APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Computer Science)

[illegible]

Author: Kevin G. Wika

This dissertation has been read and approved by the Examining Committee:

[illegible]

Dissertation Advisor: John C. Knight

[illegible]

Committee Chairman: Alfred C. Weaver

[illegible]

Committee Member: Jack W. Davidson

[illegible]

Committee Member: George T. Gillies

[illegible]

Committee Member: Andrew S. Grimshaw

Accepted for the School of Engineering and Applied Science:

[illegible]

Dean, School of Engineering and Applied Science

May 1995

Safety Kernel Enforcement of Software Safety Policies

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Kevin G. Wika

May 1995

Abstract

Computing systems in which the consequences of failure are very serious are termed *safety-critical*. Many such systems exist in application areas such as aerospace, defense, transportation, power-generation, and medicine. The software in these systems is typically large and complex, critical to system safety, and difficult to implement and verify. Even when great effort is expended to develop the software, there is no assurance that the software will operate with the required level of dependability.

We have investigated a *safety kernel* architecture that addresses part of the problem of building and verifying dependable safety-critical software. An analogous construct, the *security kernel*, has been used successfully to enforce *security policies* in classified-information systems. Similar requirements known as *safety policies* must be enforced in safety-critical systems. Other researchers have developed some basic safety kernel concepts and have proposed safety kernel designs. However, many feasibility issues have not been addressed previously. Thus, the focus of this research has been the evaluation and development of the safety kernel as a software architecture for enforcement of safety policies.

We have evaluated the feasibility of the safety kernel in four areas: policy enforcement, reliable enforcement, implementation, and verification. The first area addresses the role of the safety kernel and assesses its support for safety-critical systems. The second, area examines the requirements for reliable policy enforcement by the safety kernel. The third area focuses on the feasibility of a reuse-oriented implementation strategy. The fourth area considers the verification of the safety kernel. Work in each of these areas has been supported by our involvement with two case studies: the Magnetic Stereotaxis System and the University of Virginia Reactor.

The results presented in this dissertation demonstrate that it is feasible for the safety kernel to enforce a significant set of safety policies — policies that are directly related to device operation. Furthermore, operating in the system context, it can enforce policies reliably in spite of certain component failures. We demonstrate that a special-purpose specification language can be used to describe the safety kernel and that a source code representation of the safety kernel can be mechanically generated from this policy specification. Finally, we define the issues in verification of the safety kernel and demonstrate the feasibility of several analysis and testing techniques.

Acknowledgments

This work was supported in part by the National Science Foundation under grant number CCR-9213427, and in part by NASA under grant number NAG1-1123-FDP.

Table of Contents

1 Introduction

Computing systems in which the consequences of failure are very serious are termed *safety-critical*. Many such systems exist in application areas such as aerospace, defense, transportation, power-generation, and medicine. Public exposure to these safety-critical systems is increasing rapidly. Since the correct operation of these systems depends on software, the possibility of serious damage resulting from a software defect is considerable and growing.

The software present in safety-critical systems is frequently very large and tremendously complex. The large size and complexity can be attributed to the functionality demanded by modern applications. Functionality requirements have increased because of the many benefits of computer-based control and the availability of inexpensive yet powerful computing hardware. Hardware performance limits that formerly restricted software complexity are rarely reached because of the remarkable hardware performance now available.

Experience with safety-critical systems has shown that significant software defects tend to remain in such systems after deployment despite extensive effort on the part of the developers [16,34,41]. Building these systems to perform as desired is very difficult for a number of reasons. Even the best software development processes cannot ensure that faults are avoided completely during development. Similarly, fault detection techniques are imperfect. Research has shown, for example, that testing as an approach to verification cannot demonstrate sufficient levels of reliability because of the sheer number of tests that are required [9].

Building very small, simple software systems that achieve the extreme dependability necessary with safety-critical applications has proven to be sufficiently challenging. The complexity of large systems involving characteristics such as real-time operation and distributed processing is likely to preclude any significant assurance that the systems meet desired dependability goals if traditional techniques are used in traditional ways. The possibility, for example, of being able to test adequately a system that is comprised of upwards of 500,000 lines of source code, that executes on a network, that has sophisticated graphical operator displays, and that performs some form of real-time control seems remote at best. It is important when contemplating such an example, to keep in mind that defects in compilers and system support tools are also an issue, and that the operating system and network implementation must be viewed as part of the application. It does not matter which part is responsible when something breaks.

Although formal techniques have made substantial progress and have been applied to real systems in a number of cases, their application to large, complex systems remains elusive. It is certainly possible to demonstrate useful properties of large systems using formal techniques. For example, by using careful system design and appropriate proof techniques, it is possible to show that a concurrent system is free of deadlock. However, although this is an extremely valuable property, deadlock is just one class of fault. There are, of course, many others and, to meet typical statistical dependability goals, all faults must be eliminated or have suitably low probabilities of manifestation.

The fact that a significant number of safety-critical systems operate much of the time without serious mishap could be presented as evidence that it is possible to build systems that operate safely. This apparent success is a credit to careful engineering and commitment of vast resources to develop these systems, rather than to any techniques which can provide assurance that these systems are in fact safe. However, even if the argument is advanced that present techniques are adequate for building systems that operate safely, it is certainly the case that it would be desirable to be able to develop complex systems more cost effectively.

The goal of the research described here is to investigate the feasibility of a new approach to dealing with the situation outlined above. The premise is that no techniques exist which can routinely show that a large, complex software system is sufficiently dependable for use in a safety-critical application. We restrict our attention to systems where safety is the overriding concern, i.e., systems in which reduced service or no service is acceptable following a software error. Our approach is to assume that faults remain in the application software and to try to deal with them at execution time rather than attempting to eliminate all faults during development. The mechanism that we describe to implement this approach is a software architecture termed a *safety kernel*, a concept directly analogous to the security kernel used in security applications.

Security kernels have been covered extensively in the literature and have been implemented with a number of systems [2,15,22,45]. Safety kernels, on the other hand, have been proposed by a number of groups [33,38,44], but the development of the safety kernel concept has been limited and to the best of our knowledge, none of the proposed systems has been implemented. Given the relative novelty of the idea, the goal of this research is to develop the safety kernel concept and evaluate its feasibility in four critical areas:

- investigating the role of the safety kernel as an enforcer of safety policies,
- analyzing requirements for reliable kernel enforcement of safety policies,
- developing an implementation strategy, and
- evaluating techniques for verification of the safety kernel implementation.

Each of these areas is covered in depth in the dissertation. A central goal of our research has been to develop ideas and mechanisms that address the needs of real safety-critical systems. Toward that end, we have based our research on two case studies. The first of these case studies is the Magnetic Stereotaxis System (MSS), an experimental neurosurgical device. The second case study is the University of Virginia Research Reactor (UVAR). These applications are described in detail in the dissertation. Working with these systems has forced us to address the requirements of these two complex systems and it has also provided a context for evaluation of research ideas, processes, and tools. The case studies are not case studies in the sense of finished systems that are analyzed for strengths and weaknesses. Instead, we are actually building the systems as a means of advancing the technology employed in the development of software for these systems. A part of this effort has been the implementation of a safety kernel prototype for the MSS.

Although evaluation is often considered to be a phase applied to a research product, with the development of a concept the evaluation occurs from the initial stages through to the final product. In fact, evaluation of ideas with respect to the two case studies was iterative and continual, and ideas changed repeatedly in response to this process.

The next section of the introduction examines some of the basics of the safety kernel concept. This is followed by a discussion of the issues that have been considered in evaluating feasibility in each of the four areas. The balance of the introduction examines each of the four evaluation areas summarizing major issues in each area.

1.1 Kernels for Security and Safety

The notion of a safety kernel derives from the concept of a *security kernel* — a technique developed extensively by the security community. Informally, the goal of a security kernel is to provide assurance that a set of required fundamental properties of a computer system hold at all times during execution [2]. These properties are specified as *security policies* and are enforced by the security kernel independent of the application program. In other words, verification of the security kernel is sufficient to ensure enforcement of those policies encapsulated within the security kernel. The application program need not enforce the security policies, and it can, in fact, undertake actions that would normally lead to violation of the security policies with no danger of actual violations taking place. The result is that adherence to critical security policies can be assured by analysis of the relatively simple kernel rather than from analysis of a complex application program. This has the additional benefit of simplifying application programs by freeing them from responsibility for implementation and verification of policies that are enforced by a kernel. The general concept of a security kernel is shown in Fig. 1.

