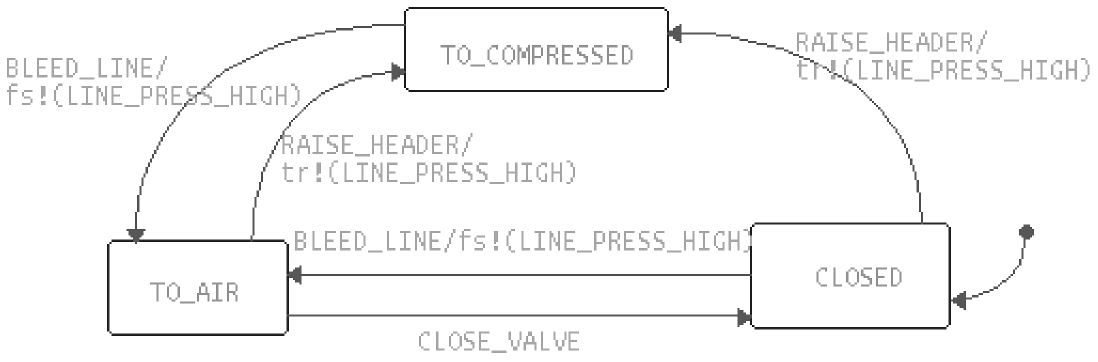
In this example, the connectors are labelled with numbers, but they can also be labelled with words.

Built-in Commands

Two built-in actions are `make_true(Condition)` and `make_false(Condition)` which are abbreviated as:

- `tr!(Condition)`
- `fs!(Condition)`

The use of these built-in commands is illustrated in the following diagram. The action taken when a transition is followed is to set the value of the condition Line_Press_High to true or false.



3.2 The PVS Notation

PVS (Prototype Verification System) is a general purpose verification system developed by SRI and available by anonymous ftp. It has an expressive model-based specification language derived from classical higher order logic that resembles pseudocode. It also provides a type checker and powerful interactive theorem prover. For information beyond what is presented here, see [But93, But96, PVSweb].

The structuring mechanism of the specification language is the theory. It serves to modularize the system and one theory can be imported into another for use by that theory. The syntax of the theory is as follows.

```
theory_name : THEORY
BEGIN
      % the theory body goes here
END theory_name
```

Comments start with `%` and continue to the end of the line. The body of a theory might consist of a list of imported theories, definitions, and functions.

```
theory_name  :  THEORY
BEGIN

IMPORTING sub_theory1
IMPORTING sub_theory2

      %definitions
      %functions

END  theory_name
```

Notation

Common mathematical notation is supported. Symbols not available on the keyboard are written as words, for instance `FORALL`, `EXISTS`, `IFF`, `IMPLIES`, `OR`, `AND`, and `NOT`. The if-then construct is provided, with the following syntax.

```
IF boolean_expression
THEN statement1
ELSE statement2
ENDIF
```

Additionally, there is a case construct.

```
CASES variable_name OF
      :     action1,
      :     action2
ENDCASES
```

Definitions

The specification language is strongly typed. Besides several built-in types, it allows user-defined types and provides type constructors, such as records and enumerated types. Built-in types include boolean, integer, natural, and positive natural. In an airplane, the number of rows might be defined as a positive natural number:

```
nrows :     posnat
```

Then a specific row would be of the following type:

```
row   :     = {n:posnat | 1<n and n<=nrows}
```

In this definition, row is positive natural number with the constraint that the row must be between `1` and `nrows`. A variable of type `row` can then be defined and given a value.

```
r     :     row
r     :=    3
```

The possible values of a boolean variable are TRUE and FALSE. This is a built-in enumerated type with two possible values. Other enumerated types can be defined with `{ }`.

```
header_status     :     TYPE = {UP, DOWN}
pump_status       :     TYPE = {ON, OFF}
line_valve_status :     TYPE = {CLOSED, TO_AIR, TO_COMPRESSED}
pressure_status   :     TYPE = {HIGH, NORMAL}
```

A variable of one of these enumerated types can be defined and given a value the same way as above.

```
pump              :     pump_status
pump              :=    ON
```

Records are defined with `[# #]` and can be nested.

```
cooling_system_status   :     TYPE =
      [#    %RECORD
            header      :      header_status,
            pump        :      pump_status,
            sec_pump    :      pump_status,
            line_valve  :      line_valve_status,
            line_pressure:     pressure_status
      #]
```

Variables of record types can be defined as above.

```
cool  :     cooling_system_status
```

Values can be assigned to all fields at once using `(# #)` or to a subset of the fields using the keyword `WITH`.

```
cool  :=    (#    header := UP,
                  pump := ON,
                  sec_pump := OFF,
                  line_valve := CLOSED,
                  line_pressure := NORMAL
            #)

cool := cool WITH [pump := OFF, line_press := HIGH]
```

The value of a field in a record is accessed as `field(record)`.

```
IF (header(cool) = DOWN AND   pump(cool) = ON)
THEN scram
ENDIF
```

Functions

In this specification notation all functions return values. The syntax of a function is `function_name (parameter_list) : return_type_name = function_definition`.

```
raise_header(cool : cooling_system_status):
       cooling_system_status =WITH [header := UP,
                                    line_valve := TO_COMPRESSED,
                                    line_pressure := HIGH ]
```

`Raise_header` takes a variable of type `cooling_system_status` and returns the new cooling system status after raising the header. `Cooling_system_status` is a record. Since the function `raise_header` only sets some of the fields and does not want to change the others, the assignment is done using `WITH`.

3.3 The Z Notation

Z (pronounced zed) was developed at Oxford University and is based on first order logic and set theory. It specifies the functionality of the system by describing pre-conditions, post-conditions, and invariants. Many tools support this notation, providing capabilities including editing, type checking, and theorem proving. More information on Z can be found in [Dil94].

Mathematical Notation

Z uses conventional mathematical notation from logic and set theory.

$\Rightarrow$	Implies
$\Leftrightarrow$	If and only if
$\wedge$	Logical And
$\vee$	Logical Or
$\neg$	Logical Not

Existential and universal quantifiers are also supported by Z. The general form for the use of a universal quantifier is:

$\forall\ VariableList \mid Predicate1 \bullet Predicate2.$

$\forall\ a : \mathbb{N} \mid (MinPosition \leq a \leq MaxPosition) \bullet Reactivity(a) \geq 0$

This can be read as, for all natural numbers a, such that a is between *MinPosition* and *MaxPosition*, it is the case that $Reactivity(a) \geq 0$.

A partial injective function is designated in Z by the symbol $\rightarrowtail\!\!\!\!\!\!+\,$. A function is a relation that maps elements of one set, the domain, to at most one element of another set, the range. Calling a function partial means that it does not have to be defined for every element in the domain. For example, the square root function is a partial function if we want the outcome to be an integer; 4, 9, and 16 have integer answers, but the numbers in between do not, so they are not defined. Injective means that each of the elements in the domain, for which the function is defined, map to different elements of the range.

The natural numbers are designated by the symbol $\mathbb{N}$ and the integers are designated by the symbol $\mathbb{Z}$. Other types can be defined as follows.

OperationStatus	==	{*Idle, Operating*}
Switch	==	{*On, Off*}

OperationStatus and *Switch* are new types. Variables of type *OperationStatus* can have a value of *Idle* or *Operating*. Variables of type *Switch* can have a value of *On* or *Off*.

Variable Declarations

Variables are declared with the syntax *VariableName* : *Type*.

$Step \quad : \quad \mathbb{N}$

This defines *Step* as a natural number. Variable identifiers are decorated with symbols such as a prime, a question mark, or an exclamation mark, to indicate different uses. If the value of *Step* is going to be changed by an operation, then it must be defined without decoration to indicate its initial value and decorated with a prime to indicate the state of the variable after the operation.

$Step' \quad : \quad \mathbb{N}$

For example, the pre-condition of the operation might be that *Step* is greater than zero and the post condition be that *Step* has been incremented by one. The pre-condition is stated using *Step* without decoration. The post-condition is stated using *Step*’ to indicate that it is the state after the operation.

$Step > 0$

$Step' = Step + 1$

Input variables are designated with a question mark. Their type is given after the colon.

$NumberInput? : \mathbb{N}$

Output variables are designated with an exclamation mark. Their type also given after the colon.

$SquareRoot! : \mathbb{N}$

The square root function would then be written as

$NumberInput? \geq 0 \quad \wedge \quad SquareRoot!^2 = NumberInput? \quad \wedge \quad SquareRoot! \geq 0$

This says that the input must be greater than or equal to zero and the square of the square root must equal the input and the square root must be greater than or equal to zero.

Schemas

The basic structuring mechanism in Z is the schema.

```
┌─── SchemaName ──────────────────────────────────
│ IncludedSchemas
│ VariableDeclarations
├──────────────
│ ScemaBody
└─────────────────────────────────────────────────
```

It can be used to define types, initialize states, and describe functions that change states. Schemas can be included in other schemas. The schema body is where the pre-conditions, post-conditions, and invariants can be specified.

Type Definition Schemas

```
┌─── Pump ────────────────────────────────────────
│ PumpSwitch    :     Switch
│ Voltage       :     ℕ
├──────────────
│ Voltage > 0   ⇒     PumpSwitch = On
└─────────────────────────────────────────────────
```

This schema defines a type named *Pump* as consisting of a *PumpSwitch* and a *Voltage*. The *PumpSwitch* is of type *Switch* which has been defined previously. The *Voltage* is a natural number. In the body of the schema is the invariant which states that, if the *Voltage* is greater than zero, this implies that the *PumpSwitch* is *On*.

Using Schemas Inside a Schema

Once the *Pump* type is defined, it can be used in other schemas just as a built-in type would be.

```
┌──── Reactor ────────────────────────────────────────
│ PrimaryPump          :       Pump
│ SecondaryPump        :       Pump
├──────────────
│ Scram = Scrammed     ⇒       ReactorStatus = Idle
└─────────────────────────────────────────────────────
```

This schema defines a type called *Reactor*. Like *Pump*, it contains variable definitions and an invariant in the body.

State Changing Schemas

The following schema is a state changing schema. This is evident by the appearance of a variable decorated with a prime. This means the state of that variable will be changed.

```
┌──── TurnOnPump ─────────────────────────────────────
│ PrimaryPump                   :       Pump
│ PrimaryPump'                  :       Pump
├──────────────
│ PrimaryPump.PumpSwitch'       =       On
└─────────────────────────────────────────────────────
```

In this schema, *PrimaryPump* is the variable that was changed. *PrimaryPump* is of type *Pump*. In the *Pump* schema above it is defined to have two elements, *PumpSwitch* and *Voltage*. The schema *TurnOnPump* changes the state of the *PumpSwitch* of the *PrimaryPump*.

The Δ Schema

It becomes tedious to write two declarations for every variable that will be changed, one without decoration and one with a prime, therefore a schema can be defined that contains only those two statements. Such schemas are named with a delta symbol.

$\Delta Pump$

$PrimaryPump$	:	$Pump$
$PrimaryPump'$	:	$Pump$

Once a delta schema is defined, $\Delta Pump$ can be used in place of the two declarations.

Schemas that Set Initial Conditions

An initializing schema can usually be identified by the name of the schema, as is the case in the following example. However, it is also clear that it initializes the state because it is a state changing schema (the delta has replaced the two declarations) and it has no pre-conditions. Thus it unconditionally sets the state.

$ReactorInit$

$\Delta Reactor$

$\forall a \mid a \in Pumps \bullet a.PumpSwitch'$	=	Off
$\forall a \mid a \in Pumps \bullet a.Voltage'$	=	0

4 Related Work

The number of formal specification notations and related toolsets is blossoming. Each of these provides a different set of capabilities, degree of formalism, and level of abstraction. The role of the specification is not clearly defined, so specification methods include varying amounts of support for general software development, requirements elicitation, design, and code generation. This makes it difficult to determine what should be included in a discussion of formal specification methods. The three notations that will be evaluated in this study, statecharts, PVS, and Z, are representative of three major types of notation; it is by no means an exhaustive list. An overview is given of the work being done involving these three notations. Following this are descriptions of selected industrial projects and research projects that utilized formal specification. Finally there is discussion of previous evaluations of formal specification methods.

4.1 State of the Art

Statecharts

Statecharts is a graphical specification language introduced by Harel [Har87, Har88]. Harel is affiliated with iLogix which commercially markets the STATEMATE [iLo87, iLo90, STM, STMweb] family of tools which include an editor for statecharts, version control, simulation, and support for structured analysis. Each tool can generate code from the statecharts in a language such as C, Ada, or VHDL. Other notations and tools also based on statecharts are Modecharts, BetterMate, SpecCharts, and RSML.

The following is a list of Statecharts users and usages. Information found in published papers is cited below. Information on the remaining industrial projects was obtained from various Web pages. If not cited, then it was found on a Web page at iLogix [iLoweb].

- iLogix, Inc., USA
    - development of the STATEMATE family of tools [iLo90, STMweb]
- University of British Columbia, CA
    - development of tools for model checking [Day93, Day94]
- University of Texas, USA
    - development of Modechart and tools [JM94, PMS95]
- Naval Research Laboratory, USA
    - development of tools for Modechart [CTLR93]
- University of California, Irvine
    - development of RSML and related tools [LHHR94, HL96]
- Boeing Commercial Airplane Group, USA
    - development, verification, integration of electrical, mechanical, avionics systems[NW96]
- R-Active Concepts, Inc., USA
    - development of BetterMate [BMweb]
- Cardiac Pacemakers, Inc. (Guidant Corporation), USA
    - pacemaker design
- Computing Devices Ltd. (CDL)
    - video processing chip
- Industrial Science and Technology (IST)
    - model of rail system
- Defense Research Agency (DRA) Malvern, UK
    - code devel. and verification of a Network Layer Security Protocol implementation
- AOA Apparatebau, DE
    - design new vacuum-flush toilet and waste systems for the Airbus A330 airplane

- Ford Motor Company of Europe
  - visually communicate complex car electronic system designs
- LSI Logic Europe
  - speed development of critical part of DSP project

Z

Z (pronounced zed) was developed by the Programming Research Group at Oxford University [Dil94, Zweb]. It is a model-based language based on first order logic and set theory. It specifies the functionality of the system by describing pre-conditions, post-conditions, and invariants. Many tools for Z exist, including editors, true type fonts, typecheckers, and verifiers, most of which are available by ftp. Some standards require the inclusion of natural language text to describe each schema. Other similar notations are VDM, Z++, ZEST, and the AMN notation of the B-method. These differ from Z in their view of preconditions and invariants as well as in their scope of applicability.

The following is a list of users and, if known, their usages of Z. Information from published papers is cited below. Other users are members of the Z Users Group (ZUG) which can be found on-line [Zweb].

- Oxford University, UK
  - development of Z and many related tools
- B-Core Ltd., UK
  - development of the B-Method
- IBM Hursley UK Laboratories, UK
  - re-engineering CICS (transaction processing system) [CGR93]
- Praxis, UK
  - develop CASE toolset (SSADM) [CGR93]
- Inmos, UK
  - design and verify microprocessors [CGR93]
- University of Washington, USA
  - researching specification of discontinuous systems [Jac95]
- George Mason University

    - Z and Category Partition Testing [AA92]
- Tektronix, USA
    - reusable architecture for oscilloscopes [CGR93]
- The University of Reading, UK
- Imperial College, London, UK
- JP Morgan, UK
- Defense Research Agency (DRA) Malvern, UK
- City University, London, UK
- University of York, UK
- Anglia Polytechnic University, UK
- University of Bradford, UK
- University of Bologna, Italy
- France Telecom CNET, France
- University of Queensland, Australia
- DST Deutsche System-Technik GmbH, Germany
- Carnegie-Mellon University, USA
- Hiroshima City University, Japan
- NJIT, USA
- University of Limerick, Ireland

PVS

PVS (Prototype Verification System) is a general purpose verification system developed by SRI and available by anonymous ftp. It has an expressive model-based specification language derived from classical higher order logic that resembles pseudocode. It also provides a type checker and powerful interactive theorem prover. For information beyond what is presented here, see [But93, But96, PVSweb]. PVS is a culmination of over 15 years of work on tools that support formal methods including work on a theorem prover named EHDM. PVS is implemented in Common Lisp and uses either GNU or X Emacs as a user interface. The system also allows specifications, theorems, and proofs to be pretty printed using LaTeX. Other tools that provide theorem proving capabilities are HOL, Nqthm (ACL/2), and EVES.

The following is a list of users and usages of PVS. Published results are cited, while uncited users and usages were found on the PVS Web site at SRI International [PVSweb]. This page also contains an extensive bibliography of papers that have been published on projects using PVS.

- Collins Commercial Avionics
  - Microprocessor Verification [SM95]
- Technical University of Eindhoven
  - Real Time Systems
  - Protocol Verification [Hoo95]
  - Software Systems [VH96]
- GEC Marconi Avionics
- Indiana University
  - Verification of an optimized fault-tolerant clock synchronization circuit [MPJ94]
  - Single Pulser Circuit [JMC94]
- Jet Propulsion Laboratory
  - Requirements analysis of critical spacecraft software [LA94]
- University of Kiel
  - Stepwise Refinement tool
  - Compiler Verification
- London University
- LSI Logic
  - Protocol specification [NRP95]
- University of Manchester
  - Verification for a Hardware Description Language
- Minnesota and Michigan State University
- NASA Langley Research Center
  - Verification of IEEE Compliant Subtractive Division Algorithms
  - Formalizing New Navigation Requirements for NASA' s Space Shuttle
- US Naval Research Laboratory
  - Verification of Timed Automata
- University of Paris VI
  - Protocol specification [HS96]

- Philips, Eindhoven
  - Digital Synthesis
- Princeton University
  - Security of Java-style Dynamic Linking
- University of Southampton
  - Support for B Abstract Machine Notation
- SRI
- Stanford University
  - Cache Coherence Protocols and Memory Models [PD96]
- Tampere University of Technology
  - Mechanized Verification for DisCo
- University of Ulm
  - Program Transformations and Compilation
- Utrecht University
  - Distributed Systems
- Verimag(Grenoble, France)
  - Automated Generation of Invariants
- University of Virginia
- Weizmann Institute
  - Introducing Temporal Properties to PVS
- University of York
  - Compiler and O/S Verification [SCweb]

4.2 Industrial Practice Using Formal Specification

iLogix provides summaries of some of the industrial applications in which their STATEMATE family of tools has been used [iLoweb]. One of the most useful features of STATEMATE is its ability to animate the models. **Cardiac Pacemakers, Inc.**, a unit of Guidant Corp., used STATEMATE to speed up development of defibrillators and pacemakers. Animations of the Statecharts models allowed them to examine interactions between features before building a prototype and to receive feedback on the design from physicians. **AOA Apparatebau** used STATEMATE to design a new waste system for the

Airbus A330 aircraft. Animation of the system allowed them to easily test single and multiple failures. **Ford of Europe** used STATEMATE to specify their on-board electronic controllers in a precise manner. This modeling allowed them to examine the behavior of the system, refine their designs, and gain confidence in the design much earlier than possible with textual descriptions. **Industrial Science and Technology** used STATEMATE to validate the vehicles for a new rail system. Through modeling and animation, many problems were found that had not been identified in the text version of the specification. If these errors had not been found until the vehicles had been built, the cost could be tremendous. The models also helped communicate the requirements of the system to subcontractors.

Animation is not the only benefit of using statecharts. The **Defense Research Agency, UK**, used STATEMATE during the implementation and verification phases of the development of a network security protocol. The hierarchical structure of the statecharts notation simplified the model and eased reasoning about and implementing the protocol. Some of the code was automatically generated in C and linked with hand-coded C++ modules. Animation was used to verify the system.

Express, a member of the STATEMATE family of tools, has been used in hardware design because it generates VHDL code. **Computing Devices Ltd.** used Express in the design and verification of a video processing chip. The capabilities of animation and code generation enabled the chip to go into from design to production in a very short time. **LSI Logic Europe** used Express in conjunction with its Coreware in the design of an ASIC chip. They noticed an improvement in productivity due to improved communication because of the graphical nature of statecharts. The statecharts notation provided a much more succinct representation of the chip than the corresponding VHDL code and simulations in Express were much easier to instantiate and took less time to run than VHDL code.

Boeing used statecharts in the development and validation of electrical, mechanical, and avionics systems as well as in their integration [NW96]. They wanted a tool that did not require the user to understand computer programming, did not require the devotion of lab space for its use, facilitated analysis of the requirements, was standardized, maintained the current level of abstraction in their requirements, and allowed for the integration of independently developed pieces. Of these, statecharts failed only in the category of standardization, since there are many variants of state machines. They found that statecharts were easily understood by non-programmers, compact, and facilitated communication and simplification of the requirements. They were especially pleased with the simulation capabilities of the tool they used (either STATEMATE or BetterMate) as it allowed for a great deal more validation than had previously occurred. The difficulties they faced in using statecharts were not in the notation or use of the tool, but rather from the lack of experience at modeling.

Most of the industrial work using PVS involves hardware design and verification. **Collins Commercial Avionics**, a division of Rockwell International, aided by NASA Langley, undertook an effort to introduce PVS into the production of their commercial microprocessor, the AAMP5 [SM95]. This was an experimental study of the applicability of formal methods in industry. The goal of the project was to increase performance over the AAMP2 and this was successful. However, an unexpected outcome was the verification of a representative set of the microcode instructions.

For software, PVS is being used for requirements analysis. **NEC Space Systems**, Jet Propulsion Laboratory, and Iowa State University applied PVS to critical spacecraft software [LA94]. This project was also intended to evaluate formal methods. They specified and analyzed the requirements for critical software for the Cassini spacecraft, a Saturn orbiter. This software was responsible for system-level fault protection.

Another area to which PVS has been applied is protocol verification. Vijay Nagasamy of **LSI Logic**, Sreeranga Rajan (then of SRI), and Preeti R. Panda of UC Irvine used PVS to specify and verify part of the implementation for the Fiber channel protocol

[NRP95]. They wrote a formal specification to assist with verification that the protocol standards were met. This was also an industrial effort that was undertaken as a research project.

SCR (Software Cost Reduction) is a formal specification method developed at the **Naval Research Laboratory** during an effort to re-engineering the flight control software for the Navy's A-7 aircraft [Hen80]. It began as a more disciplined approach to natural language specification. They organized the information about the A-7, used symbols around names to indicate type, made templates for sentences so that no information was omitted, and created tables for complete, precise descriptions of the behavior. It is the tables that have become the defining feature of SCR; they decompose the system in smaller, more manageable pieces [HJL96].

Since its introduction, the SCR methodology has been expanded, more formally defined, and used in several industrial projects including a submarine communications system [HM83] and the certification of the shutdown system for a nuclear generating station [CGR93]. The Darlington Nuclear Generating Station operated by **Ontario Hydro** had a software implementation of all of the shutdown logic and was having difficulty with licensing. With assistance from Parnas, SCR was used to verify that the code correctly implemented the shutdown logic in accordance with regulations. The requirements and the code were formalized in SCR as two separate efforts and then compared. Much of the work was done manually. The project was successful; the station received its license.

Gerhart, Craigen, and Ralston performed a study of the use of formal methods in safety-critical systems [CGR93]. The details of their study are discussed in the section of this chapter on evaluations of formal specification methods, but their work is also an excellent source of information on large commercial projects involving formal methods. More information on many of the projects described here can be obtained from their report.

Z was used at **Tektronix** in two projects, the specification of the real-time kernel for an X-ray machine [Spi90] and for a reusable architecture for a family of oscilloscopes [CGR93]. The X-ray project was a re-engineering effort from existing documentation and source code. The goal was to re-implement the system on new hardware, however, the mathematical model identified a deadlock condition. Previous documentation used diagrams to describe the system, but they showed the system in its usual state rather than describing all possible configurations, therefore it lead the programmer to make assumptions that detracted from the robustness of the software. Preconditions of each operation were computed to validate the specification.

Oxford University and **IBM Hursley Laboratories** used Z in two major projects involving IBM's Customer Information Control System (CICS) [CGR93, CW96]. This software is a large transaction processing system that was installed in thousands of sites worldwide. The first project was a re-engineering effort. Measurements found that the use of Z caused improvements in quality, reduction in errors, earlier discovery of errors, and reduction in cost of development. The second project was the formal specification of the application programming interface.

A group at **INMOS Ltd.** used formal methods, including Z, in designing their microprocessors, most notably the Transputer family of 32-bit VLSI circuits for concurrent, multiprocessor applications [CGR93]. Z was used in the specification of the IEEE Floating Point Standard and in the design of the scheduler.

Praxis has employed formal methods in several projects. Z was used in the development of a CASE toolset [CGR93]. This toolset supports the Structured Systems Analysis and Design standard. Praxis also developed part of a new air traffic management system for the UK Civil Aviation Authority [CW96]. The specification for this system was written in VDM, a notation similar to Z. Statistics were taken during the development, which involved several formal notations and processes, and compared to comparable projects in which formal methods were not used. There was no loss in productivity and there was a substantial gain in software quality.

Praxis worked with **Lockheed** on the avionics software for their C130J [CW96]. The core method was used for specification; it is a tabular notation very similar to SCR. The code was also developed rigorously, resulting in improved quality at lower costs because little rework was needed.

Although a considerable number of projects have been described here, this represents a fairly thorough summary of the major industrial software projects involving formal specification. The fact that an attempt can even be made to enumerate them reflects the limited usage of formal specification in industry.

4.3 Research Using Formal Specification

Current research involving statecharts is generally focused on exploring the usefulness of the notation and developing tools to support it. At the University of British Columbia, the formal semantics for statecharts were written in Higher Order Logic so that a simulator could be automatically generated from this definition, rather than developed as a separate effort, thus ensuring that the simulator behaves in accordance with the semantics of the notation. Additionally their toolset, recently given the name Fusion, extracts the Statecharts from the STATEMATE system and allows model checking techniques to be applied to the specification [Day93, Day94].

Modechart, a notation based on the statecharts concept, was developed as part of the SARTOR project at the University of Texas. Modechart is based on the principle of statecharts, but redefined in Real Time Logic to allow reasoning about the timing of events. Thus it provides an environment for specifying real-time systems and reasoning about their safety [JM94]. A toolset for Modechart called MT was developed at the Naval Research Laboratory which includes an editor, simulator, and model-checker [CTLR93]. A compiler that generates ESTEREL code from the Modechart specifications was later added to this toolset. The ESTEREL code can be compiled into very efficient C or Ada [PMS95].

RSML is a notation based on statecharts developed by the Irvine Safety Research Group [LHHR94]. It arose from the effort to specify the requirements for the TCAS II system. The notation was later adopted by the FAA for work on this system. This project demonstrated that formal specification can be used for a complex, process-control system and that this specification can be read and reviewed by application experts [CW96, CGR93]. Work has also been done on static analysis of RSML, including completeness and consistency checking [HL96].

Since Z is fairly widely used in industry, research in Z is focused on demonstrating its use in practical settings. Jacky specified a safety-critical control system in Z in order to demonstrate the usefulness of Z in reasoning about discontinuous features of systems [Jac95]. The example is presented in a manner that allows it to be used as a template for similar control systems. The structure of the specification is object-oriented, thus demonstrating the ability to use Z in this paradigm.

Amla and Ammann showed that Z facilitated the Category Partition Testing method [AA92]. This test case generation method requires considerable effort when applied to natural language specifications. However, most of the effort that is needed to analyze the system has already been done when creating a Z specification, so henceforth applying the Category Partition Testing method is almost trivial. They point out which elements of the two methods correspond.

Sherrell and Carver demonstrate the translation of a Z specification of a class roll system into an implementation in Haskell [SC94]. They identify the correspondence between Z schemas and Haskell data types by showing that two different Z designs indicate different Haskell implementations.

SCR was intended for use on safety-critical systems. Therefore a prominent goal was to support verification of the system. Heitmeyer, Jeffords, and Labaw describe a consistency checker for SCR specifications which automates some analysis of the model [HJL96]. In order to build this consistency checker, a formal model of the SCR notation was developed. Additionally, the utility and scalability of the tool is demonstrated in two examples. An integrated toolset for SCR has now been developed [HBGL95].

Research involving PVS falls into three major categories, the verification of hardware, protocols, and software. Hardware verification is being done at Indiana University. They developed a circuit for clock synchronization using several tools, including PVS [MPJ94]. They also joined forces with HP to do a comparative study of several reasoning tools, including PVS [PVSweb]. For this study, the "Single Pulser" circuit was modeled in several notations.

Protocol verification projects have been done at several universities. Jozef Hooman at the Technical University of Eindhoven in the Netherlands used PVS to verify part of the ACCESS.bus Protocol [Hoo95]. He also participated in a project involving the specification and verification of the Steam Boiler Control System using PVS [VH96]. Klaus Havelund of LITP, Institut Blaise Pascal, University of Paris VI experimented with the use of various verification tools on the Philips bounded retransmission protocol [HS96]. A system with infinite state space is a difficulty even for theorem provers. This is demonstrated through the verification of the general version of the protocol using PVS. A method for bounding the state space is presented. At Stanford University a method for use in the verification of concurrent systems in PVS was developed [Par96, PD96]. To demonstrate this method the FLASH cache coherence protocol was specified and verified.

Software verification is restricted almost completely to the research arena. Dave Stringer-Calvert of the University of York is using PVS to verify a compiler for a simple imperative language. His work is based on an existing specification and proof in Z by Susan Stepney [PVSweb, SCweb]. PVS was also used to prove theorems about timing in a case study by Archer and Heitmeyer [AH96]. The model was of the Generalized Railroad Crossing problem. Simon Fowler is using PVS in research on the formal verification of real-time operating system kernels [PVSweb].

The amount of research in formal specification is far more substantial than industrial usage. This summary focused on the three notations that were evaluated in this thesis and the notations closely related to them. It is by no means an exhaustive list of research in this area.

4.4 Research in the Evaluation of Formal Specification

Ardis et al. presented an evaluation of six specification methods, Modechart, VFSM, ESTEREL, LOTOS, Z, and SDL, plus the C programming language [Ard96]. The criteria used to evaluate the methods were derived from their experience in specifying a telephone switching system in each of the seven notations. They considered their criteria relevant to any reactive system.

The criteria were divided into two categories, *fundamental* and *important*, and a table was given that associated the criteria with phases of the lifecycle. Each criteria was described by a paragraph. The novelty of their approach was the inclusion of "not only academic concerns, but also the maturity of the method, its compatibility with the existing software development process and system execution environment, and its suitability for the chosen application domain" [Ard96]. The specification and toolset of each notation was examined and given a rating of +, 0, or - for each criteria.

The work of Ardis et al. inspired the approach for this thesis. Their criteria included practical issues and a chart was provided that associated the criteria with lifecycle phases. The division of criteria into categories of fundamental and important reflects the fact that some criteria are more important than others. However, there were shortcomings in their approach. No support was given for the choice of criteria and there was no explanation of the association of criteria with particular lifecycle phases. The evaluation method was not described, so it can only be assumed that the evaluation was performed solely by the authors. The authors can provide information on the feasibility of formal specification when used for writing a specification, but little else.

Gerhart, Craigen, and Ralston performed an extensive study of the current use of formal methods in industry [CGR93]. They were particularly concerned with software systems in regulated industries such as nuclear power. Their goal was not to derive a detail list of criteria, but rather to document current experience with formal methods and suggest where more research is needed. They studied the overall impact of formal methods on the project, such as the effects on client satisfaction or product cost. Their method of evaluation was similar to the one used in this thesis. They used questionnaires and interviews to obtain information from practitioners currently using formal methods. The participants of their study were superior to those used in this one because they already had extensive experience with formal methods. Their study determined that formal methods are steady gaining acceptance in industry and they expressed the need for additional studies of the use of formal methods to provide feedback to the research community.

Pfleeger and Hatton investigated the influence of formal methods in CDIS, an air-traffic-control system built by Praxis [PH97]. The development of CDIS provided a context for comparing formal and informal methods because different parts of the system had been specified using different methods. Praxis had recorded statistics on the number of faults, errors, and changes in the system during its development. Pleeger and Hatton examined these statistics for trends. Formal specification appeared to have produced simpler designs, easy testing, and high-quality code. However, they did not have all the information needed to conclude with confidence that formal specification alone caused these results.

Faulk presents short list of qualities of a “good” requirements specification [Fau95]. They are divided into two categories, semantic properties and packaging properties. A requirements specification that meets the semantic properties is complete, implementation independent, unambiguous and consistent, precise, and verifiable. A requirements specification that satisfies the packaging properties is modifiable, readable, and organized for reference and review. Rushby also provides an excellent list of criteria to consider when choosing a formal specification method in his report for NASA [Rus93].

The list is divided into criteria for the notation and criteria for the utilities. The criteria suggested by these two authors are not systematically derived from a clearly defined basis for evaluation, but from their vast experience using formal methods.

5 Basis for Evaluation

The basis on which the criteria to evaluate formal specification were derived was current software development practice. It can be described by the software lifecycle and the management activities that guide the development process. The basis must be clearly defined because it provides the defense for the list of criteria. In this chapter, the lifecycle phases and management activities are defined, followed by a discussion of the approach to deriving the criteria that will be used to evaluate formal specification methods.

5.1 Lifecycle Phases

Software engineering characterizes the lifecycle of software as consisting of phases. These phases are a list of tasks that must be completed in order to create a software product. While the activities that must be performed are generally agreed upon, the number, names, and divisions of the phases are not universal. Because of these differences, the phases and terms that will be discussed in the remainder of this paper are defined here. Another point of disagreement is the order in which these phases are performed. There are several models, most notably the Waterfall and Spiral models. The Waterfall model prescribes the completion of one phase before the beginning of the next with no backtracking. This was named the rational design process, but is not the way most software is built. In the Spiral model, the order of development of different pieces of the system is dictated by the level risk associated with them. The work presented here is not specific to a particular model. Although it may seem that the Waterfall is the model because no backtracking or iterations are explicitly discussed, there is also nothing to contradict the revisiting of phases several times during development.

Requirements Specification

The requirements specification phase is made up of two activities, eliciting the software system requirements from the customer and recording the requirements in a specification document. It is often separated into two phases, requirements and specification. There is merit to this division since the elicitation of the requirements occurs in meetings with the client, then the computer scientist uses that information to create a system specification document. However, the two activities have one objective: to clarify, define them, and record ideas about the system. The specification is an abstract description of the system and independent of the implementation machine or language except in that it must be implementable. This is arguably the most vital stage of the lifecycle since the specification document is the foundation on which the software will be designed, implemented, and verified. Every requirement of the system, functional or non-functional, must be recorded in the specification. The specification must be approved by the client as describing the desired system. It must be checked for completeness and correctness. Mechanical analysis of the specification document can be helpful. The specification document will serve as the authority on the system requirements throughout the rest of the lifecycle. Mistakes or misunderstandings in the specification document will manifest themselves in every other phase.

Design

During the design phase, a system design document is produced that meets the requirements recorded in the specification. The design document provides the details that are needed to create the concrete implementation. Whereas the specification is independent of the machine and language of implementation, the design is not. The design might be written in the specification notation, in the implementation language, or in some other notation. If the specification notation is used, the design might not be a separate document, but rather an addition to the specification. Pressman [Pre92] describes design as follows:

> *Software design is actually a multistep process that focuses on four distinct attributes of the program: data structure, software architecture, procedural detail, and interface characterization. The design process translates requirements into a representation of the software that can be assessed for quality before coding begins.*

During the design, the structure of the software system must be determined. It will be divided into modules that will be implemented by different people. The interfaces between these modules must be documented. At a more detailed level, the data structures, functions, and algorithms must be determined. There are many published design methods that are currently in use; all of them prescribe a process of detailing an implementation plan that meets the system requirements.

Implementation

The implementation phase is when running computer code is produced based on the system requirements documented in the specification and the system structure and details documented in the design. Very few decisions should be made in this late phase of the lifecycle and it is possible that implementation could be automated. The implementation is usually performed by a group of programmers. It is important that the specification and design documents be complete, precise, and unambiguous so that the pieces of the system that are implemented by different people are compatible when they are put together and will meet the requirements of the client. Verification of the pieces of the system as separate units is also the responsibility of the implementers.

Verification

The purpose of the verification phase is to ensure that the implementation meets the requirements documented in the specification. In current practice, the most common way to do this is by testing, but other methods include code reviews and theorem proving. In theory, if the refinement from specification to implementation has been checked rigorously at every step, the implementation should meet the specification. However, in practice, it is common for verification to identify a large number of changes that need to be made. This necessitates a return to the implementation phase, perhaps to the design phase, and often to the specification phase, which lengthens the development time.

Maintenance

The maintenance phases encompasses all further development of the system after it has been released to the customer. Changes might be made to correct errors that are found, to enhance the system, or as a result of changes in the requirements. Thus, the maintenance phase entails repeated cycles through all the phases. New people are introduced to the system. A common problem is that changes introduce new errors. In long-lived systems, most of the time and money is spent in the maintenance phase, so it is important to consider maintenance issues when building the software. The software system, including the specification, design, and implementation, must stand the test of time by being easy to understand and change.

