FORMAL VERIFICATION: AN EVALUATION

Matthew Gibble, John C. Knight, Luís G. Nakano, Colleen DeJong

Department of Computer Science, University of Virginia

Computer Science Report No. CS-97-13
May 1997

Table of Contents

1 Introduction

Modern software systems are becoming increasingly large and complex. It is becoming more difficult for software engineers to develop reliable software due to the size and complexity of projects. Software engineers have developed techniques called formal methods to assist developers in producing more reliable software. John Rushby of SRI International describes formal methods:

> *"Formal methods" are the use of mathematical techniques in the design and analysis of computer hardware and software; in particular, formal methods allow properties of a computer system to be predicted from a mathematical model of the system by a process akin to calculation.* [Rus93]

Formal methods provide an unambiguous notation that can be used for requirements engineering or writing the specification of a system. Although the emphasis that the software engineering community has placed on code reuse and determining better and faster methods of implementing systems would suggest most errors occur in the implementation phase of software development, formalists have data to suggest that the majority of errors occurring in existing software arises in the specification phase. By incorporating formal specification notations into the software development process, specifications can be mechanically analyzed to eliminate a vast majority of errors in the specification and therefore a large number of errors in the resulting software.

Despite the many years of praise that formal methods have received from academics, industrial use of formal methods is limited. Because industrial software developers generally use informal languages, most notably English, for their specifications, little or no mechanical analysis is applied to the specification. This is due to the ambiguous nature of informal languages. Each word or phrase does not have a unique and precise meaning, so it is virtually impossible for a computer to check for syntax, completeness, type correctness, etc. By using a formal language that is built upon a mathematical notation, developers can use computers to check for syntax, type correctness and occasionally completeness. In addition to static checking, computers can be utilized for theorem proving or model checking which allow the user to prove putative theorems and system invariants about the specification.

There are two processes that software engineers use to determine the “correctness” of software, validation and verification. Validation and verification are terms that are often misused or misunderstood. Boehm states the difference between validation and verification simply as:

Validation:	“Are we building the right product?”
Verification:	“Are we building the product right?”

Validation is a process where the developer ensures that the software that is being built correctly and accurately meets the needs and desires of the customer. The software should meet the functional requirements of the client. Verification ensures that the software behaves correctly according to the specification of the system. The software should correctly execute the tasks it is designed to accomplish [Pre92]. In general, validation is performed on the specification of the system and verification is performed on the implementation of the system. Mechanical analysis tools like theorem provers and model checkers are validation tools that help convince the customer and system designer that when implemented the system will be the “right product.”

It has been argued by many researchers that formal verification, mechanically analyzing an implementation and determining whether or not it meets its specification, is unrealistically difficult for real systems, because real systems are too large and complex. An approach that has seen some success especially in the realm of hardware verification is the use of theorem provers or model checkers to validate that the specification meets the functional requirements of the client. Since the majority of errors occur in the specification phase and not the implementation phase, elimination of the errors in the specification will ensure the system is acting properly if implemented correctly. It has also been argued that it is easier and more efficient for a programmer to translate a formal specification into an implementation rather than translating an informal specification thus producing fewer errors due to implementation phase mistakes. In addition, a number of specification languages have code generators, tools that can generate an implementation automatically from a specification. If an implementation can be generated automatically and correctly from a specification, then formal specification validation is essentially equivalent to formal verification.

Formal verification by any method is rarely used in industry, even in the safety critical realm where software failure could cause significant losses to human life or capital. Despite the praise that the formal methods community have given theorem provers and model checkers, they are not being embraced by the industrial community. This discrepancy between the formal methods community and industry is the focus of this research. Using a state-of-the-art theorem prover named PVS (Prototype Verification System) and a sample application (a nuclear reactor control system), a study of the usefulness and effectiveness of the PVS system is presented.

The case study contains practical development concerns as they apply to the production of large, complex, multi-authored systems. The underlying hypothesis is that formal specification with mechanical analysis will aid development of more reliable systems. This research documents the practical issues that occurred during an attempt to specify and partially verify a nuclear reactor control system. It is claimed that the experience report that was generated is representative of the experiences of a typical industrial engineer attempting to use these technologies. The background of the practicioner (the first author of this report) is typical of modern industrial engineers.

In the next chapter, a description of the reactor is presented. Chapter 3 discusses some related works including the state of the art of theorem provers and projects that have used theorem provers. Chapter 4 contains an in-depth discussion of the UVAR (The University of Virginia Reactor) specification and verification project. The results of the project with PVS are listed in Chapter 5. The complete PVS Specification for the reactor is presented in Appendix A.

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].

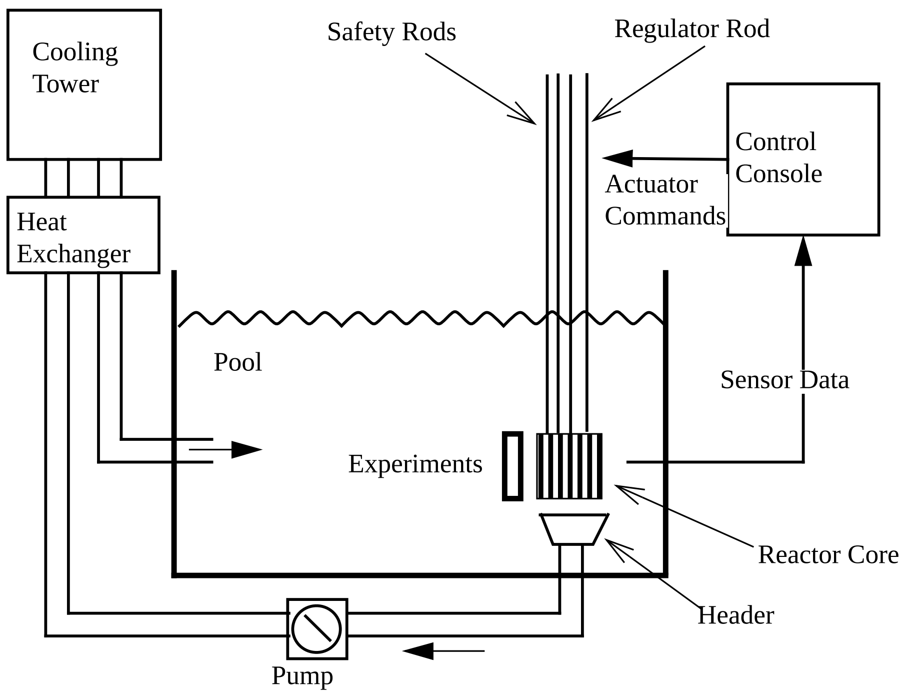


Figure 1: The University of Virginia reactor system.

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

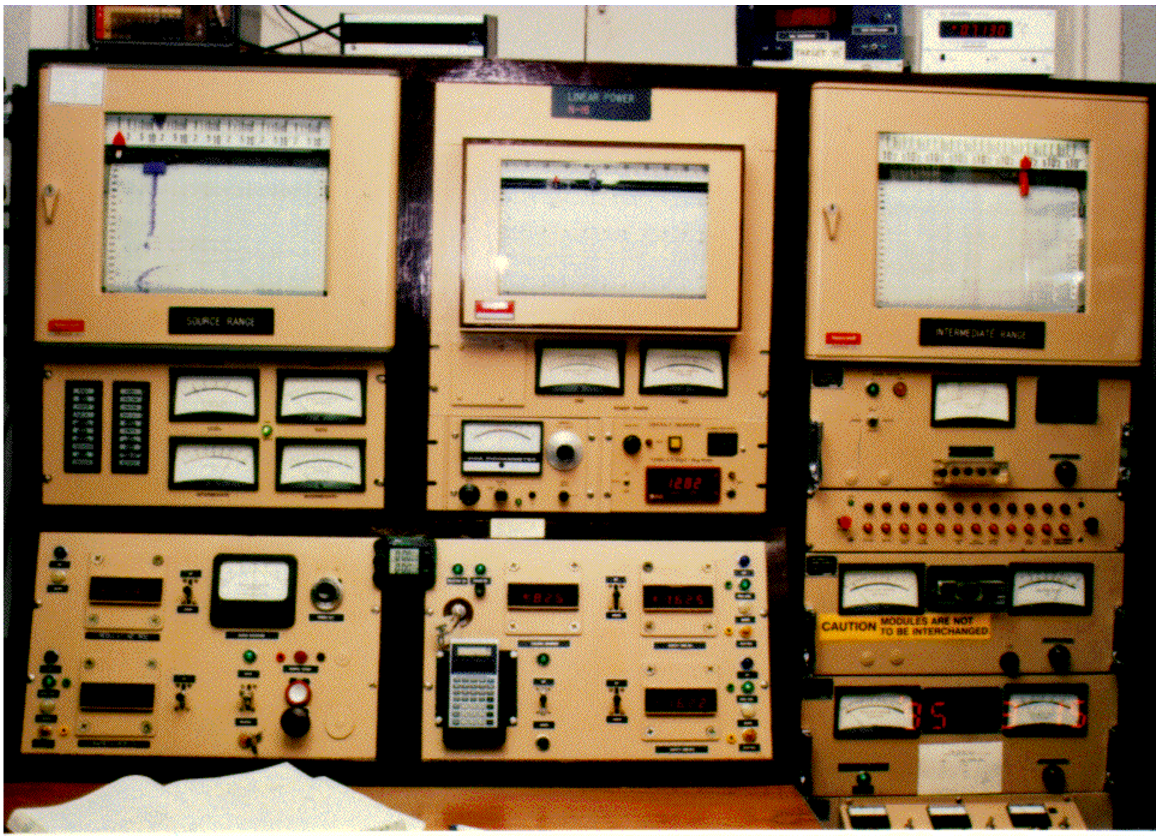


Figure 2: Partial view of the control pannel of the UVAR.

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

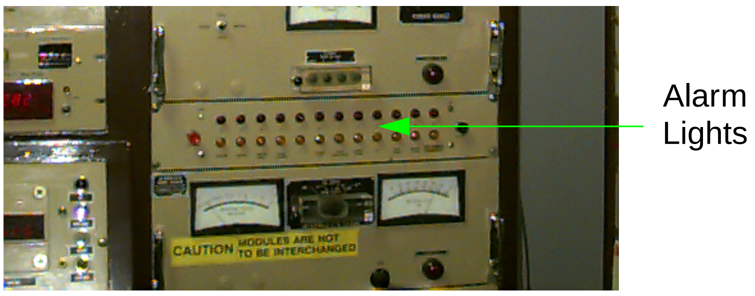


Figure 3: Lateral panel with alarm lights.