The similarity between security concerns and safety concerns is considerable [4]. *Security kernels* are used to enforce access-control policies in classified information systems. The idea of trying to exploit this technique to implement safety rather than security, i.e., the concept of a more general *safety kernel*, was proposed by Rushby [33,44], among others. A security kernel (sometimes referred to as a *reference monitor*) is in a position to enforce security policies because it controls all access to secure information and it can therefore monitor all references to that information. A safety kernel will exercise similar control over the devices in a safety-critical system and will enforce a set of *safety policies* by monitoring requests to devices, device actions, device status, application software status, and so on. The safety policies for a given application are derived from the software safety specification. Using a kernel architecture to ensure compliance with safety policies is attractive largely because of the complexity of modern safety-critical applications alluded to above. As with a security kernel, the rationale is that compliance with safety policies can be assured mostly by analysis of a relatively simple safety kernel rather than a large and complex application program.

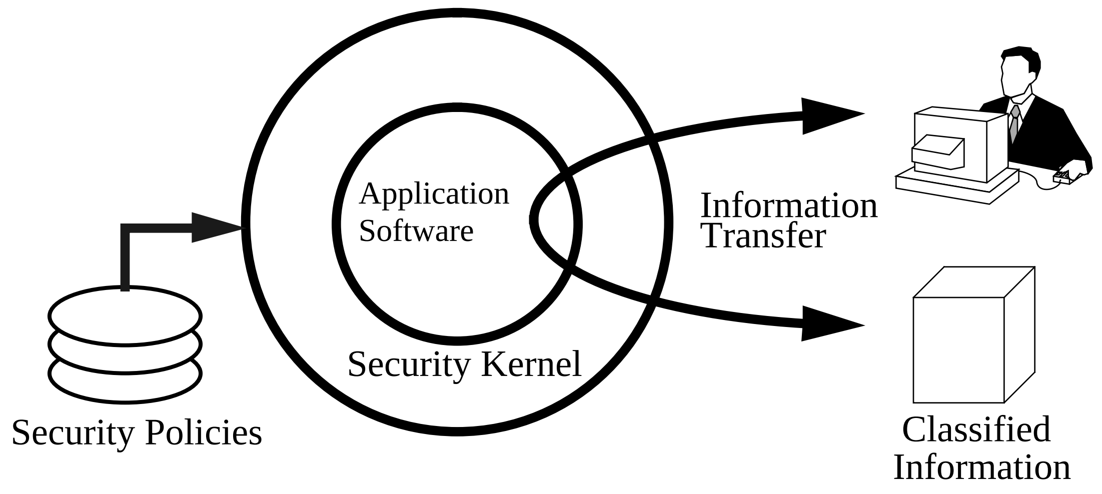

Fig. 1. Security kernel concept.

The idea that Rushby suggested is different from other architectures described as safety kernels because certain essential safety policies are enforced regardless of the actions of the application software. This is in direct analogy with security kernels that enforce access control with a similar degree of generality. Other safety-kernel architectures that have been developed tend to provide a set of services that enforce required safety policies, *if used appropriately by the application*. This is a critical distinction.

The term “kernel” as used here refers to a *policy enforcement kernel* as opposed to a traditional system kernel. In some situations (e.g., some security kernels), an enforcement kernel is also a system kernel. However, this is not necessarily essential to the success of the kernel as a policy enforcer.

Fig. 2 illustrates the safety kernel concept. The kernel is situated between the application software and the application devices. From this position, it is able to mediate all exchanges between the software and the devices and has the following benefits to a safety-critical system:

1. *Ensured enforcement of safety policies*
   The kernel structure enforces a critical set of safety policies regardless of the implementation, modification, or verification of the application software.
2. *Simplicity and verifiability of the safety kernel structure*
   The kernel is a relatively small structure and as a result facilitates the implementation and verification of safety policies that it enforces.
3. *Simplification of the application software*
   Kernel enforcement of selected safety policies frees the application software from responsibility for implementation and verification of these policies.
4. *Kernel control of devices*
   Acting as a reference monitor, the kernel is ideally situated to enforce device

control policies. Its access to devices also permits the kernel to monitor device activities for consistency with software commands. It is only acting through devices that software can cause a mishap, so it is significant and important to enforce policies governing operation of devices.

5. *Reuse of general functionality*
   The kernel architecture provides a set of general mechanisms and a framework for the abstraction of general classes of safety policies.

1.2 Feasibility Issues

Assessment of the feasibility of an idea or mechanism, depends on establishing feasibility in several areas. The first area is *technical feasibility.* This area addresses the fundamental question of concept validity and implementability. Obviously this is the first issue to be addressed in the evaluation of any product. Beyond the strict metric of technical feasibility there are a set of pragmatic concerns that we have used to guide evaluation. Discussed below, these concerns derive from the goal of producing research results that are applicable to real systems. Note that these concerns are used both in the evaluation of products, but also in choosing between design alternatives, implementation strategies, etc.

- Quality assurance
  The goal of the safety kernel is to provide for a set of safety policies in a relatively small, simple component. The intent is that this architecture would result in more dependable enforcement of the set of safety policies. In order to realize an improvement in dependability, it is essential to evaluate the safety kernel concept and design choices with respect to their impact on quality assurance. Enforcing a particular safety policy in the safety kernel simplifies the quality assurance necessary for the application. However, at the same time, it can increase the difficulty of the verification of the safety kernel.

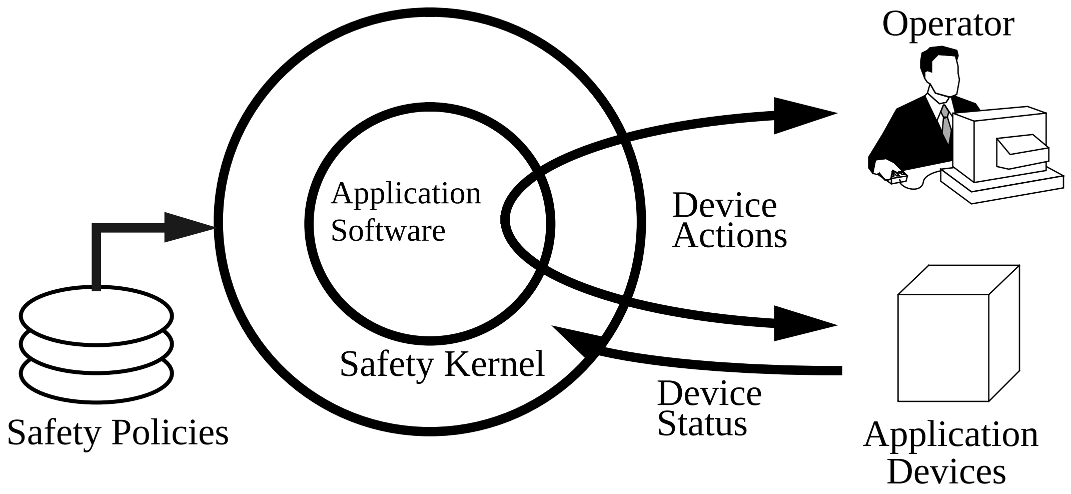

Fig. 2. Safety kernel concept.

- Cost
  In addition, to quality assurance, there are issues related to the cost-effectiveness of the safety kernel concept and design. Issues include the effort required to develop a system, the portability of the system, the dependability provided by the design, and the dependability required by an application or class of applications. As the safety kernel concept and design are evaluated, the anticipated impact on the cost of a safety-critical application will be a significant consideration.

- Functional performance
  A goal of the safety kernel is that it should be compatible with a range of real safety-critical systems. This has implications for the functional performance of the safety kernel in regards to its compatibility with various application architectures and the possible performance overhead it might impose. Although there are no firm requirements in either area, a knowledge of the needs of existing systems (e.g., the MSS and the UVAR) can be used to promote safety kernel performance and general compatibility with a range of applications.

There are several points to stress regarding the above concerns. First, they are not specific requirements, but rather guidelines for evaluation. Second, it is not possible in the abstract to directly measure any of these characteristics. However, it is possible to at least roughly evaluate a concept or mechanism and place it along a continuum. Third, when making an evaluation there will be trade-offs between each of these areas. For example, a particular design might be amenable to extensive formal verification, but the cost to implement it might be prohibitive or its performance might be unacceptable. Ultimately, choices need to be made based on the specific requirements of a system being developed. Since we are developing a general concept, our goal is to outline what the choices are and to identify the issues and trade-offs.