5.2 Management Activities

The lifecycle alone is not sufficient to describe current software development. The development must be planned and paid for. The resources must be allocated. The lifecycle phases prescribe documents that must be produced, but they don't describe how to produce them; they describe tasks that must be completed, but don't describe what order to do them in. Current practice has methods for managing a software project that have evolved over years of experience. These process considerations, such as scheduling, resource estimation, risk analysis, and quality control, are not often included in academic discussions of software development, but the are vital to the success of the project. These management activities guide the development and affect the quality of the process and the product.

Scheduling

Scheduling is a very important aspect of a software development effort. Practically every project has a deadline. It is necessary to set intermediate deadlines in order to track progress and determine if the entire effort is on schedule. There are some parts of the development can occur in parallel while others have dependencies that dictate their order. A schedule is essential for getting every task done on time and in the right order. Making a schedule requires accurate estimations of how long different activities will take. Successful estimation comes from previous experience as well as avoidance of unexpected problems.

Resource Estimation

There are many resources needed during software development, such as software, hardware, and people. The cost of these resources must be estimated to determine the cost of developing the system. New software and hardware or additional licenses for currently owned software might be needed. People must be hired or fired. Besides the cost of salaries and new equipment, there is also the issue of allocating these resources to a project. This is especially difficult if several products are being developed simultaneously. People with particular specialties must be moved from one project to another at appropriate times. Other people need to stay with a project from beginning to end to maintain continuity. Equipment must be available for everyone on the project. The allocation of resources is vital to the success of the development process.

Risk Analysis

Risk analysis is an essential part of every type of project and no less so for software development. It is important to identify the parts of the system that pose the greatest risk of failure. Additionally, there might be phases of the development that tend to cause projects to miss their deadlines or during which resources will be limited. Consideration must be given to the parts of the system that are most likely to change during the original development or during maintenance. To help ensure successful development, risks must be correctly anticipated and steps must be taken to minimize them.

Quality Control

It is the goal of software development to produce a high-quality product. However the level and characteristics of the quality desired varies between projects. Dependability may be the goal in one project, while fast execution or maintainability may be the goal in another. Whatever the quality goals are, checks must be made to ensure that the software will meet these goals. There are many ways to monitor quality, including design and code reviews and statistics on the number of defects per lines of code. The measures for quality control should be planned and enforced. The statistics should be used to make future improvements in the process.

5.3 Deriving Criteria for Evaluation

The lifecycle and management activities together describe current software development practice, so they will be the basis for the evaluation. Criteria that are generated in this way from current practice affect one or more specific phases or activities. Ideally, a study of the costs and benefits of incorporating formal specification can be performed to determine whether the introduction of formal specification would be advantageous. Such an analysis would include a weight for each criterion that reflected the goals of the project. For example, if dependability is imperative, then improvements in verification might be deemed very important; therefore the weight associated with criteria related to verification would be high. These weights would be multiplied by the costs and benefits so that the

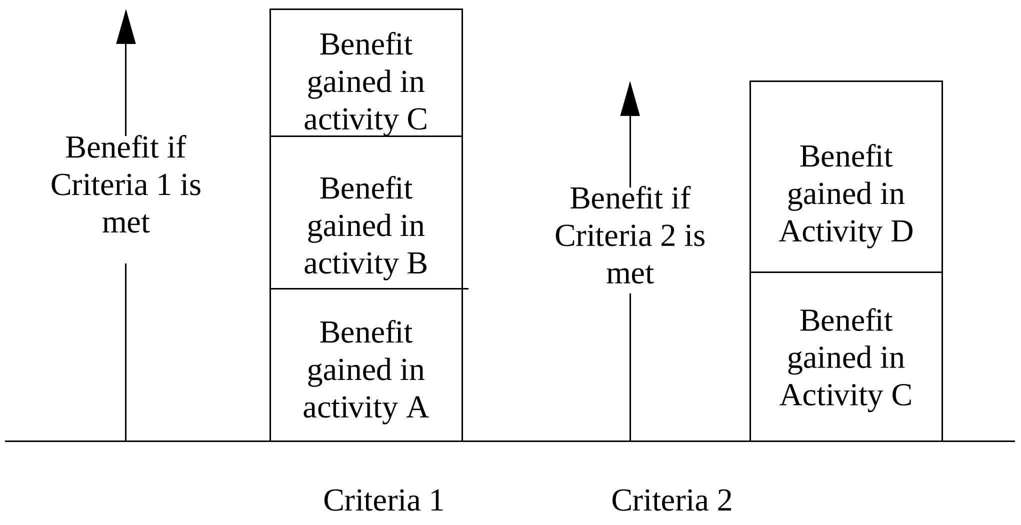

Benefit if Criteria 1 is met
Benefit gained in activity C
Benefit gained in activity B
Benefit gained in activity A
Benefit if Criteria 2 is met
Benefit gained in Activity D
Benefit gained in Activity C
Criteria 1
Criteria 2

Figure 4: Example of Cost-Benefit Analysis

important criteria affect the assessment of the specification method more than unimportant criteria. Such a cost-benefit analysis might result in data similar to that shown in Figure 1. This figure shows that the benefit of incorporating a formal specification method that meets Criteria 1 is the sum of the benefits in each of the activities that are affected by Criteria 1. The same is true for Criteria 2. The total benefit gained by incorporating the formal specification method can be calculated by summing the benefits of each criteria that the method fulfills. The total cost of using the formal specification method can be calculated in the same way, by examining each activity for costs incurred by incorporating formal specification. Such a cost-benefit analysis is highly dependent upon the goals and characteristics of the particular project and development environment. Estimations of the cost and benefits must be obtained from other similar projects.

The work described here was not aimed at a particular project, so no weights were assigned to the criteria; each was equally important. It was assumed that current practice employed a well-established informal method for development and used natural language for specification. Each of the lifecycle phases was examined and the common activities performed during that phase were enumerated. Then for each of these activities, there were two considerations, the demands placed on formal specification to meet the standards set by current practice and the new benefits that formal specification could provide. The effects on both the quality of the lifecycle artifacts and the effectiveness of the management activities from incorporation of formal specification were discussed. Demands from current practice include compatibility with current design methods and matching the quality of tools. New benefits are essential in order to amortize the cost of introducing formal specification into current practice. Such benefits might include an increase in the quality of the artifacts of the lifecycle, such as design documents or implementations, or an improvement in the development process, for example a more predictable development process is easier to schedule, estimate resources for, and there are fewer risks of missing the deadline.

This careful examination of current practice identified many demands placed on formal specification methods and many benefits that they could provide. In order for formal specification to gain acceptance in industry, these demands must be met and benefits provided. A list of criteria was compiled from the findings that resulted from the examination of current practice. These criteria were then used for evaluation of three specification methods.

6 *Demands of the Lifecycle*

In this chapter, each phase of the lifecycle was examined systematically and a list of demands that each places upon the specification notation or tools was compiled. Every phase has been included because the specification participates in every stage of the lifecycle of the software. In examining the phases, two types of requirements were considered. The ability of formal specification to take the place of natural language with as little disruption to current practice as possible. This includes compatibility with current methods and tools, as well as competition with the strengths of natural language. Secondly, the ability of formal specification to produce improvements over current practice in the development process and quality of the software produced. The phases examined here, requirements specification, design, implementation, verification, and maintenance, were defined in the previous chapter.

6.1 Requirements Specification

Requirements specification consists of two activities, writing the specification and validating the specification. The issues discussed in this phase are divided between those two categories. Writing the specification is performed by specifiers, while validation is done by both the specifiers and the client or domain expert.

Writing the Specification

- *Specification development speed*

  The length of time that it takes to develop a formal specification is a very important consideration. The formal specification method must support rapid prototyping of the specification and facilitate later elaboration of the details. One of the goals of industry is to decrease system development time. Formal specification may lengthen the early phases of the lifecycle such as writing and validating the specification, but shorten later ones like coding, testing, or maintenance because less rework is required and the system is well-documented.

- *Training, documentation, and technical support for the specification method*

  In order to assist in learning to use the formal notation and toolset, training and documentation are vital. Once formal specification is in use, documentation and technical support continue to be valuable. These types of assistance are especially important when the method is unfamiliar to the specifier and when it is not in wide use, as is currently the case.

- *Development method*

  The industrial community has years of experience developing natural language specifications. A cultural approach is used since writing has been taught in many classes throughout everyone's education. Additionally, many authors have published descriptions of successful natural language specification methods, including document layouts. None of this is true for formal specification. There are very few people with experience developing specifications using formal notations and, despite the word *method* in formal methods, little methodology is described in the literature. In order to replace natural language, formal specification must have an associated development method that can be introduced into the work environment with as little disruption as possible.

- *Coverage*

  Natural language is infinitely large and flexible and can be used to express almost any requirement of a system, whether functional or non-functional. Common shortcomings of formal specification notations are an inability to describe user interfaces or non-functional requirements. In order to be generally applicable, the formal notation must have the ability to express every concept or be designed to operate with another tool that can express it.

- *Integration with other components*

  The specification is not developed in isolation, but rather as part of the larger software development process. The specification tools must integrate with the other components of this process, such as documentation, boilerplates, management information, and executive summaries. Often a system database and version control system are used. A part or all of the specification might be inserted into another document, so the specification must have a common file format. There will likely be the desire for a hard copy of the specification. It should be easy to print the entire specification, including comments and non-functional requirements, in a straightforward manner and acquire a legible document. The formal specification method must be suited to the larger working environment.

- *Group development*

  Every software project involves more than one person. During the development of the specification, version control must be exercised, whether internal or external. It must also be possible for several people to work in parallel and combine their efforts later. Therefore, the specification method must support the idea of separate compilation. It must also allow many people to view the specification simultaneously.

- *Support for evolution*

  A specification is not built in one effort and then set in concrete; it is developed and changed over time. The specification method must support the logical evolution of specification and ease its change. Incompleteness must be tolerated. Functionality such as searching, replacing, cutting, copying, and file insertion must be provided. Modularity and information hiding must be facilitated, so that, for example, a change in a definition is automatically propagated to every usage of it. Large scale manipulations must also be supported, like moving entire sections or making them subsections.

- *Support for usability*

  The ability to locate relevant information is a vital part of the usefulness of a specification. The ability to search for regular expressions is valuable, but not sufficient. The specification is intended to serve as a means of communication. Annotating the specification with explanations, rationale, or assumptions is important for both the use of the specification in later phases and for modifications of the specification. This annotation must be easy to create and access, and it must be linked to a part of the specification, so changes effect the corresponding annotation. The specification notation should also provide structuring mechanisms to aid in navigation since the specification document is likely to be large. In a natural language document, the table of contents and index assist in the location of information; many tools allow them to be generated automatically from the text. Another useful capability seen in text editing is the use of hypertext links to a related section or glossary entry. Formal specification methods must provide similar aids to enhance the usability of the resulting specification documents.

Checking and Validation of the Specification

- *Human validation*

  During the early part of the lifecycle of the software, emphasis is on validation. The customer must check that the system described in the specification is complete and correct. The developer must also check for completeness and consistency throughout the system. Reading and understanding the specification is a minimal requirement. Another helpful capability is animation of the model. Animation demonstrates the behavior of the system. The developer might also want to develop a prototype of the software system. The specification method should facilitate validation.

- *Static analysis*

  Static analyzers can aid in identification of notational errors, incompleteness, or inconsistencies. Natural language systems provide few mechanical checks besides spelling. More extensive analysis of formal specification notations, such as type checking and completeness and consistency checks, can be performed automatically because of their formal semantics. These types of analysis can identify trivial errors and also larger problems, such as misunderstandings of the notation, omissions in the specification, or design mistakes, that would be difficult and expensive to fix if not found until later in the lifecycle. It is important that these checkers emit informative messages.

- *Extended validation*

  Tools exist that provide further capabilities that aid in validation of formal specifications. The generation of preconditions of functions can be very valuable. Properties of the specified system can be proven using theorem proving or model checking. The proof of properties such as freedom from deadlock or avoidance of dangerous states provides quality, dependability, and safety assurance. These types of checks are not mechanical; the theorems must be formulated by hand. The development of theorems requires some training. A theorem prover can then help automate the proof process by providing a language of commands that execute the steps of the proof mechanically. A model checker requires the system to have a finite state space, but can then prove the theorem automatically using an exhaustive search.

6.2 Design

- *Compatibility with design tools*

  A very strong relationship exists between the specification of a system and its design, therefore the tools should also be closely related. It should not be necessary for the designer to re-enter parts of the specification that are also part of the design. Either the specification tool must also fully support the design phase or it must be compatible with common design tools.

- *Compatibility with design methodology*

  Just as the specification tools must be the same as or compatible with popular design tools, the method for using them must also be compatible with popular design methodologies. A method for the use of a specification written in a formal notation during the design process should be described in the literature, complete with examples.

- *Communication of desired system characteristics to designers*

  In order to design the system, the designer must be able to read and understand the specification. The specification should describe the normal operating procedure, any error conditions and the response that is appropriate, and non-functional requirements such as the size or efficiency requirements. The specification should contain the answer to every question about the system, i.e. be complete. These questions could involve abstract concepts or details, so the specification must be precise, expressive, and accurate. Inaccuracy is worse than omission! The specification must use familiar notations, have rational structure, and be easy to navigate and search.

- *Facilitation of design process*

  The more easily a design can be developed from the specification, the better. The use of a formal specification could speed up the design process by describing the system clearly and precisely. The designer must take the abstract description in the specification and describe how a real system is going to implement the specification. Information hiding must be maintained and the ability to view the system at varying levels of abstraction must be provided. The specification of the system must be structured appropriately since there will likely be a strong correlation between the structure of the specification and of the design. In order to facilitate good design decisions, the specification should identify key parts of the system and make dependencies between parts of the system explicit. It should ease the understanding of the function of a section of the system or the flow of an individual data-item. The designer may want to create a system prototype, so the specification method should allow this through easy or automatic translation of the design to code.

6.3 Implementation

- *Communication of desired system characteristics to implementors*

  The implementors will need to reference the specification and design during implementation, so the two documents must complement each other. Implementors will need to read, understand, navigate, and query the specification. There should be examples of how to express features of the formal notation in an implementation language. While the specification should be implementation independent, it may be that certain features are more easily expressed in certain implementation languages. It is important that it be possible to implement every concept in the specification. The structure of the specification is vital to the implementors' understanding of which features to implement and what their relations are to other parts of the system. The ability to view the system at different levels of abstraction would enable them to focus on the relevant parts of the system. It is important that all information about a function can be found easily and the exact semantics of the specification notation should provide a clear description of the functionality. This description needs to contain the appropriate level of detail.

- *Efficient coding*

  Coding is hindered by lack of clarity in the specification and design and misunderstandings that cause rework. The more complete, precise, and detailed the specification and design are, the more smoothly coding should go. This makes the phase faster and more predictable. It could be greatly enhanced by automatic generation of code or a code framework.

- *Unit testing*

  A precise, complete, and accurate specification can greatly aid in the formulation of a unit test suit, perhaps through automatic generation. It should also minimize rework, since the requirements are well defined and unambiguously stated in the specification.

6.4 Verification

- *Effective verification*

  Formal specification can shorten the amount of time spent in verification. A high quality specification will make it clear what the requirements of the system are, so they are easy to verify and more likely to have been implemented correctly the first time. Besides providing a precise specification, formal semantics may also support easy or automatic generation of test cases. Theorem proving may eliminate the need for testing. It would be a great contribution if the specification provided an indication of when verification was complete.

- *Communication of desired system to verifiers*

  The specification defines the desired system; verification is the process of checking that the implementation meets the specification. Therefore the verifiers must be able to read and understand the specification. This is enhanced by the ability to view the specification at different levels of abstraction. They will need to navigate the specification to find information. The tools should support several viewers.

- *Integration with development environment*

  The formal specification method should be compatible with current verification methods, such as testing, inspection, and theorem proving. There should be examples available that demonstrate the use of a specification written in the notation during verification. Whatever the method of verification, the information gained during verification should be connected with the information about the rest of the system. For example, test cases might be associated with a particular section of the specification, design, and code to which they correspond. This type of linkage will speed up rework of the code and then re-verification. If a database of faults is kept, then the specification tool must be compatible with this system.

- *System testing*

  In order to test the resulting system, the specification must precisely describe the system behavior. It must state the properties of the system and its response to every situation. Without such a detailed description of the expectations for the system, it is impossible to test for compliance. The derivation of a set of test cases from a natural language specification is a lengthy process requiring extensive human analysis of the system and its operational environment and is often considered an art form. This does not lead to a fast repeatable process. Formal specification may facilitate the generation of test cases because much of this analysis has already been done and is precisely expressed in the specification. This was the hypothesis of Nina Amla and Paul Ammann and they found this to be true for at least one combination of formal specification method and testing method [AA92]. A rigorous method for generating test cases would also indicate when testing is finished.

- *Inspection*

  In an inspection, the code is subjected to human scrutiny. It must be shown that the code meets the specification and that it is written well, i.e. well-organized, structured, and in the accepted format. Using a natural language specification, it is difficult to determine whether or not the requirements have been met. Often the inspection focuses more on the form of the code than on its semantics. If a formal specification were used, the inspection could check rigorously that the transition from specification to implementation was a accurate refinement. In order to facilitate inspection, the specification must be readable by the inspectors and state the requirements precisely so that the code can be checked for compliance.

- *Formal verification*

  If the specification has been rigorously validated and, at each refinement, the design and implementation are proven to be equivalent to the specification, then the properties of the specification hold on the implementation. This requires that the specification has formal semantics based in mathematics, a verification tool exists for the notation, and that each refinement can be verified. Alternatively, the code and specification can be proven to be equivalent using theorem proving one the implementation nis complete. The application of such rigor during the development of the system practically eliminates the need for testing.

6.5 Maintenance

- *Understanding the system*

  A new person on the project should be able to study the specification and gain a broad or detailed understanding of the system. The documentation of non-functional requirements and design decisions is vital to a complete understanding of the system. It should be easy to navigate, accurate, complete, and easy to reference to find answers to questions. The structure, information hiding, and the ability to view the specification at different levels of abstraction will enhance understanding. It should also be possible to print a hardcopy of the specification document.

- *Changing the system*

  When a change is made to the system, both the code and specification must be changed. This is clearly facilitated if the two are carefully linked together so the changes needed in the code are very similar to those in the specification. Currently the specification is changed as an afterthought or not at all. Ideally the specification should be changed first to examine the effects of the change on the system. This requires that the specification be easily changed and that the document remains well-structured. Once changed, formal notations could allow static analysis, animation, or even proof of properties to be done on the new specification before the change is propagated to the code. Validation and verification of a change is important.

6.6 Generating Criteria

The specification serves as the vehicle of communication about the system throughout the lifecycle. Therefore, the specification method must accommodate the needs of every person involved in the development. The specification plays an important, but difficult role. From this careful scrutiny of current practice, a list can be generated of criteria that a formal specification method should satisfy in order to be routinely used in industry.

7 Criteria

In the previous chapter, each lifecycle phase was examined. The activities from each phase that involve the specification were listed in order to identify the demands they place on the specification notation and toolset. These demands are now translated in to a concise list of criteria that can be used to evaluate formal specification methods. Because these criteria are derived directly from the software development process rather than in an ad hoc manner based on experience with a particular project, the inclusion of each is defendable. Criteria applicable to all lifecycle phases are collected into one group; the other criteria are listed by lifecycle phase.

7.1 Common to All Lifecycle Phases

- Training in the notation and toolset is available and of appropriate length and extent
- Quality technical support for the notation and toolset is available
- The notation and toolset are reasonably easy to learn
- The size and complexity of the notation is appropriate
- The space requirements and run-time of the toolset are reasonable
- The toolset provides an easy way to print a hard copy
- The notation and toolset facilitate navigation and searching
- The toolset provides support for multiple users
- The notation and toolset provide support for differing levels of abstraction

- The resulting specification is of high quality (complete, accurate, precise)
- The notation has formal semantics
- The notation provides support for understandability (e.g. infinite-length variable names, meaningful keywords, common mathematical notation)
- The specification facilitates communication about the system

7.2 Requirements Specification Phase

Writing the Specification

- The notation and toolset decrease the time needed to write the specification
- A useful method exists for creating a specification in the notation
- Useful examples of system specifications in the notation are available
- All aspects of a system and its environment can be expressed in the notation
- The notation and toolset provide the ability to document non-functional requirements
- The toolset and notation integrate with other hardware and software in the development environment
- The toolset provides the ability to represent the specification in a common file format
- The toolset provides easy creation, manipulation, and organization of files
- The toolset allows the use of version control
- The toolset is compatible with the documentation system
- The notation and toolset support the notion of separate compilation
- The toolset tolerates incompleteness in the specification during development
- The toolset facilitates modification of the specification (small textual changes as well as large-scale changes such as moving sections)
- The notation and toolset facilitate structuring and information hiding in the specification

Checking and Validation of the Specification

- The notation and toolset decrease the time needed to validate the specification
- The specification is easily understood by a developer
- The notation and toolset facilitate completeness and consistency checking by a developer

- The notation and toolset describe static properties of the system, such as pre-conditions, post-conditions, invariants, and flow of data and control
- The toolset provides the ability to animate the specification in order to view its behavior
- The toolset provides useful static analyzers for mechanical checking
- The specification is easily understood by a client or domain expert
- The specification can be checked by a client or domain expert for completeness and correctness

7.3 Design Phase

- The specification decreases the time needed to create a design
- Useful examples of creating designs from the notation are available
- Non-functional requirements are documented in a manner useful to design
- The specification method is compatible with current design methods
- The toolset integrates with software and hardware used in design
- The specification is easily understood by a designer
- The structuring and information hiding of the specification are useful in design
- The specification facilitates design
- The specification facilitates the identification of key parts of the system
- The specification facilitates the identification of interactions or dependencies between parts of the system
- The toolset facilitates the creation of a system prototype

7.4 Implementation Phase

- The specification decreases the time needed to implement the system
- The toolset provides support for automatic code generation from the specification
- Every feature of the notation is implementable
- Useful examples of how derive code from the notation are available
- The notation does not have an affinity to certain implementation hardware or software
- The specification integrates with the design document

- The toolset is compatible with the hardware and software used in implementation
- The specification is easily understood by an implementor
- Non-functional requirements are documented in a manner useful for implementation
- The structuring and information hiding in the specification are useful to implementation
- The specification facilitates implementation
- The specification provides an appropriate level of detail about the functionality
- The specification facilitates unit testing

7.5 Verification Phase

- The specification decreases the time needed to verify the system
- Useful examples of verification based on a specification in the notation are available
- The specification methods is compatible with current verification methods
- The toolset integrates with the software and hardware used in verification
- Non-functional requirements are documented in a manner useful for verification
- The specification is easily understood by a verifier
- The specification provides the ability to determine the outcome in every situation
- The specification facilitates verification
- The toolset provides automatic test generation
- The specification facilitates code inspections
- The toolset provides the ability to perform theorem proving or model checking

7.6 Maintenance Phase

- The specification decreases the time and effort needed to maintain the system
- The specification is useful as an introduction to the system for a new maintainer
- The specification is useful as an introduction to the system for a new client or domain expert
- Non-functional requirements are documented in a manner useful for maintenance

- The specification integrates with other artifacts of the system (design, implementation, test suite)
- The specification is easily understood by a maintainer
- The specification is easy to modify (small textual changes as well as large-scale changes such as moving sections)
- The structuring and information hiding of the specification facilitates understanding of the specification
- The structuring and information hiding in the specification facilitates modification of the specification
- The specification facilitates changes to the system
- The specification can be used to validate a proposed change before changing the implementation
- The specification facilitates verification of the system after change is made

7.7 Evaluation

With this list of criteria, that were derived systematically from current practice, formal specification methods can be evaluated. This evaluation can be conducted using all of the criteria or only a portion that are deemed most relevant to a particular project. Ideally, the criteria would be given weights in accordance with the relative importance of satisfying that criteria in order to meet the goals of a particular project. The results of this evaluation can be used to assess the applicability of a formal specification method to a project or to identify features that require improvement.

8 Evaluation Method

Using the defendable list of criteria presented in the previous chapter, an evaluation of three formal specification methods, Z, PVS, and statecharts was conducted. The criteria aim to expose deficiencies in the specification methods that have kept them from receiving widespread use in industry, as well as benefits that these methods provide over the current practice of using natural language for specification. The evaluation that was conducted was not ideal due to restrictions in resources. An ideal evaluation method is briefly described, followed by the actual evaluation method used in this study.

8.1 Ideal Evaluation Method

The only way to determine for certain whether a formal specification method is beneficial to a particular project and working environment is to actually use it in that setting. To evaluate its usefulness in industrial practice in general, a formal specification method must be tested in a large number and variety of projects. The projects chosen for study should be numerous and encompass a wide range of application areas. The goals of the projects should also be as varied, including safety critical systems and embedded systems, as well as text editors and database systems. Systems with varying characteristics, such as reactive or computational-intensive, should be included. The population surveyed should consist of experienced industrial software practitioners, including clients, managers, designers, implanters, documenters, and maintainers. Data should be gathered while the formal specification method is used by these practitioners on an actual project. The project should be followed from conception through a period of maintenance. Measurements of productivity and product quality should be taken before and after the addition of formal specification to the development process, so that a comparison can be made. A study with these characteristics would require many years and the cooperation of thousands of people.

8.2 Actual Evaluation Method

The resources needed to conduct a statistically significant evaluation of even one formal specification method are too extensive for this endeavor. Instead three formal specification methods were evaluated based on one project, a nuclear reactor control system, using a limited number of criteria. For summaries of the notations and the application, see item 3 - Notation Summaries. The project was not followed through all phases of the life-cycle, only specification. A specification of the preliminary version of the control system using each of the three formal specification methods was developed by members of the research group. Then assessments of the formal specification methods were performed by three groups of participants, nuclear engineers, computer scientists, and the authors of the specifications. Below, the evaluation criteria and technique used are described for each of these groups.

- *Nuclear Engineers*

  The nuclear engineers were domain experts for the project and their role was to validate the specifications. They were familiar with the system that was being specified, but not with the formal specification methods. Their evaluation of the notations was performed during an interview in which the printout of one of the specifications was explained to them and they were asked to assess their ability to understand the specification and check it for completeness and correctness. This format was intended to approximate meetings between the specifier and the domain expert during the requirements specification phase of the lifecycle. The parts of the specifications for which they displayed ease or difficulty in understanding were noted. Their comments on the notations were recorded as anecdotal evidence. The criteria evaluated by the nuclear engineers were:

- The notation and toolset are reasonably easy to learn
- The notation facilitates navigation and searching
- The notation provides support for understandability (e.g. infinite-length variable names, meaningful keywords, common mathematical notation)
- The specification facilitates communication about the system
- The specification is easily understood by a client or domain expert
- The specification can be checked by a client or domain expert for completeness and correctness
- The specification is useful as an introduction to the system for a new client or domain expert
- The structuring and information hiding of the specification facilitates understanding of the specification
- The specification can be used to validate a proposed change before changing the implementation

- *Computer Scientists*

  The computer scientists received a brief introduction to the notations and the application, studied printouts of the specifications, and completed a questionnaire (see Appendix B) for each of the three notations that was intended to evaluate their ability to perform tasks necessary in the software development process such as learning the notation, understanding a specification written in the notation, and locating information in the specification. The questionnaires were written in a multiple choice format in order to standardize the answers, however comments were welcome and, in fact, provided some of the most useful information. The volunteers were also asked to measure the time it took them to complete certain tasks. Clearly the range of criteria that they could evaluate is limited by their lack of experience with the formal notation, however this data is nevertheless valuable because initial impressions can determine whether a formal specification method is adopted for use in a project. The criteria that were evaluated by the computer scientists were:

- The notation and toolset are reasonably easy to learn
- The size and complexity of the notation is appropriate
- The notation facilitates navigation and searching
- The notation provides support for understandability (e.g. infinite-length variable names, meaningful keywords, common mathematical notation)
- The specification facilitates communication about the system
- The specification is easily understood by a computer scientist
- The specification facilitates the identification of key parts of the system
- The specification facilitates the identification of interactions or dependencies between parts of the system
- Every feature of the notation is implementable
- The specification provides an appropriate level of detail about the functionality
- The specification is useful as an introduction to the system for a new maintainer
- The structuring and information hiding of the specification facilitates understanding of the specification
- The specification is complete

- *Authors*

  The authors of the specifications provide a perspective of the formal specification notations and toolsets that is very different from the other two groups of participants in this study. Only the authors spent a considerable amount of time writing and studying the specifications. Only the authors had experience with the toolsets. Because they contribute this vastly different perspective, it was necessary to include their assessments of the notations and toolsets. Their input could be objectionable because they were members of the research group, however these results are anecdotal, based on their experience with one project. The author of each specification was asked to complete a questionnaire (see Appendix A). In order to minimize the bias from these participants, an effort has been made to limit the subjectivity of the evidence. Anecdotal evidence from the development of the specifications was also included. The criteria that were evaluated by the authors were:

  - Training in the notation and toolset is available and of appropriate length and extent
  - Quality technical support for the notation and toolset is available

- The space requirements and run-time of the toolset are reasonable
- The toolset provides an easy way to print a hard copy
- The notation and toolset facilitate navigation and searching
- The toolset provides support for multiple users
- The notation and toolset provide support for differing levels of abstraction
- The notation provides support for understandability (e.g. infinite-length variable names, meaningful keywords, common mathematical notation)
- A useful method exists for creating a specification in the notation
- Useful examples of system specifications in the notation are available
- All aspects of a system and its environment can be expressed in the notation
- The notation and toolset provide the ability to document non-functional requirements
- The toolset and notation integrate with other hardware and software in the development environment
- The toolset provides the ability to represent the specification in a common file format
- The toolset provides easy creation, manipulation, and organization of files
- The toolset allows the use of version control
- The toolset is compatible with the documentation system
- The notation and toolset support the notion of separate compilation
- The toolset tolerates incompleteness in the specification during development
- The toolset facilitates modification of the specification (small textual changes as well as large-scale changes such as moving sections)
- The notation and toolset facilitate structuring and information hiding in the specification
- The notation and toolset facilitate completeness and consistency checking by a developer
- The notation and toolset describe static properties of the system, such as preconditions, post-conditions, invariants, and flow of data and control
- The toolset provides the ability to animate the specification in order to view its behavior
- The toolset provides useful static analyzers for mechanical checking
- The toolset provides support for automatic code generation from the specification

Many of the criteria that were derived from current practice could not be evaluated in this study. For example, criteria that required statistics to be kept during the entire life-cycle could not be evaluated since only one phase had been completed. Because of the limits of this study, the goal was to collect anecdotal evidence on a subset of the criteria. This evidence provided an early indication as to the validity of the hypothesis that formal specification methods must overcome practical hurdles before they can be accepted by industry.

9 Results

This chapter contains the results of the evaluation of three formal specification methods, Z, PVS, and statecharts, based on a subset of the criteria derived from the software development process. The formal specification methods were evaluated from three points of view, the expert in the domain of nuclear engineering, the computer scientist, and the specifier, thus there are three groups of results corresponding to these three perspectives. The nuclear engineers and computer scientists evaluated only the notation, not the toolset, while the specifiers evaluated both. The methods of evaluation were different for each set of participants and were described in the previous chapter. The evidence recorded here is anecdotal, but relevant because the participants of the study are representative of the type of people who would work with formal specification in industry. An effort has been made to document problems with the evaluation methods so that they can be taken into consideration when drawing conclusions from these results.

9.1 Nuclear Engineers

General Results

- *The role of the specification has to be understood*

  Communicating with people from a different field of expertise is always difficult because the terminology used is different and each group makes assumptions about the knowledge of the other that often prove to be incorrect. In this experiment, a particularly troublesome issue was the role of the specification in software development. One of the participants considered it computer code and wanted to see the execution to check correctness. Another considered it a summary that should be easy to read and not contain many details. Since the role of the specification is debated within the software community, it was difficult to provide the nuclear engineers with an exact definition, but the lesson learned was that it was vital to convey an understanding of the role of the specification before any further discussion.

- *Direct and indirect influence on the system are difficult to distinguish*

  A common difficulty for the nuclear engineers in understanding the specifications was with the difference between direct and indirect influence on the state of the system. The nuclear control system is reactive, meaning that it is constantly making alterations in response to input received from sensors. A change in the height of a rod causes changes in the sensor values. The height of the rod can be altered directly by the system, but the sensor values change indirectly as a result of the movement of the rod. The formal specification notations designate parts of the system that can be influenced directly differently than those that cannot, for example Z uses primes and delta schemas to indicate items that can be changed directly. These designations were a constant source of questions because, along with the changes in the system from direct influence, there are expected indirect changes in the state of the system. By no means is this an argument to abolish the separate designations for items that can be directly influenced, rather to point out a difficulty in understanding these notations that is forgotten once the notation is familiar.

- *The use of constant identifiers is problematic*

  An interesting anecdote involves the use of constants. It is customary, in fact preached, in computer science that constants should be defined in one place and given identifiers so that no "magic" numbers are used throughout the rest of the system. The reasons are that the numbers are unexplained and, if changed, require the location of every use. To most of the nuclear engineers, this organization was preferable since they did not have the set point values memorized and the values would have to be checked against other documentation in an effort separate from the general perusal of the specification. However, one participant was confused by the use of constant identifiers rather than numbers. This suggests that the ability to dynamically replace the constant identifiers with their values would be useful when the specification is viewed by certain audiences.

Z

- *Z is effective for communication*

  The Z specification was described as meaningful and useful for communication. One participant felt comfortable with the notation after a short period of time, no longer needed full translations of the schemas, and began to find errors in the specification. This participant felt that, after a few iterations of discussion and correction of the specification, he would feel that there was a mutual understanding of the system.

- *Mathematical notation is not familiar*

  A surprising discovery was that the mathematical notation used in Z was not familiar to the nuclear engineers. One participant expressed the desire for a glossary of symbols, including for all, there exists, and implies. Another asked why words, which are universally understood, were not used in place of the symbols.

- *Errors in the specification were found by the presenter*

  An additional benefit of the presentation of the formal specification to the nuclear engineers was the discovery of errors in the specification by the presenter. In this case, the presenter of the Z specification was not the author, but another computer scientist familiar with the project, and the process of explaining the specification to the nuclear engineers uncovered errors. In this sense, the presentation of the specification served as a kind of inspection of the specification. This confirms the generally accepted view of the community.

PVS

- *PVS looks like computer code*

  The first impressions of the PVS specification were that it looked like computer code, it was too long, and there was too much text. One participant said he did not even want to try to read it. Another criticism was that there were too many variables.

- *Errors in the specification were evident, despite an inability to read the specification notation*

  Although the participant was not comfortable reading the PVS notation, a detailed explanation of the specification facilitated useful discussions that identified errors in the specification and in the specifiers' understanding of the system. One way that this occurred was that the participant would ask questions to check the model. He identified a misunderstanding of the power levels of the reactor that necessitated the redesign of a section of the specification. If this error had not been found until the system had been implemented, it would have been impossible to increase the power level of the reactor above about half of the value at which it is licensed to operate. The use of meaningful variable names was key to the understanding of the specification.

- *Errors in specification were found by the presenter*

  In addition to errors found by the nuclear engineers, presenting the specification caused the specifier to discover an error in his own specification.

Statecharts

- *Statecharts's graphical notation is appealing*

  After less than an hour of introduction to the statecharts notation and specification, one participant was no longer intimidated by the notation and was able to understand the specification without assistance. The graphical notation was appealing, as well as the obvious flow of the system following the arrows. The cliche "a picture is worth a thousand words" was used repeatedly. The structure of the specification was much more evident in statecharts than the other two notations because of its hierarchical nature.

- *Statecharts is difficult to search and navigate*

  In a very detailed examination of the specification, participants complained of the difficulties of knowing the state of the whole system at once and of identifying the results of actions since the actions could affect any page of the specification. Whenever the details of a state were included in the diagram of that state rather than being saved in another file, the lack of abstraction seemed to be confusing.

- *Statecharts is easy to learn!*

  Within two hours of discussion of the specification, the participants displayed the desire to learn the syntax of the notation in order to understand the subtleties of the specification. A large number of errors were identified during the discussion of the specification and the need for additional robustness was evident. The participants found the specification easy to understand with the explanation from the specifier and felt that they could then continue to study it alone. They also felt comfortable enough with the notation that, if there were changes to be made to the system, they felt they could write statecharts of the proposed changes!

- *The statecharts specification is superior to existing documentation*

  The participants from the nuclear reactor staff felt that the specifiers understood the system better than most of the operators. They felt that they could eventually come to an agreement that the statechart specification correctly described the system and did not feel that they would have the same confidence with an English document. They said that this specification had the potential to be used in the training of their operators and perhaps even to replace their SAR which describes the control of the nuclear reactor.

9.2 Computer Scientists

Background of Participants

The participants in this portion of the study were seven computer science students. There was one undergraduate, four students working toward or finished with a master's degree, and two Ph.D. candidates. Two participants had a year or less work experience developing software, three had one to five years experience, and two had more than five years of work experience. All had knowledge of the C programming language. Regarding their experience with formal specification methods, four had no experience prior to this study, two had a segment of a course, and one had an entire course. All had some, but not extensive, knowledge of basic science and engineering and little to no knowledge of nuclear reactors.