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 Related Work

3.1 State of the Art in Theorem Provers

A theorem prover is a tool that can be used to prove putative theorems and system invariants about formal specifications. They are currently being used to validate specifications. There are a number of different theorem provers that are readily available to software engineers with a large number available by anonymous ftp. Many theorem provers are currently being used in academia for research projects while a couple have found some use in industrial projects. Following is a list of theorem provers that represent the current state of the art, that is the theorem provers that are considered powerful and general enough to be used for a wide range of applications. Each theorem prover description also includes a list of users and the projects that the theorem provers are being used for, if known.

HOL

The HOL System is a theorem prover that uses higher-order logic to prove theorems about specifications using the meta language ML. There are currently two versions of HOL. HOL88 is an older version of HOL that is built on top of Lisp. The version of ML that it uses is non-standard. HOL90 is the more recent version that uses StandardML as its meta-language and implementation language. The documentation for HOL90 is not as well-developed at present as the documentation of HOL88. Both versions of HOL are available via anonymous ftp from The University of Cambridge Computer Laboratory [HOLweb].

The following is a list of HOL users and usages, if known. Information found in published papers is cited.

- Åbo Akademi University, Finland
    - A secure tool for interactive refinement of sequential and parallel programs [AAUweb]
- Bell Laboratories
- Laboratory for Applied Logic, Brigham Young University
- Automated Reasoning Group, University of Cambridge Computer Laboratory
    - Asynchronous Transfer Mode Network verification [Cur94]
- Formal Methods Group, Department of Computing Science, University of Glasgow
    - User interface design for mechanized theorem proving [GLAweb]
- Hardware Verification Group, Institute of Computer Design and Fault Tolerance, Department of Computer Science, University of Karlsruhe, Germany
    - VHDL-based Verification [HVGweb]
    - Verification of RISC-Processors [TK94]

Users with unspecified usages were found on the HOL web page at the University of Cambridge Computer Laboratory [HOLweb].

Boyer-Moore Theorem Prover

Also called Nqthm, the Boyer-Moore theorem prover is a Common Lisp mathematical theorem prover that was developed by Robert Boyer and J Strother Moore at Computational Logic Inc. The Boyer-Moore theorem prover is available via anonymous ftp from ftp.cli.com with the most recent release of the program occurring in January, 1994. Specifications and theorems are written in LISP. The logic of Nqthm is based on a first order logic.

> *The logic of Nqthm contains two ‘extension’ principles under which the user can introduce new concepts into the logic with the guarantee of consistency. The Shell Principle allows the user to add axioms introducing ‘new’ inductively defined ‘abstract data types. The Definitional Principle allows the user to define new functions in the logic. The theorem prover is fully automatic in the sense that once a proof attempt has started, the system accepts no advice or directives from the user. The only way the user can interfere with the system is to abort the proof attempt. However, on the other hand, the theorem prover is interactive: the system may gain more proving power through its data base of lemmas, which have already been formulated by the user and proved by the system. Each conjecture, once proved, is converted into some `rules' which influence the prover's action in subsequent proof attempts* [CLIweb].

The following is a list of users and usages of the Boyer-Moore Theorem Prover with citations to the published results of the projects.

> *Since A Computational Logic was published in 1979, Nqthm has been used by several dozen users to check proofs of over 16,000 theorems from many areas of number theory, proof theory, and computer science* [CLIweb].

- Computational Logic Inc.
    - Gate-level verification of a microprocessor design [BH94]
    - Verification of a Gypsy 2.05 code generator [Moo88]
    - Mechanically checked proofs of operating system kernel specifications [Bev88]
    - Verification of a bit-slice ALU [HB89]
    - Describing and verifying synchronous circuits [Rus94]

- TSI, Inc. (Trusted Information Systems, Incorporated)
    - Verification of object code against C source code[SC96]

EVES

EVES is a system that was developed at ORA Canada and an initial version completed in May, 1990 for the Canadian Department of National Defence. [Cra92] EVES uses a language called Verdi that is based on an untyped set theory. “Verdi consists of syntactic forms for expressing specifications (what effect a program is to have), implementations (how a program is to cause an effect), and proofs (justification that a program meets its specification).”[Cra92] Verdi’s syntax is similar to that of the s-expressions of Lisp.

Actually EVES as a system contains 5 parts:

- the language Verdi
- a proof obligation generator
- an automated deduction system, called NEVER
- an interpreter
- a compiler

“The proof obligation generator automatically emits the assertions that must be proven to demonstrate certain important properties, including that code is in consonance with their specifications” [ORAweb]. NEVER is an interactive theorem prover, but it can mechanically perform large proof steps to limit some of the required interactivity. The interpreter allows EVES the flexibility of interpreting executable Verdi code. EVES is available from ORA Canada at no cost.

The following is a list of users and usages of the EVES theorem provers.

- ORA Canada
  - Set theory proofs
  - Proof of a nontrivial security property

Z/EVES is a system that applies a Z front-end to the EVES system. Z is a typed formal specification language that uses set theory as its logic. Z/EVES can be used to analyze Z specifications in a number of ways:

- syntax and type checking
- schema expansion
- precondition calculation
- domain checking
- general theorem proving

Currently Z/EVES does not support the entire Z notation, but when complete Z/EVES will support the full Z notation. [Saa95] The alpha version of Z/EVES is available at no cost from ORA Canada.

Larch

"Larch is a multi-site project exploring methods, languages, and tools for the practical use of formal specifications" [MITweb]. Early work on the project was done at MIT and at Digital Equipment in Palo Alto, California. There are actually a number of different Larch languages which are supported by LP, the Larch Prover, LSL, a checker for the Larch Shared Language, and "LCLint, a C program checker that exploits (and also checks) any accompanying LCL specifications" [MITweb].

> *The Larch family of languages supports a two-tiered, definitional style of specification. Each specification has components written in two languages: one language that is designed for a specific programming language and another language that is independent of any programming language. The former kind are Larch interface languages, and the latter is the Larch Shared Language (LSL).* [MITweb]

LP, the Larch Prover, is an interactive theorem prover for multisorted first-order logic. All of the Larch Tool Set is available via anonymous ftp from MIT.

The following is a list of users and usages of Larch.

- MIT
    - Circuit design [SGHG93]
    - Verifying timing of concurrent algorithms [LSGL94]
- Digital Equipment Systems Research Center
- Carnegie Mellon University
- Department of Computer Science, Iowa State University
- INRIA Lorraine, France
- Danish Technical University
- Aarhus University, Denmark
- University of Cincinnati, Ohio
- Odyssey Research Associates, Ithaca, New York
- Rome Laboratory, New York
    - Hardware verification
- University of St. Andrews, Scotland

Citations are made to published papers. Other users and projects listed were found on the Larch home page at MIT. [MITweb]

PVS

PVS (Prototype Verification System) is a specification language with an encapsulated environment that includes a typechecker and theorem prover. The specification language is based on a classical, typed higher-order logic. PVS was developed by members of SRI International Computer Science Laboratory in Menlo Park, California with the latest version being released on February 7, 1996. 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. PVS is available by anonymous ftp from SRI International.

The following is a list of users and usages of 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]

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

Murφ

Another mechanical analysis tool that is being used that is somewhat different than a theorem prover is the model checker. The Murφ system is a model checker that essentially enumerates all the possible states of a finite state machine. The Murφ description language consists of guarded commands that are executed non-deterministically in an infinite loop. Each guarded command consists of a condition followed by a number of actions. For every condition that evaluates to true, the actions are executed to expand all the possible states. Then through a number of state reduction techniques, the number of states is reduced to a manageable level. Another part of the description language are the state invariants. The Murφ system also contains a verifier than can verify that the invariants hold true for all reachable states.

Users:

- Stanford University
    - Verification of the cache coherence protocols in Stanford's DASH and FLASH multiprocessors.
    - Verification of link-level protocol and cache coherence protocol in Sun's S3.mp multiprocessor.
    - Verification of the cache coherence algorithm in Sun's UltraSparc-1
    - Executable specification, analyzer, and verifier for Sparc V9 memory models: TSO, PSO, and RMO.
    - Incorporated into U. of Wisconsin's Tempest customizable cache coherence protocol system.
    - Verification of part of SCI (“Scalable Coherent Interface”), IEEE Std 1596-1992. Some bugs were discovered.
    - Analysis of cryptographic and security-related protocols
    - Verification of proprietary protcols at several companies, including Fujitsu, HAL Computer Systems, HP, and IBM

3.2 Industrial Practice Using Theorem Provers

Despite the large number of research projects that have taken place, the number of industrial projects that have utilized this mechanical analysis technology is quite small. Of these industrial projects that have taken place, the majority of them are research projects as opposed to actual practice producing real products. I will detail a number of the significant projects that been performed in industry whether for sheer research purposes or for actual production-level projects.

Hardware production and verification is where a majority of the work utilizing theorem provers is being done. **Collins Commercial Avionics**, a division of Rockwell International, undertook a project that was co-sponsored by the Systems Validation Branch of NASA Langley to explore how formal specification and verification techniques could be introduced into an industrial process.

> *The project consisted of specifying in the PVS language developed by SRI a portion of a Rockwell proprietary microprocessor, the AAMP5, at both the instruction set and register-transfer levels and using the PVS theorem prover to show that the microcode correctly implemented the specified behavior for a representative subset of instructions* [SM95].

It is important to note that the purpose of the project was not to design a microprocessor that could be formally verified. The AAMP5 was designed to increase the performance of the AAMP2 by three times. The verification effort was secondary, but as the authors concluded successful. [SM95] The verification effort of the AAMP5 was undertaken as a shadow project, so it did not replace any of the normal design and verification activities that were part of Collins' design process. Also noteworthy is the fact that although the personnel at Collins played a minor, yet significant role in the actual PVS specification development, all of the verification or proofs of correctness was done by members of the SRI, International team. Although a large part of this project was completed by the members of the SRI team, the AAMP5 is an important verification effort since the AAMP5 is a commercial microprocessor as opposed to theoretical or research microprocessors that have been formally verified.

Requirements analysis is the area within the software domain that theorem provers are being used. Yoko Ampo of **NEC Space Systems** and Robyn Lutz of **Jet Propulsion Laboratory** and Iowa State University at Ames applied mechanical analysis tools, specifically PVS, to the requirements analysis of critical spacecraft software. This project consisted of specifying and analyzing the requirements for portions of the Cassini spacecraft's system-level fault-protection software (a Saturn orbiter due for launch in 1997). Unlike a number of projects in specification, this project was that the requirements were in flux while the project was being undertaken as opposed to reverse engineering where the requirements are very mature and stable. Like the AAMP5, this project was also an experimental study examining the applicability of formal methods and mechanical analysis to industrial software practices.

Judith Crow and Ben Di Vito were part of a project with **NASA Langley Research Center** in which Change Requests (software modifications) for the Global Positioning System were formalized using PVS and Murφ. This project illustrated a comparison between a theorem proving system and model checker. Like the Cassini spacecraft project, the requirements used for this project were not well defined at the time that they were being formalized. Because of the state of flux of the requirements during this project, there was no theorem proving completed, however mechanical analysis, such as typechecking, did aid in uncovering a number of errors [CD96].