1.3 Safety Policies

The safety kernel will enforce a set of safety policies, i.e., safety requirements, for an application. It is important to emphasize that for most systems the safety kernel is intended to enforce a *subset* of the safety policies for the application. Not all safety policies are suitable for enforcement by a safety kernel. As a result, several critical questions must be addressed in evaluating the contribution and role of the safety kernel. What proportion of the safety policies for an application might the safety kernel enforce? What is the relative importance of these kernel-enforced policies? Are there general classes of kernel-enforced safety policies?

The initial step to address these questions was to itemize the safety policies for the two case study applications. These policies were identified using techniques such as hazard analysis, fault tree analysis, and failure modes and effects analysis. For purposes of defining and clarifying the role of the safety kernel, it would be desirable to be able to classify the safety policies. As a part of this research, classes of safety policies have been identified based on their position on a canonical system fault tree.

Of the classes of safety policies, not all are equally suited for safety kernel enforcement. Informally, the safety policies that are enforced by the safety kernel are generally those that are closely tied to device operation and therefore have the most direct impact on safe operation of a system. Selection is based on considerations of the impact kernel enforcement has on the dependability and complexity of the safety kernel and of the system as a whole. Some policies that are not selected for complete kernel enforcement can nevertheless be enforced in a modified form as *weakened* safety policies. With a weakened safety policy, some aspect of the original safety policy is enforced by the safety kernel, but the application software is also partially responsible for enforcement of the safety policy. Enforcing a weakened safety policy is beneficial when its enforcement supports the demonstration of the dependability of the original policy.

1.4 Reliable Policy Enforcement

To be viable the safety kernel must be able to enforce safety policies reliably. That is, the probability that the safety kernel enforces its safety policies for a specified period of time, must be acceptably high. It must do so in the system context in spite of failures of devices, application software, support software, computer hardware, and so on. To establish a set of requirements for reliable kernel enforcement of safety policies, the effect of these failures on policy enforcement has been analyzed. The resulting requirements have been evaluated to assess the feasibility of reliable kernel enforcement. Techniques for meeting the various requirements have also been identified.

Utilizing the various techniques, there are many system designs that could meet the requirements for reliable enforcement of safety policies. We describe and evaluate several different designs. The selection of the best design for a particular system depends on the specific requirements for that system. Working with the reliability requirements and basic guidelines concerning the pragmatic concerns, a system design is described that is targeted for the MSS and UVAR applications.

1.5 Implementation Strategy

The basic safety kernel concept is to enforce a set of safety policies with a module that is relatively small and simple compared to the complete software system. The rationale is that this module is more amenable to verification than the complete application software and therefore can enforce policies more reliably. The success of this idea does *not* depend on the implementation techniques by which the safety kernel is realized as long as it is possible to demonstrate sufficient reliability. However, we have investigated alternatives for the implementation of the safety kernel because of the potential benefits in the areas of cost and reliability. More specifically, the existence of general classes of safety policies and experience with a variety of applications has lead us to investigate the possibility that the safety kernel architecture permits the use of a general framework that can be exploited to support abstraction of general knowledge, system design, safety policies, and software artifacts. Such a reuse-oriented framework would promote the transfer of innovative and effective concepts and artifacts from one system to another.

Reuse can be applied at many levels from abstract general system knowledge to very specific software artifacts [43]. The appropriate level of reuse for a class of systems depends on the commonalities between systems and the identification of general requirements or characteristics. Owing to the unique safety and system hardware concerns, it is obvious that a single safety kernel will not meet the needs of all systems. An alternative is the development of a safety kernel that is parameterized to permit customization for specific applications. This level of reuse would permit exploitation of established mechanisms, but it is likely that only a small class of systems could utilize a common mechanism. Another reuse option is to reuse a more abstract product such as a design. This has the potential for being applicable to a greater number of systems while still supporting the goals of reducing cost and facilitating quality assurance.

It is instructive to note that design reuse is precisely the level of reuse that occurs with security kernels. A security kernel is tailored for the processor on which it executes, for the devices it must control, and for the means by which security will be ensured in a given situation. Certainly, the notion of an access matrix is quite general; however, interfacing with devices, providing facilities for administration of the system, and dealing with other security issues (e.g., covert channels) are general problems that have solutions configured for each unique instance. In spite of this application dependence, security kernels are utilized because they provide the same benefits cited above for the safety kernel.

We have developed a system that generates an instance of the safety kernel from an application-specific, safety policy specification. The policy specification characterizes the system and component devices and specifies the safety policies that will be enforced. The combination of the information characterizing the system and information and mechanisms provided by the generator provide a context for the expression of safety policies. Policies governing application requests to the safety kernel, monitoring of devices, and response to failures are specified in this context.

A safety policy specification has been developed for the MSS and the safety kernel generator has been used to produce an operating safety kernel for the MSS. A partial safety policy specification for the UVAR has also been documented to evaluate the suitability of the implementation strategy for an application that is quite different from the MSS.

1.6 Verification

The final area of evaluation is the critical area of verification. How do we ensure that the safety kernel as implemented enforces the specified safety policies? At present there are no verification techniques that can routinely demonstrate the dependability of a system like the safety kernel. We have examined some of the potential techniques and have evaluated the role that they can play in the verification of the safety kernel. Particular emphasis has been placed on the techniques of formal verification and testing.

1.7 Contents Summary

The next chapter looks at related work that has been done in the areas of safety-critical applications, security kernels and safety kernels. The following chapter describes the two case studies. Chapter 4 examines the safety policies for the two case studies to develop a taxonomy of safety policies and identify classes of safety policies for kernel enforcement. Requirements for reliable kernel enforcement of safety policies are developed in the next chapter along with system designs for meeting the reliability requirements. An implementation strategy for the safety kernel is investigated and evaluated in Chapter 6. Chapter 7 evaluates various verification techniques for the safety kernel. The safety kernel prototype for the MSS is described in Chapter 8, along with the safety policy specification for the UVAR. Finally, a summary evaluation and topics for future research are presented in the conclusions.

2 Related Research

Previous research related to safety-critical software has focused on topics such as defining software safety, identifying methods for compiling requirements specifications, improving the technology for developing an application from a set of specifications, and verifying that an application meets its specification. In general, the research could be characterized as an attempt to develop a reliable, repeatable process for producing safe software. In addition to research of fundamental questions, considerable work has been done in industry and to a lesser degree in the research community to produce a range of operational safety-critical systems. Common domains for these applications are avionics, air traffic control, medicine and transportation. Finally, there is a collection of work that is not directly related to safety (e.g., in the area of security) but which is relevant to the proposed research. Previous work in these areas is reviewed in this chapter.

2.1 Building Dependable Software

Dependability is defined as that property of a computing system which allows reliance to be justifiably placed on the service it delivers [29]. For software, the informal notion that most people have is “Will the software perform as I wish?”. This question is not in the least precise nor testable, and a clear statement of exactly what goal has to be met by software developers is necessary. Without such a definition, the software, once built, might not possess the properties required for dependable operation of a target system.

When considered more carefully, it is clear that dependability has many different meanings. The three most important are *reliability, availability,* and *safety*:

- *Reliability*
  Reliability is the probability that a particular device will function as required in a specified environment for a particular period of time [46]. The notion of reliability is important for devices that must provide continuous service. For example, even a momentary failure of some implanted therapeutic devices is likely to have very serious consequences.

- *Availability*
  Availability is the probability that a particular device will be able to provide service at a particular time [46]. The notion of availability is important for devices that provide service where brief outages are acceptable; for some devices, even frequent outages will not cause harm provided they are very brief. Over a given period of time, however, a device that must achieve high levels of availability is required to be operational for some very large fraction of the observed time period. Interestingly, a device might have high availability yet poor reliability. For example, a patient-monitoring device that fails on average once per hour but is restarted auto-

matically in less than a second is probably in this category. For such applications, this might be perfectly acceptable.

- *Safety*
  Safety is the property that a device will not cause harm by operating incorrectly. This does not mean necessarily that such a device provides a useful service, merely that it does not cause harm. For example, a radiation therapy device that is always shut down is safe though not very useful. The goal of safety engineering is to produce useful devices that are also safe. Notice that a particular device may be useful and safe yet have low availability and reliability.

A *system failure* is said to have occurred when the system no longer complies with its specification [5]. This is an important definition because it illustrates the dependence placed on the specification. If nothing is specified about how a system is to behave under certain operating conditions then any behavior must be considered acceptable if those conditions arise. Similarly, a specification must state what is required in terms of reliability, availability, and safety, and these requirements must be technically reasonable.