Z

- *Z is fairly easy to understand and navigate*

  The Z specification was generally well-structured and this aided the participants in understanding and searching the specification. However, one participant expressed difficulty locating the definitions of types since they are not defined near their use and another suggested that the specification would be easier to search, navigate, and use for reference if there were a table of contents. The participants felt strongly that Z would aid communication about the system, however they considered it only average for use in the maintenance phase as an introduction to the system and as a reference document about the system. Familiarity with logic symbols, the smallness and simplicity of the notation, and the natural language descriptions aided the participants in understanding the specification.

- *Z is reasonably easy to learn*

  None of the participants felt very confident in their ability to use Z after this short introduction. A few of the participants felt that Z was harder to learn than a programming language, but most felt that it was as easy or easier to learn. Difficulties in learning Z were attributed to the mathematical notation, the unusual delimiters of inputs and outputs, and the unfamiliarity of the notation in general. No one thought that Z was too large of a notation and almost everyone thought the complexity of the notation was appropriate for specification.

- *Z is implementable*
  After a thorough inspection of the description of the scram logic in the specification, everyone saw ways that it could be implemented. No one was sure that the description was complete, however. Some participants found errors in the specification. Upon quick perusal of the rest of the specification, almost everyone felt that all the features of the notation were implementable. It was practically unanimous that Z provided the appropriate level of detail about the system for a specification.

PVS

- *PVS received low marks in structure, understandability, searching, and navigation*
  Although PVS is structured like code in the C programming language which all participants claimed a lot or extensive knowledge of, it received low ratings in the areas of structure, understandability, and searching. One participant cited the formatting as hindering understanding. It was deemed average to bad for use during the maintenance phase as an introduction to the system or as a reference document. The answers were widely varied as to whether PVS would aid communication between people involved in the software development process.

- *Responses about the ease of learning PVS were mixed*
  None of the participants felt confident using PVS after this short introduction. Most felt that PVS was as easy or easier to learn than a programming language, but a few felt that it was harder to learn. No one thought that the PVS notation had too few features and most people thought that it had the appropriate amount of complexity, while a few felt that it was too complex. Difficulties in learning the notation were attributed to the size and complexity of the notation and the difficulty in understanding the key-

words and constructs. However, some participants felt that the keywords and constructs were easy to learn and PVS was similar to other notations they were familiar with.

- *PVS seems implementable*

  After examining the scram logic in the PVS specification, everyone saw ways that it could be implemented, but a few saw some problems. No one was certain whether the description of the scrams was complete. After a quick inspection of the rest of the specification, the participants felt that everything was implementable. There was a wide range of responses when asked whether PVS provided the appropriate level of detail for a specification.

Statecharts

- *Statecharts is easy to understand*

  Statecharts was described as well-structured and this aided the participants in understanding the specification. Difficulties in understanding the specification were attributed to the global nature of events and the division of the specification over many pages. The responses indicated strongly that statecharts would aid communication between people in the development of a software product.

- *Statecharts is difficult to navigate and search*

  The structure of statecharts aided in searching, but one participant noted that the specification would be easier to navigate, search, and use as reference, if it had a table of contents. It was deemed average for use in the maintenance phase as an introduction to the system and as a reference document.

- *Statecharts was rated fairly easy to learn*

  The participants did not feel confident in their ability to specify a system using statecharts at this point. Difficulties in learning statecharts were attributed to the notation being unlike any notation they had seen before and the constructs being difficult to understand. However some people felt that statecharts was easy to learn because the

Statecharts		PVS			Z	
total for 3 successful searches	average successful search	total for 2 successful searches	average successful search	time for 1 unsuccessful search	total for 3 successful searches	average successful search
11:00	3:40	7:00	3:30	15:00	4:00	1:20
4:33	1:31	2:27	1:14	4:33	3:31	1:10
10:55	3:38	5:30	2:45	15:00	6:39	2:13
6:15	2:05	3:52	1:56	:40	4:14	1:25
8:30	2:50	3:50	1:55	4:00	3:45	1:15
2:10	:43	3:15	1:38	3:50	2:37	:52
2:45	:55	1:45	:53	5:30	1:50	:37

Table 5: Time to Locate Specific Information (min:sec)

notation was familiar, graphical, small and simple, and the constructs were easily understood. Most of the participants thought that statecharts was as easy or easier to learn than a programming language.

- *Statecharts can be implemented*

  After studying the scram logic described in the statecharts specification, everyone saw ways to implement it, however no one was certain the description was complete. After a quick survey of the specification, almost every participant thought that all the features of the notation were implementable. It was almost unanimous that statecharts provided the appropriate level of detail about the system. Most of the participants thought that statecharts notation contained the appropriate level of complexity.

Searching the Specification

The computer scientists were asked to measure the amount of time required to locate three particular pieces of information in each of the specifications. In the statecharts and Z specifications, the answers to all three questions were present in the specification, however the answer to one of the questions was not present in the PVS specification. This was not an intended feature of the experiment, but in retrospect it would have been interesting to collect this data on all the specifications since it indicates the amount of time needed for the participant to be sure the answer is not present. table 5 contains the data from the seven participants.

- *Z was easiest to search, then PVS*

  Although most people had the impression that statecharts was the easiest to search because of the limited amount of text, in fact the time needed to search the statecharts specification was consistently higher. Z had the lowest times overall. The results for PVS may have been affected by having only two successful searches.

9.3 Authors

Z

- *Training, documentation, and tools are available*

  The Z specification was written in Framemaker for Windows using a Z font. Training is becoming more widely available for learning to both read and write Z. In addition, there are books, papers, newsgroups, and conferences about Z. Complete specifications for real systems have been published. Toolsets are also becoming more widely available. There are typecheckers, static analyzers, and theorem provers for Z. However, there is little to no training in the use of these tools.

- *The expressivity of Z has limits*

  The Z notation supports integers, but not real numbers or the declaration of constant identifiers. Semantics for timing principles are not built-in to the notation, but they can be expressed in Z and extensions are available that provide this capability. Z is not suited for describing the behavior of a user-interface, so it needs to be able to integrate with a tool that can. Non-functional requirements can not be described in the Z notation, but Z is conventionally accompanied by natural language text in which these requirements can be documented.

- *The Z notation is built for readability*

  Infinite-length identifiers are permitted, common mathematical notation used, tabs can be used to format the text, and lower and upper case letters can be used in identifier names. The one convention that is not supported is the use of underscores in identifier names. Different levels of abstraction, structuring, and information hiding can be used in Z, but this is not enforced.

- *The use of a text editor has many benefits*

  The editor used in this project was a text editor, so it tolerates incompleteness during composition and supports printing and regular expression matching. It also provides a selection of common file formats and is clearly compatible with a documentation system. Because it is text editor, rather than an editor specific to Z, it does not allow the user to view the specification at different levels of abstraction.

- *Group development issues were not encountered*

  The editor does not provide internal version control, but can be used with external version control systems. It supports multiple users and separate compilation since it is easy to manipulate text. These issues are more critical in other tools, such as a typechecker or theorem prover, which were not evaluated in this study.

- *Symbols in the notation make Z difficult to compose and modify*

  It is very time consuming to compose a Z specification because the symbols of the notation are not found on a traditional keyboard. A symbol pad or elaborate key sequences must be used. Additionally, in Framemaker, the schemas are represented as figures and the Z text is contained in text frames. Modification of these is tedious.

PVS

- *Documentation and tutorials are available*

  PVS 2.0 requires about 40 megabytes of memory and was used on Solaris. Training to learn to write the notation exists, but is not readily available. Documentation and tutorials for the toolset have been published and were very useful. Technical support for the toolset is available via email. There is documentation of the notation and toolset available, as well as specifications of example systems. Real systems have been specified using PVS, but the complete specifications are not available. The toolset provides type-checking and theorem proving capabilities.

- *The PVS notation is fairly expressive*

  The notation can represent integers and real numbers. Constants can be defined, although not in a straightforward manner. Timing is not built-in, but can be specified. It is possible to document non-functional requirements as comments. The notation provides infinite-length identifiers, meaningful keywords, and upper and lower case letters and underscores are allowed in identifiers. Common mathematical notations that require symbols not on the keyboard are expressed with the English words, such as for all, exists, and implies. Tabs to format the text, however, are not well-supported. User-interfaces cannot be represented in PVS, so it needs to be compatible with a tool that can. The notation supports, but does not enforce, different levels of abstraction.

- *The editor is capable, but not user-friendly*

  Emacs is the user-interface for PVS and saves the files as text which is clearly compatible with any documentation system. The files can also be pretty-printed in LaTex. The specification is easy to navigate, search, and modify. The user-interface is not very user-friendly.

- *Group development is not well supported*

  Multiple users can compose files, but typechecking and other capabilities that use more than one file require the files to be in one PVS context, which usually means they must be in the same directory. PVS does not have built-in version control and it is unclear whether external version control can be used.

- *There is no published method for building a PVS specification*

  It took several months for the specifier to develop a successful approach to structuring theories.

Statecharts

- *The documentation is useful, but training and tutorials are scarce*

  The statecharts specification was written using Express 3.1.3 of the STATEMATE family of tools running on SunOS 4.1.3. It required 30 megabytes of memory. The tool supports three notations; of these, only statecharts was used. The graphical notation is supplemented by a set of forms that provide a means to express properties that are difficult to represent graphically. These forms were also not used in this study. Training in writing the notation exists, but is not locally available. The documentation on the notation is useful and readily available, however the manual on the toolset, which included a tutorial, could use improvement.

- *The expressiveness of statecharts has limits*

  The notation can represent integer and real numbers, but not constants. Some timing is built in, but other notations based on statecharts have extended this capability. Non-functional requirements can be documented in the notation, but it is not easy or natural to do so. User-interfaces cannot be represented. The notation does provide strong support for representing the environment effecting the system.

- *Support for readability of the text is lacking*

  While the graphical nature of the notation is very readable, support for readability of the text is limited to infinite-length identifiers and allowance of underscores. Meaningful keywords are abbreviated automatically by the editor to strings lacking readability. Tabs and extra spaces are automatically removed and all identifiers are written in capital letters.

- *Group development is over-constrained*

  External version control can not be used because the tool saves the information in a complex database of directories in the workspace of the user. The toolset does support multiple users and has built-in version control. It requires the categorization of the users into groups, such as project managers, and provides varying permission to make modifications based on these categorizations. The internal database of files and tight control of permissions makes it difficult or impossible to move files between projects, delete files, delete projects, or to group files, for example into different directories.

- *It is difficult to extract a statechart from the STATEMATE tool*

  It is difficult to print the statecharts or import them into another documentation system because they are not saved in a common file format. They can be printed using a plotter or saved as postscript, but the author of this specification found it easiest to use the XV tool to grab the image and save it in a format that can be more easily manipulated.

- *The toolset provides many capabilities*

  Code can be generated automatically from this tool. Several static checking capabilities are provided, including completeness, consistency, and non-determinism. Animation is provided which helped identify problems in the specification. There is support for structured design in the STATEMATE tool and various reports can be generated.

- *The user-interface is inconsistent and not user-friendly*

  The menuing system that drives the toolset is not intuitive. It was difficult to make changes to the specification in this notation. Because it is graphical, the spacing had to be constantly modified. Identifiers are associated with graphical objects, so if the object was moved, deleted, or a naming conflict arose, often identifiers would be automatically deleted. These identifiers could be conditions on a transition consisting of several lines of text. There is no support for regular expression matching.

- *Statecharts enforces structure, but not information hiding*

  The notation enforces a hierarchical structure that creates levels of abstraction and the editor supports the ability to view the specification at different levels of abstraction. While structuring was well-supported, information hiding was not. Most elements are global.

Shortcomings in the Evaluation Method

A shortcoming of this evaluation was that the nuclear engineers, while familiar with the reactor, had not been in discussions about the proposed computer system before. This had several repercussions. Terminology used by the participants was not consistent with each other or with the specifications. They did not know which parts of the system were being modeled. It would have been more realistic if the nuclear engineers had been part of the project from the start of the development of the specifications and if the evaluation consisted of multiple interviews spread over a period of time.

The Z specification contained a mixture of Z and natural language, whereas the other two specifications contained only the formalisms. This natural language may have aided the understandability of the specification. The natural language also grouped together the schemas pertaining to a particular part of the system, such as the schemas dealing with alarms. This structure is not imposed by the formalism, but helped in navigation and understanding.

Only a subset of the notations being evaluated were used in the specification, so difficulties with other features may occur. This also may have affected the ratings of the size, complexity, and difficulty of learning the notations.

The evaluations by the nuclear engineers and computer scientists were done with paper versions of the specifications, so no benefits or problems associated with the toolsets were studied by them. Also, no experimentation was done with a natural language specification, so no conclusions can be drawn about the usefulness of these notations in comparison to natural language. Additionally, the role the specification in the software development process was not explained to the computer scientists, so their judgements about the appropriateness of the level of detail may be unsubstantiated.

9.4 Implications

- *Modeling is hard*

  An issue that is not often mentioned is that modeling is hard. Even if the syntax of the notation is simple, the modeling concepts are difficult. This is true for computer scientists learning a formal specification notation and it is true for nuclear engineers studying a completed specification for the first time. Nobe and Warner discuss their difficulty with modeling when using statecharts in [NW96].

- *Formal notations are easy to learn to read*

  Although none of the participants could read the formal specifications before receiving an introduction to the notation, they were able to learn the notations well enough to understand the specifications in a short amount of time. Interviews with the nuclear engineers lasted no longer than two hours and they could learn to read one formal notation fairly well in that time. The computer scientists were asked to spend approximately an hour to an hour and a half with each notation. This time included reading a brief introduction to the notation. Like the nuclear engineers, they also felt fairly comfortable reading the notations in this short time frame.

- *Specification is a group effort*

  Creating a specification takes a lot of time and effort on the part of the specifiers. It may be written by one person or a small group. The goal of the specification, however, is communication between the specifiers and the clients or domain experts, so input from all parties should be heeded. In this study, the nuclear engineers requested a lot of changes in identifier names and organization of the specification. Since they must accept the specification as a description of the desired system, every effort must be made on the part of the specifiers to accommodate the suggestions of the client or domain expert in order to make the model intuitive and the terminology familiar. If there is existing documentation on the system, the specification should have a one-to-one correspondence with this documentation to facilitate checking for consistency.

- *Meaningful variable names are key to understanding*

  Despite effort by the specifiers to choose meaningful variable names, the identifiers were still confusing to the nuclear engineers. Appropriate variable names made the notation almost immediately understandable, while poor choices led to lengthy discussions. All of the nuclear engineers wanted to change the variable names.

- *There is no road back to natural language specification.*

  Once the nuclear engineers had experience with one or more of the formal specification notations, they said they would never trust a natural language specification again. They were impressed by the level of understanding of the system that was required to write the specifications and felt that with natural language they could never be sure that the words were not just copied down with little understanding of the system. While they would have liked some natural language to accompany the formal specifications, they wanted to retain the formalisms.

- *Inspections of the specification are priceless*

  Many errors, poor structure, and confusing identifiers can be eliminated from the specification through inspection and discussion with other specifiers. During the development of the three specifications used in this study, several informal inspections occurred and resulted in major revisions. The presentation of the specification to other specifiers or to a client or domain expert often caused the specifier to discover errors in his own specification.

- *Completeness should be checked by a computer*

  When asked about the completeness of the specifications, the participants in this study balked at the idea. In protest, one calculated the space of cases that would have to be checked. Regardless of the enormity of the state space, completeness of the specification is vital to its success as a reference document about the system. Completeness checks are not an activity for humans, rather for a computer. Research is ongoing in tools that check completeness. These would be excellent additions to the toolsets that support these formal notations.

- *Formal specifications appear to be implementable, but more study is needed*

  The computer scientists in this evaluation felt that they could implement the specifications, but they were not required to demonstrate this ability. Clearly the ease of implementation is an important criteria for formal notations and further study should be done on this issue.

- *Support for navigation and searching is lacking*

  Practically every participant complained of the difficulty of navigating and searching the specifications. They expressed the desire for a table of contents or some similar overview of the structure. Tool support for searching was also deficient. If the specification is to be used for reference, it must be easy to navigate and search.

- *User-interfaces are not friendly*

  Little emphasis seems to have been spent on making the user-interfaces for these toolsets friendly, yet this is the first impression that a new user receives of the capability of the toolset. In order for formal specification to gain popularity in industry, the user-interface must not be more difficult to learn than the formal notation!

- *Compatibly with other software packages is vital*

  The specification toolset must integrate into the larger development environment. All three of the notations evaluated were lacking the ability to specify user-interface behavior, so they need to be compatible with a tool that has that capability. Printing, exporting, and importing portions or all of the specifications are necessary functionalities of the toolset. The use of existing text editors by PVS and Z was more successful than the indigenous editor used for statecharts, but none fully satisfied the needs of a specification editor.

9.5 Questions Raised

- *Should a client or domain expert be able to read a formal specification without assistance?*

  Since a natural language specification can be read by a client or domain expert without aid, it can be mailed to them for extensive examination. If the specification written in a formal notation can not be read by the client or domain expert without help, then all of the examination, discussion, and checking of the specification must be done in meetings with the specifier. This would take a lot of time and the subtleties of the model might be missed if the client or domain expert can not read the notation. If the formal specification could be read by the client or domain expert after some introduction, then they could study it at their own leisure. The question then focuses on how much introduction is reasonable to require of the client or domain expert.

- *Should the specification always be referenced on-line?*
  In this study, the formal specifications were presented to the nuclear engineers on paper, however there might be benefits to letting them view it on-line. For example, statecharts provides the ability to animate the model and this could aid the client or domain expert in checking the behavior of the system. If the formal specification is to be viewed on-line, then questions again arise about the client or domain expert's ability to view the specification without the assistance of the specifier and, if they can view it alone, the availability of the toolset and the ease of learning to use it.

- *Should natural language be included in Z specifications?*
  It is customary to write Z interspersed with natural language descriptions of each schema. When one participant was presented with the Z specification without any introduction to the notation, he read only the natural language and ignored the Z altogether. After an explanation of a few schemas, however, he felt that he could understand the specification without aid. Another participant was presented the specification without the natural language and each schema was explained. He felt that, with a short natural language description of each schema, he could read the Z specification easily. When interspersed with natural language, the Z specification seemed to work much better as a stand-alone document than any of the three formalisms alone.

10 Conclusions

During this evaluation of Z, Statecharts, and PVS, the well-known benefits of formal specification were seen. The precision of the notation and automated analysis tools have the potential to provide industrial practitioners with much needed improvements in the development process and product quality. However, significant refinements are essential before cost-effective usage in industry is possible. These shortcomings pose substantial barriers to the acceptance of formal specification in industry.

Considerable further research is needed before formal specification methods will be ready for routine industrial use. Currently, formal specification methods generally consist of a notation supported by immature tools. A method for creating and using formal specifications is needed. Improvements must be made in the notations and tools. They need to support the development of large, multi-authored systems. The resources required to learn to use the formal specification methods must be widely available before they will be adopted by industry.

Conclusions such as these resulted from this study and indicated the following research agenda:

- *Formal specification method as well as notation*
  In order to become routinely used in industry, there must be a method for using formal specifications. This method should prescribe the steps needed to specify the requirements of a system using the formal notation. It must be evident that formal specifica-

tion can be incorporated into current industrial practice. The method for using formal specification must be compatible with other methods currently in use. Inspections of the specification should be included in its development. Investigation into the use of formal specification during interaction with clients is needed.

- *Attention to vital scale-up issues*

  Current formal specification notations and tools are not sufficient for use in large, multi-authored systems. They must facilitate group development. Many different programmers in different locations and on different platforms often work together on a project.

  Some of the specific requirements that are dictated by the need to specify large systems are:

  - *Group development*
  - *Specification navigation*
  - *Specification evolution*

- *Preparation for wide-spread use*

  The requirements that need to be met to permit the wide-spread use of formal specifications are the following:

  - *Availability of training*
  - *Quality of documentation for the notation, tools, and method*
  - *Availability of specifications of real systems*
  - *Availability of support from other users*

- *Application of lessons from programming language research*

  The requirements suggested by prior research in the theory of programming languages and design that could be adopted in formal specifications are the following:

  - *Structuring mechanisms*
  - *Information hiding*
  - *Use of symbolic constants*

- *Maturity of toolset*

  The requirements for high-quality tool support in the use of formal specifications are the following:

  - *Compatibility with other software and hardware*
  - *Improvements in usability*
  - *Association with tools that provide expressibility that is lacking in the notation*
  - *Provide unified working environment*
  - *Provide analysis tools*

- *Social and cultural acceptance of science in software development*

  The approach to evaluation employed here was successful in identifying flaws in three prominent formal specification methods. Although the flaws that were identified were not previously unknown, the use of current software development practice as the basis for this evaluation provided rationale for the choice of criteria that led to the identification of these flaws. These criteria, together with their systematic derivation, form a detailed agenda for the improvement of formal specification methods. Although there may be other barriers, unless these criteria are satisfied, there is little hope of formal specification obtaining widespread use in industry.

11 References

[AA92] Amla, Nina, and Paul Ammann. "Using Z Specifications in Category Partition Testing." COMPASS '92. Proceedings of the Seventh Annual Conference on Computer Assurance, p.3-10.

[AH96] Archer, Myla M., and Constance L. Heitmeyer. "Mechanical Verification of Timed Automata: A Case Study." Proceedings of the 1996 Real-Time Technology and Applications Symposium, 1996.

[Ard96] Ardis, Mark A., et al. "A Framework for Evaluating Specification Methods for Reactive Systems: Experience Report." IEEE Transactions on Software Engineering 22(6):378-389, June 1996.

[BMweb] R-Active. "R-Active Concepts, Inc." http://www.r-active.com/ (Jan. 1997).

[But93] Butler, Ricky W. "An Elementary Tutorial on Formal Specification and Verification using PVS 2." NASA Technical Memorandum 108991, NASA Langley Research Center, Hampton, VA, June 1993. Revised June 1995.

[But96] Butler, Ricky W. "An Introduction to Requirements Capture using PVS: Specification of a Simple Autopilot." NASA Technical Memorandum 110255, NASA Langley Research Center, Hampton, VA, May 1996.

[CGR93] Craigen, Dan, and Susan Gerhart, and Ted Ralston. "An International Survey of Industrial Applications of Formal Methods." U.S. Department of Commerce, March 1993.

[CTLR93] Clements, P. C., and C. L. Heitmeyer, and B. G. Labaw, and A. T. Rose. "MT: A Toolset for Specifying and Analyzing Real-Time Systems." Proceedings of the Real-Time Symposium, Raleigh-Durham, NC, December 1-3, 1993, pp. 12-22.

[CW96] Clarke, Edmund M., and Jeannette M. Wing. "Formal Methods: State of the Art and Future Directions." ACM Workshop on Strategic Directions in Computing Research--Group Report: Formal Methods, June 14-15, 1996, Cambridge, MA, USA.

[Day93] Day, Nancy. "A Model Checker for Statecharts (Linking CASE tools with Formal Methods)." Technical Report 93-35, Department of Computer Science, University of British Columbia, October 1993.

[Day94] Day N., and J. Joyce, and M. Donat. "S: A Machine Readable Specification Notation based on Higher Order Logic." Proceedings of the 1994 International Meeting on Higher Order Logic Theorem Proving and its Applications, Lecture Notes in Computer Science, vol. 859, pp.285-299, Springer-Verlag.

[Dil94] Diller, Antoni. Z: An Introduction to Formal Methods. Chichester: John Wiley & Sons,1994.

[Fau95] Faulk, Stuart. "Software Requirements: A Tutorial." Technical Report NRL/MR/5546--95-7775, Naval Research Laboratories, November 14, 1995.

[Hal90] Hall, Anthony. "Seven Myths of Formal Methods." IEEE Software 7(5):11-19, September 1990.

[Har87] Harel, David. "Statecharts: A Visual Formulation for Complex Systems." Science of Computer Programming 8(3):231-274, June 1987.

[Har88] Harel, David. "On Visual Formalisms." Communications of the ACM. 31(5):514-530, May 1988.

[HBGL95] Heitmeyer, C., and A. Bull, and C. Gasarch, and B. Labaw, "SCR*: A Toolset for Specifying and Analyzing Requirements." Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95), Gaithersburg, MD, June 25-29, 1995, pp. 109-122.

[Hen80] Heninger, Kathryn L. "Specifying Software Requirements for Complex Systems: New Techniques and Their Application." IEEE Transactions on Software Engineering 6(1):2-13, January, 1980.

[HJL96] Heitmeyer, Constance L., and Ralph D. Jeffords, and Bruce G. Labaw. "Automated Consistency Checking of Requirements Specifications." ACM Transactions on Software Engineering and Methodology 5(3):231-261, July 1996.

[HL96] Heimdahl, Mats P. E., and Nancy G. Leveson. "Completeness and Consistency in Hierarchical State-Based Requirements." IEEE Transactions on

Software Engineering 22(6):363-377, June, 1996.

[HM83] Heitmeyer, C., and J. McLean. "Abstract Requirements Specification: A New Approach and Its Application." IEEE Transactions on Software Engineering 9(5):580-589, September 1983.

[Hoo95] Hooman, Jozef. "Verifying part of the ACCESS.bus Protocol using PVS." Proceedings 15th Conference on the Foundations of Software Technology and Theoretical Computer Science, LNCS 1026, Springer-Verlag, pages 96-110, 1995.

[HS96] Havelund, Klaus, and N. Shankar. "Experiments in Theorem Proving and Model Checking for Protocol Verification." Proceedings of Formal Methods Europe (FME '96), Springer-Verlag Lecture Notes in Computer Science No. 1051, pp. 662-681, March 1996, Oxford, UK.

[iLo87] Harel, D., and A. Pnueli, and J. P. Schmidt, and R. Sherman. "On the Formal Semantics of Statecharts (Extended Abstract)." Proceedings, Symposium on Logic in Computer Science, pp.54-64, Ithaca, New York, 22-25 June 1987. The Computer Society of the IEEE.

[iLo90] Harel, David, et al. "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." IEEE Transactions on Software Engineering 16(4):403-414, April 1990.

[iLoweb] iLogix. "i-Logix Success Stories." http://www.ilogix.com/company/success.htm (6 May 1997).

[Jac95] Jacky, Jonathan. "Specifying a Safety-Critical Control System in Z." IEEE Transactions on Software Engineering 21(2):99-106, February 1995.

[JM94] Jahanian, F., and A. Mok. "Modechart: A Specification Language for Real-Time Systems." IEEE Transactions on Software Engineering 20(12):933-947, December, 1994.

[JMC94] Johnson, Steven D., and Paul S. Miner, and Albert Camilleri. "Studies of the Single Pulser in Various Reasoning Systems." Available: http://www.csl.sri.com/pvs-users.html (6 May 1997).

[LA94] Lutz, Robyn R., and Yoko Ampo. "Experience Report: Using Formal Methods for Requirements Analysis of Critical Spacecraft Software." Proceedings of the 19th Annual Software Engineering Workshop, pp.231--248, Greenbelt, MD, December 1994. NASA Goddard Space Flight Center.

[LHHR94] Leveson, Nancy G., and Mats P. E Heimdahl, and Holly Hildreth, and Jon Damon Reese. "Requirements Specification for Process-Control Systems."

IEEE Transactions on Software Engineering 20(9):684-707, September, 1994.

[MPJ94] Miner, Paul S., and Shyamsundar Pullela, and Steven D. Johnson. "Interaction of Formal Design Systems in the Development of a Fault-Tolerant Clock Synchronization Circuit." Computer Science Department, Indiana University, Technical Report No. 405, April 1994.

[NRP95] Nagasamy, Vijay, and Sreeranga Rajan, and Preeti R. Panda. "Fibre channel protocol: Formal specification and verification." Sixth Annual Silicon Valley Networking Conference. SysTech Research, April 1995.

[NW96] Nobe, C. R., and W. E. Warner. "Lessons Learned from a Trial Application of Requirements Modeling using Statecharts." Proceedings the Second International Conference on Requirements Engineering, April 15-18, 1996, pp. 86-93.

[Par96] Park, Seungjoon. "Computer Assisted Analysis of Multiprocessor Memory Systems." Ph.D. Thesis, Department of Electrical Engineering, Stanford University, June 1996.

[PD96] Park, Seungjoon, and David Dill. "Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions." 8th ACM Symposium on Parallel Algorithms and Architectures, Padova, Italy, June 1996.

[PH97] Pfleeger, Shari Lawrence, and Les Hatton. "Investigating the Influence of Formal Methods." Computer 30(2):33-43, February, 1997.

[PMS95] Puchol, Carlos, and Aloysius K. Mok, and Douglas A. Stuart. "Compiling Modechart Specifications." Technical Report CS-TR-95-38, Department of Computer Science, University of Texas at Austin, October 1, 1995.

[Pre92] Pressman, Roger S. "Softwate Engineering: A Practicioner's Approach." McGraw-Hill, New York, NY, 1992.

[PVSweb] Rushby, John. "SRI International Computer Science Laboratory." http://www.csl.sri.com/pvs-users.html (6 May 1997).

[Rus93] Rushby, John. "Formal Methods and the Certification of Critical Systems." Technical Report CSL-93-7, SRI International, December 1993.

[SC94] Sherrell, Linda B., and Doris L. Carver. "Experiences in Translating Z Designs to Haskell Implementations." Software--Practice and Experience 24(12):1159-1178, December 1994.

[SCweb] Stringer-Calvert, David. "High Integrity Systems Engineering Group."

http://www.york.ac.uk/~dwjsc100/compilers.html (6 May 1997).

[SM95] Srivas, Mandayam K., and Steven P. Miller. "Formal Verification of an Avionics Microprocessor." Technical Report SRI-CSL-95-4, Computer Science Laboratory, SRI International, June 1995.

[Spi90] Spivey, J. Michael. "Specifying a Real-Time Kernel." IEEE Software 7(5):21-28, September 1990.

[STM] iLogix. The Languages of Statemate. iLogix, 22 Third Avenue, Burlington, MA 01803. November 1987.

[STMweb] iLogix. "Statemate Family Overview." http://www.ilogix.com/products/statemat.htm (6 May 1997).

[UVAR] University of Virginia Reactor, "The University of Virginia Nuclear Reactor Facility Tour Information Booklet". http://minerva.acc.virginia.edu/~reactor

[UvarSC] University of Virginia Reactor Safety Committee, "University of Virginia Reactor Safety Analysis Report", http://minerva.acc.virginia.edu/~reactor

[VH96] Vitt, Jan, and Jozef Hooman. "Assertional Specification and Verification using PVS of the Steam Boiler Control System." Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control, LNCS 1165, Springer-Verlag, pages 453-472, 1996.

[Zweb] Bowen, Jonathan. "The Z Notation." http://www.comlab.ox.ac.uk/archive/z.html (6 May 1997).