Another application of theorem provers that has received some attention is protocol verification. Vijay Nagasamy of **LSI Logic**, Sreeranga Rajan (then of SRI), and Preeti R. Panda of UC Irvine investigated the formal specification and verification of a fibre Channel Protocol core [NRP95]. "They specified a portion of an implementation of the protocol in PVS and used the PVS model checker to examine its properties [PVSweb]." In addition they provide a formal specification of the protocol with hopes that the machine readable specification can aid in checking whether an implementation satisfies the standards. Like the previous examples, this was a research project that was undertaken.

A project is underway at **Trusted Information Systems, Incorporated** where the Boyer-Moore theorem prover is being used to verify object code against C source code. Instead of verifying an industrial strength compiler which can be very difficult, Sakthi Subramanian and Jeffrey V. Cook of TIS are using the Boyer-Moore theorem prover to verify that the object code correctly implements the C source code. The ultimate goal of the project is to produce a verification system for C programs [SC96].

Perhaps the single greatest industrial engineering project that took advantage of formal methods and mechanical analysis tools is the microprocessor design that is being done at **Computational Logic, Inc.** Beginning with the FM 8501 and progressing to the FM9001, CLI has formally specified and verified a number of microprocessors with the Boyer-Moore theorem prover. “The FM9001 is a general-purpose 32-bit microprocessor whose gate-level netlist design implementation was developed using a theorem-proving environment in conjunction with a traditional CAD system [BH94].” “Rigorous testing has not uncovered any situation where the manufactured device fails to meet its specification [BH94].”

Computational Logic has also formally verified an assembler, Piton, and a high-level language compiler for the μ-Gypsy language. [Moo88]

3.3 Research and Experimental Projects Involving Theorem Provers

The majority of the work in the field of theorem provers, formal verification, and mechanical analyses has been experimental projects usually performed by academics in conjunction with some level of participation of industrial partners. These are projects that are undertaken for the most part strictly as academic pursuits with no real application of the project occurring in industry, although projects of this nature often lead to innovations in industry.

Like the industrial practice of mechanical analyses, a large number of projects utilizing mechanical analysis, specifically theorem provers is in hardware verification. Unlike industrial practice however, there are a large number of existing projects that entail some sort of software verification. There are also a number of more theoretical projects that consist of proving theorems about algorithms or number theory. Finally, there are a number of sites that are developing the current theorem prover technologies and attempting to modify these technologies and provide more advanced tools to better serve the community.

Like industrial practice a large number of people are using theorem provers for hardware verification. A large number of hardware projects involve verifying microprocessors, both general purpose and embedded systems. Unlike the industrial projects discussed earlier, these microprocessors and hardware are theoretical. The vast majority of these hardware projects will not make it to actual hardware, but will remain specifications. The following is a list of projects utilizing theorem provers for hardware verification.

Boyer-Moore

- J Strother Moore - Mechanically Verified Hardware Implementing an 8-bit parallel IO Byzantine Agreement Processor [Moo92]

HOL:

- Paul Curzon of University of Cambridge - Asynchronous Transfer Mode (ATM) [Cur94]
- David Fura of Boeing and Philip Windley of the University of Idaho - processor interface unit for a fault -tolerant embedded processor [Kal93]
- Catia Angelo of Katholieke Universiteit Leuven - verification of generic and parameterized hardware modules in the CATHEDRAL silicon compiler environment [Ang94]
- Glynn Winskel of the University of Aarhus - low-level circuit verification in HOL [Kal93]
- Phillip J. Windley of the University of Idaho - Implementation of a Verified Microprocessor with Security Features [Win90]

- Tim Leonard of Digital Equipment Corporation - formal verification of Digital's chips [Kal93]
- S. Tahar of the University of Karlsruhe and R. Kumar of Forschungszentrum Informatik - Formal verification of RISC-pipeline conflicts [TK94]
- J. Frößl and T. Kropf of the University of Karlsruhe - Property Verification of Real-Time MOS-Transistor Circuits [FK95]

PVS:

- Paul S. Miner, Shyamsundar Pullela, and Steven D. Johnson of Indiana University - circuit for clock synchronization using various reasoning tools [MPJ94]
- Steven D. Johnson, Paul S. Miner, and Albert Camilleri of Indiana University, NASA Langley, and HP, did a study of the "Single Pulser" circuit for TPCD using various reasoning tools [JMC94]

A single focus of theorem prover research that has seen very positive results is in the area of protocol verification. Network protocols, authentication protocols and cache coherence protocols are a few of the projects that have been documented. As Vijay Nagasamy, Sreeranga Rajan and Preeti Ranjan Panda note that manual checking of an implementation and whether or not it satisfies a protocol can be tedious. The use of a mechanically described protocol provides mechanical verification of an implementation against a protocol. A number of projects have been completed that use mechanical analysis for verifying various protocols:

EVES:

- Dan Craigen and Mark Saaltink of Odyssey Research Associates - Authentication protocol analysis

PVS:

- Jozef Hooman of the Technical University of Eindhoven in the Netherlands - Verifying Part of the ACCESS.bus Protocol Using PVS [Hoo95]
- Klaus Havelund of LITP, Institut Blaise Pascal, University of Paris VI - Experiments in Theorem Proving and Model Checking for Protocol Verification [HS96]
- Seungjoon Park and David Dill of Stanford University - Cache Coherence Protocols and Memory Models [PD96]

A difference between the industrial practice and academic investigation and experimentation with theorem provers is the academic use of theorem provers for software verification. The industrial practice of verifying software with mechanical analyses tools is very limited. However, there are a number of projects in academia in which people are using theorem provers and verification systems to verify software. The verification efforts to this point have mainly been in the area of compiler verification and operating system kernel verification. A list of software verification projects include:

EVES:

- Sentot Kromodimoeljo of ORA Canada - *The one way link system/ subsystem specification*
- Dan Craigen of ORA Canada - *Application of EVES to Reverse Engineering Software*

HOL:

- Glynn Winskel and Sten Agerholm of the University of Aarhus - Program verification in HOL [Kal93]
- Michael J. Healy of Boeing - Preliminary investigation of software verification [Kal93]
- Peter Vincent Homeier of UCLA - Verification of software written in a general-purpose programming language [Kal93]
- Mark van der Voort of the University of Twente in the Netherlands - Attempting to extend HOL for use in the verification of functional languages [Kal93]

PVS:

- Dave Stringer-Calvert of the University of York - verification of a compiler for Tosca [SCWeb]
- Simon Fowler of the University of York - formal verification of real-time operating system kernels [FW96]
- Jozef Hooman and Jan Vitt of the Technical University of Eindhoven in the Netherlands - *Assertional Specification and Verification Using PVS of the Steam Boiler Control System* [VH96]

Another area that has seen a significant amount of research is formalizing distributed, parallel and concurrent systems. Theorem provers have been used to verify communication of distributed systems and algorithms.

HOL:

- J. Alves-Foss and K. Levitt - Mechanical Verification of Secure Distributed Systems [AFL92]
- A. J. Camilleri - Reasoning in CSP using the HOL Theorem Prover [Cam90]
- C. Zhang, R. Shaw, M. R. Heckman, G. D. Benson, M. Archer, K. Levitt, and R. A. Olsson - Towards a Formal Verification of a Secure Distributed System and its Applications [ZSH94]

PVS:

- Frank de Boer and Marten van Hulst of the Department of Computer Science at Utrecht University presented a paper at Formal Methods Europe (FME '96) on Local Nondeterminism in Asynchronously Communicating Processes. Marten's Ph.D. Thesis is also available.
- An overview of A Tool for Proving Invariance Properties of Concurrent Systems Automatically that uses PVS as a back-end was presented by Hassen Saidi at "Tools and Algorithms for the Construction and Analysis of Systems" (TACAS '96), Springer Verlag Lecture Notes in Computer Science no. 1055, pp. 412–416, Passau, Germany, March 1996.

Real-time systems have also been mechanically analyzed using theorem provers. The theorem prover can be used to prove properties about the real-time capabilities of systems using certain assumptions. The research in this area is fairly recent which suggests that there is a growing interest in using theorem provers for analyzing real-time systems.

HOL

- R. M. Cardell-Oliver - Formal Verification of Hard Real-Time Systems [COl92]

PVS

- Jozef Hooman at the Technical University of Eindhoven in the Netherlands, has used PVS to provide mechanical support for the Correctness of Real-Time Systems by Construction
- Mechanical Verification of Timed Automata: A Case Study, presented at the 1996 IEEE Real-Time Technology and Applications Symposium (RTAS'96), Boston, MA, June 1996.

- TAME: A Specialized Specification and Verification System for Timed Automata, by Myla Archer and Connie Heitmeyer, presented at the 17th IEEE Real-Time Systems Symposium (RTSS'96), Washington, DC, December 1996
- Verifying Hybrid Systems Modeled as Timed Automata: A Case Study, by Myla Archer and Connie Heitmeyer, to be presented at the International Workshop on Hybrid and Real-Time Systems (HART'97), Grenoble, France, March 1997.
- Simon Fowler is using PVS as part of work on the formal verification of real-time operating system kernels. The paper Formal Analysis of a Real-Time Kernel Specification was presented at FTRTFT'96, Uppsala, Sweden in September 1996, and represents the preliminary results of the work.

There has been some research done in the area of tool improvement. Many theorem provers reuse existing text editors or interfaces as their user interface. Although this can reduce bugs and speed production of the tools, interfaces are often crude and not user-friendly. Researchers have investigated the role of the user interface in theorem proving.

HOL

- Stuart Aitken, Philip Gray, Tom Melham and Muffy Thomas - Investigating Proof Activites and Human Computer Interaction. [AGMT95]

PVS

- Nicholas A. Merriam and Michael D. Harrison - Evaluating the Interfaces of Three Theorem Proving Assistants. [MH96]

3.4 Research in the Evaluation of Theorem Provers

There has been a limited amount of research done evaluating mechanical analysis tools and theorem provers, specifically. The are a few powerful theorem provers that are presently available and these tools that exist are relatively new and underused. Although a number of papers that describe the use of a specific theorem prover on a specific project attempt to generalize results to theorem provers in general, there is limited work in the field that has evaluated theorem provers and systematically detailed criteria for evaluation.

John Rushby from the Computer Science Laboratory at SRI International wrote a report for NASA entitled, “Formal Methods and the Certification of Critical Systems”, in which he explains the use of formal methods, specifically formal specification and verification, in safety-critical applications. Rushby concludes his report with a chapter on the selection of formal methods, levels and tools. Although this is not a set of criteria that can be used for evaluation, Rushby introduces a number of general questions that one must ask when deciding to use a “verification system”. Rushby distinguishes a verification system from a theorem prover saying:

> *I assume the tools under consideration provide a formal specification language, parser, typechecker, various utilities, and some support for mechanized proof checking or theorem proving. I generally refer to such a tool as a “verification system” (or simply “system”), and consider its capabilities in the order listed in the previous sentence* [Rus93].