The term *software safety* is used frequently concerning software in safety-critical systems [31]. It is generally accepted that because software acts through the other components in a system, the process of building software must focus on the role of the software in the system. However, there should be a specification of the requirements that the software must meet to operate safely in the system. It is this *software safety specification* that the software must meet and it is this specification that establishes the standard for determining the safety of software. If the system safety analysis does not identify a critical software safety requirement and an accident results during system operation, this is a case of *specification error* and not an example of unsafe software [25].

2.2 Safety-Critical Applications

Various features that could be included in a safety kernel have been built into almost every safety-critical system utilizing software. Common techniques include watchdog timers, input and output assertions [32], sequencing checkers [44], fault tolerant data structures [48], software isolation [1], and software self checking [20]. These techniques have been incorporated largely in an ad hoc fashion. Some of the systems that are presently the state of art in this area are described below.

The control of an electric generation turbine is a safety critical task in that failure to control the speed (i.e., the flow of steam) can lead to a very expensive overspeed breakup. Therefore, there is a need to ensure that steam valves will never open spuriously. In addition, the possibility of load rejection and the accompanying rapid increase in the speed of the turbine requires that valves can be closed within a few hundred milliseconds. For economic reasons, availability is also a significant system goal.

J. C. Higgs describes the design of a software-based electric turbine governing system known as MICROGOVERNOR that has several novel features [20]. The author cites the following:

1. Software assertions are utilized in order to obtain integrity through intelligent self checking rather than through duplex comparison. Failure of the self checks results in fail-safe hardware ensuring the safety of the system outputs.
2. The relatively low cost of microprocessor hardware allows a very simple distributed processing structure to be employed in which a complete, self-contained, governing channel is dedicated to each controlled steam valve path.
3. A two-level structure is used consisting of a high integrity base level and an upper level for less critical functionality. The base level contains the high integrity, high availability governing functions. The upper level consists of less important governing functions along with supervisory, coordination and management functions.

The system relies on the core software of the base level to provide the governing functions, but also to detect failure of the hardware around it. To deal with potential software failures, the system utilizes a watchdog timer that provides the system with a fail-safe means to exit the software core. Should it expire, the timer closes the steam valves. Finally, the system incorporates a state table that defines processing routes at the base level and also determines the appropriate self checks based on the present state of the turbine. This arrangement enables precise timing constraints to be applied to critical sections of the base level code. This system appears to be well formulated making good use of hardware and software strengths, and taking advantage of the characteristics of a turbine governing system.

R. H. Taylor et al. [49,50] have developed a robotic system designed to aid in hip replacement operations by performing the machining necessary for the insertion of a cementless hip implant. To ensure safe operation of the robot, a range of techniques is applied. Some basic measures include: position and velocity deadbands in the joint servos, monitoring of selected robot activities by a separate computer, and a safety time-out monitor. A force sensor is mounted on the cutter to detect excessive forces on the patient. In addition, either the controller or the surgeon can disable the system in the event of some exceptional condition. The most innovative measure is an independent motion monitoring system that runs on a separate computer. This system is intended to ensure that the cutter will stay within a specified volume relative to the bone by tracking both the motion of the bone and the motion of the robot end effector.

The applications described above employ important safety measures and could likely be shown to be quite safe in the informal sense. However, because they utilize ad hoc techniques and are uniquely constructed for a particular application, they do not significantly contribute to a general solution for developing safety-critical systems. In addition, the lack of formality in their development makes it difficult to state precisely what these systems are supposed to do - rendering it impossible to argue that the implementation provides for the safety of the application.

The problems of lack of formality, the use of ad hoc measures, and custom-built systems have been common in many areas of computer science. In a few cases, however, an area has managed to develop general, reusable solutions for basic problems. The area of computer security is the most relevant, because of certain similarities between security and safety.

2.3 Security Kernels

The security community has historically faced many of the same problems presently being encountered within the field of safety. Early security systems were ad hoc, unique systems that had no formal approach to ensuring security. The result was that the systems were very difficult to build and, once built, it was almost impossible to verify whether the required security was guaranteed. Over time however, concepts have been developed that have made the development of secure systems more general, repeatable, and more amenable to verification.

One technique employed in developing secure systems is based on the use a security kernel [2,15,22,45]. The concept behind a security kernel is to enforce basic security policies using a relatively simple central mechanism. The typical security problem is to monitor the access of users to objects (typically information). The security kernel approach is to require all references to objects to be carried out through the kernel. The kernel then checks each reference to ensure that it conforms to the specified security policy for the system. With this exclusive control of access to objects and an assumption that the kernel must be isolated to prevent modification, the security of the system can be ensured by verifying that the security kernel correctly implements the specified security policy. As a result, there is no need to ensure the trustworthiness of the software executing on top of the kernel.

The security community has also recognized that a system is only secure with respect to its security policy. There is no attempt to build a system that is secure in an informal sense - where secure would connote preventing any compromise of security. The safety community would benefit from a similarly formal approach of building systems to ensure a specified safety policy instead of striving for some informal notion of safety.

It should be noted that security kernels are not a panacea. Especially as they have been incorporated into distributed systems, there are many security concerns that cannot be addressed by the kernel architecture. In systems of this type, security requires not only preclusion of a set of harmful actions, but also successful performance of many actions (e.g., the reliable delivery of a message). In a security system, there is not much to be gained by enforcing a few security policies while leaving others unenforced. In a safety system however, enforcing a subset of safety policies has potential benefit.

2.4 Previous Research on Safety Kernels

The development of concepts such as the security kernel have not gone unnoticed in the safety community. Several authors have used the term “safety kernel” for systems or concepts that had the goal of supporting safe operation of application software. Others have suggested schemes (known by other names) that have some of the features of a safety kernel. These systems typically have some of the features of a security kernel with a relatively small component providing some form of support for software safety.

Leveson et al. [33] appear to be the first to have used the term “safety kernel.” They describe a concept based on a centralized location for a set of safety mechanisms. These mechanisms are used to enforce usage policies that are established for a given system. The policies detail how error detection and recovery will be carried out for the system.

The kernel described by Leveson provides mechanisms for detection of errors and recovery from errors. The two detection mechanisms are based on assertions and timers. Four mechanisms are provided for recovery: change-schedule, init-module, consult-error-history, and trace. The first two are used for reconfiguring the system by altering the execution of modules, and the second two aid in the selection of a recovery action by a human operator. The actions of the mechanisms are determined by policies that are provided to the kernel. These policies differ from those used in security kernels in that security policies are more global (describing what) while those used here specify responses to particular errors (describing how). The distinction is made to demonstrate that the term kernel, as used here, is not directly analogous to its use in the security context.

Neumann [40] considers the idea of a safety-trusted computing base as a part of his examination of whether the hierarchical design familiar in secure systems could be generalized to other critical applications. In describing a hierarchical design approach Neumann relies on the standard “uses” relation [42], but also introduces the notion of associating degrees of criticality with the design levels. Degrees of criticality are applied in secure systems with the most critical component, the security kernel, occupying the lowest level. Higher levels representing lower criticality include trusted servers, subsystem interfaces, end-user interfaces and finally user programs. A safety hierarchy could also incorporate criticality using, for example, the degrees of fail-operational, fail-soft, fail-safe, fail-stop-safe, and fail-unsafe (although there may not be as distinct a relation to the design levels in this case). A trusted computing base attempts to capture the critical aspects of a system and focus efforts on ensuring their implementation. A major concern with the safety-trusted computing base is that much of an application might be critical, meaning that safe operation will depend on a significant portion of the application software. This is in contrast to security systems where a relatively concise policy can be formulated and subsequently guaranteed by the security kernel and trusted functions.

Rushby has made the strongest theoretical argument for the development of a safety kernel [44]. In the process, he has more clearly defined the role of a safety kernel and addressed the concern raised by Neumann. Rushby considers whether the concept of a small component that guarantees the enforcement of some system policy (typically security) could be applied to safety-critical software systems. The observation is made that kernel structures are potentially applicable for the enforcement of properties where the following two conditions hold:

1. the properties of interest at the system level must be present at the kernel level, and
2. those properties must be expressed by a second-order assertion of the form

$$\forall \alpha \in op^{*} : P(\alpha)$$

This second-order assertion states that for any combination of operations, α, in the set $op^*$where op is the set of all functions provided by the kernel (i.e., the first condition), the predicate P over the input/output behavior of that set will hold. Informally, the assertion says that the kernel can guarantee P provided that every operation that can be performed ultimately is effected through calls to kernel functions, and also that the kernel itself cannot be modified. For example, if P specified a maximum value for a device control signal, the kernel could ensure that no greater values were sent by having exclusive access to the device and providing only operations that would control the device below the maximum value. This is precisely the situation with a security kernel, where all access to information occurs via the security kernel, permitting the enforcement of security policies (e.g., information with a given security level should never be given to an object with a lower security level).