Appendices

Appendix A
PVS Specification

```
cooling                   :    THEORY

  BEGIN

  header_status           :    TYPE = { UP, DOWN }
  pump_status             :    TYPE = { ON, OFF }
  line_valve_status       :    TYPE = { CLOSED, TO_AIR, TO_COMPRESSED }
  pressure_status         :    TYPE = { HIGH, NORMAL }
  cooling_status          :    TYPE =

      [# %RECORD
          header          :    header_status,
          pump            :    pump_status,
          sec_pump        :    pump_status,
          line_valve      :    line_valve_status,
          line_pressure   :    pressure_status
      #]


  lower_header(cool :  cooling_status) :
      cooling_status        =    cool WITH [header := DOWN]

  raise_header(cool :  cooling_status) :
      cooling_status        =    cool WITH [header := UP,
                                 line_valve := TO_COMPRESSED,
                                 line_pressure := HIGH ]

  bleed_line(cool   :  cooling_status) :
      cooling_status        =    cool WITH [line_valve := TO_AIR,
                                 line_pressure := NORMAL ]

  close_valve(cool  :  cooling_status) :
      cooling_status        =    IF line_valve(cool) = TO_AIR
                                 THEN cool WITH [line_valve := TO_COMPRESSED]
                                 ELSE cool
                                 ENDIF

  pump_off(cool     :  cooling_status) :
      cooling_status        =    cool WITH [pump := OFF]

  pump_on(cool      :  cooling_status) :
      cooling_status        =    cool WITH [pump := ON]

  sec_pump_off(cool :  cooling_status) :
      cooling_status        =    cool WITH [sec_pump := OFF]

  sec_pump_on(cool  :  cooling_status) :
      cooling_status        =    cool WITH [sec_pump := ON]
  pumps_off(cool    :  cooling_status) :
      cooling_status        =    cool WITH [pump := OFF, sec_pump := OFF]

  pumps_on(cool     :  cooling_status) :
      cooling_status        =    cool WITH [pump := ON, sec_pump := ON]


  END cooling
sensors                     :    THEORY

  BEGIN

  sensors_status            :    TYPE =

      [# %RECORD
          pool_temp         :    nat,
          pool_level        :    nat,
          pool_level_low    :    bool,
          power_indic1      :    nat,
          power_indic2      :    nat,
          water_cond        :    nat,
          react_period      :    nat,
          gamma_rad         :    nat,
          air_mont          :    nat,
          %duct_mont        :    nat,
          area_rad          :    nat,
          core_temp         :    nat,
          core_flow         :    nat,
          %line_pressure    :    bool,
%------------------------------------------------------------
          auto_ctrl_lost    :    bool,
          her_door_open     :    bool,
          dr_door_open      :    bool,
          sec_pump_off      :    bool,
          thimble_too_hot   :    bool,
          key_removed       :    bool,
          bridge_rad        :    nat,
          face_rad          :    nat,
          t_door_open       :    bool,
          ehatch_open       :    bool,
          r1_up             :    bool,
          r1_down           :    bool,
          r1_seated         :    bool,
          r1_mag_eng        :    bool,
          r2_up             :    bool,
          r2_down           :    bool,
          r2_seated         :    bool,
          r2_mag_eng        :    bool,
          r3_up             :    bool,
          r3_down           :    bool,
          r3_seated         :    bool,
          r3_mag_eng        :    bool
          #]


  raise_shim_rods_10(sensors       :  sensors_status) :
      sensors_status        =    sensors WITH [r1_up      := false,
                                              r1_down    := false,
                                              r1_seated  := false,
                                              r1_mag_eng := true,
                                              r2_up      := false,
                                              r2_down    := false,
                                              r2_seated  := false,
                                              r2_mag_eng := true,
                                              r3_up      := false,
                                              r3_down    := false,
                                              r3_seated  := false,
                                              r3_mag_eng := true]

  lowest_shim_rod_position(sensors :  sensors_status) :
      sensors_status        =    sensors WITH [r1_up      := false,
                                              r1_down    := true,
                                              r1_seated  := true,
                                              r1_mag_eng := true,
                                              r2_up      := false,
                                              r2_down    := true,
                                              r2_seated  := true,
                                              r2_mag_eng := true,
                                              r3_up      := false,
                                              r3_down    := true,
                                              r3_seated  := true,
                                              r3_mag_eng := true]




  END sensors
alarm_display               :    THEORY

  BEGIN

  alarm_status              :    TYPE = { BOTH_ON, YELLOW_ON, BOTH_OFF }

  alarms_status             :    TYPE =

      [# %RECORD
          %spare_alarm       :    alarm_status,
          core_temp_alarm   :    alarm_status,
          control_rod_alarm :    alarm_status,
          air_mont_alarm    :    alarm_status,
          water_cond_alarm  :    alarm_status,
          area_rad_alarm    :    alarm_status,
          her_door_alarm    :    alarm_status,
          sec_pump_alarm    :    alarm_status,
          gamma_rad_alarm   :    alarm_status,
          dr_door_alarm     :    alarm_status,
          thimble_temp_alarm:    alarm_status,
          scram_alarm       :    alarm_status
      #]


  scram(alarms                         :  alarms_status) :
      alarms_status         =    alarms WITH [scram_alarm := BOTH_ON]

  reset_scram(alarms                   :  alarms_status) :
     alarms_status       =   alarms WITH [scram_alarm := IF scram_alarm(alarms) /= BOTH_OFF
                                                         THEN YELLOW_ON
                                                         ELSE BOTH_OFF
                                                         ENDIF]

  clear_alarms(alarms                  :  alarms_status) :
      alarms_status         =    alarms WITH
          [ core_temp_alarm      :=  IF (core_temp_alarm(alarms) /= BOTH_ON)
                                     THEN BOTH_OFF
                                     ELSE BOTH_ON
                                     ENDIF,
          control_rod_alarm      :=  IF (control_rod_alarm(alarms) /= BOTH_ON)
                                     THEN BOTH_OFF
                                     ELSE BOTH_ON
                                     ENDIF,
          air_mont_alarm         :=  IF (air_mont_alarm(alarms) /= BOTH_ON)
                                     THEN BOTH_OFF
                                     ELSE BOTH_ON
                                     ENDIF,
          water_cond_alarm       :=  IF (water_cond_alarm(alarms) /= BOTH_ON)
                                     THEN BOTH_OFF
                                     ELSE BOTH_ON
                                     ENDIF,
          area_rad_alarm         :=  IF (area_rad_alarm(alarms) /= BOTH_ON)
                                     THEN BOTH_OFF
                                     ELSE BOTH_ON
                                     ENDIF,
          her_door_alarm         :=  IF (her_door_alarm(alarms) /= BOTH_ON)
                                     THEN BOTH_OFF
                                     ELSE BOTH_ON
                                     ENDIF,
          sec_pump_alarm         :=  IF (sec_pump_alarm(alarms) /= BOTH_ON)
                                     THEN BOTH_OFF
                                     ELSE BOTH_ON
                                     ENDIF,
          gamma_rad_alarm        :=  IF (gamma_rad_alarm(alarms) /= BOTH_ON)
                                     THEN BOTH_OFF
                                     ELSE BOTH_ON
                                     ENDIF,
          dr_door_alarm          :=  IF (dr_door_alarm(alarms) /= BOTH_ON)
                                     THEN BOTH_OFF
                                     ELSE BOTH_ON
                                     ENDIF,
          thimble_temp_alarm     :=  IF (thimble_temp_alarm(alarms) /= BOTH_ON)
                                     THEN BOTH_OFF
                                     ELSE BOTH_ON
                                     ENDIF,
          scram_alarm            :=  IF (scram_alarm(alarms) /= BOTH_ON)
                                     THEN BOTH_OFF
                                     ELSE BOTH_ON
                                     ENDIF
          ]


  core_temp_alarm_signal_on(alarms     :  alarms_status) :
      alarm_status          =    BOTH_ON

  core_temp_alarm_signal_off(alarms    :  alarms_status) :
      alarm_status          =    IF core_temp_alarm(alarms) /= BOTH_OFF
                                 THEN YELLOW_ON
                                 ELSE BOTH_OFF
                                 ENDIF

  control_rod_alarm_signal_on(alarms   :  alarms_status) :
      alarm_status          =    BOTH_ON

  control_rod_alarm_signal_off(alarms  :  alarms_status) :
      alarm_status          =    IF control_rod_alarm(alarms) /= BOTH_OFF
                                 THEN YELLOW_ON
                                 ELSE BOTH_OFF
                                 ENDIF

  air_mont_alarm_signal_on(alarms      :  alarms_status) :
      alarm_status          =    BOTH_ON

  air_mont_alarm_signal_off(alarms     :  alarms_status) :
      alarm_status          =    IF air_mont_alarm(alarms) /= BOTH_OFF
                                 THEN YELLOW_ON
                                 ELSE BOTH_OFF
                                 ENDIF
```

```
water_cond_alarm_signal_on(alarms    :  alarms_status) :
    alarm_status          =    BOTH_ON

water_cond_alarm_signal_off(alarms   :  alarms_status) :
    alarm_status          =    IF water_cond_alarm(alarms) /= BOTH_OFF
                               THEN YELLOW_ON
                               ELSE BOTH_OFF
                               ENDIF

area_rad_alarm_signal_on(alarms      :  alarms_status) :
    alarm_status          =    BOTH_ON

area_rad_alarm_signal_off(alarms     :  alarms_status) :
    alarm_status          =    IF area_rad_alarm(alarms) /= BOTH_OFF
                               THEN YELLOW_ON
                               ELSE BOTH_OFF
                               ENDIF

her_door_alarm_signal_on(alarms      :  alarms_status) :
    alarm_status          =    BOTH_ON

her_door_alarm_signal_off(alarms     :  alarms_status) :
    alarm_status          =    IF her_door_alarm(alarms) /= BOTH_OFF
                               THEN YELLOW_ON
                               ELSE BOTH_OFF
                               ENDIF

sec_pump_alarm_signal_on(alarms      :  alarms_status) :
    alarm_status          =    BOTH_ON

sec_pump_alarm_signal_off(alarms     :  alarms_status) :
    alarm_status          =    IF sec_pump_alarm(alarms) /= BOTH_OFF
                               THEN YELLOW_ON
                               ELSE BOTH_OFF
                               ENDIF

gamma_rad_alarm_signal_on(alarms     :  alarms_status) :
    alarm_status          =    BOTH_ON

gamma_rad_alarm_signal_off(alarms    :  alarms_status) :
    alarm_status          =    IF gamma_rad_alarm(alarms) /= BOTH_OFF
                               THEN YELLOW_ON
                               ELSE BOTH_OFF
                               ENDIF

dr_door_alarm_signal_on(alarms       :  alarms_status) :
    alarm_status          =    BOTH_ON

dr_door_alarm_signal_off(alarms      :  alarms_status) :
    alarm_status          =    IF dr_door_alarm(alarms) /= BOTH_OFF
                               THEN YELLOW_ON
                               ELSE BOTH_OFF
                               ENDIF

thimble_temp_alarm_signal_on(alarms  :  alarms_status) :
    alarm_status          =    BOTH_ON

thimble_temp_alarm_signal_off(alarms :  alarms_status) :
    alarm_status          =    IF thimble_temp_alarm(alarms) /= BOTH_OFF
                               THEN YELLOW_ON
                               ELSE BOTH_OFF
                               ENDIF
```

```
  END alarm_display
shim_rods                  :    THEORY

  BEGIN

  lamp_status              :    TYPE = { ON, OFF }

  shim_lamp_status         :    TYPE =

      [# %RECORD
          up               :    lamp_status,
          down             :    lamp_status,
          seated           :    lamp_status,
          mag_eng          :    lamp_status
      #]

  magnet_status            :    TYPE = { MAG_ON, MAG_OFF }

  scram_status             :    TYPE = { NOT_SCRAMMED, SCRAMMED }

  shim_rods_status         :    TYPE =

      [# %RECORD
          scram_state      :    scram_status,
          r1_driver        :    nat,
          r1_lamps         :    shim_lamp_status,
          r1_magnet        :    magnet_status,
          r2_driver        :    nat,
          r2_lamps         :    shim_lamp_status,
          r2_magnet        :    magnet_status,
          r3_driver        :    nat,
          r3_lamps         :    shim_lamp_status,
          r3_magnet        :    magnet_status
      #]


  scram(safety_rods             :  shim_rods_status) :
      shim_rods_status     =    safety_rods WITH [scram_state := SCRAMMED,
                                r1_magnet := MAG_OFF, r2_magnet := MAG_OFF,
                                r3_magnet := MAG_OFF]

  reset_scram(safety_rods       :  shim_rods_status) :
      shim_rods_status     =    safety_rods WITH [scram_state := NOT_SCRAMMED]

  r1_magnet_on(safety_rods      :  shim_rods_status) :
      shim_rods_status     =    safety_rods WITH [ r1_magnet := MAG_ON ]

  r2_magnet_on(safety_rods      :  shim_rods_status) :
      shim_rods_status     =    safety_rods WITH [ r2_magnet := MAG_ON ]

  r3_magnet_on(safety_rods      :  shim_rods_status) :
      shim_rods_status     =    safety_rods WITH [ r3_magnet := MAG_ON ]

  all_magnets_on(safety_rods    :  shim_rods_status) :
      shim_rods_status     =    safety_rods WITH [r1_magnet := MAG_ON,
                                r2_magnet := MAG_ON, r3_magnet := MAG_ON]

  lower_shim_rods(safety_rods   :  shim_rods_status) :
      shim_rods_status     =    safety_rods WITH [ r1_driver := 0,
                                r2_driver := 0, r3_driver := 0]

  raise_shim_rods(safety_rods   :  shim_rods_status,
                  height        :  posnat           ) :
      shim_rods_status     =    safety_rods WITH [ r1_driver := height,
                                r2_driver := height, r3_driver := height]
  END shim_rods
control_rod                :    THEORY

  BEGIN

  control_status            :    TYPE = { AUTOMATIC_CONTROL, MANUAL_CONTROL }

  control_rod_status        :    TYPE =
      [# %RECORD
          control           :    control_status,
          position          :    posnat
      #]

  start_auto_control(control_rod   :  control_rod_status) :
      control_rod_status    =    control_rod WITH [control := AUTOMATIC_CONTROL]

  start_manual_control(control_rod :  control_rod_status) :
      control_rod_status    =    control_rod WITH [control := MANUAL_CONTROL]

  move_control_rod(control_rod     :  control_rod_status,
                  height           :  posnat            ) :
      control_rod_status    =    IF control(control_rod) = MANUAL_CONTROL
                                 THEN control_rod WITH [ position := height]
                                 ELSE control_rod
                                 ENDIF

  END control_rod
rods                        :    THEORY

  BEGIN

  IMPORTING shim_rods
  IMPORTING control_rod

  rod_status                :    TYPE =

      [# %RECORD
          shim_rods         :    shim_rods_status,
          control_rod       :    control_rod_status
      #]

  Rods                      :    VAR rod_status
  position                  :    VAR nat

  start_auto_control(Rods   :  rod_status) :
      rod_status            =    Rods WITH [ control_rod :=
                                 start_auto_control(control_rod(Rods))]

  start_manual_control(Rods :  rod_status) :
      rod_status            =    Rods WITH [ control_rod :=
                                 start_manual_control(control_rod(Rods))]

  scram(Rods                :  rod_status) :
      rod_status            =    Rods WITH [ shim_rods := scram(shim_rods(Rods))]

  reset_scram(Rods          :  rod_status) :
      rod_status            =    Rods WITH [ shim_rods :=
                                 reset_scram(shim_rods(Rods))]

  r1_magnet_on(Rods         :  rod_status) :
      rod_status            =    Rods WITH [ shim_rods :=
                                 r1_magnet_on(shim_rods(Rods))]

  r2_magnet_on(Rods         :  rod_status) :
      rod_status            =    Rods WITH [ shim_rods :=
                                 r2_magnet_on(shim_rods(Rods))]

  r3_magnet_on(Rods         :  rod_status) :
      rod_status            =    Rods WITH [ shim_rods :=
                                 r3_magnet_on(shim_rods(Rods))]

  lower_shim_rods(Rods      :  rod_status) :
      rod_status            =    Rods WITH [ shim_rods :=
                                 lower_shim_rods(shim_rods(Rods))]

  all_magnets_on(Rods       :  rod_status) :
      rod_status            =    Rods WITH [shim_rods :=
                                 all_magnets_on(shim_rods(Rods))]

  move_shim_rods(Rods       :  rod_status,
                  height    :  posnat    ) :
      rod_status            =    Rods WITH [ shim_rods :=
                                 move_shim_rods(shim_rods(Rods), height)]

  END rods

power_level                 :    THEORY

  BEGIN

  range_switch_2_status     :    TYPE = { LOW_MODE, HIGH_MODE }

  operating_power_status    :    TYPE = { LOW_POWER, HIGH_POWER }

  operating_status          :    TYPE = { IDLE_CHECKED, IDLE_UNCHECKED,
                                                            POWER_TO_LOW,
POWER_TO_HIGH, OPERATING }

  power_level_status        :    TYPE =

      [# %RECORD
          sp_limit          :    nat,
          set_point         :    nat,
          operating         :    operating_status,
          power_level       :    operating_power_status,
          range_switch_2    :    range_switch_2_status
      #]


  scram(power            :  power_level_status) :
      power_level_status    =    power WITH [ operating :=
                                              IF operating(power) = IDLE_UNCHECKED
                                              THEN IDLE_UNCHECKED
                                              ELSE IDLE_CHECKED
                                              ENDIF ]

  range_sw_to_low(power  :  power_level_status) :
      power_level_status    =    power WITH [ range_switch_2 := LOW_MODE,
                                 sp_limit := 250, set_point := 230 ]

  range_sw_to_high(power :  power_level_status) :
      power_level_status    =    power WITH [ range_switch_2 := HIGH_MODE,
                                 sp_limit := 2500, set_point := 2230 ]

  power_to_low(power     :  power_level_status) :
      power_level_status    =    power WITH [ operating := POWER_TO_LOW ]

  power_to_high(power    :  power_level_status) :
      power_level_status    =    power WITH [ operating := POWER_TO_HIGH ]
  checked(power          :  power_level_status) :
      power_level_status    =    power WITH [ operating := IDLE_CHECKED ]

  problem(power          :  power_level_status) :
      power_level_status    =    power WITH [ operating := IDLE_UNCHECKED ]

  low_power_on(power     :  power_level_status) :
      power_level_status    =    power WITH [operating := OPERATING,
                                 power_level := LOW_POWER]

  high_power_on(power    :  power_level_status) :
      power_level_status    =    power WITH [operating := OPERATING,
                                 power_level := HIGH_POWER]

  END power_level

reactor                     :    THEORY

  BEGIN

  IMPORTING cooling
  IMPORTING alarm_display
  IMPORTING rods
  IMPORTING power_level
  IMPORTING sensors

  states                    :    TYPE =

      [# %RECORD
          rods              :    rod_status,
          cooling_system    :    cooling_status,
          alarms            :    alarms_status,
          power_level       :    power_level_status,
          sensors           :    sensors_status
      #]

  events                    :    TYPE =

      {
       scram,              raise_header,       lower_header,       pump_off,
       pump_on,            bleed_line,         close_valve,        reset_scram,
       open_truck_door,    open_escape_hatch,  remove_key,         sb_console_pressed,
       sb_rdoor_pressed,   sb_bdoor_pressed,   evacuation1,        evacuation2,
       evacuation3,        evacuation4,        clear_alarms,       clear_scram_light,
       r1_magnet_on,       r2_magnet_on,       r3_magnet_on,       range_sw_to_high,
       range_sw_to_low,    start_auto_control, start_man_control,  check_power_ind,
       check_alarms,       test,               startup
      }


  END reactor
check_sensors               :    THEORY

  BEGIN

  IMPORTING reactor


  check_sensors(st : states) :
      states                =    st WITH
          [ rods                 :=  rods(st) WITH
              [ shim_rods        :=  shim_rods(rods(st)) WITH
                  [ r1_lamps     :=  r1_lamps(shim_rods(rods(st))) WITH
                      [ up       :=  IF r1_up(sensors(st))
```

```
                                      THEN ON
                                      ELSE OFF
                                      ENDIF,
                      down       :=   IF r1_down(sensors(st))
                                      THEN ON
                                      ELSE OFF
                                      ENDIF,
                      seated     :=   IF r1_seated(sensors(st))
                                      THEN ON
                                      ELSE OFF
                                      ENDIF,
                      mag_eng    :=   IF r1_mag_eng(sensors(st))
                                      THEN ON
                                      ELSE OFF
                                      ENDIF
                      ],

                  r2_lamps       :=   r2_lamps(shim_rods(rods(st))) WITH
                      [ up       :=   IF r2_up(sensors(st))
                                      THEN ON
                                      ELSE OFF
                                      ENDIF,
                      down       :=   IF r2_down(sensors(st))
                                      THEN ON
                                      ELSE OFF
                                      ENDIF,
                      seated     :=   IF r2_seated(sensors(st))
                                      THEN ON
                                      ELSE OFF
                                      ENDIF,
                      mag_eng    :=   IF r2_mag_eng(sensors(st))
                                      THEN ON
                                      ELSE OFF
                                      ENDIF
                      ],

                  r3_lamps       :=   r3_lamps(shim_rods(rods(st))) WITH
                      [ up       :=   IF r3_up(sensors(st))
                                      THEN ON
                                      ELSE OFF
                                      ENDIF,
                      down       :=   IF r3_down(sensors(st))
                                      THEN ON
                                      ELSE OFF
                                      ENDIF,
                      seated     :=   IF r3_seated(sensors(st))
                                      THEN ON
                                      ELSE OFF
                                      ENDIF,
                      mag_eng    :=   IF r3_mag_eng(sensors(st))
                                      THEN ON
                                      ELSE OFF
                                      ENDIF
                      ]
                  ]
              ]
          ]

  END check_sensors


check_scrams                :    THEORY
```

```
BEGIN

IMPORTING reactor


scram_rods(st             :  states) :
    states                =    st WITH [rods := scram(rods(st)),
                               alarms := scram(alarms(st)),
                               power_level := scram(power_level(st))]

not_scrammed_rods(st      :  states) :
    bool                  =    IF scram_state(shim_rods(rods(st))) = NOT_SCRAMMED
                               THEN true
                               ELSE false
                               ENDIF

check_scrams(st           :  states) :
    states                =    IF scram_state(shim_rods(rods(st))) /= SCRAMMED
                             THEN
                                   IF (power_level(power_level(st)) = HIGH_POWER
                                       AND (line_pressure(cooling_system(st)) = HIGH
                                           OR core_flow(sensors(st))<960))
                                   OR (pump(cooling_system(st)) = ON
                                       AND header(cooling_system(st)) = DOWN)
                                   OR (pump(cooling_system(st)) = OFF
                                       AND header(cooling_system(st)) = UP)
                                OR power_indic1(sensors(st)) > sp_limit(power_level(st))
                                OR power_indic2(sensors(st)) > sp_limit(power_level(st))
                                   OR bridge_rad(sensors(st)) > 30
                                   OR face_rad(sensors(st)) > 2
                                   OR pool_level_low(sensors(st)) = true
                                   OR pool_level(sensors(st)) < 231
                                   OR pool_temp(sensors(st)) > 108
                                   OR react_period(sensors(st)) < 33
                                   OR t_door_open(sensors(st)) = true
                                   OR ehatch_open(sensors(st)) = true
                                   OR key_removed(sensors(st)) = true
                                   THEN scram_rods(st)
                                   ELSE st
                                   ENDIF
                               ELSE st
                             ENDIF


tran_reset_scram(st       :  states) :
    states                =    IF not_scrammed_rods(check_scrams(st))
                               THEN st WITH [rods := reset_scram(rods(st)),
                                   alarms := reset_scram(alarms(st))]
                               ELSE st
                               ENDIF

tran_truck_door_open(st   :  states) :
    states                =    st WITH [sensors := sensors(st)
                               WITH [ t_door_open := true]]

tran_escape_hatch_open(st :  states) :
    states                =    st WITH [sensors := sensors(st)
                               WITH [ ehatch_open := true]]

tran_key_removed(st       :  states) :
    states                =    st WITH [sensors := sensors(st)
                               WITH [key_removed := true]]

END check_scrams
```

```
check_alarms                :    THEORY

  BEGIN

  IMPORTING reactor


  tran_clear_alarms(st      :  states) :
      states                =    st WITH [alarms := clear_alarms(alarms(st))]

  check_alarms(st           :  states) :
      states                =    st WITH
          [ alarms               :=  alarms(st) WITH
              [core_temp_alarm   :=  IF (core_temp(sensors(st)) > 0)
                                     THEN core_temp_alarm_signal_on(alarms(st))
                                     ELSE core_temp_alarm_signal_off(alarms(st))
                                     ENDIF,
              control_rod_alarm  :=  IF (auto_ctrl_lost(sensors(st)) = true)
                                     THEN control_rod_alarm_signal_on(alarms(st))
                                     ELSE control_rod_alarm_signal_off(alarms(st))
                                     ENDIF,
              air_mont_alarm     :=  IF (air_mont(sensors(st)) > 0)
                                     THEN air_mont_alarm_signal_on(alarms(st))
                                     ELSE air_mont_alarm_signal_off(alarms(st))
                                     ENDIF,
              water_cond_alarm   :=  IF (water_cond(sensors(st)) > 2)
                                     THEN water_cond_alarm_signal_on(alarms(st))
                                     ELSE water_cond_alarm_signal_off(alarms(st))
                                     ENDIF,
              area_rad_alarm     :=  IF (area_rad(sensors(st)) > 0)
                                     THEN area_rad_alarm_signal_on(alarms(st))
                                     ELSE area_rad_alarm_signal_off(alarms(st))
                                     ENDIF,
              her_door_alarm     :=  IF (her_door_open(sensors(st)) = true)
                                     THEN her_door_alarm_signal_on(alarms(st))
                                     ELSE her_door_alarm_signal_off(alarms(st))
                                     ENDIF,
              sec_pump_alarm     :=  IF (sec_pump_off(sensors(st)) = true)
                                     THEN sec_pump_alarm_signal_on(alarms(st))
                                     ELSE sec_pump_alarm_signal_off(alarms(st))
                                     ENDIF,
              gamma_rad_alarm    :=  IF (gamma_rad(sensors(st)) > 0)
                                     THEN gamma_rad_alarm_signal_on(alarms(st))
                                     ELSE gamma_rad_alarm_signal_off(alarms(st))
                                     ENDIF,
              dr_door_alarm      :=  IF (dr_door_open(sensors(st)) = true)
                                     THEN dr_door_alarm_signal_on(alarms(st))
                                     ELSE dr_door_alarm_signal_off(alarms(st))
                                     ENDIF,
              thimble_temp_alarm :=  IF (thimble_too_hot(sensors(st)) = true)
                                     THEN thimble_temp_alarm_signal_on(alarms(st))
                                     ELSE thimble_temp_alarm_signal_off(alarms(st))
                                     ENDIF
              ]
          ]

  tran_clear_scram_light(st :  states) :
      states                =    st WITH [alarms := reset_scram(alarms(st))]


  END check_alarms
check_conditions            :    THEORY
```

```
  BEGIN

  IMPORTING check_sensors
  IMPORTING check_scrams
  IMPORTING check_alarms

  END check_conditions
transition                 :    THEORY

  BEGIN

  IMPORTING reactor
  IMPORTING check_conditions


  tran_raise_header(st                  :  states) :
      states               =    st WITH [cooling_system :=
                                raise_header(cooling_system(st))]

  tran_lower_header(st                  :  states) :
      states               =    st WITH [cooling_system :=
                                lower_header(cooling_system(st))]

  tran_pump_off(st                      :  states) :
      states               =    scram_rods(st WITH [cooling_system :=
                                pump_off(cooling_system(st))])

  tran_pump_on(st                       :  states) :
      states               =    scram_rods(st WITH [cooling_system :=
                                pump_on(cooling_system(st))])

  tran_bleed_line(st                    :  states) :
      states               =    st WITH [cooling_system :=
                                bleed_line(cooling_system(st))]

  tran_close_valve(st                   :  states) :
      states               =    st WITH [cooling_system :=
                                close_valve(cooling_system(st))]

  tran_scram(st                         :  states) :
      states               =    scram_rods(st)

  tran_r1_magnet_on(st                  :  states) :
      states               =    st WITH [ rods := r1_magnet_on(rods(st))]

  tran_r2_magnet_on(st                  :  states) :
      states               =    st WITH [ rods := r2_magnet_on(rods(st))]

  tran_r3_magnet_on(st                  :  states) :
      states               =    st WITH [ rods := r3_magnet_on(rods(st))]

  tran_pumps_on(st                      :  states) :
      states               =    scram_rods(st WITH [ cooling_system :=
                                pumps_on(cooling_system(st))])

  tran_all_drivers_to_lowest_position(st :  states) :
      states               =    st WITH [ rods := lower_shim_rods(rods(st)),
                                sensors := lowest_shim_rod_position(sensors(st))]

  tran_all_magnets_on(st                :  states) :
      states               =    st WITH [rods := all_magnets_on(rods(st))]

  tran_all_drivers_up_10(st             :  states) :
      states               =    st WITH [ rods :=
                                 move_shim_rods(rods(st), 10),
                                 sensors := raise_shim_rods_10(sensors(st))]

  tran_range_sw_to_high(st               :  states) :
      states                =    st WITH [ power_level :=
                                 range_sw_to_high(power_level(st))]

  tran_range_sw_to_low(st                :  states) :
      states                =    st WITH [ power_level :=
                                 range_sw_to_low(power_level(st))]

  tran_start_auto_control(st             :  states) :
      states                =    st WITH [ rods := start_auto_control(rods(st))]

  tran_start_manual_control(st           :  states) :
      states                =    st WITH [ rods := start_manual_control(rods(st))]

  tran_check_power_ind(st                :  states) :
      states                =
          IF control(control_rod(rods(st))) = AUTOMATIC_CONTROL
                 AND (power_indic1(sensors(st)) > (6/5 * set_point(power_level(st)))
              OR power_indic2(sensors(st)) > (6/5 * set_point(power_level(st)))
              OR power_indic1(sensors(st)) < (4/5 * set_point(power_level(st)))
              OR power_indic2(sensors(st)) < (4/5 * set_point(power_level(st))))
          THEN tran_start_manual_control(st)
          ELSE st
          ENDIF

  tran_check_alarms(st                   :  states) :
      states                =    check_alarms(st)

  scrammed(st                            :  states) :
      bool                  =    IF scram_state(shim_rods(rods(st))) = SCRAMMED
                                 THEN true
                                 ELSE false
                                 ENDIF

  not_scrammed(st                        :  states) :
      bool                  =    IF scram_state(shim_rods(rods(st))) = NOT_SCRAMMED
                                 THEN true
                                 ELSE false
                                 ENDIF

  not_header_up(st                       :  states) :
      bool                  =    IF header(cooling_system(st)) = DOWN
                                 THEN true
                                 ELSE false
                                 ENDIF

  not_seated(st                          :  states) :
      bool                  =    IF seated(r1_lamps(shim_rods(rods(st)))) /= ON
                                     OR seated(r2_lamps(shim_rods(rods(st)))) /= ON
                                     OR seated(r3_lamps(shim_rods(rods(st)))) /= ON
                                 THEN true
                                 ELSE false
                                 ENDIF

  not_mag_eng(st                         :  states) :
      bool                  =    IF mag_eng(r1_lamps(shim_rods(rods(st)))) /= ON
                                     OR mag_eng(r2_lamps(shim_rods(rods(st)))) /= ON
                                     OR mag_eng(r3_lamps(shim_rods(rods(st)))) /= ON
                                 THEN true
                                 ELSE false
                                 ENDIF
  not_seated_off_and_down_off(st          :  states) :
      bool                 =    IF seated(r1_lamps(shim_rods(rods(st)))) = ON
                                     OR seated(r2_lamps(shim_rods(rods(st)))) = ON
                                     OR seated(r3_lamps(shim_rods(rods(st)))) = ON
                                     OR down(r1_lamps(shim_rods(rods(st)))) = ON
                                     OR down(r2_lamps(shim_rods(rods(st)))) = ON
                                     OR down(r3_lamps(shim_rods(rods(st)))) = ON
                                 THEN true
                                 ELSE false
                                 ENDIF


  check(st                                :  states) :
      states               =    check_sensors(check_alarms(check_scrams(st)))

  reset_and_raise(st                      :  states) :
                      states                                                       =
check(tran_raise_header(check_sensors(check_alarms(check_scrams(tran_reset_scram(st))))))

  bleed_close_and_reset(st                :  states) :
                      states                                                       =
check(tran_reset_scram(check(tran_close_valve(check(tran_bleed_line(st))))))

  turn_pump_on(st                         :  states) :
      states               =    check(tran_pump_on(check(bleed_close_and_reset(st))))

  test_step1(st                           :  states) :
      states               =    reset_and_raise(st)

  test_step2(st                           :  states) :
      states               =    check(turn_pump_on(test_step1(st)))

  test_step3(st                           :  states) :
     states              =    check(tran_pump_off(check(tran_reset_scram(test_step2(st)))))

  test_step4(st                           :  states) :
      states               =    check(tran_reset_scram(test_step3(st)))


  perform_tests(st                        :  states) :
      states               =    IF operating(power_level(st)) = IDLE_UNCHECKED
                                 OR operating(power_level(st)) = IDLE_CHECKED
                                 THEN IF not_scrammed(test_step1(st))
                                      THEN check(tran_scram(test_step1(st)))
                                          WITH [ power_level := problem(power_level(st))]
                                      ELSIF not_scrammed(test_step2(st))
                                      THEN check(tran_scram(test_step2(st)))
                                          WITH [ power_level := problem(power_level(st))]
                                      ELSIF not_scrammed(test_step3(st))
                                      THEN check(tran_scram(test_step3(st)))
                                          WITH [ power_level := problem(power_level(st))]
                                      ELSE check(test_step4(st))
                                          WITH [ power_level := checked(power_level(st))]
                                      ENDIF
                                 ELSE st
                                 ENDIF

  low_step1(st                            :  states) :
                      states                                                       =
check(tran_all_drivers_to_lowest_position(check(tran_reset_scram(st))))

  low_step2(st                            :  states) :
                      states                                                       =
check(tran_all_drivers_up_10(check(tran_all_magnets_on(low_step1(st)))))
```

```
  startup_low(st                        :  states) :
      states                =    IF not_seated(low_step1(st))
                                 THEN check(tran_scram(low_step1(st)))
                                     WITH [ power_level := problem(power_level(st))]
                                 ELSIF not_mag_eng(low_step1(st))
                                 THEN check(tran_scram(low_step1(st)))
                                     WITH [ power_level := problem(power_level(st))]
                                 ELSIF not_seated_off_and_down_off(low_step2(st))
                                 THEN check(tran_scram(low_step2(st)))
                                     WITH [ power_level := problem(power_level(st))]
                                 ELSE check(tran_start_auto_control(low_step2(st)))
                                     WITH [ power_level := low_power_on(power_level(st))]
                                 ENDIF

  high_step1(st                         :  states) :
      states                =    reset_and_raise(st)

  high_step2(st                         :  states) :
      states                =    bleed_close_and_reset(tran_pumps_on(high_step1(st)))

  high_step3(st                         :  states) :
      states                =    check(tran_all_drivers_to_lowest_position(high_step2(st)))

  high_step4(st                         :  states) :
                      states                                                         =
check(tran_all_drivers_up_10(check(tran_all_magnets_on(high_step3(st)))))

  startup_high(st                       :  states) :
      states                =    IF not_scrammed(high_step1(st))
                                 THEN check(tran_scram(high_step1(st)))
                                     WITH [ power_level := problem(power_level(st))]
                                 ELSIF not_header_up(high_step1(st))
                                 THEN check(tran_scram(high_step2(st)))
                                     WITH [ power_level := problem(power_level(st))]
                                 ELSIF not_seated(high_step3(st))
                                 THEN check(tran_scram(high_step3(st)))
                                     WITH [ power_level := problem(power_level(st))]
                                 ELSIF not_mag_eng(high_step3(st))
                                 THEN check(tran_scram(high_step3(st)))
                                     WITH [ power_level := problem(power_level(st))]
                                 ELSIF not_seated_off_and_down_off(high_step4(st))
                                 THEN check(tran_scram(high_step4(st)))
                                     WITH [ power_level := problem(power_level(st))]
                                 ELSE check(tran_start_auto_control(high_step4(st)))
                                     WITH [ power_level := high_power_on(power_level(st))]
                                 ENDIF

  startup(st                            :  states) :
      states                =    IF operating(power_level(st)) = IDLE_CHECKED
                                     AND range_switch_2(power_level(st)) = LOW_MODE
                                 THEN startup_low(st
                                     WITH [power_level := power_to_low(power_level(st))])
                                 ELSIF operating(power_level(st)) = IDLE_CHECKED
                                     AND range_switch_2(power_level(st)) = HIGH_MODE
                                 THEN startup_high(st
                                     WITH [power_level := power_to_high(power_level(st))])
                                 ELSE st
                                 ENDIF

  check_new_state(st                    :  states) :
      states                =    check(st)

  nextstate(st                          :  states,
            event                       :  events) :
      states                =    check_new_state(
          CASES event OF
          raise_header      :    tran_raise_header(st),
          lower_header      :    tran_lower_header(st),
          pump_off          :    tran_pump_off(st),
          pump_on           :    tran_pump_on(st),
          bleed_line        :    tran_bleed_line(st),
          close_valve       :    tran_close_valve(st),
          open_truck_door   :    tran_truck_door_open(st),
          open_escape_hatch :    tran_escape_hatch_open(st),
          remove_key        :    tran_key_removed(st),
          scram             :    tran_scram(st),
          reset_scram       :    tran_reset_scram(st),
          sb_console_pressed:    tran_scram(st),
          sb_rdoor_pressed  :    tran_scram(st),
          sb_bdoor_pressed  :    tran_scram(st),
          evacuation1       :    tran_scram(st),
          evacuation2       :    tran_scram(st),
          evacuation3       :    tran_scram(st),
          evacuation4       :    tran_scram(st),
          clear_alarms      :    tran_clear_alarms(st),
          clear_scram_light :    tran_clear_scram_light(st),
          r1_magnet_on      :    tran_r1_magnet_on(st),
          r2_magnet_on      :    tran_r2_magnet_on(st),
          r3_magnet_on      :    tran_r3_magnet_on(st),
          range_sw_to_high  :    tran_range_sw_to_high(st),
          range_sw_to_low   :    tran_range_sw_to_high(st),
          start_auto_control:    tran_start_auto_control(st),
          start_man_control :    tran_start_manual_control(st),
          check_power_ind   :    tran_check_power_ind(st),
          check_alarms      :    tran_check_alarms(st),
          test              :    perform_tests(st),
          startup           :    startup(perform_tests(st))
          ENDCASES
  )

  END transition
verified_theorems           :    THEORY

  BEGIN

  IMPORTING transition


  lamps1                    :   shim_lamp_status =
      (# up   := OFF,
      down    := ON,
      seated  := ON,
      mag_eng := ON #);

  lamps2                    :   shim_lamp_status =
      (# up   := OFF,
      down    := ON,
      seated  := ON,
      mag_eng := ON #);

  lamps3                    :   shim_lamp_status =
      (# up   := OFF,
      down    := ON,
      seated  := ON,
      mag_eng := ON #);


  initial_cooling           :   cooling_status =

      (# pump            := OFF,
      header             := DOWN,
      sec_pump           := OFF,
      line_valve         := CLOSED,
      line_pressure      := NORMAL
      #);

  initial_sensors           :   sensors_status =

      (# pool_temp       := 75,
      pool_level         := 240,
      pool_level_low     := false,
      power_indic1       := 0,
      power_indic2       := 0,
      water_cond         := 0,
      react_period       := 50,
      gamma_rad          := 0,
      air_mont           := 0,
      area_rad           := 0,
      core_temp          := 0,
      core_flow          := 0,
      auto_ctrl_lost     := false,
      her_door_open      := false,
      dr_door_open       := false,
      sec_pump_off       := true,
      thimble_too_hot    := false,
      key_removed        := false,
      bridge_rad         := 25,
      face_rad           := 1,
      t_door_open        := false,
      ehatch_open        := false,
      r1_up              := false,
      r1_down            := true,
      r1_seated          := true,
      r1_mag_eng         := true,
      r2_up              := false,
      r2_down            := true,
      r2_seated          := true,
      r2_mag_eng         := true,
      r3_up              := false,
      r3_down            := true,
      r3_seated          := true,
      r3_mag_eng         := true
      #);

  initial_alarms            :    alarms_status =

      (# core_temp_alarm := BOTH_OFF,
      control_rod_alarm  := BOTH_OFF,
      air_mont_alarm     := BOTH_OFF,
      water_cond_alarm   := BOTH_OFF,
      area_rad_alarm     := BOTH_OFF,
      her_door_alarm     := BOTH_OFF,
      sec_pump_alarm     := BOTH_OFF,
      gamma_rad_alarm    := BOTH_OFF,
      dr_door_alarm      := BOTH_OFF,
      thimble_temp_alarm := BOTH_OFF,
      scram_alarm        := BOTH_OFF
      #);

  initial_shim_rods         :    shim_rods_status =

      (# scram_state := NOT_SCRAMMED,
      r1_driver      := 0,
      r1_lamps       := lamps1,
      r1_magnet      := MAG_OFF,
      r2_driver      := 0,
      r2_lamps       := lamps2,
      r2_magnet      := MAG_OFF,
      r3_driver      := 0,
      r3_lamps       := lamps3,
      r3_magnet      := MAG_OFF
      #);

  initial_control_rod        :    control_rod_status =
      (# control     := MANUAL_CONTROL,
         position    := 0
      #);

  initial_high_power_level   :    power_level_status =

      (# sp_limit        := 2500,
      set_point          := 2230,
      operating          := IDLE_UNCHECKED,
      power_level        := HIGH_POWER,
      range_switch_2     := HIGH_MODE
      #);

  initial_power_level       :    power_level_status =

      (# sp_limit        := 250,
      set_point          := 230,
      operating          := IDLE_UNCHECKED,
      power_level        := LOW_POWER,
      range_switch_2     := LOW_MODE
      #);

  st0                       :    states =

      (#cooling_system           :=  initial_cooling,
      sensors                    :=  initial_sensors,
      alarms                     :=  initial_alarms,
      rods                       :=  (# shim_rods       := initial_shim_rods,
                                     control_rod        := initial_control_rod
                                     #),
      power_level                :=  initial_power_level
      #);

  st0prime                  :    states =

      (#cooling_system           :=  initial_cooling,
      sensors                    :=  initial_sensors,
      alarms                     :=  initial_alarms,
      rods                       :=  (# shim_rods       := initial_shim_rods,
                                     control_rod        := initial_control_rod
                                     #),
      power_level                :=  initial_high_power_level
      #);

  is_initial(st : states): bool = st = perform_tests(st)


  reachable_in(n : posnat, st : states): RECURSIVE bool =
                        IF n =0  THEN st = st0
                        ELSE
                        EXISTS (pst : states, event : events) : st = nextstate(pst,event)
                        AND reachable_in(n-1, pst)
                        ENDIF MEASURE n

  is_reachable(st : states): bool = EXISTS (n : posnat) : reachable_in(n,st)
```