By Rushby’s definition, a theorem prover would be an element of a verification system. Rushby’s general questions when choosing a verification system are:

- Does the system have adequate documentation and examples?
- Has the system been used on real problems?
- Is it easy to learn? And does it provide effective support for experienced users?
- Does the system support the selected formal methods and analyses effectively?
- Is the system generic or specific to a particular logic and language?
- Does the system support the implementation language concerned?

Similarly, Rushby lists questions for specification languages, utilities and theorem provers. Although Rushby has developed a significant set of questions and ideas to consider when evaluating formal methods’ tools, his list is not justified by means of the methods of real software production, the software lifecycle and project management activities.

Dan Craigen of ORA Canada references Rushby in his paper, “Formal Methods, EVES, and Safety Critical Systems” saying:

> *While there is, perhaps, a slight bias towards the technology developed at SRI International, they are a useful set of questions to contemplate. An example of a bias, through omission, is the lack of any questions referring*

> *to the use of a separate proof checker to confirm proofs discovered by a theorem prover.*

Craigen goes on to answer Rushby's questions for the EVES verification system. Unfortunately, Craigen does not add any questions to Rushby's list or any additional topics that should be covered when evaluating verification systems.

After verifying components of the Fairisle Asynchronous Transfer Mode communications network switch in HOL, Paul Curzon, Ian Leslie and Mike Gordon developed a list of practical concerns they encountered when attempting to specify and verify the ATM network. Although Curzon, Leslie and Gordon are not evaluating or comparing theorem provers, they discuss practical issues that hindered their ease of completing their project in a timely manner. Among their observations were:

- Proof scripts are beneficial especially when maintaining and reusing specifications.
- The specification design plays a role in the ease of verification. Although there are a number of functionally equivalent specifications for a specific problem, the design of the specification is critical to the ease of verification.
- Large proofs took dramatically longer to prove than smaller ones. Support for large proofs is needed. As an example, having tools or techniques to find similar theorems that have been proved.
- Proving theorems is a time-consuming project. Theorem proving should be used on specifications that have been analyzed for blatant errors while theorem proving should be reserved for only the most subtle errors to prevent wasted proof attempts.

Although these are not criteria, observations generated from actual experience will help produce usable tools to be used by industry [Cur94].

The majority of the use that verification systems have found is in the hardware verification area. To aid the hardware verifiers, Angelo, Verkest, Claesen and De Man compared HOL and Boyer-Moore for use in formal hardware verification. They evaluated these two theorem provers on the basis of maintainability, reusability and documentation of proofs. Unlike our approach where the evaluation is based on the software and the production of "better" software, Angelo, Verkest, Claesen and De Man choose three rather abstract terms and evaluate the theorem provers based upon these abstract ideas without demonstrating why these abstract ideas are relevant and important to hardware verification. The authors picked concepts that are generally considered important throughout the software engineering community, however, the authors never relate the relevance of maintainability, reusability and documentation of proofs to hardware production [AVCD91].

More recently, formalists have hypothesized that the combination of theorem proving and model checking would be the most effective method of theorem proving. By evaluating the strengths and weaknesses of theorem provers and model checkers as a whole, researchers have concluded that allowing theorem provers to use model checkers as primitive prove steps on small, constrained state spaces exploits the benefits of both model checkers and theorem provers. Model checkers are good at expanding and exploring relatively small state spaces to verify certain properties. Theorem provers are strong at handling larger proofs. By incorporating both techniques, small state spaces can be explored efficiently by a model checker, while the larger proof in general is handled by the theorem prover [RSS95].

4 Approach

The approach of this research was to develop a control system for the University of Virginia Reactor (UVAR) in order to investigate the effectiveness of using PVS by a typical engineer on a real software project. Specifications undergo phases of development, just as software undergoes phases such as: requirements, specification, implementation, verification and maintenance. This chapter will detail the specification and theorem proving activities that took place during the development and refinement of the specification.

4.1 Initial Specification Development and Theorem Proving

After completion of an initial simplified specification, research focussed on determining appropriate and relevant theorems to prove. After gaining an abstract understanding of the reactor and inspection of some related works, such as Butler's simple autopilot specification [But96], the initial state of the reactor was defined and a theorem that the initial state, `st0`, of the reactor is not in a scrammed state was successfully proved. In a scrammed state, the electrical current of the safety rod driver magnets is turned off and the rods are dropped into the reactor halting the nuclear reaction.

```
st0_good: LEMMA good(st0)
```

The proof of this theorem was rather trivial and fairly uninteresting. Following this, a second theorem was proved, that the reactor is always in a "good" state, that is "not scrammed", unless the scram event occurs.

```
after_initial_good: THEOREM (good(nextstate(st, event)) AND
      good(st)) IMPLIES event = reset_scram OR event = reset OR
      event = fail_header OR event = fail_pump
```

Virtually all theorem proving attempts applied the automated theorem proving capabilities of PVS and none of the more advanced capabilities of PVS were explored. By doing so, there was minimal human-computer interaction which can speed the theorem proving effort. Because the specification was so simple this was not a significant issue at this juncture. Although both proof attempts were successful, it became clear later that the simplified nature of this version of the reactor specification was significantly responsible for the ease of the proof efforts.

Problems encountered:

During this phase of initial specification development, the problems encountered consisted of unfamiliarity with the specification language and theorem prover, and the slow speed of development.

- *Difficulty with specification notation*
  A rather large amount of time was spent during this phase writing and revising specifications in order to create a syntactically correct specification. Some of the specific problems encountered involved accessing and modifying certain data structures. These problems were partly due to the fact that the specifier was unfamiliar with the language prior to this specification effort. However, it is important to note that the documentation was of little help in solving many problems and the fact that the PVS record structure is different than most popular programming languages hindered the ease of solving some rather trivial problems.

- *Difficulty with theorem proving notation and concepts*
  Like the specification language, the theorem prover was not familiar to the specifier prior to this research effort. Theorem proving in general was a task to which the specifier had little or no prior exposure which hindered the effectiveness and efficiency of the initial specification development. The documentation was a significant aid in the theorem proving effort, however the time needed to initially understand the basic commands of the theorem prover was significant.

- *Dramatic increase in time needed to prove theorems*

  Finally, the speed of development was very slow during this initial specification phase. As problems with the specification language were solved, the time required to prove theorems increased dramatically even following seemingly small and insignificant changes to the specification. As mentioned above, the theorems that were proved in this initial specification phase were proved using PVS's automated theorem proving capabilities. After expanding the specification to incorporate the alarms of the reactor, proving that the reactor was in a safe state increased the execution time of the theorem prover by as much as 10 times. The execution time was the CPU time as measured by the PVS system. Numerous trials were attempted at various times with various network and processor loads which ruled out excessive loads on the system as a possible explanation.

4.2 Specification Refinement

Following some additional input from sources more familiar with the UVAR reactor, the specification went through substantial changes and revisions. Rather than a reactor with simply a header, a pump and a few scram conditions, the reactor specification was changed to a more comprehensive model the significant elements of the reactor. The shim or safety rods, the control rod, the alarm lights, and the rod drivers were all introduced in this second iteration of the specification.

Following this specification refinement, and a much greater understanding of the nuclear reactor control system, a proof was attempted of a theorem that stated the reactor could not be operating in the high power mode with either the pump off or the header down. This is a significant theorem, because it describes a safety policy that ensures that certain disastrous states are not entered. If the reactor is operating in high power mode and the header is not up or the pump is not on, then the reactor is not being cooled sufficiently. The consequences of this scenario could be severe.

The first problem was to precisely state the theorem in such a manner that it would be manageable to prove and that the theorem stated was equivalent to the safety policy that was important to reactor experts. In English, the safety policy can be stated as: for every possible state, if the reactor is operating and is in high power mode, then the header is up and the pump is on. Translating this directly into PVS produces:

```
header_up_pump_on_in_high_power : LEMMA FORALL (st : states) :
      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
```

However, attempts to prove this theorem will be unsuccessful because the theorem is false. All of the possible states are not reachable from the initial state. A similar problem was encountered by Butler in his simplified autopilot [But96], so his strategy for checking for reachable states was emulated. There are a number of possible states for which this theorem is not true, however those states are not reachable from the initial state. Therefore the set of states considered must be additionally constrained by reachability. States that are not reachable do not need to be considered.

The definition in English of a reachable state is any state that can be arrived at from the initial state, `st0`, after transitioning on any number, `n`, of legal events. In order to use the same definition regardless of the number of events, `reachable_in` is defined in PVS using recursion, and states that a state, `st`, is reachable if there exists a reachable state, `pst`, from which there is an event that causes a transition from `pst` to `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)
```

In order to make the proof of a theorem, such as:

```
header_up_pump_on_in_high_power
```

that uses a recursive definition (such as `reachable_in`) more manageable, it became clear that it would be necessary to use mathematical induction as a proof strategy. PVS has a pre-defined induction strategy, but it was difficult to state the theorem header_up_pump_on_in_high_power in a manner that was amenable to induction. Namely, it was not immediately clear that the variable of induction should be n, the number of transitions made.

By using the theorem as it was previously stated, it was not possible to use induction because n had not been defined. In order to use induction it became necessary to state the theorem in an alternate, but equivalent way and to introduce an additional definition startup_on_n, similar to reachable_in, that ensures that the startup event was encountered. The following is the final version of the theorem header_up_pump_on_in_high_power and its related definitions that was used to attempt to prove that at any reachable state, if the reactor was operating in high mode, then the pump is on and the header is up:

```
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

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
```

This definition of header_up_pump_on_in_high_power differs from the previous version only in the addition of the two lines using the two definitions, startup_on_n and is_reachable, and the introduction of n.

The theorem header_up_pump_on_in_high_power was not successfully proved. It is not clear whether the reason for this was that the proof strategy was not valid, the system specification violated this theorem, or just that additional lemmas and definitions were needed to assist the theorem prover with the proof. Perhaps this is data to support claims made by opponents of formal verification that theorem provers are very difficult to use on real software systems because of their complexity.

When attempting a proof with a mechanized theorem prover, it is important to define and prove a number of smaller lemmas in order to make the entire proof effort more manageable. In an attempt to make the proof of header_up_pump_on_in_high_power simpler, various lemmas were stated and proofs attempted. For example, it was specified and proved that if the reactor was not in an operating state, the only event that could occur to initialize operation would be the startup event. This was the basis for the induction:

```
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
```

The startup event causes the pump to be turned on and the header to be moved up. It can be shown then that the pump is on and the header is up when the reactor is initially operational:

```
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
```

A large number of lemmas were developed during the proof attempt of `header_up_pump_on_in_high_power`. These lemmas describe sub-tasks of the main proof and must each be proved individually, but then they can be used in the main proof. The purpose and syntax of all of these lemmas are not explained in detail here, but the development effort required to perform even this unsuccessful proof attempt is apparent. The following lemmas are those used in the final version of `header_up_pump_on_in_high_power`, not every lemma that was formulated during the complete attempt:

```
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
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

startup_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

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_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
```

```
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
```

Problems encountered:

During this phase of specification development, difficulties were encountered with the use of the theorem prover for more complex theorems that could not be proved automatically in PVS.

- *Difficulty identifying theorems to prove.*

  It became clear that choosing appropriate theorems to prove is difficult. Although there are a number of safety properties imperative for the nuclear reactor control system to maintain, system specifiers felt that these properties were inherent in the specification and it was not necessary to spend time attempting proofs of these properties. For example, it is an important property of the control system that the only method of recovering from a scram is to press the reset scram button. Although this might be an important property to specify as a theorem, it is fairly explicitly stated in the specification.

- *Difficulty writing theorems that were both readable and provable.*

  As discussed above, once the theorems were determined, it was difficult to specify the theorems in such a manner that was easily understood by readers of the specification and was easily provable using the theorem prover. The syntax of the theorem was just as influential as its semantics on the ease of the theorem proving effort.

5 Results

In this chapter, the experiences using PVS with preparation and analysis of the UVAR specification are presented.

5.1 Parser

The parser is responsible for checking that a specification adheres to the syntax of the PVS specification language. The PVS parser is invoked any time the typechecker or theorem prover are used and the current specification has not been parsed.

- *The parser can only find a single error per execution regardless of the number of errors that may be present in a specification.*

  Initial static analysis should be fast in order to find simple errors and aid initial specification development. Current programming language compilers find as many syntax errors as possible when run, specification parsers should be equally powerful, otherwise initial specification development will be slow. This is especially critical if a “prototype” specification is being developed where speed of development is essential.

5.2 Typechecker

PVS is a strongly typed language which means that every variable and function have an explicit type, such as integer, boolean, or other user-defined types. The PVS typechecker checks that all values have a type and that the types are consistent throughout the specification.

- *The typechecker can only find a single error per execution regardless of the number of errors that may be present in a specification.*

  It is necessary that as many errors as possible be detected at an early stage of mechanical analysis for the analysis to be as fast and useful as possible. The PVS typechecker is neither. Modern programming language compilers find and report all the static typecheck errors that can be determined on a single compiler execution. Specification language typecheckers are essentially no different than programming language typecheckers, and yet the PVS typechecker is substandard.

- *The user must typecheck a specification every time that the system is started regardless of whether or not the specification has changed since last typechecked.*

  Typechecking an industrial-sized specification can be extremely time-consuming. An unchanged, typechecked specification must be typechecked every time PVS is initiated. This is unnecessary and can create delays in the development process.

5.3 Specification Interface

- *The interface doesn't alert the user with sufficient emphasis that the specification has successfully been typechecked.*

  PVS simply indicates the last theory that has been typechecked with a tiny flag in the emacs window that changes from tc for typechecking to Ready which is not documented. See Fig. 3 on page 64 and Fig. 4 on page 65.

- *It appears that the user cannot change the tabbing environment within PVS.*

  Part of the specification process is communicating the ideas and concepts that are present in the specification to the client. This process of communication is aided greatly if the specification text can be formatted in an organized manner. PVS hinders this effort by not allowing the user to change the tabbing environment. The editing environment will allow the user to edit the tab stops, but when the user attempt to install the changes, the changes are not incorporated into the editing environment.

- *Support for tabular specifications is marred by the editor's lack of expressivity.*

  A method of specification that is gaining popularity is that of tabular specifications. PVS has added support for tabular specifications that can meet the tabular standards of a number of different specification methods (SCR, decision tables, etc.). However, by using the simple text editor Emacs, tabular specifications are not only difficult to read and decipher, but they are also rather laborious to write. The specifier must constantly type in characters to represent the lines in a table. The editor should support some method of creating and editing tables efficiently if large-scale tabular specifications are going to be built with PVS.

```
two_d_tables: THEORY

BEGIN

  IMPORTING Parnas_examples

  normal2(y,x:real):real =
    TABLE
           %--------------------------------------------------%
           |[ y=27           | y>27            | y<27          ]|
     %--------------------------------------------------------%
     | x=3 | 27+sqrt(27)     | 54+sqrt(27)     | y^2 +3        ||
     %--------------------------------------------------------%
     | x<3 | 27+sqrt(-(x-4)) | y+sqrt(-(x-5)) | y^2 + (x-3)^2 ||
     %--------------------------------------------------------%
     | x>3 | 27+sqrt(x-3)    | 2*y+sqrt(x-3)  | y^2 + (3-x)^2 ||
     %--------------------------------------------------------%
    ENDTABLE

END two_d_tables
```

- *Navigation is extremely labored.*
  Specification navigation is a fundamental concept that is needed in order to aid communication of the specification to the client. Navigation refers to the ease of finding desired information in a document efficiently. Navigation is also necessary when making changes to the specification. PVS has a number of functions that add navigation possibilities, but they are very difficult to use effectively. PVS provides significant information for the user such as: the location (file) of the declaration of a function or variable, the locations within files where this function or variable is used, or the actual declaration of the variable or function. The interface does not display the file where the declaration resides automatically, the user must invoke the display manually so editing is slow. PVS can provide the user with the declaration of a variable or function, but the declaration is a read-only copy that cannot be modified for use in the specification. The sort of navigation that PVS provides does not aid the user as much as it could.

The specification should serve as a method of communication between the computer scientists and the domain experts. The specification will be viewed often for communicating ideas without making any revisions to the specification. However, when first viewing a completed specification in PVS, it is necessary to typecheck and restore all the theories used in the specification, before the navigation capabilities can be used. Since typechecking can be very time consuming, this is a great inconvenience simply to navigate the specification.

- *Interface is prone to latencies.*

  The large amounts of computing power required by the theorem prover make virtually any other activity impossible until the theorem prover has completed its computations. For instance, although PVS supports editing of a specification while theorem proving, it is ineffective if the theorem prover is doing computation since the interface is delayed by the computation.

- *All files of a specification must be in the same directory.*

  In a multiple-user specification project, it is advantageous if all the specification files do not have to be in the same directory. Each individual user should have local files of specific parts of the specification. However, there must be some shared files or globally available files. PVS requires all the files of a specification to be in the same directory in order to use the analysis procedures. This hinders group development.

5.4 Theorem Prover Interface

- *Information is not presented in a useful manner during proof attempts.*

  Information should be presented in such a manner that it is relatively clear and easy to discern the next proof command that should be used in order to arrive closer to the ultimate proof goal. Following a single step in a PVS proof attempt, the user can be presented with literally pages and pages of information. The user is responsible for determining the next proof command that should be used from these pages and pages of information. The information that PVS presents is too abundant to easily decipher the next proof step that is necessary.

- *Theorem prover redisplays proof results after an invalid command.*

  This is a problem if the most recent proof attempt resulted in pages and pages of information, the display may take many minutes to display. There is no need to redisplay the information if all that occurred was an invalid proof command.

  Similarly, if a proof command is run that results in no change on the status of the proof, the theorem prover will alert you that no change occurred, but will also redisplay the current proof information. The current proof information is still visible, it is not necessary to redisplay existing information if the proof command resulted in no change.

- *There is an overabundance of output following certain proof commands.*

  The `grind` command essentially allows the theorem prover to apply strategies to the proof in order to prove the current proof goal without the assistance of a human user. The resulting output generated by `grind` can create pages and pages of output which is not only difficult to decipher if the command should not result in a successful proof, but it is also time-consuming for the emacs interface to output to the screen.

  PVS allows you to combine singular primitive commands into a composite command that sequentially applies the individual basic commands to the current proof. When executing a composite command the output from each individual basic command is displayed despite the fact that the result is simply going to be used to execute another command. When the final basic command has been completed, although the resulting output has been sent to the screen, the output is once again displayed on the screen as the result of the composite command. Thus, the final proof result is outputted twice.

  In an attempt to reduce the amount of information presented to the user, PVS allows the user to hide certain sequents or formulas. This essentially deletes the sequent, but saves it for retrieval if needed later in the proof attempt. PVS does not however hide two identical sequents automatically. Hiding sequents reduces the amount of information somewhat, but PVS does not allow you to hide clauses of a formula. For example a sequent of the form:

```
IF j
THEN w
ELSIF k
```

```
THEN x
ELSE y
ENDIF
ELSE z
ENDIF
```

It is rather irrelevant what `w`, `x`, `y`, or `z` are. The user must prove either that condition `j` is true or false in order to simplify this formula. The values of `w`, `x`, `y`, and `z` are of secondary importance. The user can only use `w`, `x`, `y`, or `z` after proving the truth value of `j`. If PVS could provide a means of hiding some of this excess information automatically or at least allow the user to hide some of this excess information, the output would be more easily navigable.

- *Proof tree display often lags behind the current proof attempt.*

  Using the display commands that PVS provides, allows the user to view a tcl representation of the current proof tree. It illustrates the goals and subgoals of the current proof so that the user has some idea of the state of the current proof. This tree display, however, often lags behind the current state of the proof and demands that either the user do without the display or that the user wait until the proof tree display can properly generate the representation of the tree. Proof attempts are time-consuming without waiting for this display. It takes substantially longer if the user is interested in using this proof tree display window.

- *Navigation of a proof attempt is virtually impossible.*

  As has been mentioned before, there is often an overabundance of information presented to the user during a proof attempt. With such an abundance of information, it is necessary that some sort of navigation mechanism exists in order to be able to decipher the information. The only sort of navigation that exists for the theorem prover is the search mechanism that is present in Emacs. A user can do regular expression searching backwards and forwards, but that is the extent of navigation capabilities available while theorem proving.

5.5 Theorem Prover

- *Proofs of type definitions are not automated.*

  After defining an enumerated type and attempting a significant proof, it became necessary to prove that the only values that a variable of that enumerated type could have were the values that were enumerated in its definition. There is no automated case analysis that can be done on enumerated types.

- *There is no way to interrupt a proof command gracefully.*

  The only method of interrupting the current proof command is hitting CTRL C twice and then typing: `(restore)`

- *Some amount of LISP knowledge is required to prove effectively.*

  The primitive proof commands of PVS give the user some useful primitives that can be combined for more powerful techniques. The theorem prover is written in LISP and thus taking advantage of all of the most powerful techniques require an understanding of LISP. LISP is not generally a language that is used or taught extensively in industry and rather unlike most popular languages used in industry.

- *Certain syntaxes are undesirable for typechecking, and possibly theorem proving.*

  The use of embedded record initialization made the typechecking of a simple specification explode from 3 seconds to 544 seconds. See Fig. 1 on page 60 and Fig. 2 on page 62 respectively.

  The modularization mechanism that PVS provides is theories. To use types and definitions from another theory, the user must use the command `INCLUDING` and the theory name. When a theory is typechecked all of the theories that it includes will be typechecked also. It is not necessary for file names to correspond to theory names in PVS. For example, it is possible, but not required, for the user to have a file named temp that includes no theory named temp. If the user copies all the files of an entire spec to another directory and typechecks, if the name of a file is not also the name of a theory, that file is not typechecked and creates a typecheck error. This is an inconsistency that could create problems in a large, multiple-user project, where individual users may each have a local copy of a specification.

- *Certain proof commands have no counterpart.*

  The `expand` command expands function or identifier names into their respective definitions. There is no contract command that contracts an equivalent definition into its identifier name. This aids some amount of abstraction during a proof attempt and could help reduce the amount of information that is presented during a proof attempt. The method that is used to modify fields of a record without reassigning the entire record is using the `WITH` command. There is no automated mechanism or simple command that will take a series of `WITH` statements and simplify the statement such as simplifying:

  ```
  record WITH [ field1 := value1] WITH [field2 := value2] WITH
  [field1 := value3]
  ```

  to

  ```
  record WITH [ field1 := value3] WITH [field2 := value2]
  ```

- *Certain seemingly intuitive proofs require much work to prove.*

  Proofs using a simple case analysis require extensive human guidance to prove, such as:

  ```
  IF operating(checked(power_level(pst!1))) = IDLE_UNCHECKED
  THEN IDLE_UNCHECKED
  ELSE IDLE_CHECKED
  ENDIF
    = OPERATING
  ```

  This statement is comparing the result of the `IF THEN ELSE` statement to `OPERATING`. Regardless of the value of `operating(checked(power_level(pst!1))`, the resulting value of `operating(checked(power_level(pst!1)))` is either `IDLE_UNCHECKED` or `IDLE_CHECKED` neither of which are equal to `OPERATING` by definition. The theorem prover does not recognize this and requires the user to determine the value of `operating(checked(power_level(pst!1)))` or do an explicit case analysis of `operating(checked(power_level(pst!1)))` before the proof can be completed.

  PVS cannot simplify some basic theorems of discrete mathematics, such as:

  $$\neg[(\forall x)P(x)] \equiv (\exists x)\neg[P(x)]$$

  Although the proof of such a statement is rather simple, there are instances where one form is more readable or usable than another. In order to use the equivalent form, the user must prove a theorem that explicitly states the equivalence of the two forms.

- *An interrupted proof must be rerun in order to restore the state of the theorem prover.*
  Running a partially completed proof of a theorem can take hours to complete. This only leaves the user at an unfinished state. This is a great hindrance to efficient theorem proving when the correct method of proof is not known. If the user explores a strategy that proves unsuccessful, the user must either rerun the original partially completed proof or undo a number of proof commands to back up to the previous state of the proof. Both are time consuming.

5.6 General Verification System

- *System does not support running a proof and editing simultaneously and effectively.*
  Although PVS does allow the user to edit specifications and theorem prove simultaneously, it is not effective. As mentioned earlier, the theorem prover requires a significant amount of computing power and CPU time to execute. There is often a noticeable delay when attempting to edit a specification while the theorem prover is attempting a proof. In addition, the system allows the user to change a specification while the theorem prover is running without alerting the user that the theorem prover is executing. This could result in an outdated proof of a theorem since the proof was completed on the original, unmodified specification.

- *Combination of model checking and theorem proving requires additional language features.*
  Many formalists have advocated using a combination of model checking and theorem proving. PVS supports model checking as a primitive proof command. The use of model checking with PVS requires the specifier to learn and use additional language constructs. This adds to the size and complexity of the specification language which is already rather large and complex. It also requires the user to learn new poorly documented techniques on model checking and combining model checking and theorem proving.

- *Memory is not managed in a convenient manner.*
  When attempting to use the `(grind)` command on `basic_last_lemma`, the following

error message appeared:

```
Error: An explicit gc call caused tenuring and a need for 83099648
more bytes of heap.  The operating system will not make the space
available because of a lack of swap space or some other operating
system imposed limit.

[condition type: STORAGE-CONDITION]
```

- *Meaningful identifier names are not encouraged.*

  It is a generally accepted software engineering principle to use reasonable and meaningful names for identifiers. This aids in communicating the meaning of the identifiers not only to clients, but to other software engineers as well. In order to successfully prove theorems it is often necessary to expand the definitions of many different functions and identifiers. It is often the case that it is necessary to expand the definition of a single function or identifier many times during a single proof attempt. It is possible to define auto-rewrite rules that could automatically expand the definition. However, in order to keep the proof succinct and readable it is often desirable to expand only the necessary definitions. In order to expand an individual instance of an identifier, it is necessary to type in the identifier name and the formula it occurs in instead of using a shortcut key sequence which expands all of the occurrences of that identifier. In an industrial setting this would encourage the specifier to use as short identifier names as possible to avoid excessive typing that can slow down the theorem proving process.

  The purpose of the specification is communication with the client concerning the functional requirements of the system. Despite communication being a primary goal, the communicative value of the specification can be lost when using a theorem prover in order to make proving theorems easier. As stated above, function definitions may be expanded many times throughout a proof attempt. In an attempt to minimize the information generated during a proof attempt, a function's definition would be kept minimal. This might involve combining statements, using some more complicated syntax which will be more unreadable by the customer.

- *The user cannot save multiple proof attempts easily.*

  Proving theorems is a somewhat long and arduous process. It is often necessary to make many different attempts at a proof before the theorem is successfully proved.

PVS allows the user to save a proof attempt when aborting before success. Unfortunately, PVS does not allow the user to easily save multiple proof attempts. It may be the case that a seemingly unsuccessful and unfruitful proof attempt was the right strategy that was missing a key lemma. If PVS does not allow the user to save multiple proof attempts, this information could be lost.

PVS will allow the user to attempt a proof after the theorem has already been proved. In addition, the successful proof attempt can be overwritten. If this is done and the resulting attempt was not successful the status of the proof simply is "unfinished." PVS can not be responsible for saving the user from him or herself, but it should not be allowed for a successful proof attempt to be completely overwritten and lost.

- *Incorporating PVS specifications into a specification document is difficult.*

  PVS has a prettyprint function defined which takes a proof or specification and generates LaTex code to present a nicely formatted document. However, specification documents are typically large, written in systems other than LaTex, and contain a myriad of material. Incorporating LaTex output into such documents is problematic.

- *There is very little support for evolution.*

  After expanding the specification slightly, the same proofs can last much longer than previous simpler attempts. Any industrial or research theorem proving is done primarily after the specification has been validated by the domain experts. Theorem proving is very complex and time-consuming and should be used as a final verification technique.

  > *Formal proof should be used to find obscure bugs and sort out subtleties in the understanding of why the design is believed to be correct. For such problems formal verification may be the only solution. Debugging of specifications, designs and implementations by traditional methods such as testing, code walk-throughs, comparison of netlists and even model checking where appropriate should be conducted first. The formal verification task will then be much simpler.*[Cur95]

5.7 Documentation

- *There is no literature (books and courses) that describes how to build specifications* *using PVS.*

Specification is a process that is different than implementation. The processes themselves are different, not just the languages used. There is no manual or tutorial describing how best to specify systems using PVS.

Specification is an abstract description of the functional requirements of a software system. There is much disagreement even among experts in the field as to the amount of information that should be present in a specification. There is no described method of specification that discusses the amount of abstraction that should be used in a specification.

- *There is no literature (books and courses) that describes how to best use a theorem prover.*

  Existing documentation explains what individual proof commands mean and the effect that a certain proof command may have on a small proof, but there is no documentation discussing proof strategies to use. Theorem proving with a mechanical analysis tool is not the same process as a human-engineered proof. It has been documented that theorem proving is a difficult task, but there is no existing documentation that describes methods of theorem proving, based upon the experience of others.

- *The existing documentation is difficult to understand and is incomplete.*

  There are syntaxes that are used throughout the community that are not well described in the documentation. Brief descriptions for individual keywords are primarily what exist, but there is little or no discussion of when or how to use these features. Examples include the `ASSUMING` and `USING` keywords. There is also little documentation describing how to do model checking in PVS although there are papers written that describe this practice.

- *There are few papers that discuss a method to determine useful proofs to prove.*

  Once a specification has been written, then theorem proving can occur. However, the problem of what theorems should be proved is significant. There are few papers that discuss the method that users have used to determine theorems to prove.

- *There are virtually no examples of formally verified software.*

  The majority of verification efforts using PVS are for hardware. There are a few examples of using PVS to document and aid requirements capture, but not enough to provide industrial users with real examples of how to use PVS for a significant software verification effort.

- *Papers that document formal verification examples rarely describe proof techniques used*

  There is little effort being made by the community of users of PVS to share their proofs or proof techniques of significant theorems on significant specifications. Much like implementation languages, users agree that specification and proof reuse is an important step that must be taken, but there exist very few libraries and those that do exist are very small. In addition to the fact that papers do not provide the in-depth details of the proofs, there is little information on the techniques that are being used to successfully prove significant theorems. The hardware arena is the only arena where a significant effort has been made to generalize theorem proving techniques.

- *The existing information on bug fixes and other pertinent information is virtually unusable.*

  The mailing list archive for 1996 is 568K of unformatted text messages. It is difficult to decipher pertinent information since it is so long and the mailing-list archive includes conference announcements and calls for papers, etc.

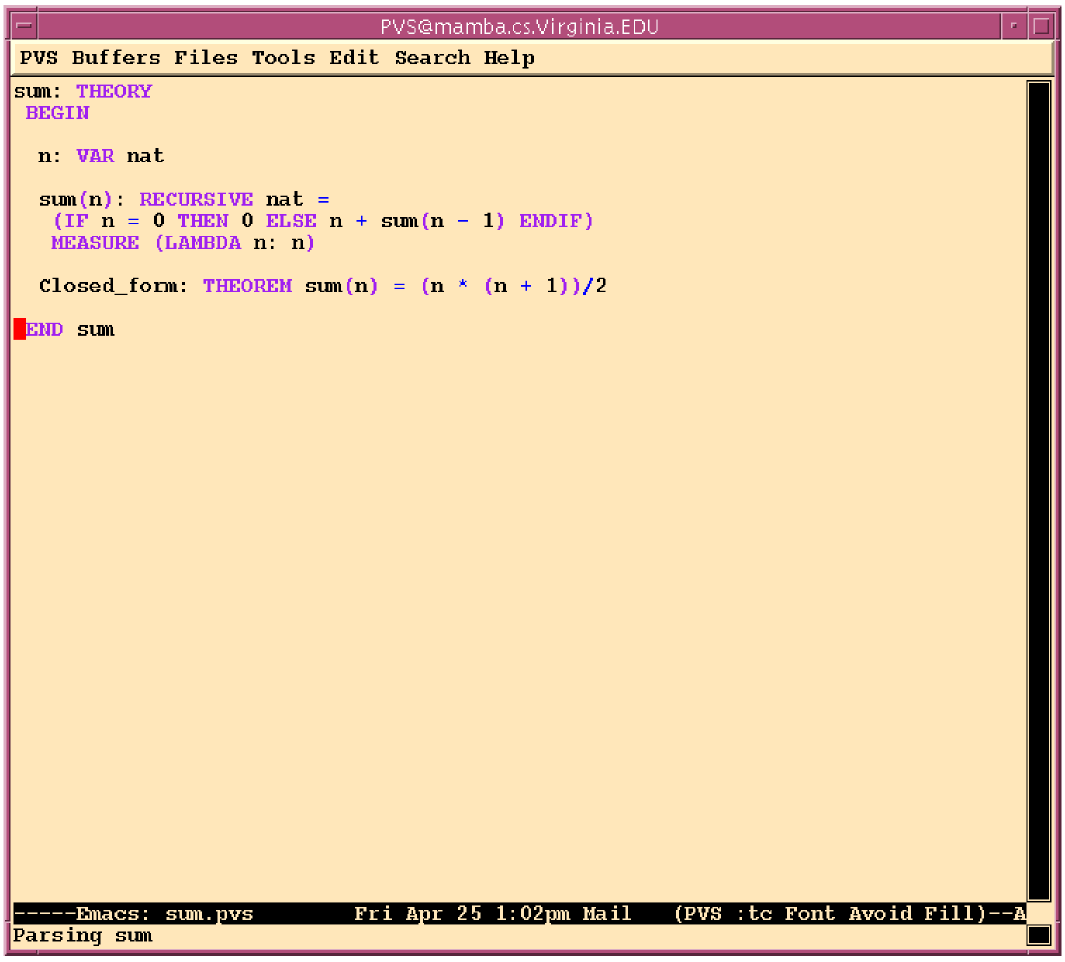
Figure 1