Second-order assertions define conditions that should always hold and are particularly well suited to describing actions that should never occur (negative properties). Rushby contends that kernels can exert control over the occurrence of “bad behaviors” via the functions that they do provide. Enforcement of positive behaviors is much more doubtful because it is difficult to ensure the proper use of functions that are provided. Positive behaviors can generally be described using first order logic and can thus be verified by demonstrating conformance to pre- and post-conditions. Rushby concludes by offering a potential (although quite limited) design for a safety kernel. The design is based on a separation kernel that restricts and monitors communication between modules and resource managers that are responsible for the safe operation of devices. Communications are monitored to enforce a required status as specified in a “policy base.” In particular, Rushby mentions the potential for enforcing policies that specify valid sequences of operations.

A report by the NATO ad hoc Working Group on Munition Related Safety Critical Computing Systems [39] mentions a safety kernel that it defines as follows:

> Safety Kernel: An independent computer program that monitors the state of the system to determine when potentially unsafe system states occur or when transitions to potentially unsafe system states may occur. The Safety Kernel is designed to prevent the system from entering the unsafe state and return it to a known safe state.

The report also details other safety requirements and provides a fairly comprehensive list of the ad hoc techniques that are commonly applied to safety-critical systems.

More recently, Moffett et al. [38] have proposed a concept for enforcement of policies in distributed systems. They identify two types of policies. Obligation policies describe actions that must be initiated in order to enforce the policy. Authority policies place restrictions on actions within a system and are only invoked when a request is made for a particular action.

All of the research described in this section emphasizes the use of a relatively small software component to enforce safety properties or provide services required by safety-critical software. This type of structure is utilized for two reasons. First, for a given safety-critical system, a significant subset (e.g., functionality with properties identified by Rushby) of the safety specification can be implemented in a small component. Second, smaller components are simpler and more amenable to verification than the application software, thereby providing increased assurance that the subset of the safety specification has been met.

Previous research on safety kernels, especially that by Rushby, has established some basic kernel concepts. However, little progress has been made in defining what policies might be enforced, how the safety kernel would be incorporated into a system, what requirements must be met to dependably enforce policies, and so on. As a result, the particular techniques presented are basically ad hoc. For example, the kernel proposed by Leveson et al. provides mechanisms for detecting and recovering from software errors. Although these mechanisms would likely be necessary (but not sufficient) for supporting safety-critical software, their overall impact on software safety is not evaluated.

A critical shortcoming of the work described above is that none of these concepts or designs have been implemented, let alone evaluated, with actual safety-critical applications. Research performed on paper or on “toy” systems does not have the benefit of dealing with the challenges posed by an actual safety-critical system.

2.5 Software Fault Tolerance

The fundamental safety kernel concept of tolerating faults in the application software is certainly not new. Software fault tolerance techniques have been investigated by many researchers and are used in several operational systems. This class of techniques has the objective of surviving the effects of faults by dealing with them at run-time. In essence, if a fault exists and causes a problem (such as generation of a wrong output), the software would survive the effects of the fault by detecting the problem and automatically masking its effects.

Coping with the effects of faults during execution and thereby possibly improving the dependability of the software is the goal of *software fault tolerance*. Researchers have proposed various methods for building fault-tolerant software with the hope that they might provide substantial improvements in the dependability of software for safety-critical applications. Most of the proposed methods for achieving software fault tolerance have been adapted from similar approaches used to cope with hardware faults. Very high levels of hardware dependability are achieved using hardware fault tolerance but the faults that are tolerated are for the most part degradation faults.

To tolerate software faults, the commonly advocated techniques rely on redundant implementations, i.e., the availability of several different programs that all implement the same specification. Software engineers assume that the various programs are sufficiently different that they contain different faults. Fault-tolerant software built in this manner is referred to as design-diverse software. A well-known method for building design-diverse software is N-version programming [6,11]. N-version programming requires multiple (i.e., “N”) programs (i.e., versions) to be built for a given specification, a decision algorithm or *voter* for choosing which outputs to use (the majority for example), and a monitor or execution-time support system to manage the versions. The versions are developed separately and frequently use different development tools and techniques. At execution time, the versions usually operate in parallel in the application environment; each receives identical inputs, and each produces its version of the required outputs. The monitor collects the outputs and the decision algorithm determines what output should be used by the enclosing system. For example, if the outputs are not all the same but should be, the majority value might be selected, if there is one. If the outputs are floating-point numbers, the mean or median might be used. Errors are detected by differences in the outputs of the various versions and, if N is three or more, faults are tolerated if a majority of the versions produce acceptable outputs.

There are several technical problems with design-diverse software that should be recognized by practitioners but frequently are not. The problems are, in some cases, quite subtle. Each has an impact on the final performance of a design-diverse software system, and the effects of each must be weighed against any benefits that design diversity might provide. Certainly there are benefits that can accrue from the use of design diversity, such as its application to secure systems [21], but it is not a panacea for achieving high software dependability. It is possible that, by chance, the faults in the various implementations are sufficiently different that their effects are masked. But this cannot be assured with high probability. This problem with the assumption of independent failures has been well documented along with other possible difficulties [8,13,14,26,27,30].

A key difference between the general application of software fault tolerance and its manifestation in the safety kernel is that whereas general software fault tolerance is targeted at tolerating *all* software faults, the safety kernel is focused on a very specific set of failures. This permits the safety kernel to be much simpler and smaller than the application software. As a result, considerable effort can be expended to build a dependable safety kernel — effort that would not be feasible if the safety kernel were a replicated version of the application software.

2.6 Summary

Although safety-critical systems have been built that implement their safety specification effectively, these systems have been built using ad hoc methods. The related area of security has developed general methods based on a kernel approach that have made the development of certain types of secure systems a general, repeatable, and verifiable process. Rushby has recognized the potential for applying the concept of a kernel for the enforcement of safety policies in safety-critical systems. In the process, Rushby also identified the conditions that must be met in order to qualify a policy as being kernel enforced. The research described here expands on this work to clarify the safety kernel concept and to address questions that arise when applying the concept to a real system.

3 Case Studies

Given the research goal of developing ideas, processes, and tools that address the needs of safety-critical systems, it is essential to have access to safety-critical systems that exhibit the complexity and challenging requirements of real systems. As such, the Magnetic Stereotaxis System and the University of Virginia Research Reactor have provided a context for problem exploration and evaluation of research results. In many cases simplistic solutions were eliminated because they could not satisfy the requirements of these systems. The case studies are not case studies in the sense of finished systems that are analyzed for strengths and weaknesses. Instead, we are actually building the systems as a means of advancing the technology employed in the development of software for these systems. This chapter describes the operation, system hardware, and safety issues of these two case studies.

3.1 Magnetic Stereotaxis System

The safety kernel is being developed in the context of a case study with the *Magnetic Stereotaxis System* (MSS). This is an investigational device for performing human neurosurgery that was originated by researchers from the Department of Physics and the Department of Neurosurgery at the University of Virginia [17,19,53]. Presently, Stereotaxis, Inc. is preparing the latest version of the MSS for animal trials and, ultimately, human clinical trials that will be conducted at the Barnes Hospital of Washington University in St. Louis.

The MSS operates by manipulating a small permanent magnet (known as a "seed") within the brain using an externally applied magnetic field (see Fig. 3). By varying the composite structure (and, hence, the gradient) of the external field, the seed can be moved along a non-linear path and positioned at a site requiring therapy, e.g., a tumor. The device can be used for hyperthermia by radio-frequency heating of the seed from an external source, for chemotherapy by using the seed to deliver drugs to a site within the brain, or for biopsy by affixing an appropriate miniature instrument to the seed. The MSS concept promises to be far less traumatic to the patient than present invasive approaches to such treatments. The state of the MSS is that the concept is fully defined, the majority of the basic research in physics is complete, and a fully functional prototype is nearing completion for demonstration and evaluation. A program of animal trials and subsequent human testing using the prototype is expected to begin in the near future.