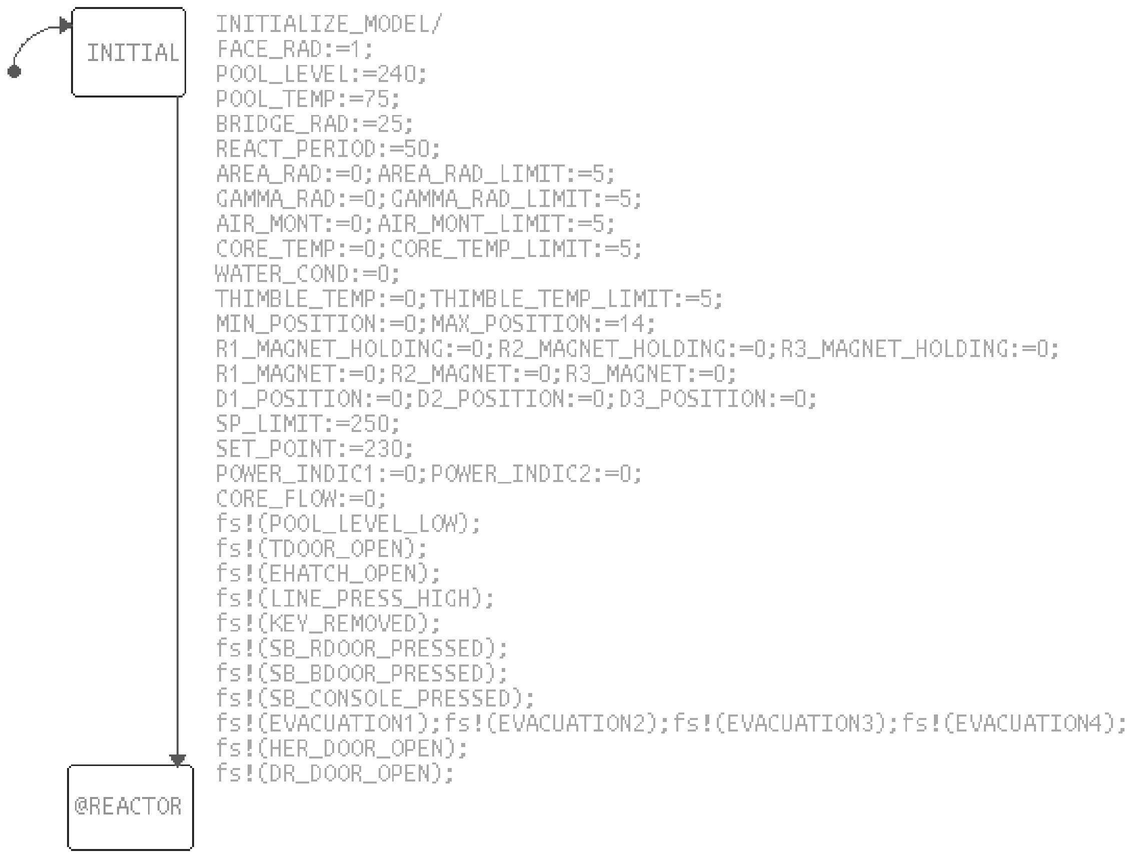
```
  startup_on_n(n : posnat, st : states):  RECURSIVE bool =
                      IF n = 1
                      THEN EXISTS (pst : states) : is_reachable(pst)
                         AND st = nextstate(pst, startup)
                         AND operating(power_level(st)) /= OPERATING
                   ELSE EXISTS (pst : states, event : events) : st = nextstate(pst,event)
                    AND startup_on_n(n-1, pst)
                      AND event /= startup
                    ENDIF MEASURE n

  startup_encountered(st : states): bool =
                      is_reachable(st)
                      AND EXISTS (n : posnat) : startup_on_n(n, st)
                      AND FORALL (p : posnat) : p < n AND NOT(startup_on_n(p, st))

  no_startup_on_n(n : posnat, st : states): RECURSIVE bool =
                      IF n = 1
                      THEN EXISTS (event : events) : st = nextstate(st0,event)
                      AND event /= startup
                   ELSE EXISTS (pst : states, event : events) : st = nextstate(pst,event)
                    AND no_startup_on_n(n-1, pst)
                    AND event /= startup
                    ENDIF MEASURE n

  startup_not_encountered(st : states): bool =
                      is_reachable(st)
                      AND FORALL (n : posnat) : no_startup_on_n(n, st)

%--------------------------VERIFIED THEOREMS-----------------------------------

  case_analysis: LEMMA FORALL (event : events) :
      event = scram
      OR event = raise_header
      OR event = lower_header
      OR event = pump_off
      OR event = pump_on
      OR event = bleed_line
      OR event = close_valve
      OR event = reset_scram
      OR event = open_truck_door
      OR event = open_escape_hatch
      OR event = remove_key
      OR event = sb_console_pressed
      OR event = sb_rdoor_pressed
      OR event = sb_bdoor_pressed
      OR event = evacuation1
      OR event = evacuation2
      OR event = evacuation3
      OR event = evacuation4
      OR event = clear_alarms
      OR event = clear_scram_light
      OR event = r1_magnet_on
      OR event = r2_magnet_on
      OR event = r3_magnet_on
      OR event = range_sw_to_high
      OR event = range_sw_to_low
      OR event = start_auto_control
      OR event = start_man_control
      OR event = check_power_ind
      OR event = check_alarms
      OR event = test
      OR event = startup

  checking_scrammed:   LEMMA FORALL (st: states) : scrammed(st) IMPLIES scrammed(check(st))
```

```
basic_lemma1:   LEMMA not_scrammed(test_step1(st0)) = false

basic_lemma2:   LEMMA not_scrammed(test_step2(st0)) = false

basic_lemma3:   LEMMA not_scrammed(test_step3(st0)) = false

basic_last_lemma:   LEMMA FORALL (st: states) : st = nextstate(st0, test)
                    IMPLIES operating(power_level(st)) = IDLE_CHECKED

check_alarms_lemma: LEMMA FORALL (st: states, pst: states) : is_reachable(pst)
                    AND operating(power_level(pst)) /= OPERATING
                    AND st = nextstate(pst, check_alarms)
                    IMPLIES operating(power_level(st)) /= OPERATING


testing_lemma: LEMMA FORALL (st : states, pst : states) : is_reachable(pst)
               AND operating(power_level(pst)) /= OPERATING
               AND st = nextstate(pst, test)
               IMPLIES operating(power_level(st)) /= OPERATING

startup1_lemma: LEMMA FORALL (st : states, pst : states) : is_reachable(pst)
                AND operating(power_level(pst)) /= OPERATING
                AND st = nextstate(pst, startup)
                IMPLIES operating(power_level(st)) = OPERATING



    induction_step:
        LEMMA FORALL (st : states, pst : states, event : events) : is_reachable(pst)
        AND operating(power_level(pst)) /= OPERATING
        AND st = nextstate(pst, event)
        AND event /= startup
        IMPLIES operating(power_level(st)) /= OPERATING

    induction_step1:
        LEMMA FORALL (st : states) : is_reachable(st)
        AND operating(power_level(st)) = OPERATING
        IMPLIES startup_encountered(st)

        if_high_testing_high:
            LEMMA FORALL (pst : states) : is_reachable(pst)
            AND range_switch_2(power_level(pst)) = HIGH_MODE
            IMPLIES range_switch_2(power_level(perform_tests(pst))) = HIGH_MODE

        if_next_high:
            LEMMA FORALL (st : states, pst : states) : is_reachable(pst)
            AND st = nextstate(pst, startup)
            AND power_level(power_level(st)) = HIGH_POWER
            IMPLIES range_switch_2(power_level(pst)) = HIGH_MODE

    if_startup_header_up_pump_on:
        LEMMA FORALL (st : states, pst : states) : is_reachable(pst)
        AND operating(power_level(pst)) /= OPERATING
        AND st = nextstate(pst, startup)
        AND operating(power_level(st)) = OPERATING
        AND power_level(power_level(st)) = HIGH_POWER
        IMPLIES pump(cooling_system(st)) = ON
        AND header(cooling_system(st)) = UP

    if_header_falls_scram:
        LEMMA FORALL (st : states, pst : states, event : events) : is_reachable(pst)
        AND operating(power_level(pst)) = OPERATING
        AND power_level(power_level(pst)) = HIGH_POWER
        AND st = nextstate(pst, event)
        AND header(cooling_system(st)) = DOWN
```

```
          IMPLIES operating(power_level(st)) /= OPERATING

      if_pump_off_scram:
          LEMMA FORALL (st : states, pst : states, event : events) : is_reachable(pst)
          AND pump(cooling_system(pst)) = ON
          AND operating(power_level(pst)) = OPERATING
          AND power_level(power_level(pst)) = HIGH_POWER
          AND st = nextstate(pst, event)
          AND pump(cooling_system(st)) = OFF
          IMPLIES operating(power_level(st)) /= OPERATING

      startup_lemma:
          LEMMA FORALL (st : states) : st = nextstate(st0, startup)
          IMPLIES operating(power_level(st)) = OPERATING

      if_high_was_high:
        LEMMA FORALL (st : states, pst : states, event : events) : st = nextstate(pst, event)
          AND operating(power_level(st)) = OPERATING
          AND power_level(power_level(st)) = HIGH_POWER
          AND event /= startup
          IMPLIES power_level(power_level(pst)) = HIGH_POWER

      if_high_was_high1:
          LEMMA FORALL (st : states, pst : states) : st = nextstate(pst, startup)
          AND operating(power_level(st)) = OPERATING
          AND power_level(power_level(st)) = HIGH_POWER
          AND operating(power_level(pst)) = OPERATING
          IMPLIES power_level(power_level(pst)) = HIGH_POWER

  header_up_pump_on_in_high_power :
     LEMMA FORALL (n : posnat, st : states, pst : states, event : events) : startup_on_n(n,
st)
      AND is_reachable(pst)
      AND st = nextstate(pst, event)
      AND operating(power_level(st)) = OPERATING
      AND power_level(power_level(st)) = HIGH_POWER
      IMPLIES header(cooling_system(st)) = UP
      AND pump(cooling_system(st)) = ON


%******************************THEOREMS***********************************


  running: LEMMA IF operating(power_level(startup(perform_tests(st0)))) /= IDLE_UNCHECKED
           THEN operating(power_level(startup(perform_tests(st0)))) = OPERATING
           ELSE scram_state(shim_rods(rods(startup(perform_tests(st0))))) = SCRAMMED
           ENDIF

  power_up: LEMMA IF operating(power_level(startup(perform_tests(st0)))) /= IDLE_UNCHECKED
            THEN (operating(power_level(startup(perform_tests(st0)))) = OPERATING
            AND power_level(power_level(startup(perform_tests(st0)))) = LOW_POWER)
            ELSE scram_state(shim_rods(rods(startup(perform_tests(st0))))) = SCRAMMED
            ENDIF

%  high_power: LEMMA reachable(st)


  test_prime: LEMMA FORALL (st : states) : is_reachable(st)
              AND operating(power_level(st)) = OPERATING
              IMPLIES NOT(startup_not_encountered(st))



  basic_lemma:   LEMMA not_scrammed(test_step1(st0)) IFF FALSE
```

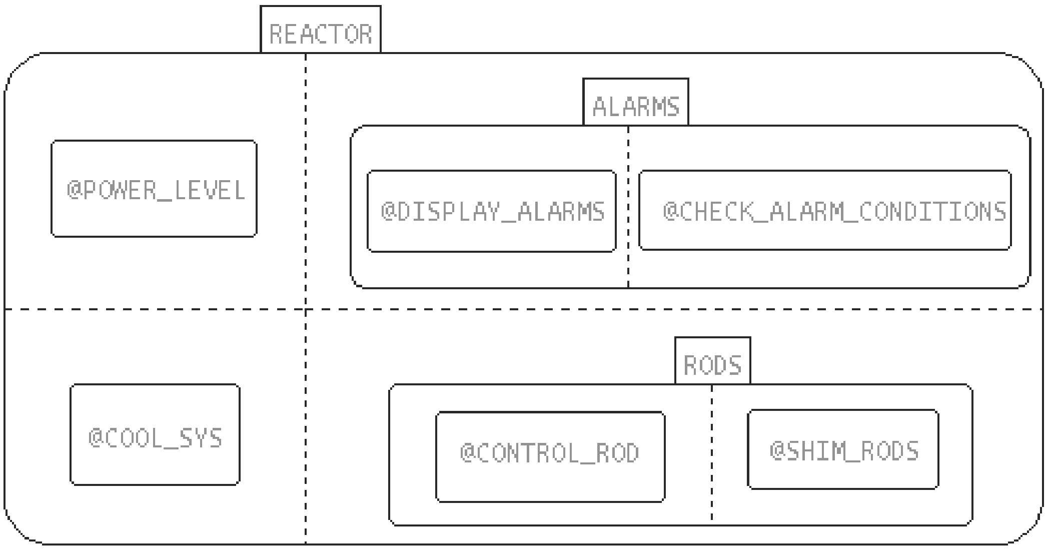
```
basic1_lemma:   LEMMA not_scrammed(test_step2(st0)) IFF FALSE

test2: LEMMA FORALL (st : states, event : events) : st = nextstate(st0, event)
             AND operating(power_level(st)) = OPERATING
            AND power_level(power_level(st)) = HIGH_POWER
             IMPLIES event = startup

END verified_theorem
```

Appendix B
Statechart Specification

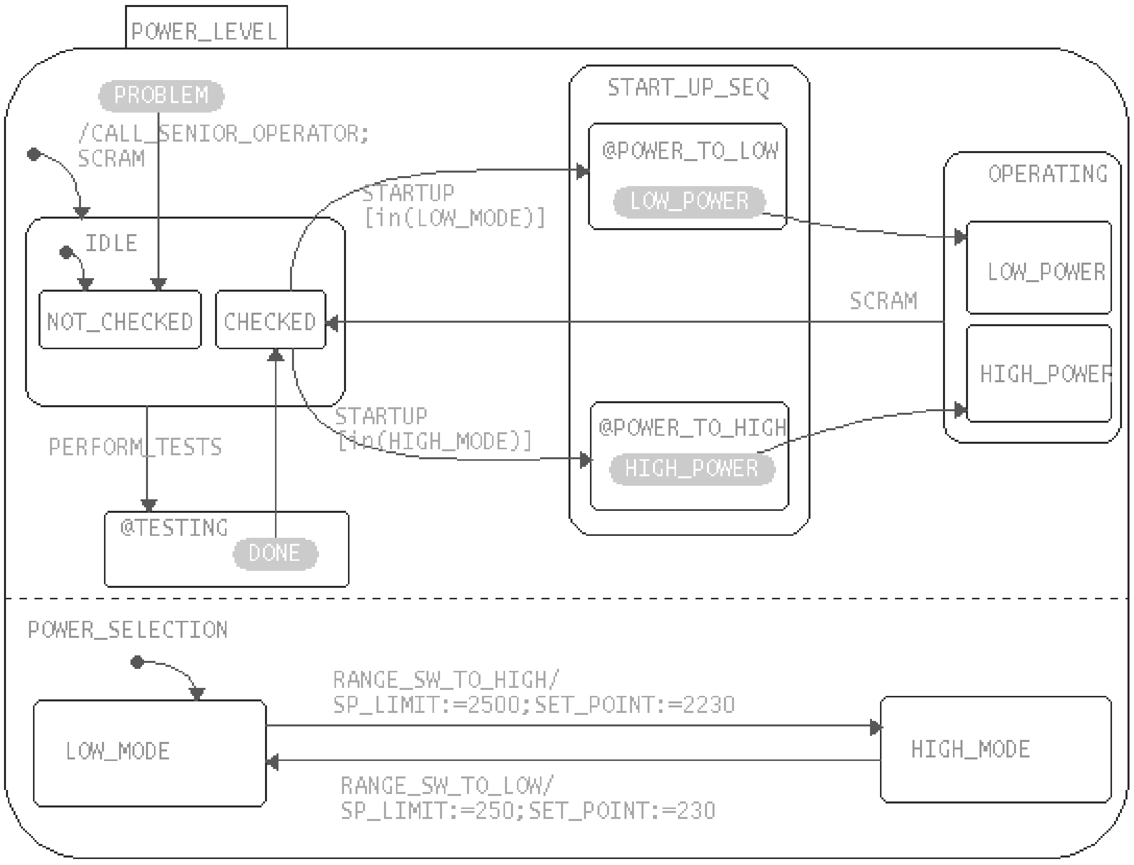


```
INITIALIZE_MODEL/
FACE_RAD:=1;
POOL_LEVEL:=240;
POOL_TEMP:=75;
BRIDGE_RAD:=25;
REACT_PERIOD:=50;
AREA_RAD:=0;AREA_RAD_LIMIT:=5;
GAMMA_RAD:=0;GAMMA_RAD_LIMIT:=5;
AIR_MONT:=0;AIR_MONT_LIMIT:=5;
CORE_TEMP:=0;CORE_TEMP_LIMIT:=5;
WATER_COND:=0;
THIMBLE_TEMP:=0;THIMBLE_TEMP_LIMIT:=5;
MIN_POSITION:=0;MAX_POSITION:=14;
R1_MAGNET_HOLDING:=0;R2_MAGNET_HOLDING:=0;R3_MAGNET_HOLDING:=0;
R1_MAGNET:=0;R2_MAGNET:=0;R3_MAGNET:=0;
D1_POSITION:=0;D2_POSITION:=0;D3_POSITION:=0;
SP_LIMIT:=250;
SET_POINT:=230;
POWER_INDIC1:=0;POWER_INDIC2:=0;
CORE_FLOW:=0;
fs!(POOL_LEVEL_LOW);
fs!(TDOOR_OPEN);
fs!(EHATCH_OPEN);
fs!(LINE_PRESS_HIGH);
fs!(KEY_REMOVED);
fs!(SB_RDOOR_PRESSED);
fs!(SB_BDOOR_PRESSED);
fs!(SB_CONSOLE_PRESSED);
fs!(EVACUATION1);fs!(EVACUATION2);fs!(EVACUATION3);fs!(EVACUATION4);
fs!(HER_DOOR_OPEN);
fs!(DR_DOOR_OPEN);
```

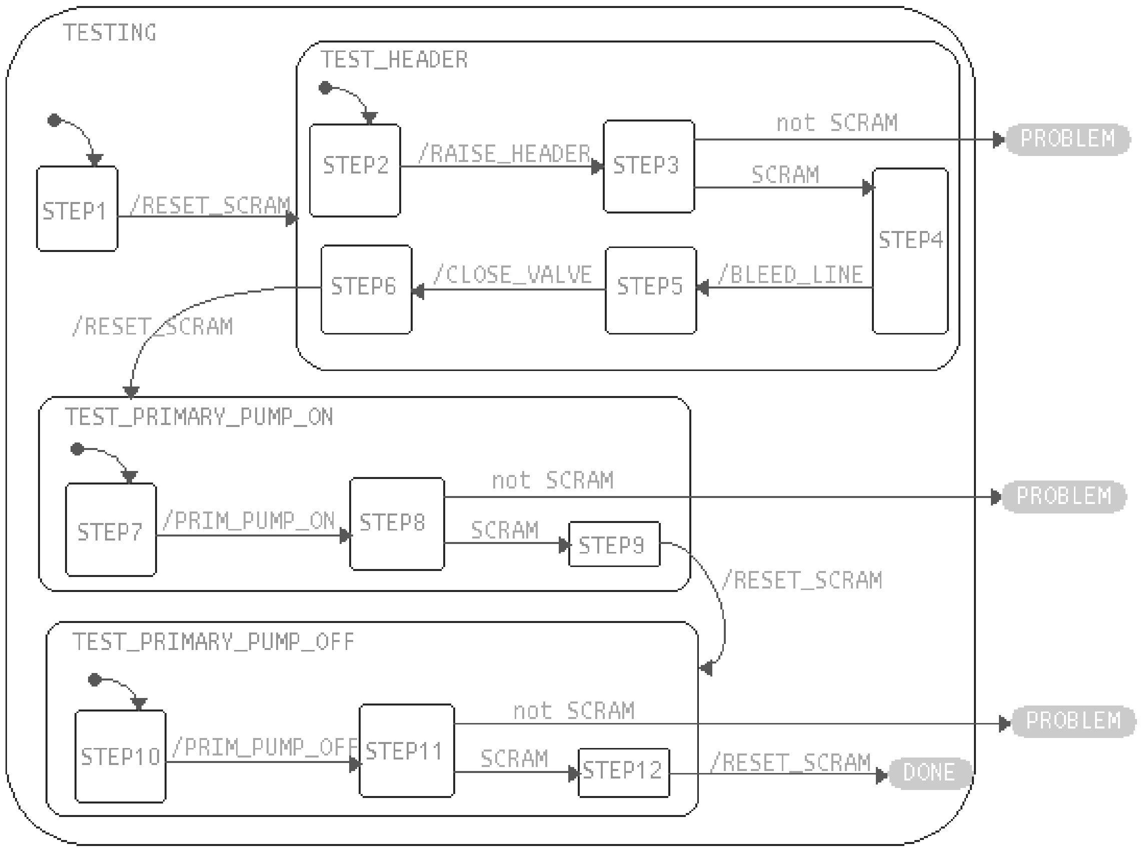
1 Reactor



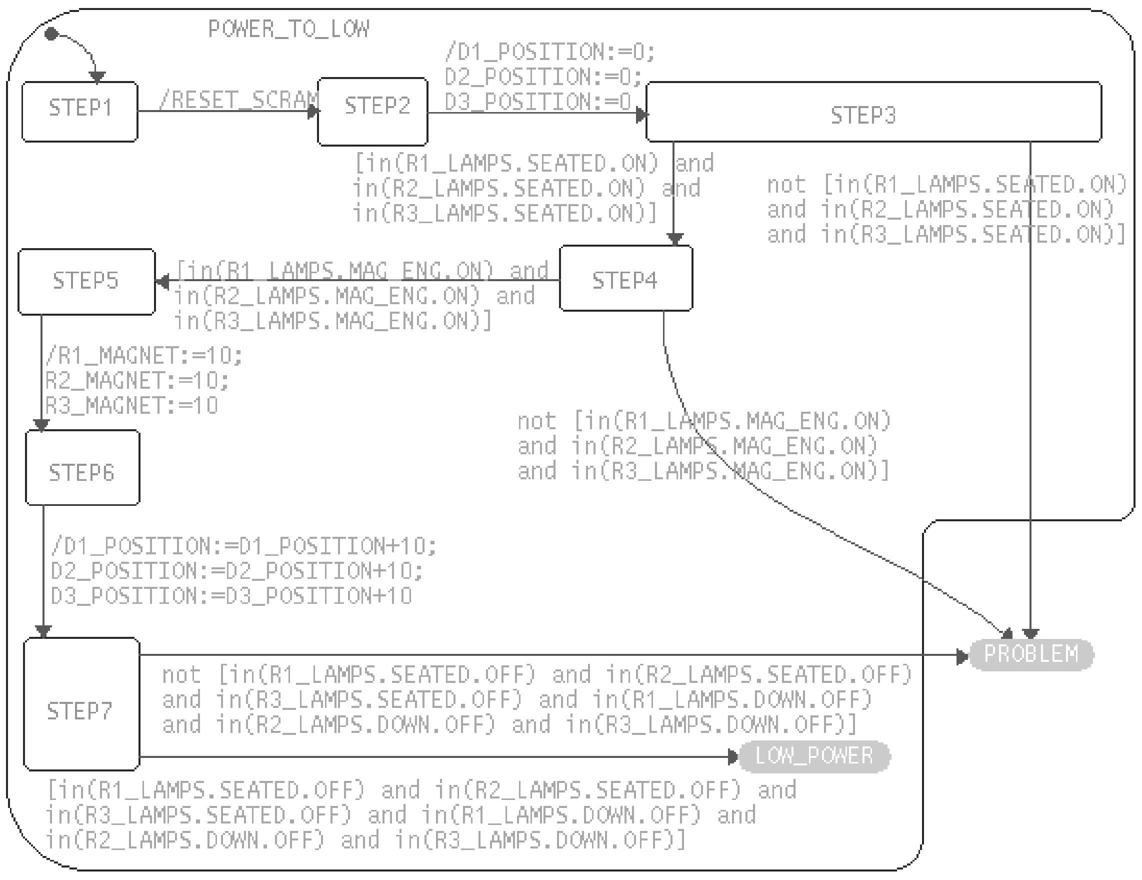
1.1 Power Level



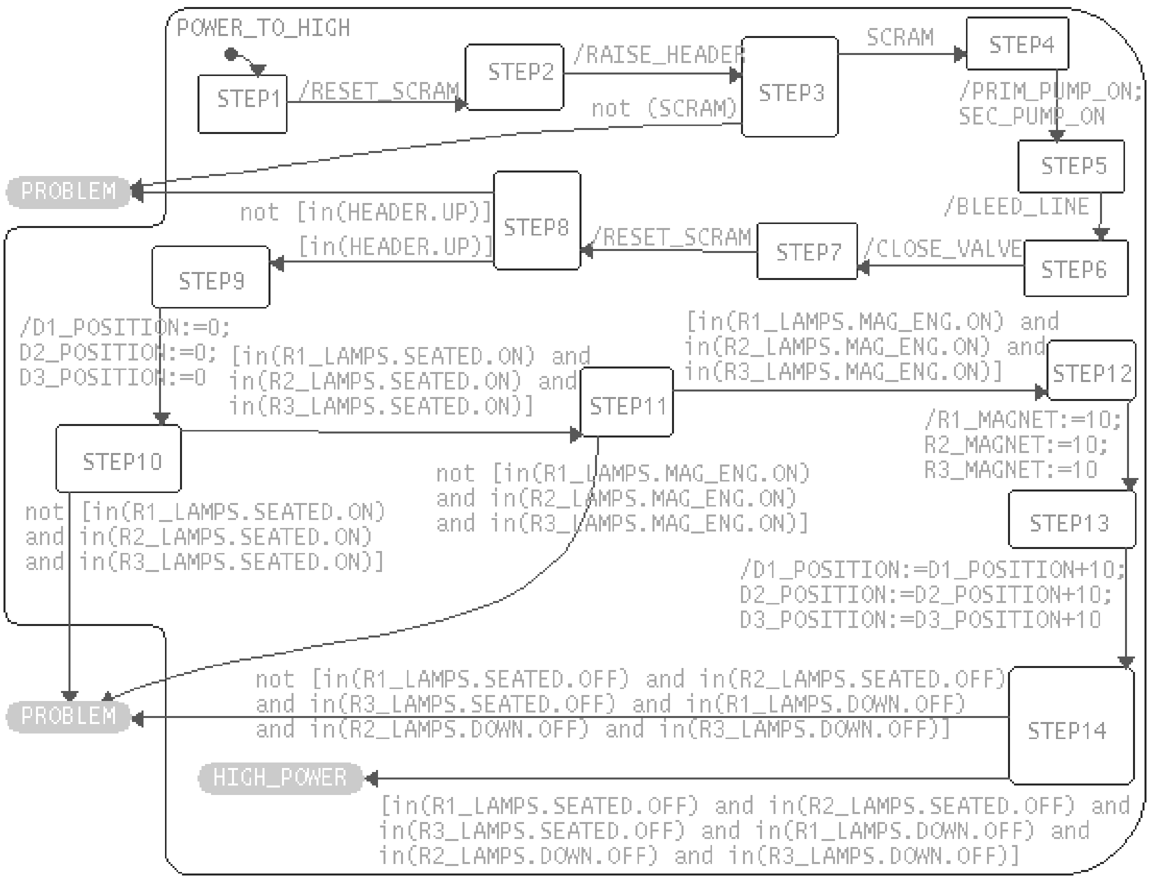
1.1.1 Testing


TESTING
TEST_HEADER
STEP1
/RESET_SCRAM
STEP2
/RAISE_HEADER
STEP3
not SCRAM
PROBLEM
SCRAM
STEP4
STEP6
/CLOSE_VALVE
STEP5
/BLEED_LINE
/RESET_SCRAM
TEST_PRIMARY_PUMP_ON
STEP7
/PRIM_PUMP_ON
STEP8
not SCRAM
PROBLEM
SCRAM
STEP9
/RESET_SCRAM
TEST_PRIMARY_PUMP_OFF
STEP10
/PRIM_PUMP_OFF
STEP11
not SCRAM
PROBLEM
SCRAM
STEP12
/RESET_SCRAM
DONE

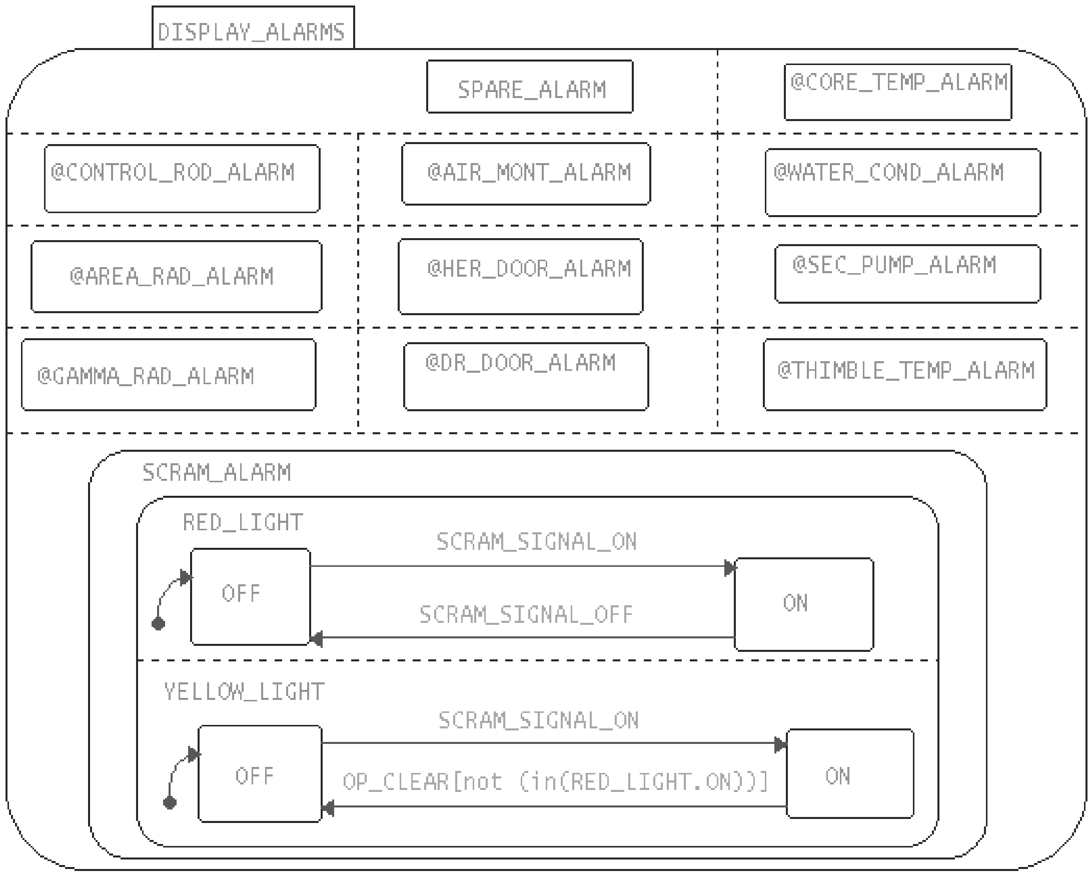
1.1.2 Power to Low


POWER_TO_LOW
STEP1
/RESET_SCRAM
STEP2
/D1_POSITION:=0;
D2_POSITION:=0;
D3_POSITION:=0
STEP3
[in(R1_LAMPS.SEATED.ON) and
in(R2_LAMPS.SEATED.ON) and
in(R3_LAMPS.SEATED.ON)]
not [in(R1_LAMPS.SEATED.ON)
and in(R2_LAMPS.SEATED.ON)
and in(R3_LAMPS.SEATED.ON)]
STEP4
[in(R1_LAMPS.MAG_ENG.ON) and
in(R2_LAMPS.MAG_ENG.ON) and
in(R3_LAMPS.MAG_ENG.ON)]
STEP5
/R1_MAGNET:=10;
R2_MAGNET:=10;
R3_MAGNET:=10
not [in(R1_LAMPS.MAG_ENG.ON)
and in(R2_LAMPS.MAG_ENG.ON)
and in(R3_LAMPS.MAG_ENG.ON)]
STEP6
/D1_POSITION:=D1_POSITION+10;
D2_POSITION:=D2_POSITION+10;
D3_POSITION:=D3_POSITION+10
STEP7
PROBLEM
not [in(R1_LAMPS.SEATED.OFF) and in(R2_LAMPS.SEATED.OFF)
and in(R3_LAMPS.SEATED.OFF) and in(R1_LAMPS.DOWN.OFF)
and in(R2_LAMPS.DOWN.OFF) and in(R3_LAMPS.DOWN.OFF)]
LOW_POWER
[in(R1_LAMPS.SEATED.OFF) and in(R2_LAMPS.SEATED.OFF) and
in(R3_LAMPS.SEATED.OFF) and in(R1_LAMPS.DOWN.OFF) and
in(R2_LAMPS.DOWN.OFF) and in(R3_LAMPS.DOWN.OFF)]

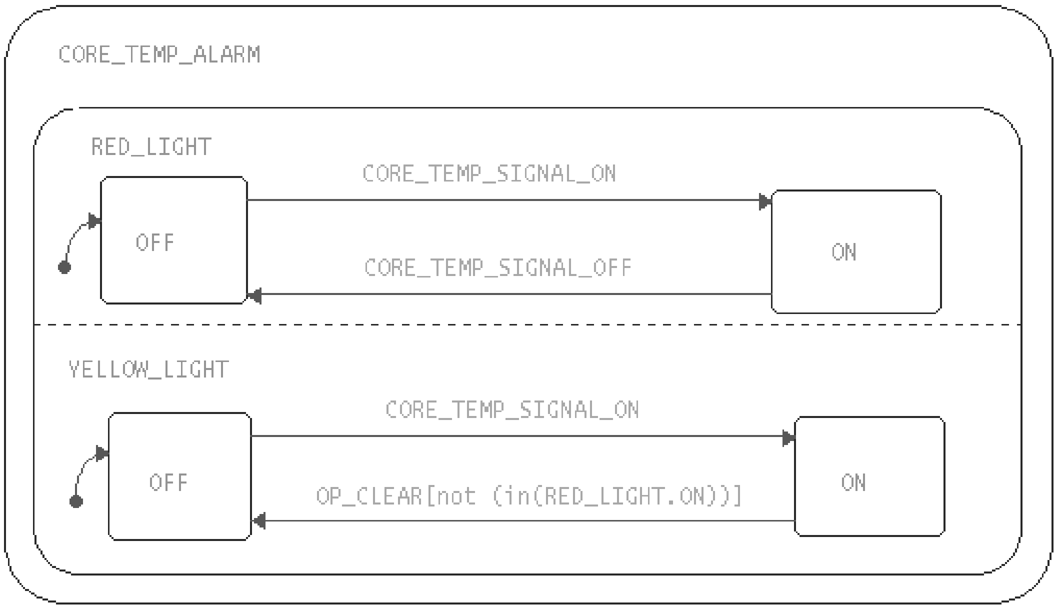
1.1.3 Power to High


POWER_TO_HIGH
STEP1
/RESET_SCRAM
STEP2
/RAISE_HEADER
STEP3
SCRAM
STEP4
/PRIM_PUMP_ON;
SEC_PUMP_ON
not (SCRAM)
PROBLEM
STEP5
/BLEED_LINE
not [in(HEADER.UP)]
STEP8
[in(HEADER.UP)]
/RESET_SCRAM
STEP7
/CLOSE_VALVE
STEP6
STEP9
/D1_POSITION:=0;
D2_POSITION:=0;
D3_POSITION:=0
[in(R1_LAMPS.SEATED.ON) and
in(R2_LAMPS.SEATED.ON) and
in(R3_LAMPS.SEATED.ON)]
[in(R1_LAMPS.MAG_ENG.ON) and
in(R2_LAMPS.MAG_ENG.ON) and
in(R3_LAMPS.MAG_ENG.ON)]
STEP11
STEP12
STEP10
/R1_MAGNET:=10;
R2_MAGNET:=10;
R3_MAGNET:=10
not [in(R1_LAMPS.MAG_ENG.ON)
and in(R2_LAMPS.MAG_ENG.ON)
and in(R3_LAMPS.MAG_ENG.ON)]
STEP13
not [in(R1_LAMPS.SEATED.ON)
and in(R2_LAMPS.SEATED.ON)
and in(R3_LAMPS.SEATED.ON)]
/D1_POSITION:=D1_POSITION+10;
D2_POSITION:=D2_POSITION+10;
D3_POSITION:=D3_POSITION+10
not [in(R1_LAMPS.SEATED.OFF) and in(R2_LAMPS.SEATED.OFF)
and in(R3_LAMPS.SEATED.OFF) and in(R1_LAMPS.DOWN.OFF)
and in(R2_LAMPS.DOWN.OFF) and in(R3_LAMPS.DOWN.OFF)]
PROBLEM
STEP14
HIGH_POWER
[in(R1_LAMPS.SEATED.OFF) and in(R2_LAMPS.SEATED.OFF) and
in(R3_LAMPS.SEATED.OFF) and in(R1_LAMPS.DOWN.OFF) and
in(R2_LAMPS.DOWN.OFF) and in(R3_LAMPS.DOWN.OFF)]