```
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 #);


  st0                       :    states =

      (#cooling_system           :=  (# pump             := OFF,
                                     header             := DOWN,
                                     sec_pump           := OFF,
                                     line_valve         := CLOSED,
                                     line_pressure      := NORMAL
                                     #),
      sensors                    :=  (# 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
                                     #),
      alarms                     :=  (# 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
                                     #),
     rods                 :=  (# shim_rods      := (# 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
                                                           #),
                                     control_rod        := MANUAL_CONTROL
                                     #),
      power_level                :=  (# sp_limit        := 250,
                                     set_point          := 230,
                                     operating          := IDLE_UNCHECKED,
                                     power_level        := LOW_POWER,
                                     range_switch_2     := LOW_MODE
                                     #)
                  #);

END verified_theorems
```

Figure 2

```
verified_theorems            :    THEORY

  BEGIN

  IMPORTING transition

  st0                        :    states =

      (#cooling_system            :=  (# pump             := OFF,
                                      header             := DOWN,
                                      sec_pump           := OFF,
                                      line_valve         := CLOSED,
                                      line_pressure      := NORMAL
                                      #),
      sensors                     :=  (# 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
                                      #),
      alarms                      :=  (# 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
                                      #),
 rods                 :=  (# shim_rods      := (# scram_state := NOT_SCRAMMED,
                                                            r1_driver      := 0,
                                                  r1_lamps      := (# up   := OFF,
                                                                      down    := ON,
                                                                     seated  := ON,
                                                                  mag_eng := ON #),
                                                       r1_magnet      := MAG_OFF,
                                                            r2_driver      := 0,
                                                  r2_lamps      := (# up   := OFF,
                                                                      down    := ON,
                                                                     seated  := ON,
                                                                  mag_eng := ON #),
                                                       r2_magnet      := MAG_OFF,
                                                            r3_driver      := 0,
                                                  r3_lamps      := (# up   := OFF,
                                                                      down    := ON,
                                                                     seated  := ON,
                                                                  mag_eng := ON #),
                                                       r3_magnet      := MAG_OFF
                                                            #),
                                      control_rod        := MANUAL_CONTROL
                                      #),
       power_level              :=  (# sp_limit        := 250,
                                      set_point          := 230,
                                      operating          := IDLE_UNCHECKED,
                                      power_level        := LOW_POWER,
                                      range_switch_2     := LOW_MODE
                                      #)
                   #);

 END verified_theorems
```

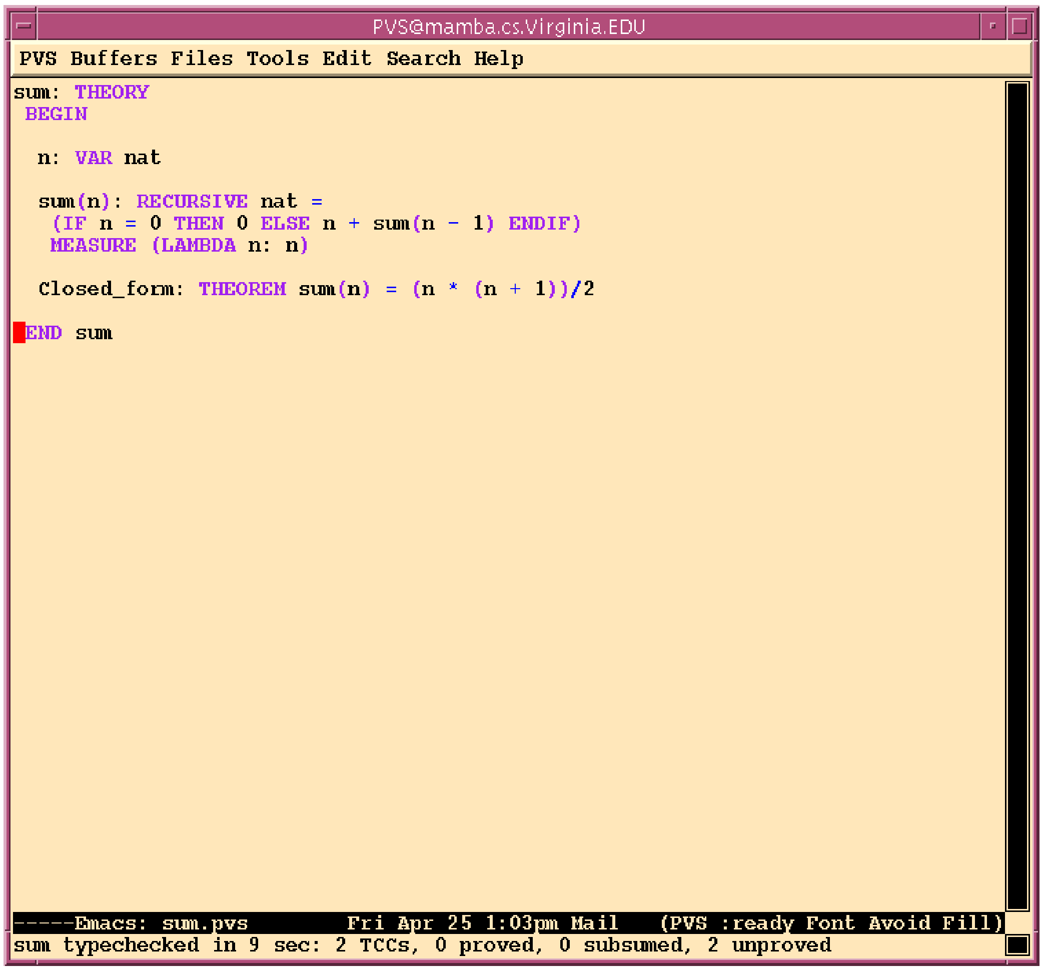
Figure 3


PVS@mamba.cs.Virginia.EDU
PVS Buffers Files Tools Edit Search Help

```
sum: THEORY
 BEGIN

  n: VAR nat

  sum(n): RECURSIVE nat =
   (IF n = 0 THEN 0 ELSE n + sum(n - 1) ENDIF)
   MEASURE (LAMBDA n: n)

  Closed_form: THEOREM sum(n) = (n * (n + 1))/2

 END sum
```

```
-----Emacs: sum.pvs         Fri Apr 25 1:02pm Mail    (PVS :tc Font Avoid Fill)--A
Parsing sum
```

Figure 4


PVS@mamba.cs.Virginia.EDU
PVS Buffers Files Tools Edit Search Help

```
sum: THEORY
 BEGIN

  n: VAR nat

  sum(n): RECURSIVE nat =
   (IF n = 0 THEN 0 ELSE n + sum(n - 1) ENDIF)
   MEASURE (LAMBDA n: n)