Fig. 4 shows the hardware used by the MSS to effect and monitor movement of the magnetic seed within the patient's brain. The patient is positioned at the center of six superconducting electromagnets. Under the direction of the computer, power supplies and current controllers regulate the electric current in the electromagnets, thereby producing the magnetic field that acts on the seed. Along both of the coil axes perpendicular to the patient’s body, an x-ray source and camera produce fluoroscopic images for tracking the seed.

During an operation with the MSS, a neurosurgeon directs the movement of the seed from a console that displays preoperative Magnetic Resonance (MR) images. The computer takes movement requests and computes the electromagnet currents required to produce the desired seed movement. During seed movement, a computer vision system analyzes the images from the fluoroscopes to locate the seed and markers affixed to the patient’s skull. Visible on both the MR and x-ray images, the markers enable the position of the seed to be transformed into the MR frame of reference and subsequently superimposed on the MR images.

When the MSS is in operation, there are a large number of events that could lead to patient injury. The complete set is determined by a hazard analysis including the use of techniques such as system fault-tree analysis. Some examples of events that could lead to patient injury include:

- Failure of electromagnets or current controllers.
- Incorrect calculation of currents required to provide a requested movement.

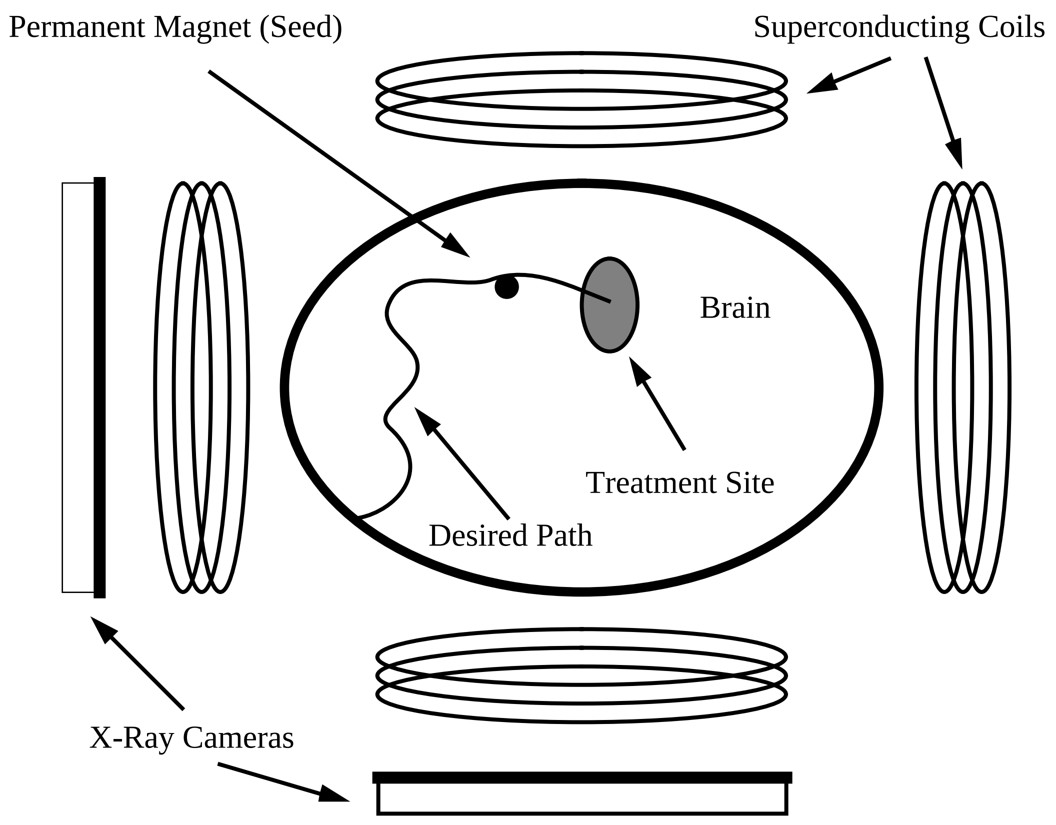

Fig. 3. Seed guidance by MSS.

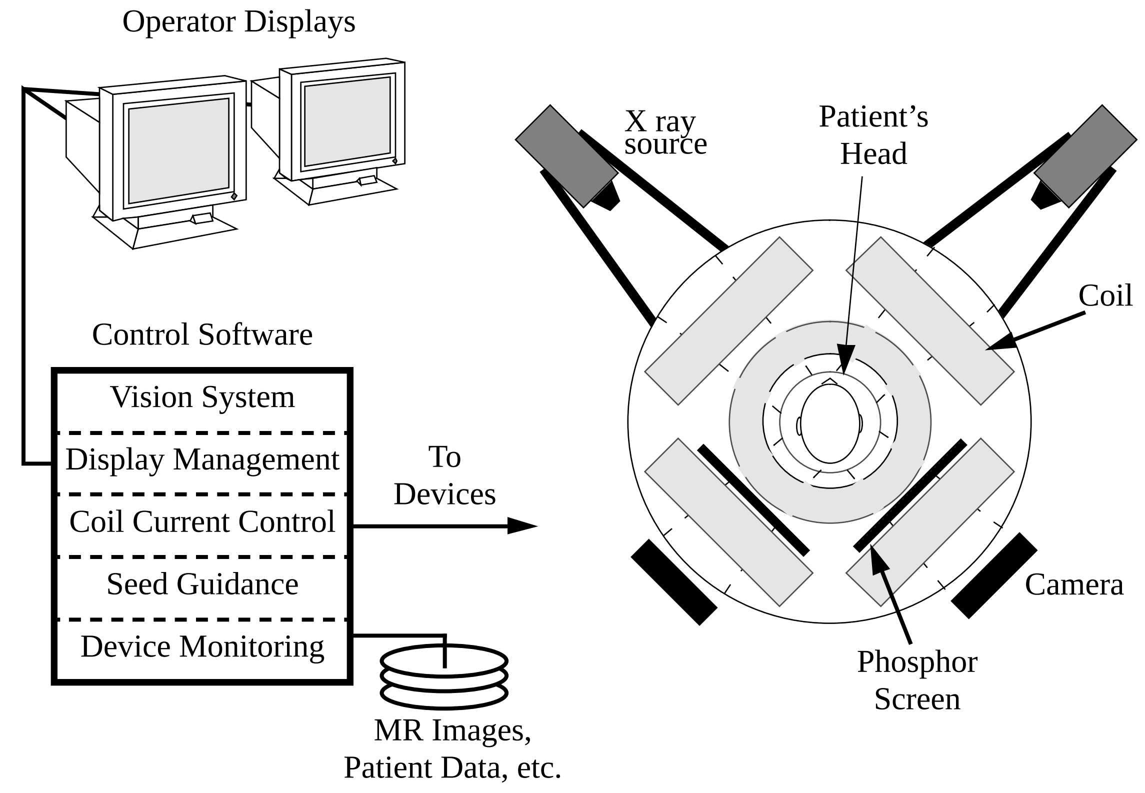

Fig. 4. Magnetic Stereotaxis System.

- Misrepresentation of the position of the seed on the MR images.
- Inappropriate control of currents by the computer.
- Erroneous movement commands by the human operator.
- Failure to respond promptly to an increase in seed velocity.
- Incorrect response to the failure of an electromagnet or current controller.
- X-ray overdose.

Each of these could be the result of numerous different faults and, in fact, the software could either initiate or prevent many of these failures.

For the MSS, safety is the key requirement. If a failure occurs it is both feasible and acceptable to shut the system down, ensure the safety of the patient, and then address the cause of the failure — thus system reliability is not a concern. At the same time, availability is not a critical requirement because use of the MSS can be scheduled with little more consequence than inconvenience.