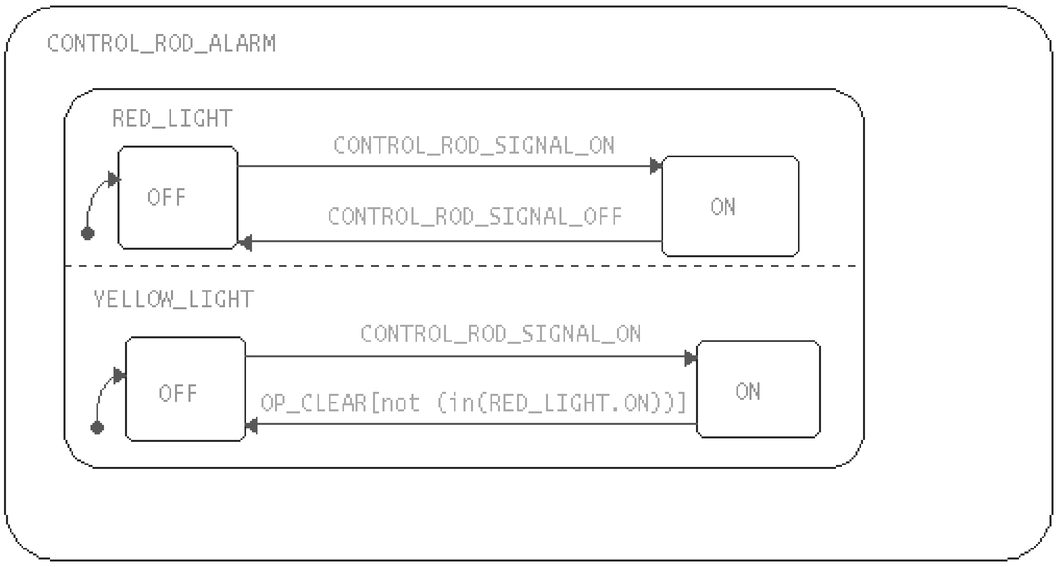
2.2a Alarms: Display Alarms



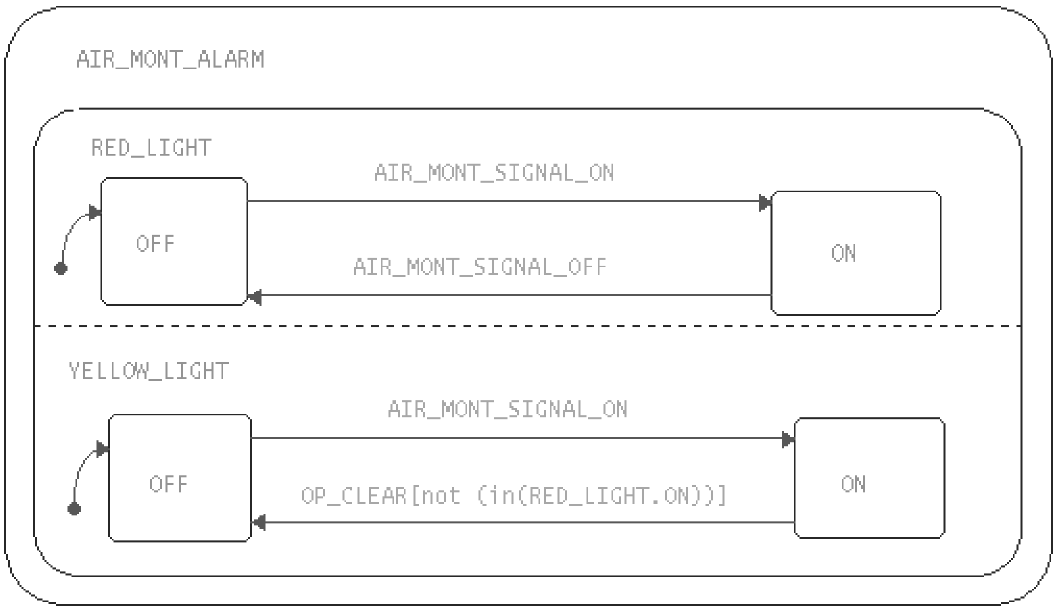
2.2a.1 Core Temperature Alarm



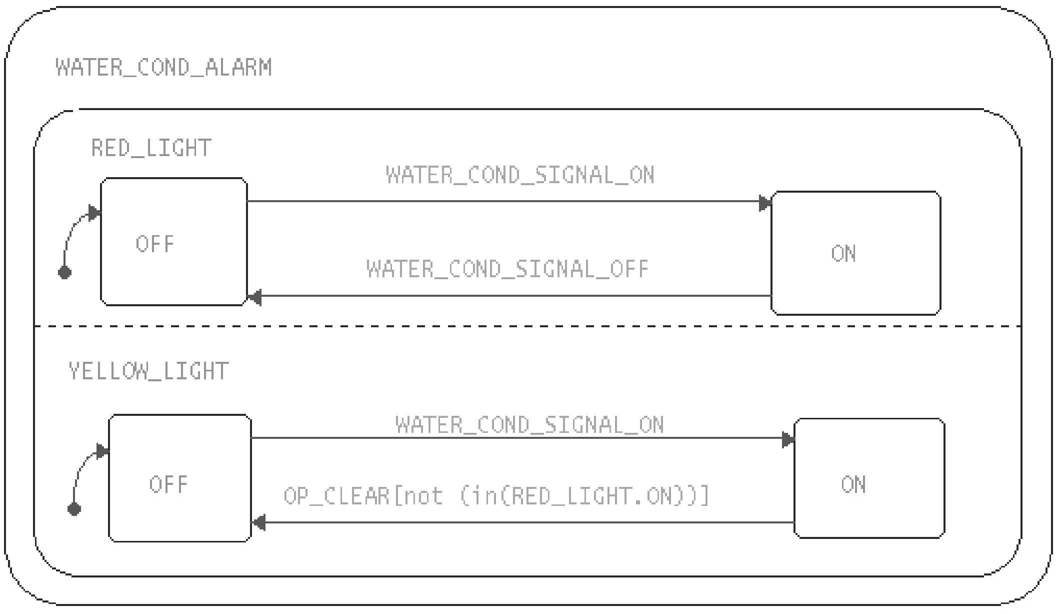
2.2a.2 Control Rod Alarm



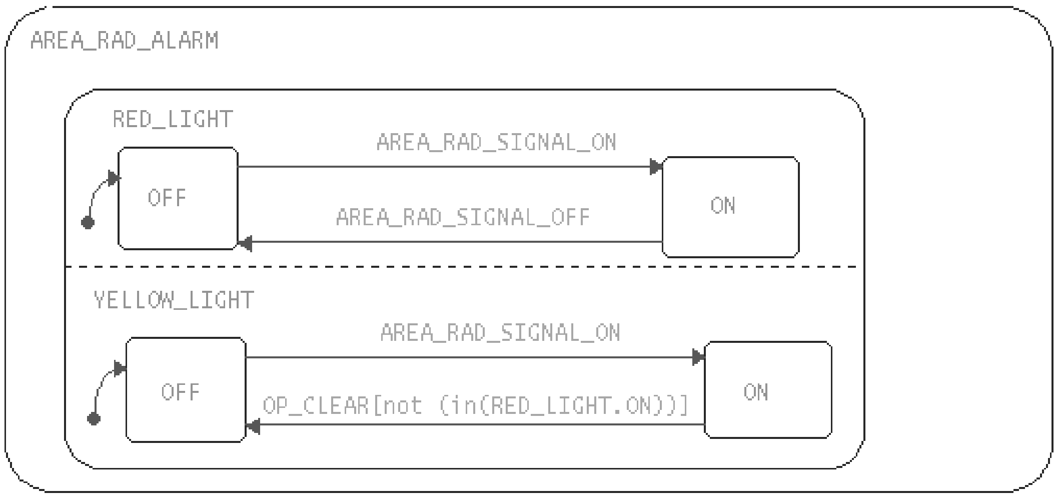
2.2a.3 Air Monitor Alarm



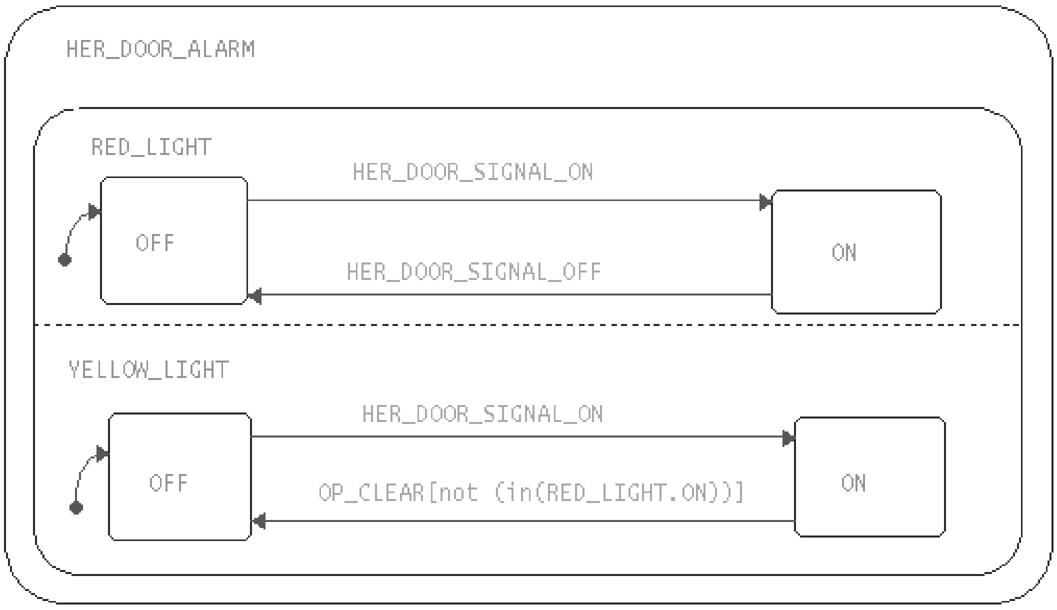
2.2a.4 Water Conductivity Alarm



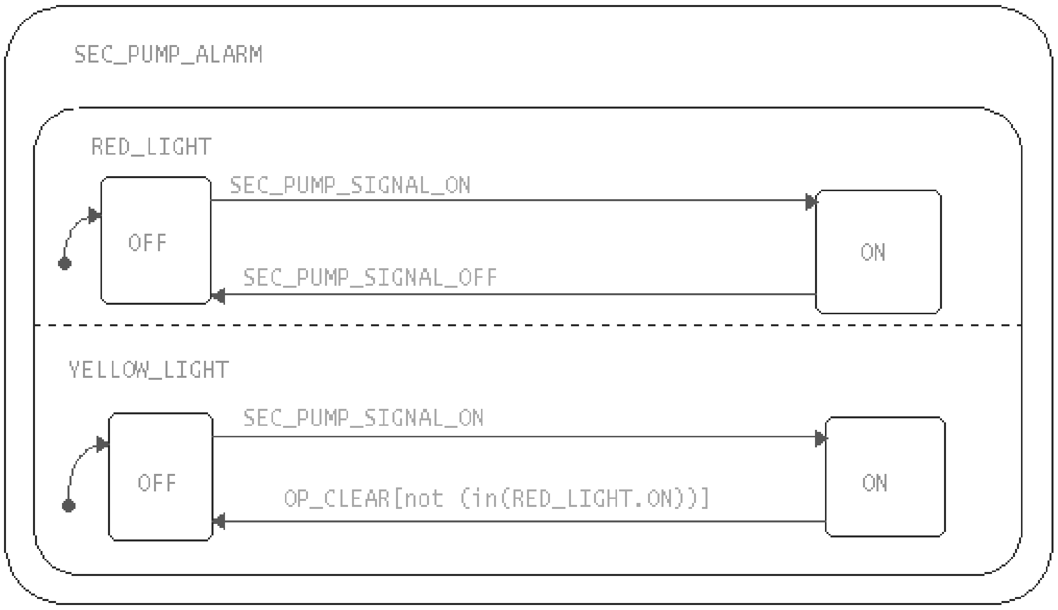
2.2a.5 Area Radiation Alarm



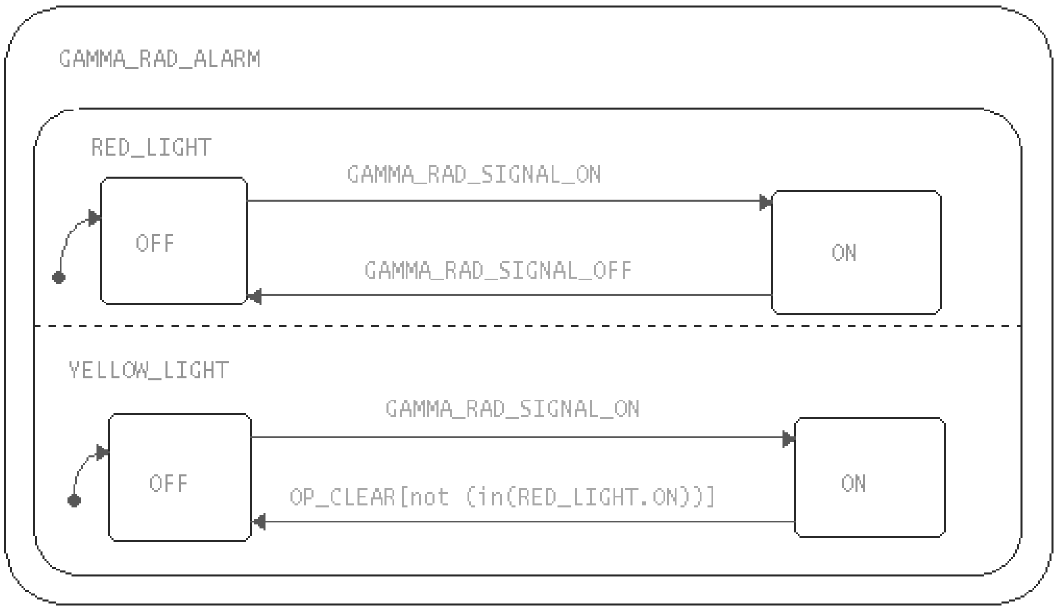
2.2a.6 Heat Exchange Room Door Alarm



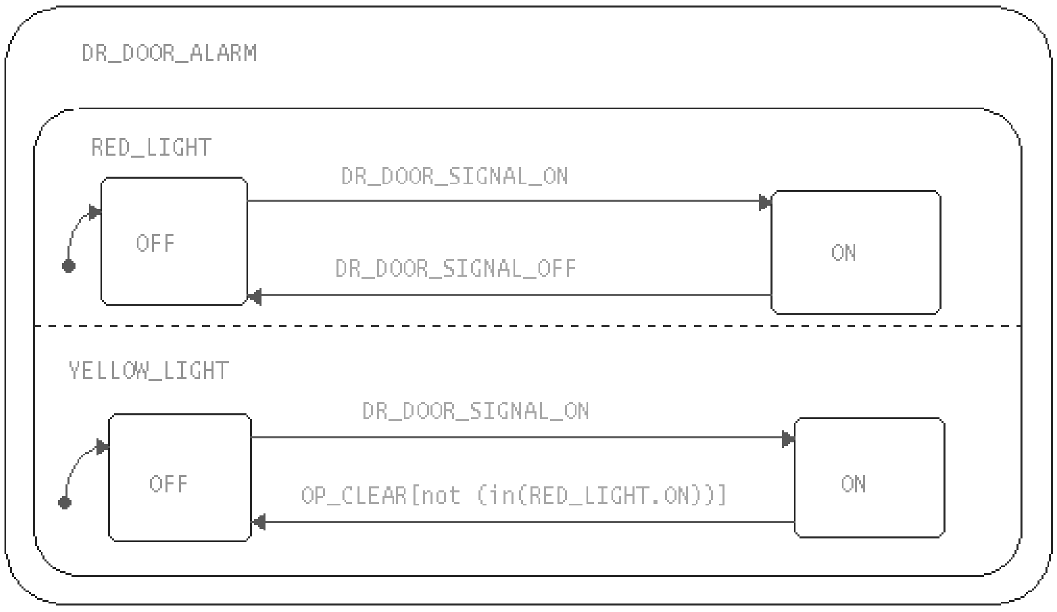
2.2a.7 Secondary Pump Alarm



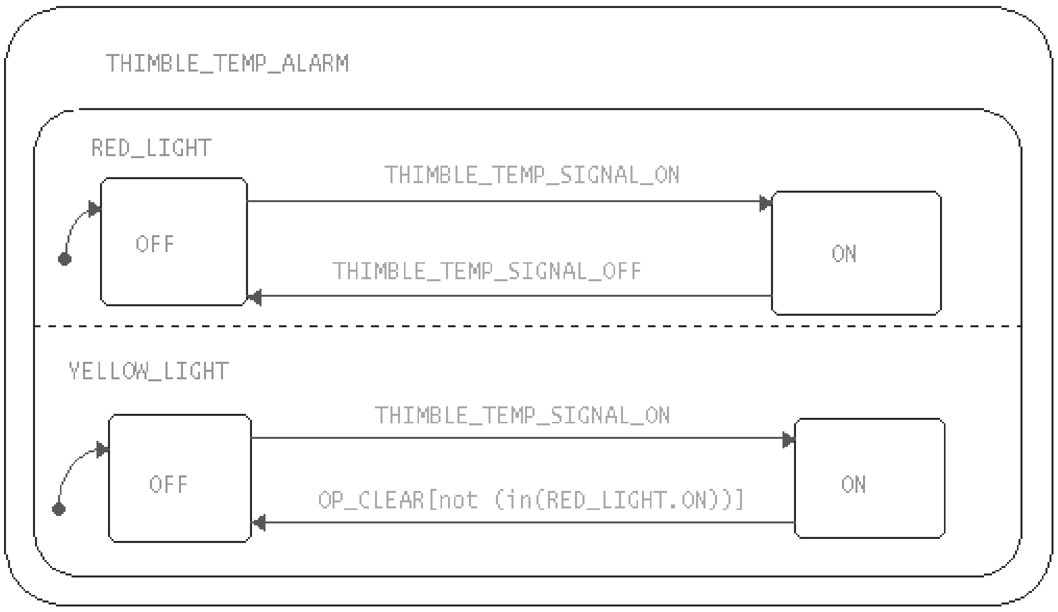
2.2a.8 Gamma Radiation Alarm



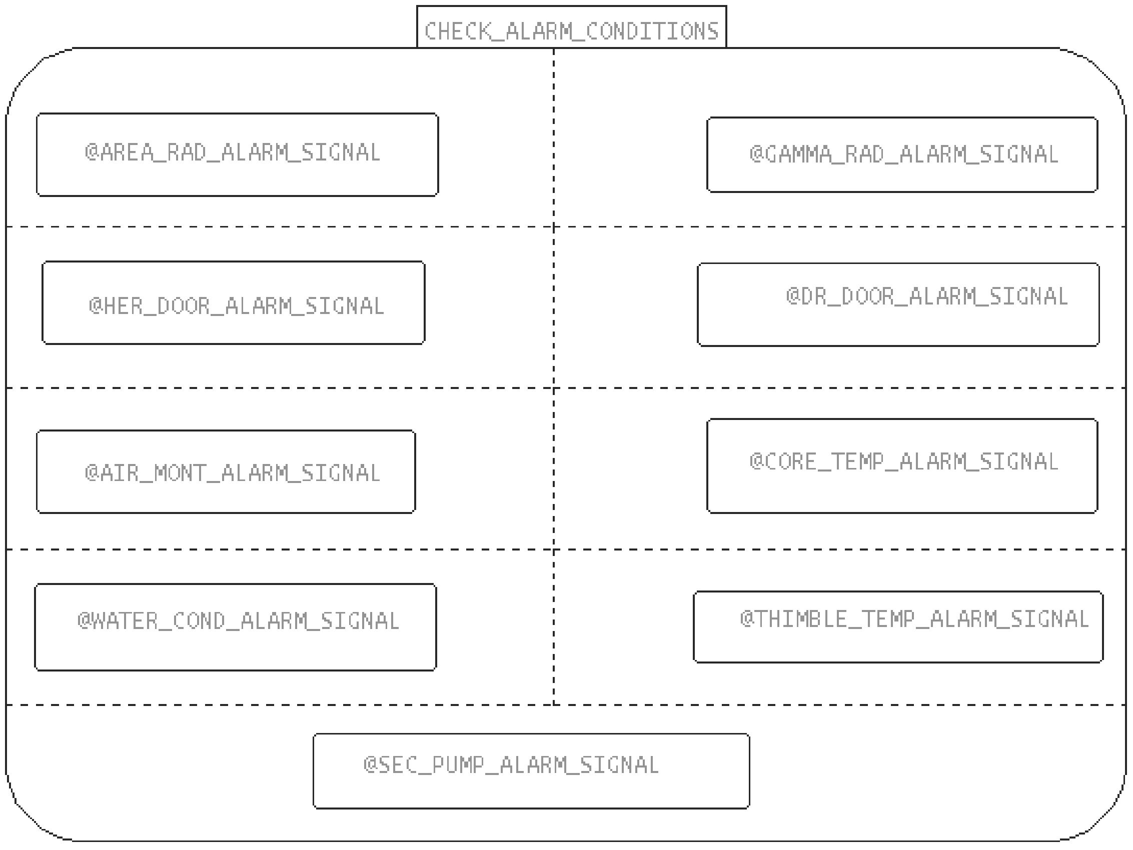
2.2a.9 Demineralizer Room Door Alarm



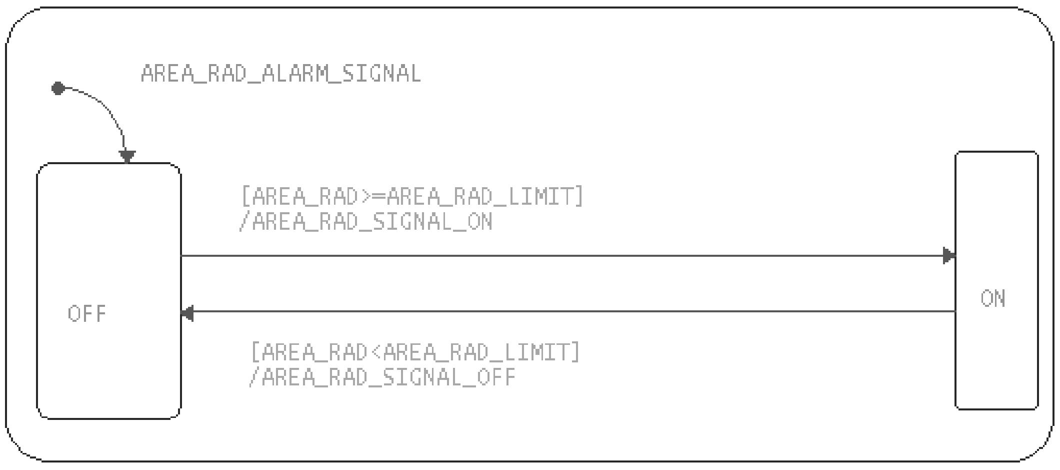
2.2a.10 Thimble Temperature Alarm



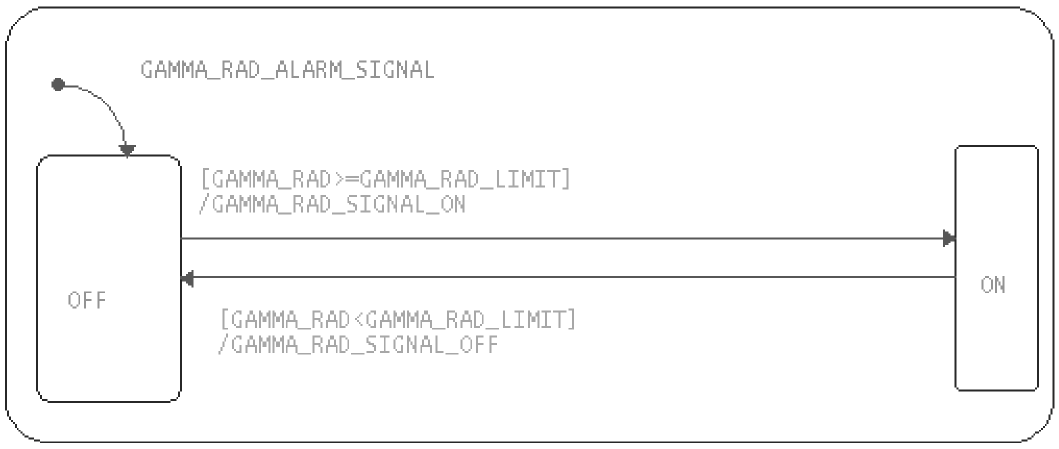
3.2b Alarms: Check Alarm Conditions



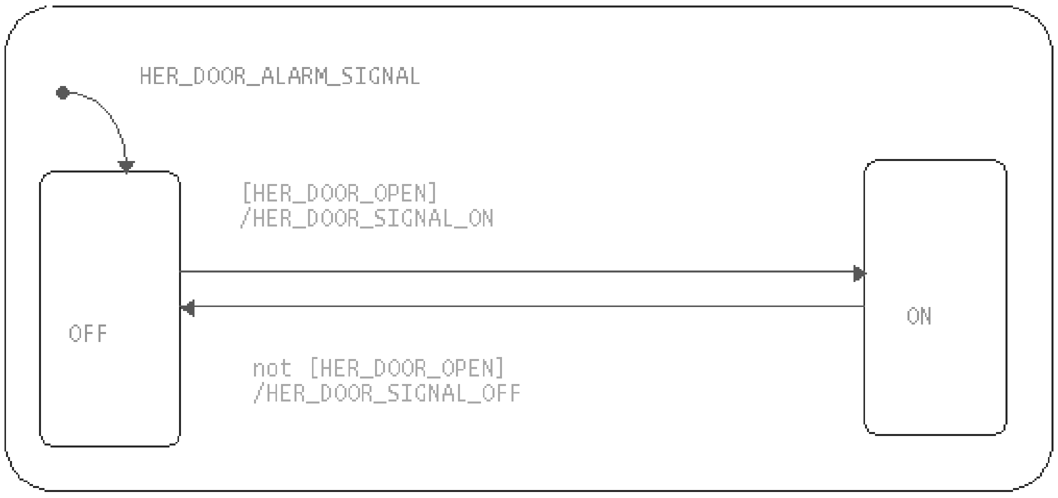
3.2b.1 Area Radiation Alarm Signal



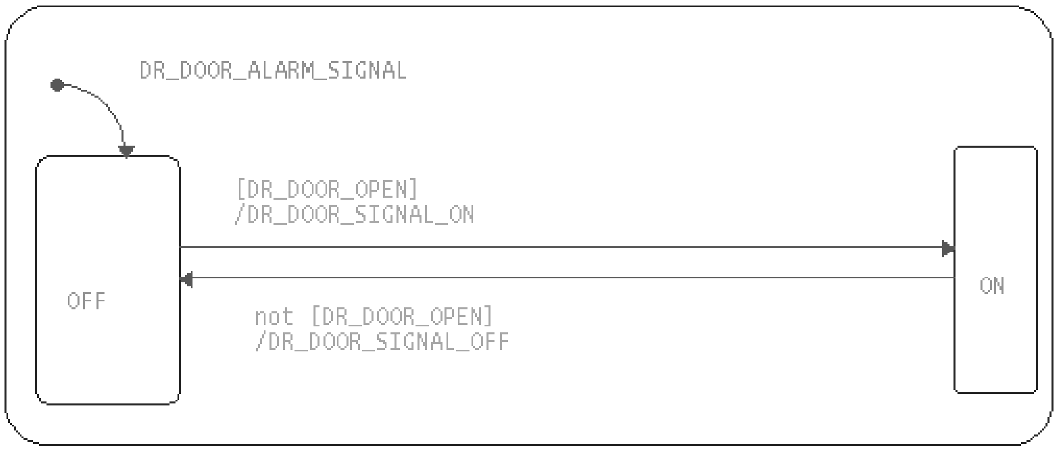
3.2b.2 Gamma Radiation Alarm Signal



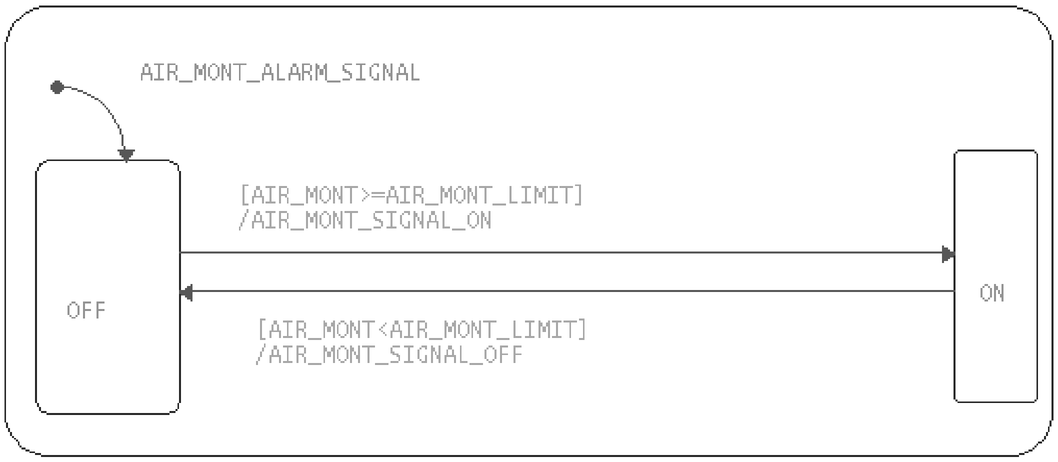
3.2b.3 Heat Exchanger Room Door Alarm Signal



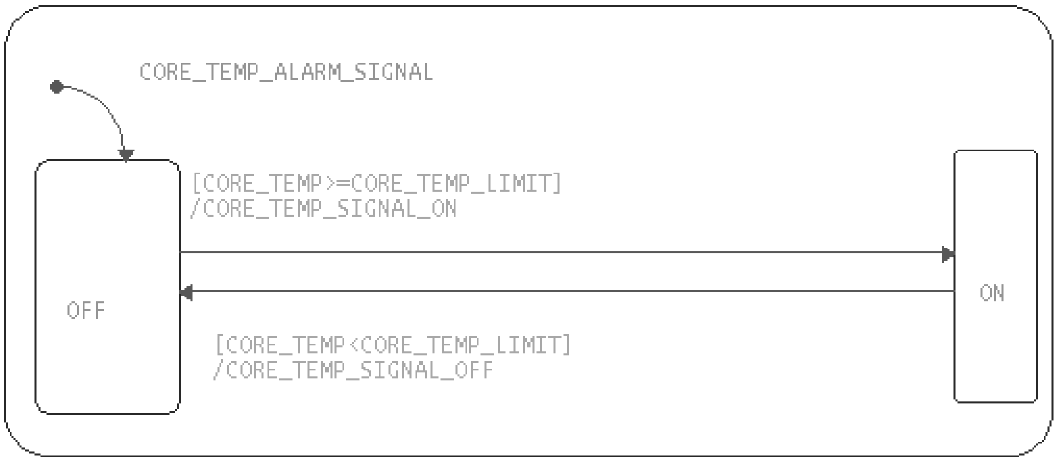
3.2b.4 Demineralizer Room Door Alarm Signal



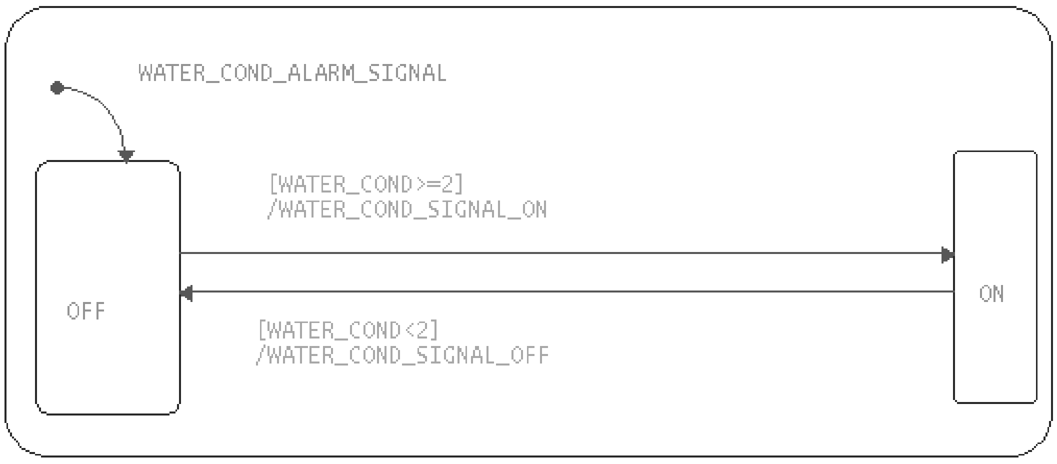
3.2b.5 Air Monitor Alarm Signal



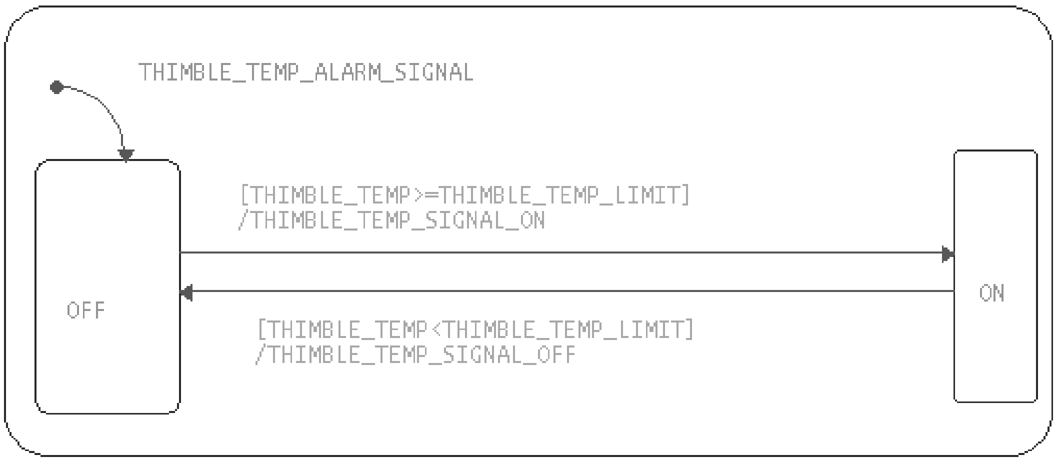
3.2b.6 Core Temperature Alarm Signal



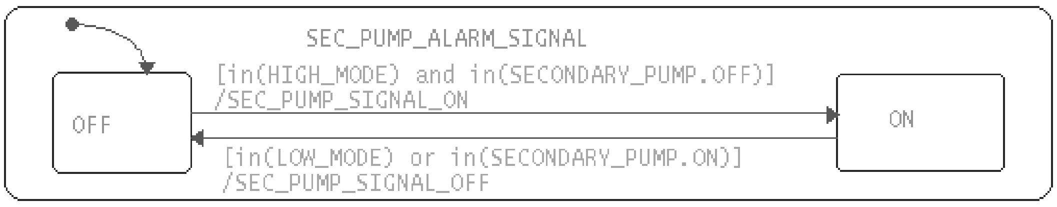
3.2b.7 Water Conductivity Alarm Signal



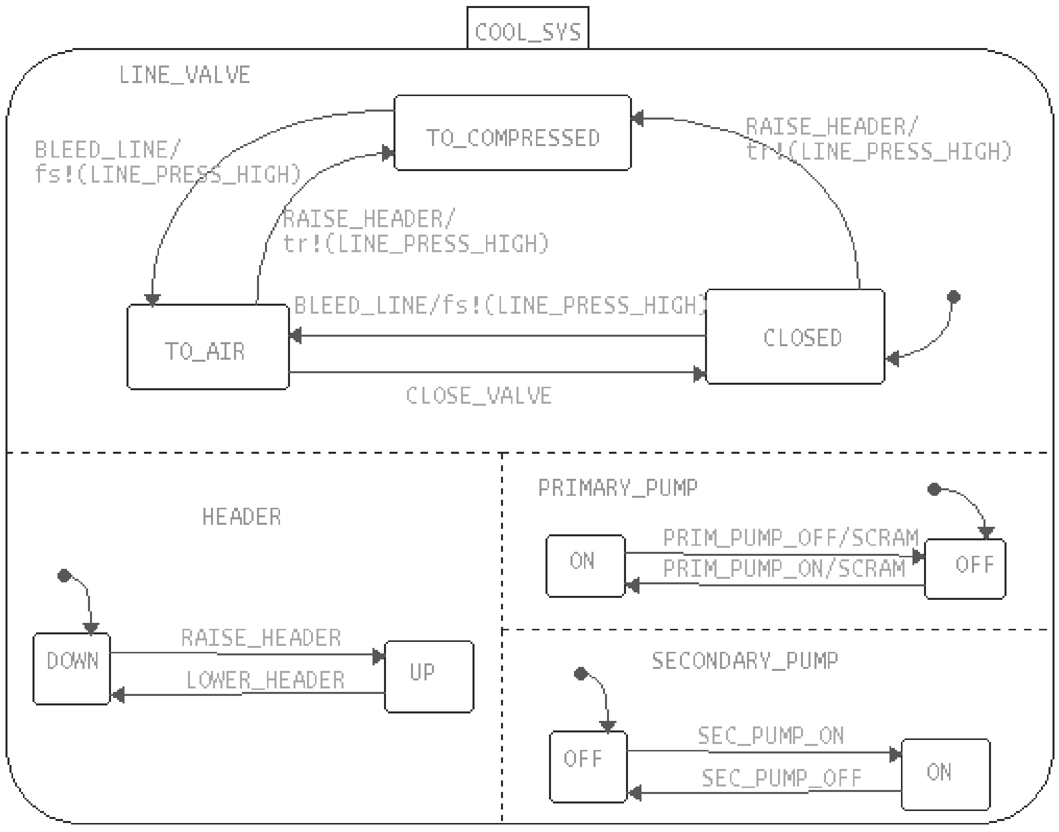
3.2b.8 Thimble Temperature Alarm Signal


THIMBLE_TEMP_ALARM_SIGNAL
[THIMBLE_TEMP>=THIMBLE_TEMP_LIMIT]
/THIMBLE_TEMP_SIGNAL_ON
OFF
ON
[THIMBLE_TEMP<THIMBLE_TEMP_LIMIT]
/THIMBLE_TEMP_SIGNAL_OFF