  Closed_form: THEOREM sum(n) = (n * (n + 1))/2

 END sum
```

```
-----Emacs: sum.pvs         Fri Apr 25 1:03pm Mail    (PVS :ready Font Avoid Fill)
sum typechecked in 9 sec: 2 TCCs, 0 proved, 0 subsumed, 2 unproved
```

6 Conclusion

Despite the added reliability that formalists claim can be achieved using formal methods, formal methods have found little place in industrial practice. The primary argument of industry against the use of formal methods has been that formal specification and theorem proving create too many delays early in the development process and slow overall software development. Formal methods proponents have countered with the argument that initial costs and delays due to training and the inexperience of engineers with this technology will be realized. However, once engineers are properly trained to use formal methods the delays that industry complains of will disappear as will the additional costs that industry claims make formal methods unreasonable and not applicable to real software development.

The results of this research demonstrate how the two opposing views have spent too long advocating their own views without investigating the problems explained by the other side. The initial delays and costs that industry has complained about were definitely evident in this specification development. Although the initial costs of injecting an entirely new process into software development can not be avoided, the inclusion of extensive, and well-organized documentation would help ease the initial costs of learning a new language and a new process, as well as acting as reference materials for more experienced practitioners. This type of documentation is not currently available. Likewise, industry has failed to notice some of the significant experiences and results with formal methods such as the

AAMP5 verification effort.

After investigating the PVS verification system from the point-of-view of a typical engineer, it is clear that PVS is not ready for widespread industrial use. Unlike much of the related work with which has praised PVS, the experiences were not detailed by experts or developers of the system, but the typical, inexperienced, untrained engineer. Although there are wide-ranging deficiencies present in PVS, the two major deficiencies that will hinder PVS' widespread use in industry are the user interface and it's lack of usability and the speed of the theorem prover. While programming tools are becoming visual taking advantage of point-and-click interfaces attempting to make the implementation phase more efficient, PVS remains primarily a command line interface that is difficult and inefficient. Likewise, injecting theorem proving into the existing software development cycle with the underdeveloped theorem proving techniques and strategies that are common to the state-of-the-art is too great a cost for industry to take.

Although the correctness and completeness of the specification were aided greatly by the use of PVS, the speed of development was not only hindered by the injection of the formal notation, but by shortcomings of the verification system itself. It is imperative that future research consist of not only improving the speed of theorem provers, but also attempts to determine better methods of proving theorems including clearer and more receptive interfaces. Industry and formalists must work together to discuss the benefits and shortcomings of the tools and methods not from opposing sides, but complimentary groups, as tool and notation developers and customers or users.

7 References

PVS

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

[But93] Ricky W. Butler. 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. Available, with PVS specification files, from <http://atbwww.larc.nasa.gov/ftp/larc/PVS-tutorial>; use only files marked “revised.”

[But96] Ricky W. Butler. An introduction to requirements capture using PVS: Specification of a simple autopilot. NASA Technical Memorandum 110255, NASA Langley Research Center, Hampton, VA, May 1996.

[CD96] Judith Crow and Ben L. Di Vito. “Formalizing Space Shuttle Software Requirements,” Proceedings of the ACM SIGSOFT Workshop on Formal Methods in Software Practice, January 1996.

[FW96] Simon Fowler and Andy Wellings. Formal Analysis of a Real-Time Kernel Specification, presented at FTRTFT96, Uppsala, Sweden.

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

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

[JMC94] Steven D. Johnson, Paul S. Miner, and Albert Camilleri. Studies of the Single Pulser in Various Reasoning Systems,

[LA94] Robyn Lutz and Yoko Ampo. Experience report: Using formal methods for requirements analysis of critical spacecraft software. In Proceedings of the 19th Annual Software Engineering Workshop, pages 231--248, Greenbelt, MD, December 1994. NASA Goddard Space Flight Center.

[MH96] Nicholas Merriam and Michael Harrison. Evaluating the Interfaces of Three Theorem Proving Assistants, Proceedings of Design, Specification, and Verification of Interactive Systems 1996.

[MPJ94] Paul S. Miner, 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] Vijay Nagasamy, Sreeranga Rajan, and Preeti R. Panda. Fibre channel protocol: Formal specification and verification. In Sixth Annual Silicon Valley Networking Conference. SysTech Research, April 1995.

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

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

[PVSweb] Information on PVS can be found at the Computer Science Laboratory at SRI International at: <http://www.csl.sri.com/pvs-users.html>, 1997.

[RSS95] An Integration of Model Checking with Automated Proof Checking. Proceedings of Computer Aided Verification, 1995.

[SCWeb] Information on David W.J. Stringer-Calvert's work on the Trusted Compilation Project at the University of York can be found at <http://www.york.ac.uk/~dwjsc100/compilers.html> 1997

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

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

Boyer-Moore

[Bev88] William R. Bevier. Kit: A Study in Operating System Verification, Computational Logic Inc., Technical Report 28, August 1988.

[BH94] Bishop C. Brock and Warren A. Hunt, Jr. An Overview of the Formal Specification and Verification of the FM9001 Microprocessor, Fall 1994 available from (<http://www.cli.com/hardware/fm9001.html>)

[CLIweb] Information on the Boyer-Moore Theorem Prover can be found at: <http://www.cli.com/software/nqthm/index.html>, 1997

[HB89] Warren A. Hunt, Jr. and Bishop C. Brock. The Verification of a Bit-Slice ALU, Computational Logic Inc., Technical Report 49, September 1989.

[Moo88] J Strother Moore. Piton: A verified Assembly Level Language, Technical Report 22, Computational Logic, Inc., September, 1988.

[Moo92] J Strother Moore. Mechanically Verified Hardware Implementing an 8-Bit Parallel IO Byzantine Agreement Processor, Technical Report 69, Computational Logic, Inc., 1992.

[Rus94] David M. Russinoff. Specification and Verification of Gate-Level VHDL Models of Synchronous and Asynchronous Circuits, Computational Logic Inc., Technical Report 99, May 10, 1994.

[SC96] Sakthi Subramanian and Jeffrey V. Cook. Mechanical Verification of C Programs, Proceedings of the ACM SIGSOFT Workshop on Formal Methods in Software Practice, January, 1996.

EVES

[Cra92] Dan Craigen. An Introduction to EVES, Its Motivations, and Its Evolution, ORA Canada TR-92-5440-09, March 1992.

[Cra94] Dan Craigen. Formal Methods, EVES, and Safety Critical Systems, ORA Canada FR-94-5479-04, January 1994.

[Cra95] Dan Craigen. Application of Reverse Engineering Software, ORA Canada

FR-95-5476-02, March 1995.

[CS96] Dan Craigen and Mark Saaltink. Using EVES to Analyze Authentication Protocols, ORA Canada TR-96-5508-05, March 1996.

[Kro96] Sentot Kromodimoeljo. The One Way Link System/Subsystem Specification, ORA Canada TR-96-5508-03

[ORAweb] Information on the EVES theorem prover at ORA at: <http://ora.on.ca/eves.html>, 1997.

[Saa95] Mark Saaltink. The Z/EVES system, ORA Canada, Sept. 1, 1995

HOL

[AAUweb] Information on the research being done with HOL to build a secure tool for interactive refinement of sequential and parallel programs at Åbo Akademi University in Finland can be found at: <http://www.abo.fi/~mbutler/pmg/mech.html>, 1997.

[AFL92] J. Alves-Foss and K. Levitt, Mechanical Verification of Secure Distributed Systems in Higher Order Logic, in Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, August 1991, edited by M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley (IEEE Computer Society Press, 1992), pp. 263-278.

[AGMT95] Stuart Aitken, Philip Gray, Tom Melham and Muffy Thomas, Interactive Proof Discovery: An Empirical Study of HOL Users. User Interface Design for Theorem Proving Systems: an International Workshop, Glasgow, July 1995.

[Ang94] C. M. Angelo. Formal Hardware Verification in a Silicon Compilation Environment by means of Theorem Proving, Ph.D. Dissertation, Katholieke Universiteit Leuven (February, 1994).

[AVCD91] C. M. Angelo, D. Verkest, L. Claesen, and H. De Man. Formal Hardware Verification in HOL and in Boyer-Moore: A Comparative Analysis, in Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, August 1991, edited by M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley (IEEE Computer Society Press, 1992), pp. 340-347.

[Cam90] A. J. Camilleri, Reasoning in CSP via the HOL Theorem Prover, in Next Decade in Information Technology: Proceedings of the 5th Jerusalem Conference on Information Technology, Israel, 22-25 October 1990, (IEEE

Computer Society Press, 1990), pp. 173-183.

[COl92] R. M. Cardell-Oliver, The Formal Verification of Hard Real-Time Systems, Ph.D. Dissertation, Technical Report Number 255, University of Cambridge Computer Laboratory (May 1992).

[Cur94] P. Curzon. The Formal Verification of an ATM Network, in Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (ACM Press, 1994), p. 392.

[Cur95] P. Curzon, Problems Encountered in the Machine-assisted Proof of Hardware. In Correct Hardware Design and Verification Methods, Eds. Paul E. Camurati and Hans Eveking, pp. 56-70, Springer-Verlag, 1995.

[FK95] J. Frößl and T. Kropf. Verifying Real-Time Properties of MOS-Transistor Circuits, in Proceeding of the European Design and Test Conference (EDTC), (Paris), pp. 314-319, IEEE Computer Society Press, March 1995.

[GLAweb] Information on the research being done with HOL and User interface design for mechanized theorem proving can be found at: <http://www.dcs.glasgow.ac.uk/research/fmt/projects/user.html>, 1997.

[HOLweb] Information on the HOL system can be found at the Cambridge University Computer Laboratory HOL page at: <http://www.cl.cam.ac.uk/Research/HVG/HOL/>, 1997.

[HVGweb] Information on the research being done in the Hardware Verification Group at the University of Karlsruhe in Germany is available at: <http://goethe.ira.uka.de/hvg/research.html>, 1997

[Kal93] HOL Around the World ‘93 compiled by Sara Kalvala. August 1993. Available from the Automated Reasoning Group at the University of Cambridge. <http://www.cl.cam.ac.uk/Research/HVG>

[TK94] S. Tahar and R. Kumar. Implementational Issues for Verifying RISC-Pipeline Conflicts in HOL, in Higher Order Logic Theorem Proving and Its Applications: 7th International Workshop, Valletta, Malta, September 1994: Proceedings, edited by T. F. Melham and J. Camilleri, Lecture Notes in Computer Science, Volume 859 (Springer-Verlag, 1994), pp. 424-439.

[Win90] Phillip J. Windley. The Verification of AVM-1, Division of Computer Science, The University of California, Davis, Research Report CSE-90-21, July, 1990.

[ZSH94] C. Zhang, R. Shaw, M. R. Heckman, G. D. Benson, M. Archer, K. Levitt, and R. A. Olsson, Towards a Formal Verification of a Secure Distributed

System and its Applications, in Supplementary Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and its Applications, edited by T. Melham and J. Camilleri, University of Malta (Valletta, 1994).

Larch

[LSGL94] Victor Luchangco, Ekrem Soylemez, Stephen Garland and Nancy Lynch. Verifying Timing Properties of Concurrent Algorithms, Proceedings of the Seventh International Conference on Formal Description Techniques for Distributed Systems (FORTE94), Chapman and Hall.

[MITweb] Information on the Larch Project can be found at MIT's Larch web page at: <http://larch-www.lcs.mit.edu:8001/larch/index.html>, 1997.

[SGHG93] James B. Saxe, John V. Guttag, James J. Horning, and Stephen J. Garland. Using transformations and verification in circuit design, Formal Methods in System Design Vol. 3, No. 3, December 1993, pages 181-209.

Criteria for Evaluating Theorem Provers

[AVCD93] C.M. Angelo, D. Verkest, L. Claesen and H. Deman. On the Comparison of HOL and Boyer-Moore for Formal Hardware Verification. Formal Methods in System Design, vol. 2, no. 1 (February 1993), pp. 45-72.

[KA96] D.J. King and R.D. Arthan. Development of Practical Verification Tools. Published in The ICL Systems Journal. Volume 11. Issue 1. May 1996.

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

[Rus95] John Rushby. Formal Methods and their Role in the Certification of Critical Systems. Technical Report CSL-95-1, Computer Science Laboratory, SRI International, March 1995.

Software Engineering

[Cha89] R.N. Charette. Software Engineering Risk Analysis and Management. McGraw-Hill/Intertext, 1989.

[Pre92] Roger S. Pressman. Software Engineering A Preactitioner's Approach Third Edition. McGraw-Hill, Inc. New York 1992.

[DGNK97] Colleen DeJong, Matthew Gibble, Luís Nakano and John Knight. Formal specification: A systematic evaluation. Department of Computer Science, The University of Virginia. Computer Science Report No. CS-97-09, May 1997.

University of Virginia Nuclear Reactor

[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

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
```