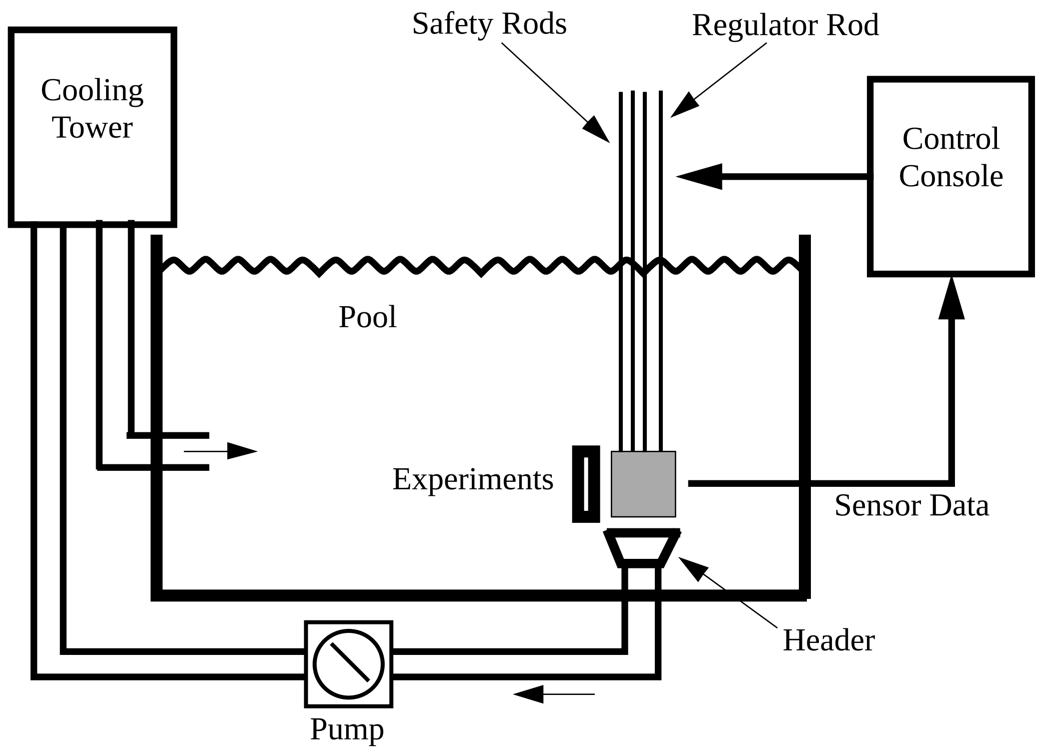

Fig. 5. University of Virginia 2.0 MW Research Reactor.

3.2 University of Virginia Reactor

The target of the second case study is the nuclear research reactor currently operated by the University of Virginia (see Fig. 5). It is a 2 MW, thermal, concrete-walled pool reactor. It was originally constructed in 1959 as a 1 MW system, and it was upgraded to 2 MW in 1973. Though only a research reactor rather than a power reactor, the issues raised are significant and can be related to the problems faced by full-scale reactor systems.

The system operates using 20 to 25 plate-type fuel assemblies placed in a rectangular array. There are three scramable safety rods, and one non-scramable regulating rod that can be put in automatic mode. The primary process variables that are measured are: 1) Gross output, by movable fission chamber; 2) Neutron flux, by ion chamber; 3) Start-up neutron flux and period, by BF_3 counter; 4) Core inlet and outlet temperatures, by thermocouples; 5) Primary system flow, by pressure gauge; 6) Control and regulating rod positions, by potentiometer; 7) Gross gamma-ray dose, by ion chamber; 8) Various limit-set switches to monitor pool level, etc.

As with the MSS, there are a large number of events that could lead to a reactor accident with the potential to cause extensive damage. Some examples of events that could result in hazards include uncontrolled withdrawal of the reactor control rods, loss of water in the reactor pool, failure of a coolant pump, and high radiation levels outside of the reactor pool.

4 Safety Policies

The goal of the safety kernel is to facilitate the development and verification of safety-critical software. The safety kernel will provide this support by ensuring that certain safety requirements are satisfied independent of (and even in spite of) the application software. Analogous with the security kernel, these requirements for a safety-critical system are known as *safety policies*. To this point, the discussion of kernel enforcement of safety policies has largely treated the kernel and policies from an abstract point of view, i.e., with the kernel seen as a policy enforcer and a policy as some requirement that must be ensured to support system safety. Before more specific requirements and a design can be pursued, the understanding of the safety kernel role as a policy enforcer must become more detailed and complete.

This chapter describes the results of an analysis of safety policy data from the two safety-critical applications used as case studies, the MSS and the UVAR. The goal of the analysis was to answer questions concerning safety policies. The questions addressed include the following:

- Where does a safety policy come from?
- What are distinguishing characteristics of safety policies?
- How are safety policies organized?
- Are safety policies similar across applications?
- What safety policies are most appropriate for safety kernel enforcement?
- How are safety policies identified for safety kernel enforcement?
- What portion of safety policies could be best enforced by a safety kernel?

These questions are important for defining the role of the safety kernel, evaluating its contribution to a safety-critical system, and informing choices related to implementation of an instance of the safety kernel.

The approach that we have taken to address these questions is to conduct an experiment in which the two safety-critical applications provide a target for study of safety policies. For each application, safety policy data was gathered from a range of sources including reference to approved safety documents [52] and interviews with systems engineers. Although likely not complete, the safety policy data represents a wide variety of requirements and we believe it is sufficiently comprehensive to permit the questions posed above to be answered. The safety policy data was analyzed to attempt to characterize the policies and to identify an underlying organization of the policies. A policy taxonomy resulted from this analysis, and it in turn was evaluated against the policy data.

With several classes of safety policies identified, these classes were evaluated for possible enforcement by the safety kernel. Using properties identified by Rushby and guidelines developed as a part of this experiment, several classes of safety policies were identified for enforcement. The final phase was to analyze the policy data to determine which policies were included in the classes of kernel-enforced policies. From this analysis, preliminary conclusions were made concerning the effectiveness of the kernel and the degree of support it provides to safety-critical software.

4.1 Safety Policy Classes

The safety policies that have been identified for the MSS and UVAR are presented in Appendix A. The set of MSS safety policies is relatively static, but will change some as the application software is modified. The safety policies dealing with UVAR system safety are well understood and static. However, the UVAR policy data is incomplete with respect to the application software because that software is still in the process of being specified.

In the initial analysis of the safety policy data, the focus was on identifying characteristics that might permit a logical organization of the safety policies. Characteristics considered included the purpose of a policy, the origination and destination of the information to which the policy is applied, and the generality of the policy. Although each brought some order to the policies, none was sufficiently systematic, complete, or effective for classifying all of the policies and relating the policies to the safety kernel. The identification of a useful characteristic occurred when we considered where safety policies come from and how they were obtained.

Safety-critical applications have a similar ultimate safety policy. This policy states that the risk of a system failure causing injury to people or resulting in excessive damage must be acceptably low. In the instantiation of a system, this general policy is decomposed into many component policies that are intended to ensure system safety. To facilitate the identification of component safety policies, systems and software engineers perform a system safety analysis utilizing techniques such as hazard and fault tree analysis to evaluate interactions of components.

In a system safety analysis, an initial step is the identification of hazards that could result in injury or damage. Each hazard is subsequently placed at the root of a fault tree and the failures that could result in the hazard are analyzed [35]. From this analysis, a complete fault tree is developed which details the failure conditions that could lead to a particular hazard. The exact form of a fault tree depends on the hazard being considered and the details of the particular application. However, working with the two case studies and other systems, we have identified a *canonical fault tree* for systems incorporating computer-controlled devices. Shown in Fig. 6, this canonical representation documents the general types of failures that can occur and produce a hazard. Since hazards are caused by the action or inaction of system devices, the focus of this fault tree is the devices, device interaction, the computer that controls the devices, and the control inputs to the computer.