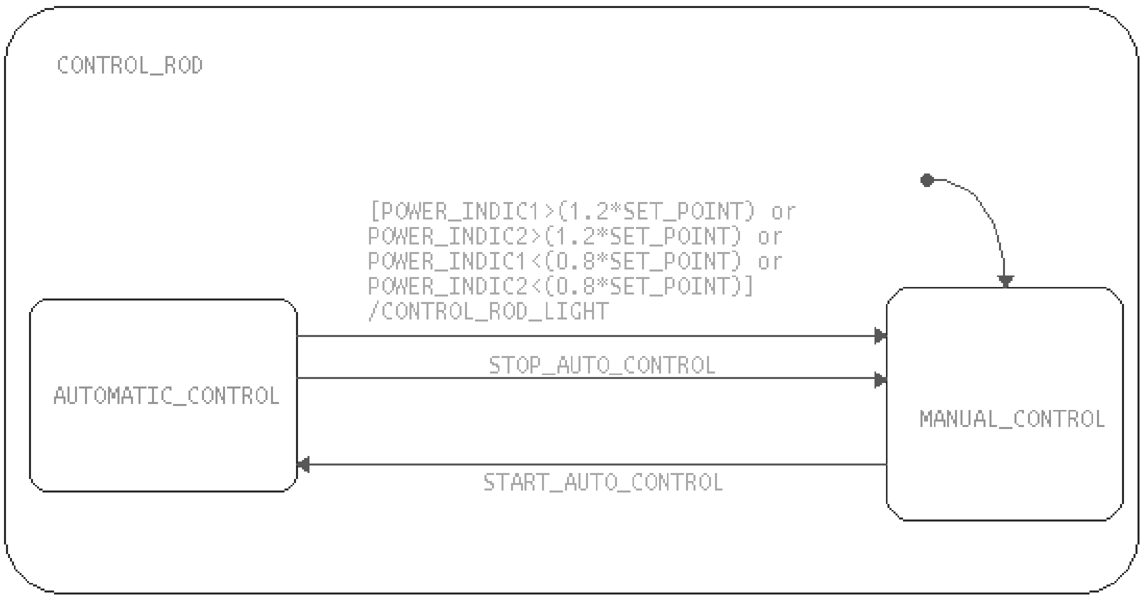
3.2b.9 Secondary Pump Alarm Signal


SEC_PUMP_ALARM_SIGNAL
[in(HIGH_MODE) and in(SECONDARY_PUMP.OFF)]
/SEC_PUMP_SIGNAL_ON
OFF
ON
[in(LOW_MODE) or in(SECONDARY_PUMP.ON)]
/SEC_PUMP_SIGNAL_OFF

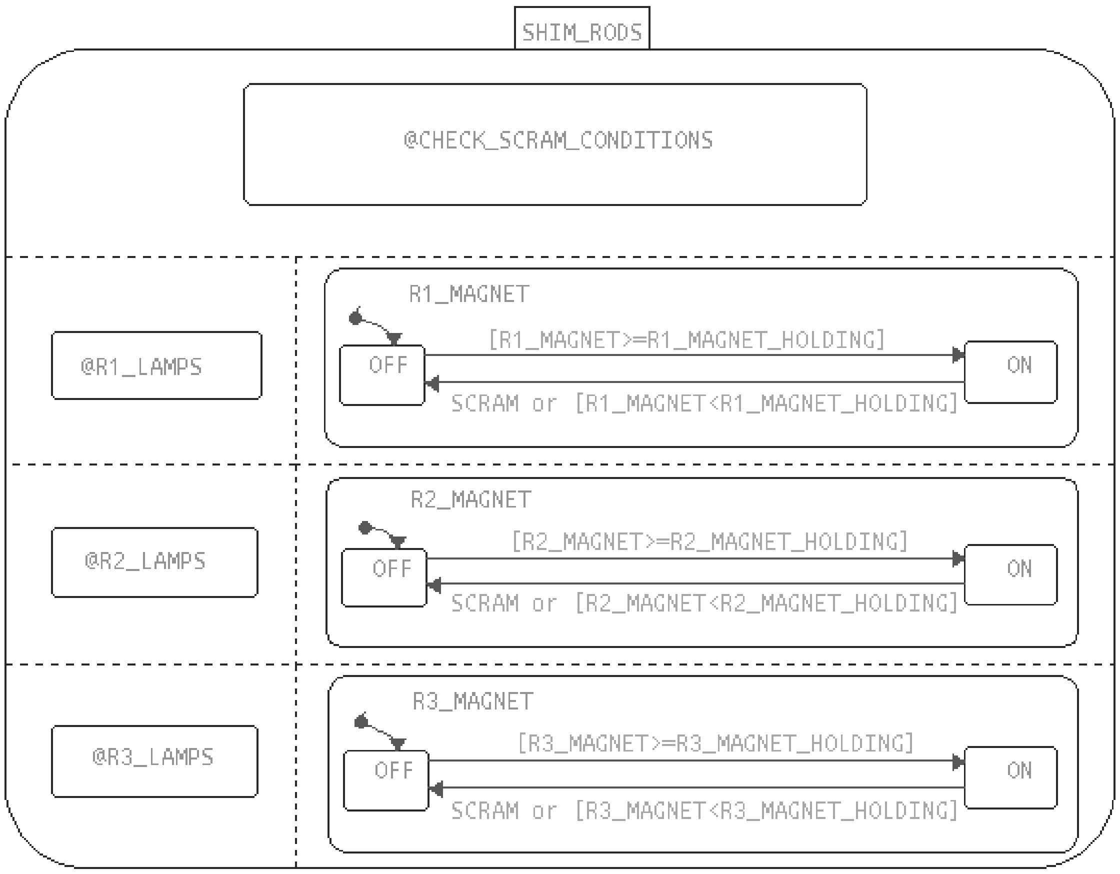
3.3 Cooling System



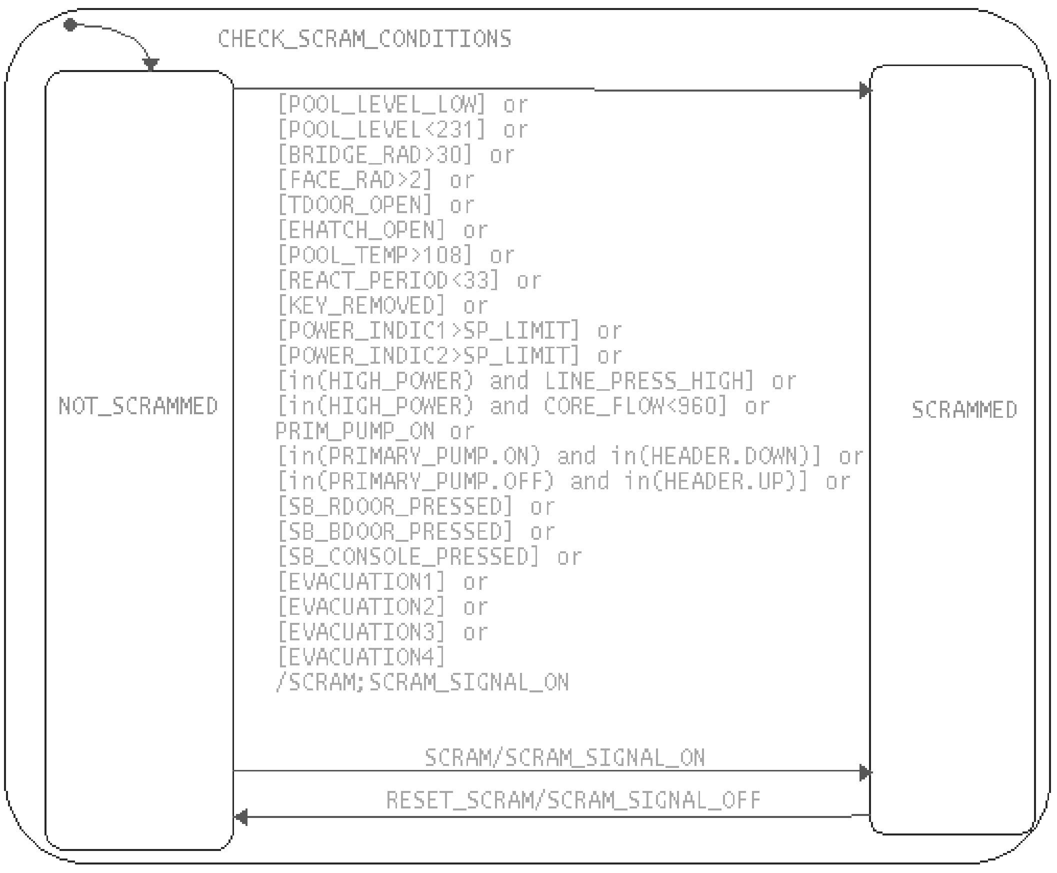
4.4a Rods: Control Rod


CONTROL_ROD
[POWER_INDIC1>(1.2*SET_POINT) or
POWER_INDIC2>(1.2*SET_POINT) or
POWER_INDIC1<(0.8*SET_POINT) or
POWER_INDIC2<(0.8*SET_POINT)]
/CONTROL_ROD_LIGHT
STOP_AUTO_CONTROL
AUTOMATIC_CONTROL
MANUAL_CONTROL
START_AUTO_CONTROL

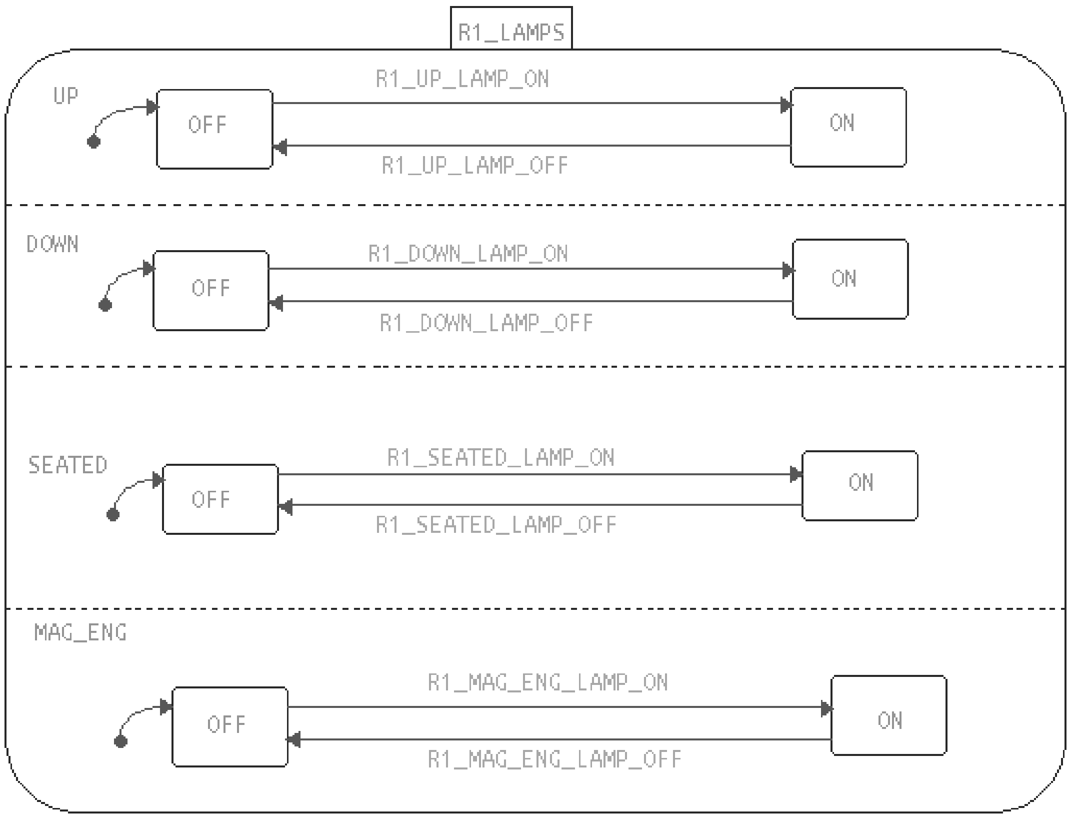
5.4b Rods: Shim Rods


SHIM_RODS
@CHECK_SCRAM_CONDITIONS
R1_MAGNET
@R1_LAMPS
[R1_MAGNET>=R1_MAGNET_HOLDING]
OFF
ON
SCRAM or [R1_MAGNET<R1_MAGNET_HOLDING]
R2_MAGNET
@R2_LAMPS
[R2_MAGNET>=R2_MAGNET_HOLDING]
OFF
ON
SCRAM or [R2_MAGNET<R2_MAGNET_HOLDING]
R3_MAGNET
@R3_LAMPS
[R3_MAGNET>=R3_MAGNET_HOLDING]
OFF
ON
SCRAM or [R3_MAGNET<R3_MAGNET_HOLDING]

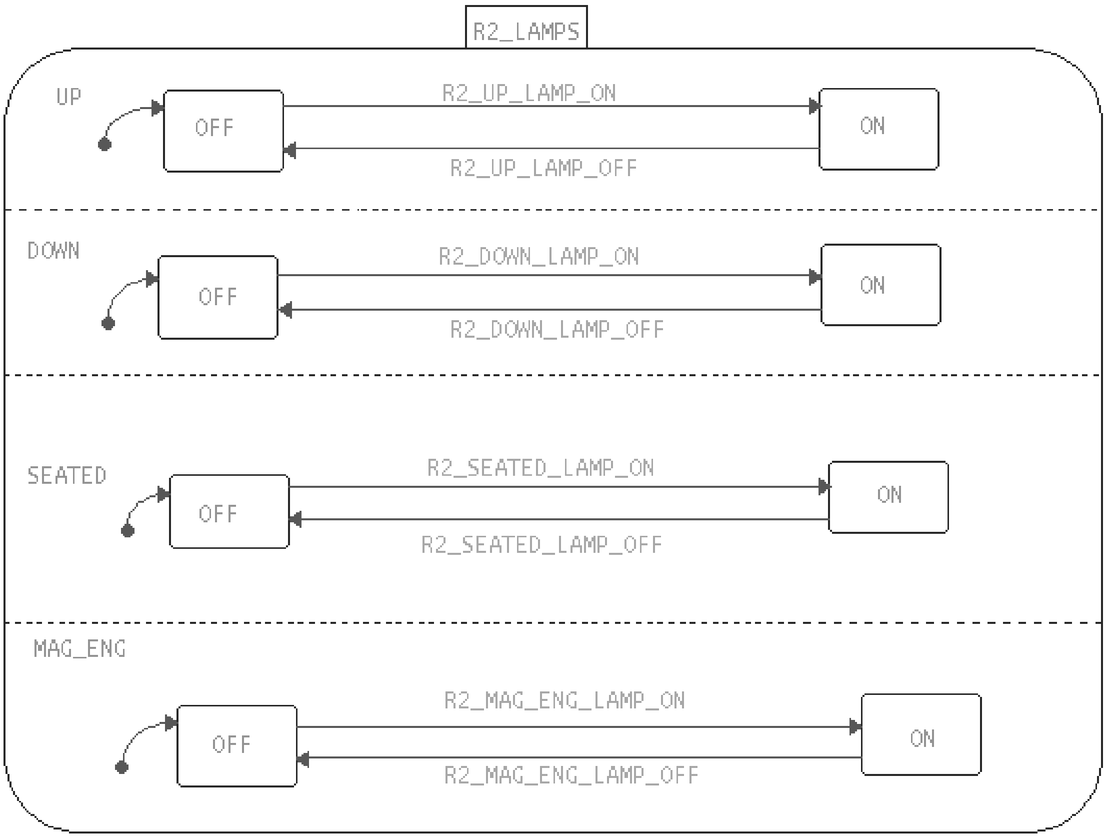
5.4b.1 Check Scram Conditions



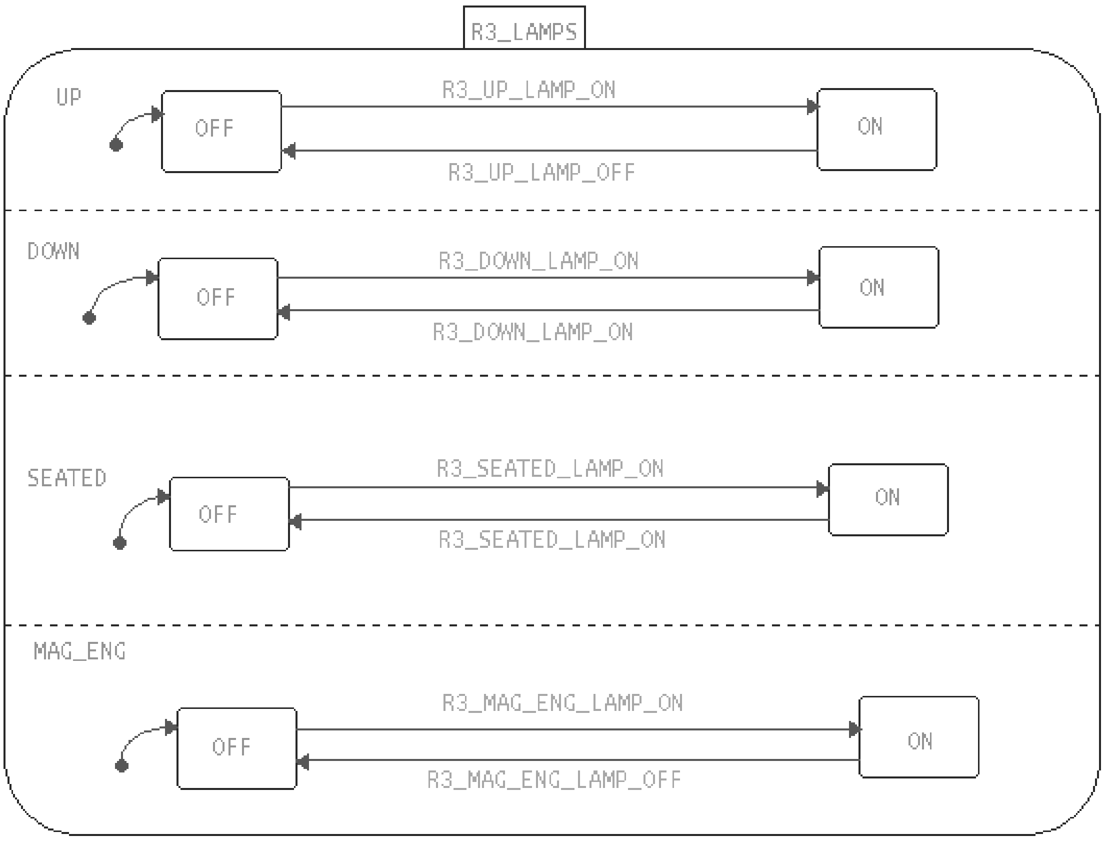
5.4b.2 Rod1 Lamps



5.4b.3 Rod2 Lamps



5.4b.4 Rod3 Lamps



Appendix C
Z Specification

1 Introduction

This document describes the University of Virginia Nuclear Reactor in terms of reactor system, scram conditions, alarms and start-up procedures. For each informal description, there is a corresponding Z formal specification.

The informal specification is taken from a pre-specification document that was elaborated previously. In the original documentation, there are three different limits for monitored variables describing reactor operation. In this document, only the actual limits used in the scram conditions and alarms are used.

2 Reactor System Specification

The reactor system specification will consist of only of information that is necessary to describe the reactor operation, without any regard to scram conditions or alarms. The following signals are necessary for the description of several parts of the system:

OperationStatus	==	{*Idle, Operating*}
ScramStatus	==	{*Scrammed, NotScrammed*}
PowerSelector	==	{*LowPower, HighPower*}
Switch	==	{*On, Off*}
VerticalPosition	==	{*Up, Down*}
OpenClose	==	{*Open, Close*}
Button	==	{*Pressed, NotPressed*}
Signal	==	{*High, Low*}
Valve	==	{*CompressedAirToLine, LineClosed, LineToAtmosphere*}

Using these signals, the schema describing the shim rods is:

ShimRod

$Position$	:	$\mathbb{N}$
$MagCurrent$	:	$\mathbb{N}$
$MinHoldingCurrent$	:	$\mathbb{N}$
$Engaged$	:	$Switch$
$MinPosition$	:	$\mathbb{N}$
$MaxPosition$	:	$\mathbb{N}$
$Reactivity$	:	$Position \mathrel{\rightarrowtail\!\!\!\!\!\!+} \mathbb{N}$
$MinPosition$	$\leq$	$Position$
$Position$	$\leq$	$MaxPosition$
$\forall\ a : \mathbb{N} \mid (MinPosition \leq a \leq MaxPosition)$	$\bullet$	$Reactivity(a) \geq 0$

The schema describing the control rod is:

ControlRod

$Position$	:	$\mathbb{N}$
$AutoControl$	:	$Switch$
$MinPosition$	:	$\mathbb{N}$
$MaxPosition$	:	$\mathbb{N}$
$Reactivity$	:	$Position \mathrel{\rightarrowtail\!\!\!\!\!\!+} \mathbb{N}$
$MinPosition$	$\leq$	$Position$
$Position$	$\leq$	$MaxPosition$
$\forall\ a : \mathbb{N} \mid (MinPosition \leq a \leq MaxPosition)$	$\bullet$	$Reactivity(a) \geq 0$

The schema will describing pumps in the system is:

Pump

$PumpSwitch$	:	$Switch$
$Voltage$	:	$\mathbb{N}$
$Voltage > 0$	$\Rightarrow$	$PumpSwitch = On$

The schema describing the signals provided by the different sensors in the system is:

Inputs

$PowerIndic1?$	:	$\mathbb{N}$
$PowerIndic2?$	:	$\mathbb{N}$
$SPLimitLow?$	:	$\mathbb{N}$
$SPLHigh?$	:	$\mathbb{N}$
$PoolLevelHigher19ft3in?$	:	$HighLow$
$PoolLevelMonitor?$	:	$\mathbb{N}$
$BridgeRad?$	:	$\mathbb{N}$
$FaceRadiation?$	:	$\mathbb{N}$
$AirLinePressureAbove2psi?$	:	$HiLow$
$TruckDoor?$	:	$OpenClose$
$EscapeHatch?$	:	$OpenClose$
$SwitchAtRoomDoor?$	:	$Button$
$SwitchAtBackDoor?$	:	$Button$
$EvacuationAlarm1?$	:	$Switch$
$EvacuationAlarm2?$	:	$Switch$
$EvacuationAlarm3?$	:	$Switch$
$EvacuationAlarm4?$	:	$Switch$
$PoolTemperature?$	:	$\mathbb{N}$
$ReactorPeriod?$	:	$\mathbb{N}$
$Flow?$	:	$\mathbb{N}$
$KeySwitch?$	:	$Switch$
$ManualScram?$	:	$Button$
$ResetButton?$	:	$Button$
$ArgonRadiationIndicator?$	:	$Signal$
$CoreGammaAlarm?$	:	$Signal$
$SpareSignal?$	:	$Signal$
$AirMonitorSignal?$	:	$Signal$
$HeatExchangerDoor?$	:	$Door$
$DemireralizerRoomDoor?$	:	$Door$
$CoreDiffTempIndic?$	:	$\mathbb{N}$
$DiffTempAlarmLevel?$	:	$\mathbb{N}$
$WaterCondIndic?$	:	$\mathbb{N}$
$WaterCondAlarmLevel?$	:	$\mathbb{N}$
$ThimbleTemperature?$	:	$\mathbb{N}$
$ThimbleTempAlarmLimit?$	:	$\mathbb{N}$
$HeaderDown?$	:	$Signal$
$ShimRod1Seated?$	:	$Signal$
$ShimRod2Seated?$	:	$Signal$
$ShimRod3Seated?$	:	$Signal$
$TouchingDriver1?$	:	$Signal$
$TouchingDriver2?$	:	$Signal$
$TouchingDriver3?$	:	$Signal$

With the previous schemas, the schema describing the reactor system is:

Reactor

ReactorStatus	:	*OperationStatus*
Scram	:	*ScramStatus*
PowerSelection	:	*PowerSelector*
HeaderPosition	:	*VerticalPosition*
*ShimRod*1	:	*ShimRod*
*ShimRod*2	:	*ShimRod*
*ShimRod*3	:	*ShimRod*
ShimRods	=	{*ShimRod*1, *ShimRod*2, *ShimRod*3}
RegulatingRod	:	*ControlRod*
PrimaryPump	:	*Pump*
SecondaryPump	:	*Pump*
Pumps	=	{*PrimaryPump*, *SecondaryPump*}
CoolingTower	:	*Switch*
HeaderValve	:	*Valve*
Scram = Scrammed	⇒	*ReactorStatus = Idle*

The schema that sets the initial conditions on the reactor is:

ReactorInit

Δ*Reactor*

ReactorStatus'	=	*Idle*
Scram'	=	*NotScrammed*
PowerSelection'	=	*LowPower*
HeaderPosition'	=	*Down*
CoolingTower'	=	*Off*
HeaderValve'	=	*LineToAtmosphere*
∀ *a* ∣ *a* ∈ *Rods* • *a.Position*'	=	*a.MinPosition*
∀ *a* ∣ *a* ∈ *ShimRod* • *a.TouchingDrive*'	=	*False*
∀ *a* ∣ *a* ∈ *ShimRod* • *a.MagCurrent*'	=	0
∀ *a* ∣ *a* ∈ *Pumps* • *a.PumpSwitch*'	=	*Off*
∀ *a* ∣ *a* ∈ *Pumps* • *a.Voltage*'	=	0

3 Scram Condition Description

Power Range #1 or Power Range #2 Exceeded

Informal Description:

- There are two different channels for measuring power level.
- The reactor is scrammed if the power level goes above 250 kW in natural convection mode or 2.5 MW in forced convection mode, in either of the channels.

Z Formal Specification:

It is necessary to represent the two different measures given by the two different channels. In this specification, the values from the different channels will be represented by the natural numbers *PowerIndic*1? and *PowerIndic*2?. The same will hold true for the scram limits *SPLimitLow* and *SPLimitHigh*? of the power level. All sensor information is present in the schema *Inputs*.

Given these representations, the schema for the scram caused by exceeding the power range limit is:

CheckPowerRange

$\Delta Reactor$
Inputs

$$\begin{array}{lcl}
Scram & = & NotScrammed \\
(PowerIndic1? > SPLimitLow? \wedge PowerSelection = LowPower) & \Rightarrow & Scram' = Scrammed \\
(PowerIndic2? > SPLimitLow? \wedge PowerSelection = LowPower) & \Rightarrow & Scram' = Scrammed \\
(PowerIndic1? > SPLimitHigh? \wedge PowerSelection = HighPower) & \Rightarrow & Scram' = Scrammed \\
(PowerIndic2? > SPLimitHigh? \wedge PowerSelection = HighPower) & \Rightarrow & Scram' = Scrammed
\end{array}$$

Pool Water Level Low

Informal Description

- There are two different sensors: electrical conductivity and mechanical switch.
- The pool level electrical conductivity sensor is able to measure intermediate values.
- The pool level mechanical switch sensor gives a boolean result: the water level is either above 19'3 " or below 19'3".
- The reactor is scrammed if any of the sensors report that the water level above the core has dropped below 19'3".

Z Formal Specification

To represent this scram condition, it is necessary to represent the different sensors. The electrical conductivity sensor can be represented by a natural number *PoolLevelMonitor*?, representing the number of inches of water above the core. The mechanical switch provides only a *Hi* or *Low* Boolean value *PoolLevelHigher*19*ft*3*in*?. All this sensor information is in the *Inputs* schema.Given this representations, the schema for this scram condition is:

```
┌─── CheckPoolWaterLevel ─────────────────────────────
│ ΔReactor
│ Inputs
├────────────
│ Scram                              =    NotScrammed
│ PoolLevelHigher19ft3in? = Low      ⇒    Status' = Scrammed
│ PoolLevelMonitor? < 231            ⇒    Status' = Scrammed
└─────────────────────────────────────────────────────
```

Bridge Radiation Level High

Informal Description

- The sensor is an ion chamber placed above the pool.
- The reactor is scrammed if the radiation level on the ground floor goes above 2 mR/h.

Z Formal Specification

The bridge radiation level can be represented by *BridgeRad*?, a natural number describing the radiation level at the bridge when measured in mR/h. This sensor information is described by the *Inputs* schema. The schema corresponding to this scram condition is:

```
┌─── CheckBridgeRadiationLevel ───────────────────────
│ ΔReactor
│ Inputs
├────────────
│ Scram             =    NotScrammed
│ BridgeRad? > 30   ⇒    Scram' = Scrammed
└─────────────────────────────────────────────────────
```

Face Radiation Level High

Informal Description

- The sensor is an ion chamber placed at ground level.
- The reactor is scrammed if the radiation level on the ground floor goes above 2 mR/h.

Z Formal Specification

The face radiation level can be described by *FaceRadiation*?, a natural number describing the radiation level at the core face when measured in mR/h. This sensor information is described by the *Inputs* schema. The schema corresponding to this scram condition is:

```
┌── CheckFaceRadiation ─────────────────────────────────────
│ ΔReactor
│ Inputs
├──────────────
│ Scram                  =     NotScrammed
│ FaceRadiation? > 2     ⇒     Scram' = Scrammed
└───────────────────────────────────────────────────────────
```

Primary Pump Switch Turned from Off to On

Informal Description

- There is a contact sensor is on the switch.
- When the switch goes from off to on, the reactor is scrammed.
- The reactor is scrammed by the action of turning the pump on.

Z Formal Specification

The schema representing this scram condition is:

```
┌── CheckSwitchPrimaryPumpTurnsOn ──────────────────────────
│ ΔReactor
├──────────────
│ Scram                            =     NotScrammed
│ PrimaryPump.PumpSwitch           =     Off
│ PrimaryPump.PumpSwitch' = On     ⇒     Scram' = Scrammed
└───────────────────────────────────────────────────────────
```

Primary Pump Power Turned from On to Off

Informal Description

- The sensor measures the voltage to the motor.
- The reactor is scrammed when the power goes from on to off.
- This pump should only be turned off with the reactor completely stopped.

Z Formal Specification

The schema representing this scram condition is:

```
┌─── CheckPrimaryPumpPowerTurnsOff ───────────────────────
│ ΔReactor
├──────────────
│ Scram                          =     NotScrammed
│ PrimaryPump.Voltage            >     0
│ PrimaryPump.Voltage' = 0       ⇒     Scram' = Scrammed
└─────────────────────────────────────────────────────────
```

Pump On with Header Down

Informal Description

- The reactor is scrammed if the flow header is down and the primary pump is turned on.

Formal Z Specification

The schema representing this scram condition is:

```
┌─── CheckPumpAndHeader ──────────────────────────────────
│ ΔReactor
├──────────────
│ Scram                          =     NotScrammed
│ PrimaryPump.PumpSwitch         =     On
│ HeaderPosition = Down          ⇒     Scram' = Scrammed
└─────────────────────────────────────────────────────────
```

Pressure in the Air Line to the Header

Informal Description

- The reactor is scrammed if, during forced convection mode operation, the pressure in the air line that is used to raise the flow header goes above 2 psi.
- The sensor breaks a circuit if the pressure goes above 2 psi.
- This scram is used to make sure that the head is held in position only by the water flow caused by the primary pump.
- If the primary pump stops and the reactor has not been scrammed by the primary pump off scram, the water flow will reduce and the header will fall due to gravity, causing this sensor to scram the reactor.

Z Formal Specification

The pressure in the air line to the header is sensed by a device that is only able to inform if the pressure is above 2 psi or below 2 psi. Therefore, the sensor can be modeled by a signal that can either be *Hi* or *Low*. The signal corresponding to this sensor, *AirLinePressureAbove2psi?*, is described in the *Inputs* schema. Using this modeling, the schema representing this scram condition is:

```
┌──── CheckPressureInAirLineToHeader ──────────────────
│ ΔReactor
│ Inputs
├──────────────
│ Scram                               =     NotScrammed
│ AirLinePressureAbove2psi? = Hi      ⇒     Scram' = Scrammed
└──────────────────────────────────────────────────────
```

Truck Door Open

Informal Description

- The sensor is a mechanical switch.
- The reactor is scrammed if this door is opened.
- The door gives access to the reactor room.
- The door is used to remove old fuel from the reactor pool.

Z Formal Specification

The schema representing this scram condition is:

```
┌──── CheckTruckDoor ──────────────────────────────────
│ ΔReactor
│ Inputs
├──────────────
│ Scram                    =     NotScrammed
│ TruckDoor? = Open        ⇒     Scram' = Scrammed
└──────────────────────────────────────────────────────
```

Escape Hatch Open

Informal Description

- The sensor is a mechanical switch.
- If the escape hatch is open, the reactor is scrammed.

Z Formal Specification

The *Door* type defined previously will be used to describe the *EscapeHatch?*, defined in the *Inputs* schema. The schema describing this scram condition is:

```
── CheckEscapeHatch ──────────────────────────────────
│ ΔReactor
│ Inputs
├───────────────
│ Scram                    =      NotScrammed
│ EscapeHatch? = Open      ⇒      Scram' = Scrammed
└─────────────────────────────────────────────────────
```

Manual Switch at Room Door Pressed

Informal Description

- The sensor is a mechanical switch.
- The reactor is scrammed if this switch is pressed.

Z Formal Specification

The schema describing this scram condition is:

```
── CheckManualSwitchAtRoomDoor ───────────────────────
│ ΔReactor
│ Inputs
├───────────────
│ Scram                          =      NotScrammed
│ SwitchatRoomDoor? = Pressed    ⇒      Scram' = Scrammed
└─────────────────────────────────────────────────────
```

Manual Switch at Back Door Pressed

Informal Description

- The sensor is a mechanical switch.
- The reactor is scrammed if this switch is pressed.

Z Formal Specification

Using the *Button* construction defined previously, the schema describing this scram condition is:

```
── CheckManualSwitchAtBackDoor ───────────────────────
│ ΔReactor
│ Inputs
├───────────────
│ Scram                          =      NotScrammed
│ SwitchatBackDoor = Pressed     ⇒      Scram' = Scrammed
└─────────────────────────────────────────────────────
```

Evacuation Alarm On

Informal Description

- There are four mechanical switches and alarms.
- The reactor is scrammed if any of this switches is pressed.

Z Formal Specification

To represent this scram condition, all the alarms and mechanical switches will be treated as being a single type of entity, a *EvacuationAlarm* that can be either *On* or *Off*. Using this abstraction, the schema describing this scram condition is:

```
── CheckEvacuationAlarm ──────────────────────────────
 ΔReactor
 Inputs
──────────────
 Scram                   =    NotScrammed
 EvacuationAlarm1? = On  ⇒    Scram' = Scrammed
 EvacuationAlarm2? = On  ⇒    Scram' = Scrammed
 EvacuationAlarm3? = On  ⇒    Scram' = Scrammed
 EvacuationAlarm4? = On  ⇒    Scram' = Scrammed
──────────────────────────────────────────────────────
```

Pool Water Temperature Too High

Informal Description

- The reactor is scrammed if temperature goes above 108 °F.
- The goal is to keep the water at 75 °F.
- Pool temperature rarely exceeds 95 °F.

Z Formal Specification

To represent this scram condition, the temperature will be represented by a natural number *PoolTemperature*?, corresponding to the pool water temperature in degrees Farenheits. This sensor signal is described in the *Inputs* schema. Using this representation, the schema describing this scram condition is:

```
┌─── CheckPoolTemperature ──────────────────────────
│ ΔReactor
│ Inputs
├──────────────
│ Scram                     =      NotScrammed
│ PoolTemperature?> 108     ⇒      Scram' = Scrammed
└───────────────────────────────────────────────────
```

Reactor Period Too Short

Informal Description

- The reactor is scrammed if the reactor period drops below 3.3 seconds.

Z Formal Specification

The period is represented in tenth of seconds by the natural value *ReactorPeriod?*. This signal is described in the *Inputs* schema. The schema describing his scram condition is:

```
┌─── CheckReactorPeriod ────────────────────────────
│ ΔReactor
│ Inputs
├──────────────
│ Scram                     =      NotScrammed
│ ReactorPeriod? < 33       ⇒      Scram' = Scrammed
└───────────────────────────────────────────────────
```

Water Flow Through the Core too Low

Informal Description

- The sensor is composed of one orifice and associated pressure lines that measure the differential pressure.
- The reactor is scrammed if the flow across the core drops bellow 960 gal/min, with the reactor in the forced convection mode.

Z Formal Specification

The water flow through the core can be represented by the real number *Flow?* that reflects the number of gallons per minute that is passing through the core. This sensor signal is described in the *Inputs* schema. Using this representation for the water flow, the schema describing this scram condition is:

CheckWaterFlowThroughTheCore

$$\begin{array}{lll}
\Delta Reactor & & \\
Inputs & & \\
\hline
Scram & = & NotScrammed \\
Flow? < 960 & \Rightarrow & Scram' = Scrammed
\end{array}$$

Key Switch Off

Informal Description

- There is a lock in the panel.
- The lock is a mechanical switch,
- The reactor is scrammed if this key is moved to the Off position.
- The reactor can' t be started without this key on the On position.

Z Formal Specification

The schema describing this scram condition is:

CheckKeySwitch

$$\begin{array}{lll}
\Delta Reactor & & \\
Inputs & & \\
\hline
Scram & = & NotScrammed \\
KeySwitch? = Off & \Rightarrow & Scram' = Scrammed
\end{array}$$

Manual Scram Button Pressed

Informal Description

- The sensor for this scram is a hard contact mechanical switch.
- This is an emergency button to scram the reactor located on the operator console.
- The manual scram button does not comes back after pressed.
- The operator has a reset button to restore this button to the previous position.
- The reactor is scrammed if this button is pressed.

Z Formal Specification

The schema describing this scram condition is:

CheckManualScram

$$\begin{array}{|l}
\Delta Reactor \\
Inputs \\
\hline
Scram = NotScrammed \\
ManualScram? = Pressed \Rightarrow Scram' = Scrammed \\
\hline
\end{array}$$

Reactor Already Scrammed

Informal Description

- If the reactor has been scrammed, it stays scrammed.

Z Formal Specification

All of the previous schemas have as a pre-condition that the status of the reactor is *NotScrammed*. The schema defining what should happen if the reactor is on the *Scrammed* condition is:

CheckReactorAlreadyScrammed

$$\begin{array}{|l}
\Delta Reactor \\
\Delta Inputs \\
\hline
((Scram = Scrammed) \wedge (ResetButton? = NotPressed)) \Rightarrow Scram' = Scrammed \\
((Scram = NotScrammed) \vee (ResetButton? = Pressed)) \Leftrightarrow Scram' = NotScrammed \\
\hline
\end{array}$$

Global Scram Condition

The global scram condition is simply composed by stating that the reactor is scrammed if any of the individual scram conditions is met. The schema describing the global scram condition is:

```
┌─── GlobalScramCondition ──────────────────────────
│ CheckPowerRange
│ CheckPoolWaterLevel
│ CheckBridgeRadiationLevel
│ CheckFaceRadiation
│ CheckSwitchPrimaryPumpTurnsOn
│ CheckPrimaryPumpPowerTurnsOff
│ CheckPumpAndHeader
│ CheckPressureInAirLineToHeader
│ CheckTruckDoor
│ CheckEscapeHatch
│ CheckManualSwitchAtRoomDoor
│ CheckManualSwitchAtBackDoor
│ CheckEvacuationAlarm
│ CheckPoolTemperature
│ CheckReactorPeriod
│ CheckWaterFlowThroughTheCore
│ CheckKeySwitch
│ CheckManualScram
│ CheckReactorAlreadyScrammed
└───────────────────────────────────────────────────
```

4 Alarms

Informal Description

- There are several alarm types.
- Each alarm is indicated individually.
- Sensor signals used by the alarm system are described in the *Inputs* schema.

Z Formal Specification

The schema describing the different alarm types is:

```
┌─── AlarmSystem ───────────────────────────────────
│ Alarms  ==     {Scram, ServoLost, ArgonHi, CoreGamma,
│                Spare, AirMonitor, HeatXDoor, DeminDoor,
│                CoreDiffTemp, ConductHigh, SecondaryPump,
│                ThimbleTemperature}
│ AlarmStates:   {Raised, Lowered}
└───────────────────────────────────────────────────
```

Scram Alarm

Informal Description

- A scram alarm is generated if the reactor is in a scram condition.

Z Formal Specification

The schema describing this alarm is:

$$
\begin{array}{|l}
\textit{ScramAlarm} \\
\hline
\Delta \textit{AlarmSystem} \\
\Delta \textit{Reactor} \\
\hline
\textit{Scram} = \textit{Scrammed} \Leftrightarrow \textit{AlarmConditions}'(\textit{Scram}) = \textit{Raised} \\
\hline
\end{array}
$$

Servo Control Lost

Informal Description

- An alarm is generated if automatic control over the control rod is lost.

Z Formal Specification

The schema describing this alarm is:

$$
\begin{array}{|l}
\textit{ServoLostAlarm} \\
\hline
\Delta \textit{AlarmSystem} \\
\Delta \textit{Reactor} \\
\hline
\textit{if Reactor.RegulatingRod.AutoControl} = \textit{Hi} \wedge \textit{Reactor.RegulatingRod.AutoControl}' = \textit{Low} \\
\qquad \textit{then AlarmConditions}'(\textit{ServoLost}) = \textit{Raised} \\
\qquad \textit{else AlarmConditions}'(\textit{ServoLost}) = \textit{Lowered} \\
\hline
\end{array}
$$

Argon Radiation High

Informal Description

- An alarm is generated if argon radiation levels are high.

Z Formal Specification

The following schema describes this alarm:

```
── ArgonRadiationHighAlarm ──────────────────────────
 ΔAlarmSystem
 Inputs
─────────────
 if ArgonRadiationIndicator? = Hi
         then AlarmConditions'(ArgonHi) = Raised
         else AlarmConditions'(ArgonHi) = Lowered
─────────────────────────────────────────────────────
```

Core Gamma Radiation High

Informal Description

- An alarm is generated if core gamma radiation is high.

Z Formal Specification

The schema describing this alarm is:

```
── CoreGammaAlarm ───────────────────────────────────
 ΔAlarmSystem
 Inputs
─────────────
 if CoreGammaAlarm? = Hi
         then AlarmConditions'(CoreGamma)=Raised
         else AlarmConditions'(CoreGamma)=Lowered
─────────────────────────────────────────────────────
```

Spare Alarm (Not used)

Informal Description

- An alarm is generated if there is a high signal on a spare indicator.

Z Formal Specification

The schema describing this alarm is:

```
── SpareAlarm ───────────────────────────────────────
 ΔAlarmSystem
 Inputs
─────────────
 if SpareSignal? = High
         then AlarmConditions'(Spare)=Raised
         else AlarmConditions'(Spare)=Lowered
─────────────────────────────────────────────────────
```

Air Monitor Indicates High Level

Informal Description

- An alarm is generated if the air monitor indicates a high level of radiation.

Z Formal Specification

The schema describing this alarm is:

```
── AirMonitorAlarm ──────────────────────────────
│ ΔAlarmSystem
│ ΔReactor
│ Inputs
├──────────────
│ if AirMonitorSignal? = High
│           then AlarmConditions'(AirMonitor)=Raised
│           else AlarmConditions'(AirMonitor)=Lowered
└─────────────────────────────────────────────────
```

Heat Exchanger Door Open

Informal Description

- An alarm is generated if the heat exchanger door is open.

Z Formal Specification

The schema describing this alarm is:

```
── HeatExchangerAlarm ───────────────────────────
│ ΔAlarmSystem
│ Inputs
├──────────────
│ if HeatExchangerDoor? = Open
│           then AlarmConditions'(HeatXDoor)=Raised
│           else AlarmConditions'(HeatXDoor)=Lowered
└─────────────────────────────────────────────────
```

Demineralizer Room Door Open

Informal Description

- An alarm is generated if the demineralizer room door is open.

Z Formal Specification

The schema describing this alarm is:

```
── DemineralizerRoomAlarm ──────────────────────────
ΔAlarmSystem
Inputs
──────────
if DemireralizerRoomDoor? = Open
        then AlarmConditions'(DeminDoor)=Raised
        else AlarmConditions'(DeminDoor)=Lowered
────────────────────────────────────────────────────
```

Core Differential Temperature Too High

Informal Description

- An alarm is generated if the core differential temperature is too high.

Z Formal Specification

The schema describing this alarm is:

```
── CoreDiffTempAlarm ───────────────────────────────
ΔAlarmSystem
Inputs
──────────
if CoreDiffTempIndic?≥DiffTempAlarmLevel?
        then AlarmConditions'(CoreDiffTemp)=Raised
        else AlarmConditions'(CoreDiffTemp)=Lowered
────────────────────────────────────────────────────
```

High Water Conductivity

Informal Description

- An alarm is generated if the water conductivity is too high.

Z Formal Specification

The schema describing this alarm is:

```
── ConductHighAlarm ────────────────────────────────
ΔAlarmSystem
Inputs
──────────
if WaterCondIndic?≤ WaterCondAlarmLevel?
        then AlarmConditions'(ConductHigh)=Raised
        else AlarmConditions'(ConductHigh)=Lowered
────────────────────────────────────────────────────
```

Secondary Pump Off With Reactor In High Power Mode

Informal Description

- An alarm is generated if the secondary pump is off and the reactor is in high power mode.

Z Formal Specification

The schema describing this alarm is:

```
┌── SecondaryPumpAlarm ──────────────────────────────
│ ΔAlarmSystem
│ Reactor
├──────────
│ if (SecondaryPump.PumpSwitch = Off ∧ PowerSelection = HighPower)
│         then AlarmConditions'(SecondaryPump)=Raised
│         else AlarmConditions'(SecondaryPump)=Lowered
└────────────────────────────────────────────────────
```

Hot Thimble Temperature

Informal Description

- An alarm is generated if the thimble temperature is too high.

Informal Description

The schema describing this alarm is:

```
┌── ThimbleTemperatureAlarm ─────────────────────────
│ ΔAlarmSystem
│ Inputs
├──────────
│ if ThimbleTemperature? ≥ ThimbleTempAlarmLimit?
│         then AlarmConditions'(ThimbleTemperature)=Raised
│         else AlarmConditions'(ThimbleTemperature)=Lowered
└────────────────────────────────────────────────────
```

Global Alarm Conditions

Informal Description

- All alarm checks are performed concurrently.

Z Formal Specification

The schema describing the global alarm conditions is:

```
┌─── Alarms ──────────────────────────
│ AlarmSystem
│ ScramAlarm
│ ServoLostAlarm
│ ArgonRadiationHighAlarm
│ CoreGammaAlarm
│ SpareAlarm
│ AirMonitorAlarm
│ HeatExchangerAlarm
│ DemineralizerRoomAlarm
│ CoreDiffTempAlarm
│ ConductHighAlarm
│ SecondaryPumpAlarm
│ ThimbleTemperatureAlarm
└─────────────────────────────────────
```

The schema used to set initial conditions for the alarms is:

```
┌─── InitAlarmSystem' ────────────────
│ AlarmSystem'
├──────────
│ ∀ a : Alarms | AlarmConditions'(a)   =   Lowered
└─────────────────────────────────────
```

5 Startup

Informal Description

- There are two different processes for start-up.
- If the reactor is to operate in high-power mode, the header has to be up and the primary pump is to be on.
- If the reactor is to operate in low-power mode, there is no need for the header to be up and the primary pump can be off.

Z Formal Specification

Progress through the start-up process will be indicated by step numbers. A step number of -1 will indicate that the start-up procedure is to be stopped. The schema that represents the step number is:

```
┌─── Step ────────────────────────────
│ StepNum   :   ℤ
└─────────────────────────────────────
```

The schema that sets the initial value for the step number is:

```
┌── StepInit ─────────────────────────────────
│ ΔStep
├──────────────
│ StepNum'        =        0
└─────────────────────────────────────────────
```

The schema that indicates the output signals used to call the operator and to start the control algorithm is:

```
┌── Outputs ──────────────────────────────────
│ CallOperator!              :        Signal
│ StartControlAlgorithm!     :        Signal
└─────────────────────────────────────────────
```

The schema that sets the initial value for the outputs is:

```
┌── OutputsInit ──────────────────────────────
│ ΔOutputs
├──────────────
│ CallOperator!'             =        Low
│ StartControlAlgorithm!'    =        Low
└─────────────────────────────────────────────
```

Step 1

Informal Description

- Reset reactor scram.

Z Formal Specification

The schema describing this step is:

```
┌── StartUpStep1 ─────────────────────────────
│ ΔReactor
│ ΔScram
│ ΔStep
├──────────────
│ StepNum         =        0
│ Scram'          =        NotScrammed
│ StepNum'        =        1
└─────────────────────────────────────────────
```

Step 2

Informal Description

- If the reactor is to operate in high-power mode, admit air to header until it raises to the grid plate.

- If the reactor is to operate in low-power mode, go to step 9.

Z Formal Specification

The schema describing this step is:

```
┌── StartUpStep2 ─────────────────────────────────────────────
│ ΔReactor
│ ΔScram
│ ΔStep
├──────────
│ StepNum          =         1
│ if PowerSelection=         HighPower
│            then HeaderValve'= CompressedAirToLine ∧ StepNum' = 2
│            else StepNum' = 8
└─────────────────────────────────────────────────────────────
```

Step 3

Informal Description

- Verify that a scram was generated; if not, stop the procedure and call the senior operator.

Z Formal Specification

An output signal *CallOperator*! is used to call the operator. Using this signal, the schema describing this step is:

```
┌── StartUpStep3 ──────────────────────────────────────
│ ΔReactor
│ ΔScram
│ ΔStep
│ ΔOutputs
├──────────
│ StepNum                                =        2
│ If Scram                               =        NotScrammed
│            then CallOperator!' = High  ∧        StepNum' = -1
│            else Scram' = NotScrammed   ∧        StepNum' = 3
└──────────────────────────────────────────────────────
```

Step 4

Informal Description

- Start the primary and secondary pumps.

Z Formal Specification

The schema describing this step is:

```
┌── StartUpStep4 ───────────────────────────────────────────
│ ΔReactor
│ ΔScram
│ ΔStep
├──────────────
│ StepNum                               =      3
│ Reactor.PrimaryPump.PumpSwitch'       =      On
│ Reactor.SecondaryPump.PumpSwitch'     =      On
│ StepNum'                              =      4
└───────────────────────────────────────────────────────────
```

Step 5

Informal Description

- Turn the line to the header valve so that the line is conected to the atmosphere.

Z Formal Specification

The schema describing this step is:

```
┌── StartUpStep5 ───────────────────────────────
│ ΔReactor
│ ΔScram
│ ΔStep
├──────────────
│ StepNum                 =      4
│ Reactor.HeaderValve'    =      LineToAtmosphere
│ StepNum'                =      5
└───────────────────────────────────────────────
```

Step 6

Informal Description

- Close valve on the air line to the header.

Z Formal Specification

The schema describing this step is:

*StartUpStep*6

$\Delta Reactor$
$\Delta Scram$
$\Delta Step$

StepNum	=	5
Reactor.HeaderValve'	=	*LineClosed*
StepNum'	=	6

Step 7

Informal Description

- Reset reactor scram.

Z Formal Specification

The schema describing this step is:

*StartUpStep*7

$\Delta Reactor$
$\Delta Scram$
$\Delta Step$

StepNum	=	6
Scram'	=	*NotScrammed*
StepNum'	=	7

Step 8

Informal Description

- Check that the header remains up.

Z Formal Specification

*StartUpStep*8

$\Delta Reactor$
$\Delta Scram$
$\Delta Step$
$\Delta Outputs$
$Inputs$

$StepNum = 7$
$\text{if } HeaderDown? = Hi$
$\quad \text{then } CallOperator!' = High \wedge StepNum' = -1$
$\quad \text{else } StepNum' = 8$

Step 9

Informal Description

- Bring all the shim rod drivers to the lowest position.

Z Formal Specification

The schema describing this step is:

*StartUpStep*9

$\Delta Reactor$
$\Delta Scram$
$\Delta Step$

$StepNum = 8$
$\forall a : a \in ShimRods \bullet a.Position' = a.MinPosition$
$StepNum' = 9$

Step 10

Informal Description

- Verify that the seated lamps are on for each individual rod; if not, stop the procedure and call the senior operator.

Z Formal Specification

The schema describing this step is:

```
┌── StartUpStep10 ─────────────────────────────────────────
│ ΔReactor
│ ΔScram
│ ΔStep
│ ΔOutputs
│ Inputs
├──────────────────
│ StepNum                                   =     9
│ if      (ShimRod1Seated? = Hi             ∧
│         ShimRod2Seated? = Hi              ∧
│         ShimRod3Seated? = Hi)
│         then StepNum'                     =     10
│         else CallOperator!' = High        ∧     Step'=-1
└──────────────────────────────────────────────────────────
```

Step 11

Informal Description

- Verify that the magnetically engage lamp corresponding to each of them is on; if not, stop the procedure and call the senior operator.

Z Formal Specification

The schema describing this step is:

```
┌── StartUpStep11 ─────────────────────────────────────────
│ ΔReactor
│ ΔScram
│ ΔStep
│ ΔOutputs
│ Inputs
├──────────────────
│ StepNum                        =     10
│ if      (TouchingDriver1? = Hi  ∧
│         TouchingDriver2? = Hi   ∧
│         TouchingDriver3? = Hi)
│         then StepNum'           =     11
│         else CallOperator!' = Hi ∧    StepNum'=-1
└──────────────────────────────────────────────────────────
```

Step 12

Informal Description

- Turn on the magnetic currents on the shim rod drivers.

Z Formal Specification

The schema describing this step is:

$$\begin{array}{|lcl}
\hline
\multicolumn{3}{|l}{\quad \mathit{StartUpStep}12} \\
\Delta \mathit{Reactor} & & \\
\Delta \mathit{Scram} & & \\
\Delta \mathit{Step} & & \\
\hline
\mathit{StepNum} & = & 11 \\
\forall a{:}a \in \mathit{ShimRods} \bullet a.\mathit{MagCurrent}' & = & a.\mathit{MinHoldingCurret}+1 \\
\mathit{StepNum}' & = & 12 \\
\hline
\end{array}$$

Step 13

Informal Description

- Raise the shim rod drivers.

Z Formal Specification

The schema describing this step is:

$$\begin{array}{|lcl}
\hline
\multicolumn{3}{|l}{\quad \mathit{StartUpStep}13} \\
\Delta \mathit{Reactor} & & \\
\Delta \mathit{Scram} & & \\
\Delta \mathit{Step} & & \\
\hline
\mathit{StepNum} & = & 12 \\
\forall a{:}a \in \mathit{ShimRods} \bullet a.\mathit{Position}' & = & 10 \\
\mathit{StepNum}' & = & 13 \\
\hline
\end{array}$$

Step 14

Informal Description

- Verify that the seated position indicator lamp and the rod down lamp indicator go off; if not, stop the procedure and call the senior operator.

Z Formal Specification

The schema describing this step is:

```
── StartUpStep14 ──────────────────────────────────────
│ ΔReactor
│ ΔScram
│ ΔStep
│ ΔOutputs
│ Inputs
├──────────────
│ StepNum                              =        13
│ if        (ShimRod1Seated? = Low              ∧
│           ShimRod2Seated? = Low               ∧
│           ShimRod3Seated? = Low )
│           then StepNum’                       =        14
│           else CallOperator!’ = High          ∧        StepNum’ = -1
└──────────────────────────────────────────────────────
```

Step 15

Informal Description

- Request power level from operator and start control algorithm for reactor.

Z Formal Specification

The schema describing this step is:

```
── StartUpStep15 ──────────────────────────────────────
│ ΔReactor
│ ΔScram
│ ΔStep
│ ΔOutputs
├──────────────
│ StepNum                    =        14
│ StartControlAlgorithm!’    =        High
│ StepNum’                   =        15
└──────────────────────────────────────────────────────
```

Start-Up Procedure

Informal Description

The start-up procedure can be described as being the execution of one of the steps of the start-up.

Z Formal Specification

The description of the start-up procedure is:

$StartUpProcedure \quad \widehat{=} \quad StartUpStep1 \quad \vee$

$$
\begin{array}{l}
StartUpStep2 \lor \\
StartUpStep3 \lor \\
StartUpStep4 \lor \\
StartUpStep5 \lor \\
StartUpStep6 \lor \\
StartUpStep7 \lor \\
StartUpStep8 \lor \\
StartUpStep9 \lor \\
StartUpStep10 \lor \\
StartUpStep11 \lor \\
StartUpStep12 \lor \\
StartUpStep13 \lor \\
StartUpStep14 \lor \\
StartUpStep15
\end{array}
$$

6 Reactor Specification

The complete reactor specification can be described by:

$$
\begin{array}{ll}
ReactorSystem \mathrel{\widehat=} & (Reactor \land GlobalScramCondition \land Alarms) \land \\
& (StartUp \lor Operate)
\end{array}
$$

Appendix D
Questionnaire for Authors

1 Non-subjective

What language/tool/platform did you use for the specification?

1.1 Language Issues

Is training for the language available?

How much of the language does it cover?

Does the language structuring support different levels of abstraction?

Does the language have formal syntax and semantics?

Is there documentation for this language?

Are there published examples of specifications for real systems written in this language?

Are there published examples of designs derived from a specification in this language?

Are there published examples of implementations from specifications in this language?

Are there published examples of verification based on a specification in this notation?

Can the following be represented in the language:

- Integers?
- Real numbers?
- Constants?
- Timing?

Is it possible to document nonfunctional requirements or design decisions in the specification notation?

is it possible in this notation to specify features that are not implementable in a language such as C or C++?

Does your specification contain any features which are not implementable in C or C++?

Is the notation built for readability?

- Infinite-length identifier names?
- Meaningful keywords?
- Common mathematical notation?
- Accomodation of tabs for readability?
- Allowance of upper and lower case in identifiers?
- Allowance of underscores in identifiers?

1.2 Toolset Issues

Is training for the toolset available?

How much of the toolset does it cover?

Is technical support available to answer questions about the toolset?

How large is the toolset (in computer memory)?

Is it possible to print a hard copy from the toolset?

Does the toolset support multiple users?

Does the editor allow the document be viewed at different levels of abstraction?

What common file format(s) is supported by the toolset?

Was the file format compatible with a natural language text editor?

Did the toolset support the notion of separate compilation?

Does the toolset have its own version control system?

Is it possible to use external version control with this toolset?

Is there documentation for the toolset?

Are there tutorials for this toolset?

Can code be automatically generated from a specification in this notation?

Can test cases be automatically generated from a specification in this notation?

Did the toolset tolerate incompleteness during development of the specification?

Does the toolset support regular expression matching?

Does the toolset provide static analysis of the specification?

- Pre-conditions?
- Post-conditions?
- Invarients?
- Data flow?
- Completeness?
- Consistency?

Is further validation/verification provided, such as:

- Model annimation?
- Theorem proving?
- Model checking?

2 Subjective

How easy was it for you to understand the specification that you'd written?

- ❑ Very difficult
- ❑ Fairly difficult
- ❑ Fairly easy
- ❑ Very easy

How well does the formal method facilitate communication about the system?

- ❑ Hinders communication
- ❑ Allows communication
- ❑ Improves communication

Evaluate the size and complexity of the language.

- ❑ Too small and simple
- ❑ Appropriately small and simple
- ❑ Appropriately big and complex
- ❑ Too big and complex

How difficult is it to learn this notation?

- ❑ Very difficult
- ❑ Fairly difficult
- ❑ Fairly easy
- ❑ Very easy

How difficult is it to learn the modeling skills needed to use this notation?

- ❑ Very difficult
- ❑ Fairly difficult
- ❑ Fairly easy
- ❑ Very easy

Were the documentation and examples of the language useful and sufficient?

- ❑ They did not exist/Were not useful or sufficient
- ❑ Not very useful and sufficient
- ❑ Fairly useful and sufficient
- ❑ Very useful and sufficient

Is it convenient and natural to document nonfunctional requirements or design decisions in the specification notation?

- ❑ Not possible
- ❑ Very inconvenient and unnatural
- ❑ Fairly inconvenient and unnatural
- ❑ Fairly convenient and natural

- ❑ Very convenient and natural

Does the notation facilitate the identification of key parts of the system, thus aiding in the design phase?

- ❑ No aid
- ❑ Little aid
- ❑ Some aid
- ❑ A lot of aid

Does the notation facilitate the identification of interactions or dependencies between parts of the system, thus aiding in the design phase?

- ❑ No aid
- ❑ Little aid
- ❑ Some aid
- ❑ A lot of aid

Can a specification in this notation provide sufficient, but not too much, detail for implementation?

- ❑ Too little detail
- ❑ A little detail
- ❑ A lot of detail
- ❑ Too much detail

How well was structuring and information hiding supported by the language and toolset?

- ❑ Poorly supported
- ❑ Fairly well supported
- ❑ Well supported
- ❑ Excellently supported

Were the documentation and tutorials of the toolset useful and sufficient?

- ❑ They did not exist/Were not useful or sufficient
- ❑ Not very useful and sufficient
- ❑ Fairly useful and sufficient
- ❑ Very useful and sufficient

If technical support for the toolset exists, how high is the quality?

- ❑ Does not exist/Was not used
- ❑ Poor
- ❑ Fair
- ❑ Good
- ❑ Excellent

Are the time and space requirements of the toolset reasonable?

❑ Unreasonably large or slow
❑ Tollerably large or slow
❑ Of reasonable size and speed
❑ Small and fast

How easy is it to print a hard copy from the toolset?

❑ Could not print
❑ Very difficult
❑ Fairly difficult
❑ Fairly easy
❑ Very easy

Was the file format(s) easy to manipulate?

❑ Very difficult
❑ Fairly difficult
❑ Fairly easy
❑ Very easy

Was it easy to create, manipulate, and organize files?

❑ Very difficult
❑ Fairly difficult
❑ Fairly easy
❑ Very easy

How user-friendly was the interface of the toolset?

❑ Very unfriendly
❑ Fairly unfriendly
❑ Fairly friendly
❑ Very friendly

How easy was it to make modifications (small and large) to the system?

❑ Very difficult
❑ Fairly difficult
❑ Fairly easy
❑ Very easy

How easy was it for you to navigate the specification (on-line) that you'd written?

❑ Very difficult
❑ Fairly difficult
❑ Fairly easy
❑ Very easy

How easy was it for you to search the specification(online) that you'd writ-

ten?

- ❑ Very difficult
- ❑ Fairly difficult
- ❑ Fairly easy
- ❑ Very easy

How useful is the static analysis provided by the toolset?

- ❑ Does not exist/Not useful
- ❑ Not very useful
- ❑ Fairly useful
- ❑ Very useful

Appendix E
Questionnaire for Computer Scientists

1 Directions

This is part of an evaluation of four formal specification languages. Please take your time learning the languages, but do not expect to become an expert. We are interested in your thoughts about the languages even though you will not have much experience with them. You should spend about an hour to an hour and a half with each of the four languages. If at any point you have a question about one of the languages or about part of this survey, do not hesitate to ask. The multiple choice format is to make answering easier for you as well as to standardize the answers for us, but if there is an idea that is not captured in the choices, you are encouraged to comment in the margins. Remember that this is a test of the language, not you. Do the sections of the packet in the following order:

1. Complete "Background Questions"
2. Read "Application Summary"
3. For each language:
   - Read the summary of the language
   - Complete "Structure and Navigation"
   - Complete "Implementation"
   - Complete "Maintenance"
   - Complete "General Questions"

2 Background Questions

1. How much coursework have you received in computer science? Please indicate all that apply. Include any degree in progress.
   ❑ None

- ❑ Bachelors degree in computer science
- ❑ Masters degree in computer science
- ❑ Ph.D. in computer science
- ❑ Course(s) outside of a university
- ❑ Course(s) at a university that did not go toward a degree in computer science

How much work experience do you have in developing software?

- ❑ None
- ❑ A year or less
- ❑ 1-5 years
- ❑ more than 5 years

Indicate your knowledge of the C programming language.

- ❑ Little to none
- ❑ Some
- ❑ Quite a lot
- ❑ Very extensive

How much instruction have you received in formal methods? Include courses in progress.

- ❑ None
- ❑ A segment of one course
- ❑ One entire course
- ❑ Two or more courses

How much experience do you have using formal methods?

- ❑ Little to none
- ❑ Some
- ❑ A lot

Indicate your knowledge of science and engineering fields such as electronics, mechanics, physics, and chemistry.

- ❑ None
- ❑ Only the basics
- ❑ Some knowledge
- ❑ Extensive knowledge

Indicate your knowledge of the equipment and functionality of a nuclear reactor.

- ❑ None
- ❑ Only the basics
- ❑ Familiarity
- ❑ Intimate knowledge

3 Structure and Navigation

3.1 First language

Take some time to examine the first specification. You do not need to memorize, rather to get a feel for the notation and the structure of the document. When you have finished studying it, complete the following exercise. Perform the following steps:

1. Read the question.
2. Start a timer.
3. Find the answer to the question.
4. Stop the timer.
5. Record your answer, the page on which it was found, and the time needed to find it.

Remember that this is intended to measure the ease with which the specification can be navigated, not your abilities, so time each question separately and as accurately as possible.

1. The reactor is scrammed if the pool water level is too low. At what value is this scram signal generated?

answer:________________ page:________________

time:_______________

What is the effect of the gamma radiation level becoming too high?

answer:________________ page:________________

time:_______________

What is the initial state of the primary pump?

answer:________________ page:________________

time:______________

3.2 Second language

Take some time to examine the second specification. You do not need to memorize, rather to get a feel for the notation and the structure of the document. When you have finished studying it, complete the following exercise. Perform the following steps:

1. Read the question.
2. Start a timer.
3. Find the answer to the question.
4. Stop the timer.
5. Record your answer, the page on which it was found, and the time needed to find it.

Remember that this is intended to measure the ease with which the specification can be navigated, not your abilities, so time each question separately and as accurately as possible.

1. What is the effect of the air monitor measuring a high radiation level?
    answer:_______________ page:_______________

time:_______________

What is the initial state of the air line valve?

answer:_______________ page:_______________

time:_______________

The reactor is scrammed if the reactor period is too low. At what value is this scram signal generated?

answer:_______________ page:_______________

time:_______________

3.3 Third language

Take some time to examine the third specification. You do not need to memorize, rather to get a feel for the notation and the structure of the document. When you have finished studying it, complete the following exercise. Perform the following steps:

1. Read the question.

2. Start a timer.

3. Find the answer to the question.

4. Stop the timer.

5. Record your answer, the page on which it was found, and the time needed to find it.

Remember that this is intended to measure the ease with which the specification can be navigated, not your abilities, so time each question separately and as accurately as possible.

1. What is the initial state of the secondary pump?
answer:_______________ page:_______________

time:______________

The reactor is scrammed if the pool water temperature is too high. At what value is this scram signal generated?

answer:_______________ page:_______________

time:_______________

What is the effect of the area radiation level becoming too high?

answer:_______________ page:_______________

time:_______________

3.4 Fourth language

Take some time to examine the first specification. You do not need to memorize, rather to get a feel for the notation and the structure of the document. When you have finished studying it, complete the following exercise. Perform the following steps:

1. Read the question.

2. Start a timer.

3. Find the answer to the question.

4. Stop the timer.

5. Record your answer, the page on which it was found, and the time needed to find it.

Remember that this is intended to measure the ease with which the specification can be navigated, not your abilities, so time each question separately and as accurately as possible.

1. What is the effect of the core temperature becoming too high?
answer:________________ page:________________

time:_______________

The reactor is scrammed if the flow through the core is too low At what value is this scram signal generated?

answer:________________ page:________________

time:_______________

What is the initial state of the header?

answer:________________ page:________________

time:______________

3.5 All languages

How well structured was the specification?

❑ There was no structure.
❑ There was some structure, but it was hard to identify or illogical.
❑ It was fairly well structured.
❑ It was clearly and logically structured.

How much did the structure of the document assist you in finding the information requested in the exercises above?

❑ None, I had to do a linear search.
❑ Some, I could eliminate sections after a quick glance.
❑ A lot, I was able to identify likely locations for the information.
❑ The structure allowed me to find the information immediately.

How did the quantity of text affect your search?

- ❑ There was very little text, so it was easy to scan.
- ❑ The amount of text did not adversely affect my search.
- ❑ The quantity of text slowed me down somewhat.
- ❑ There was so much text I felt that I was looking for a needle in a haystack.

How effective was the structuring of the specification in aiding understanding

- ❑ The structure was illogical, so it didn't aid understanding.
- ❑ The structure helped group things and made it easier to understand.
- ❑ Because of the structure, it was easy to understand.

Evaluate the aid from the specification in identifying key parts of the system.

- ❑ It provided no help. There was no identification of major components.
- ❑ After extensive study of the specification, the key components could be identified.
- ❑ It helped identify key components.
- ❑ The key components were easily identified.

Evaluate the aid from the specification in identifying the interactions or dependencies between parts of the system.

- ❑ It provided no help. There was no identification of the relationships between components.
- ❑ After extensive study of the specification, the interactions could be identified.
- ❑ It helped identify the interactions between components.
- ❑ The interactions were easily identified.

4 Implementation

Envision that you are an implementer assigned to write the code for the module containing the scram conditions. Look carefully at the section(s) of the specification related to the scrams. Develop an implementation scheme. Study the properties of the system as described by the specification.

1. After some thought, can you think of a way to implement this section?
   - ❑ From the specification I can conceive of more than one possible implementation.
   - ❑ I see one possible implementation.
   - ❑ I have some thoughts about the implementation, but see some problems.
   - ❑ I don't see any way to implement this.

Evaluate the level of detail in the specification.

- ❑ There are too many implementation details included and it over constrains the implementer.

- ❑ The concepts expressed are not described precisely enough; the implementer is left with questions about functionality.
- ❑ The functionality is complete and the description is abstract enough that the implementer given the implementation decisions.

Is every possible behavior for this section of the system described by the specification (i.e. is the specification complete)?

- ❑ It is impossible to tell.
- ❑ It's hard to tell, but it's doubtful.
- ❑ It's hard to tell, but it looks like it.
- ❑ Missing cases are evident.
- ❑ A missing case would be easily identified and there aren't any.

Is every feature of the specification notation implementable? Take some time to look through more of the specification than just the section on the scram conditions.

- ❑ Everything is definitely implementable.
- ❑ Everything seems implementable.
- ❑ There are some features which are dubious.
- ❑ There are one or more features which are not implementable.

5 Maintenance

Envision now that this system as already been built and you are assigned to maintain the code, fixing bugs and adding new features. Previously you had no involvement with the project.

1. How useful would this specification be as an introduction to the system?
   - ❑ It is a very good introduction; it is complete, concise, and easy to understand.
   - ❑ It is an average summary document, useful for introduction.
   - ❑ It too hard to understand to be an introduction.

How useful would this specification be as a reference document?

- ❑ It is very useful for reference since it is well organized, complete, and concise.
- ❑ It is average as a reference because some things are hard to find or unclear.
- ❑ It is poor as a reference because it is hard to find things and the information is incomplete and/or unclear.

6 General Questions

1. How well would the specification facilitate communication between people involved in the development of the system?
   - ❑ It would confuse more than help.
   - ❑ It would slow down communication because it is hard to understand.
   - ❑ It would aid communication because it is precise and unambiguous.
   - ❑ It would aid communication because it is easy to understand and unambiguous.

Rate the size of the language (number of features/keywords/constructs).

- ❑ Larger than C
- ❑ About the same as C
- ❑ Smaller than C

How appropriate is the size of the language?

- ❑ Too big
- ❑ About right
- ❑ Too small

Rate the complexity of the language (number of ways to combine constructs).

- ❑ More complex than C
- ❑ About the same complexity as C
- ❑ Less complex than C

How appropriate is the level of complexity of the language?

- ❑ Too complex
- ❑ About right
- ❑ Not complex enough

How confident are you in your current ability to write a specification in this language?

- ❑ I could specify a large, complex system with minimal assistance
- ❑ I could specify a simple system with minimal assistance
- ❑ I would need a lot of help to use this language for any system

Rate the difficulty of learning this language.

- ❑ Impossible
- ❑ More difficult than a programming language
- ❑ The same as a programming language
- ❑ Less than a programming language

Identify the source(s) of difficulty in learning the language. If there are more than one, please number them starting with (1) the largest cause of difficulty. If a source of difficulty is not in the list, please add it in the *margin.*

- ❑ The notation is unlike anything I've seen before
- ❑ The language is very large and complex
- ❑ Keywords or other built-in language elements do not convey their meaning

Identify the feature of the language that makes it easy to learn. If there are more than one, please number them starting with (1) the most helpful feature. If something is not in the list, please add it in the margin.

- ❑ I have worked with similar notations before
- ❑ The language is small and simple
- ❑ Keywords or other built-in language elements effectively convey their meaning