The initial safety policies for a system are high-level policies that define safe operation of the devices. These policies are typically stated in an abstract manner without any of the details that enable system components to enforce the policies. However, working with the

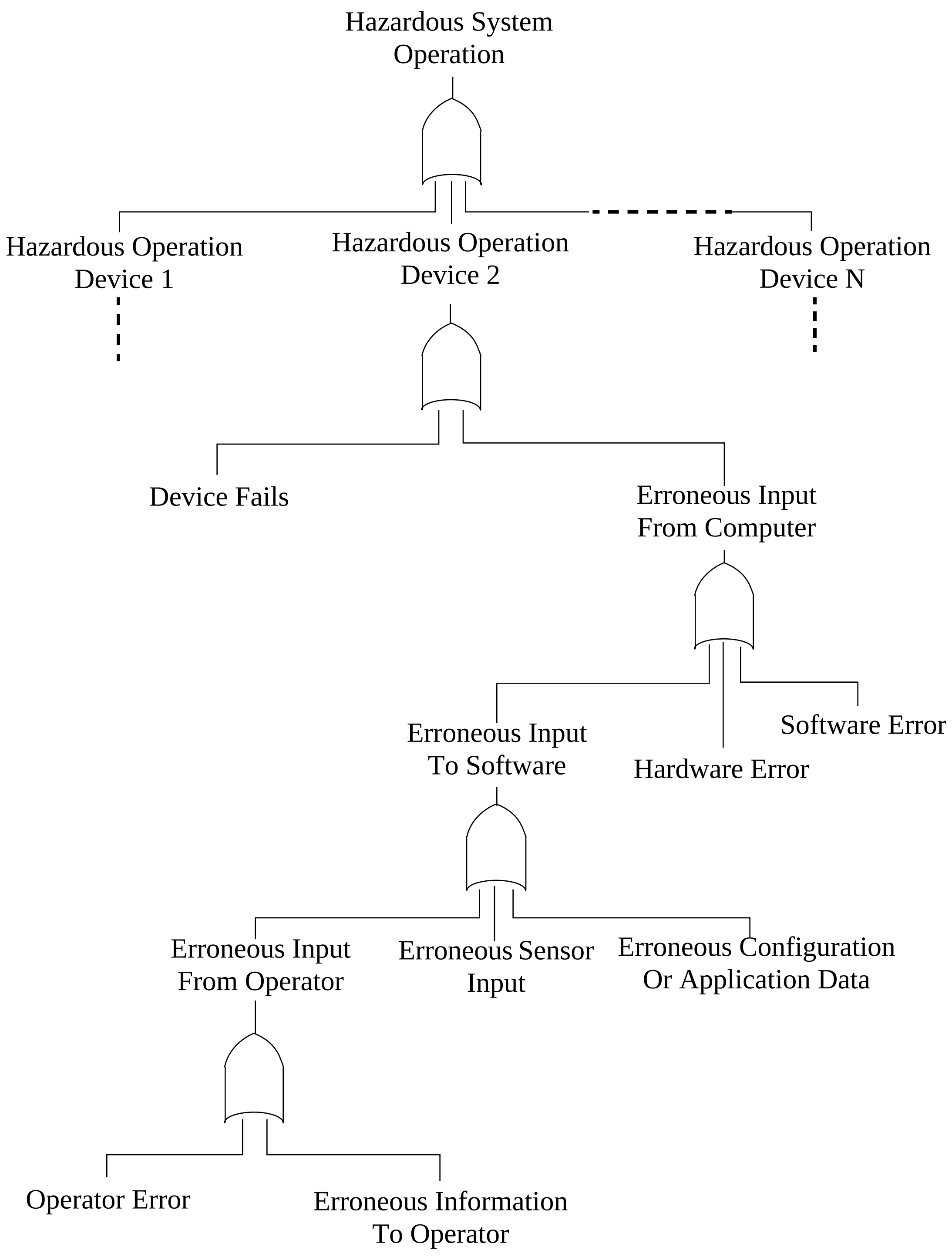
Hazardous System Operation
Hazardous Operation Device 1
Hazardous Operation Device 2
Hazardous Operation Device N
Device Fails
Erroneous Input From Computer
Erroneous Input To Software
Hardware Error
Software Error
Erroneous Input From Operator
Erroneous Sensor Input
Erroneous Configuration Or Application Data
Operator Error
Erroneous Information To Operator

Fig. 6. Segment of canonical system fault tree showing software-related nodes.

fault trees and details of the system, safety policies are developed that specify the requirements the components must meet to ensure that failures do not result in hazardous operation. In this manner, safety policies are developed to address each of the types of failures.

The canonical fault tree contains several classes of failures that are identified by the labels on the fault tree. Failures are grouped according to these classes. In the same way, safety policies are classified based on the type of failure that the safety policy addresses. The result is a taxonomy of safety policies. Obviously, additional types of failures, and thus safety policies, could be identified by further subdividing the fault tree. However, from our experience the classes listed below provide a sufficiently detailed set of classes for analysis — particularly for evaluating the enforcement role of the safety kernel. The twelve classes of safety policies are the following:

- System Operation
- Device failure
- Application software error
- Software input
- Sensor input
- Operator error
- Device Operation
- Device input from computer
- Failure response
- Operator input to the software
- Configuration or application data
- Operator information

One of the classes of safety policies, failure response policies, does not actually appear on the canonical fault tree. This is because nodes for this policy are only added to the tree whenever a safety policy is added that calls for detection of a particular failure. Addition of a policy of this type results in a modification of the fault tree. To illustrate, consider the type of failure *device fails* on the canonical fault tree. One way to deal with such a failure would be to detect the failure and respond to it. The effect on the tree is to replace the single *device fails* node with a subtree as shown in Fig. 7. Now, the failure will only result in a hazard if the device fails and either the failure cannot be detected or there is not an effective failure response.

The classes of safety policies that appear lower on a branch of the canonical fault tree address failures that could cause policies higher on the branch to be violated. For example, system operation policies are concerned with safe device interaction and device operation. Policies from the class software error address specific software failures that could lead to a failure at the system level, i.e., the violation of a system operation policy. Violation of any of the lower policies could result in a system operation failure.

Is the identified set of safety policies complete? In other words, is there a policy class corresponding to any type of failure that could be identified? Given that the canonical fault tree can be further decomposed, the classes identified here are obviously not complete. On the other hand, we believe a strong argument can be made that a critical portion of the tree is in fact complete. This argument is presented below.

The class *system operation* is complete by definition because it includes any policy that addresses a system-level failure. Since a system can only affect its environment through

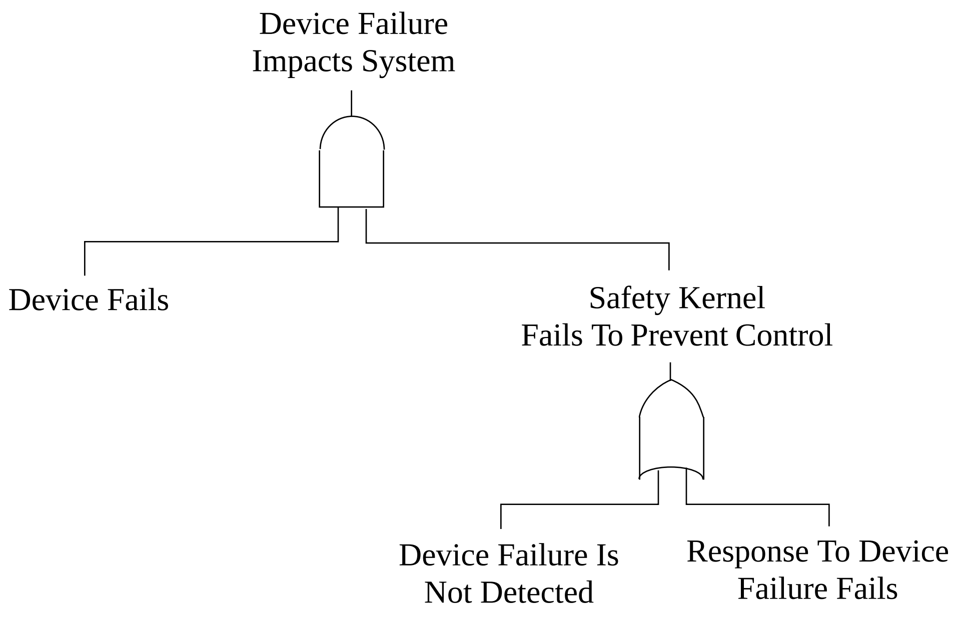

Fig. 7. Subtree resulting from addition of detection and failure response policies.

actions of devices, *device operation* policies address all failures that could lead to a system-level failure. Continuing to the third level, a device that fails in its operation, does so either because it was commanded erroneously or because the device itself has failed. At the next level, the hardware and software of the computer calculate output values to devices based on input to the computer. If there is a failure, in the computer output, the only possible sources are either the hardware, software, or input data. Thus to this level, the classes of policies are complete in that they account for all of the types of failures that could lead to a hazard. Below this level, however, the tree is likely to be incomplete simply because there are many more types of failures that could be enumerated. The lower portions of the tree are less critical for evaluation of the safety kernel, however, because the classes corresponding to these failures are not likely to be kernel-enforced.

The safety policies in Appendix A have been classified according to the safety policy classes developed in this section. All of the safety policies for the two case studies have been categorized using this grouping. Given the classes of safety policies, the question to be addressed is, “What will the role of the safety kernel be in their enforcement?”. This question is addressed in the next section.

4.2 Issues in Kernel Enforcement

4.2.1 Kernel Enforceability

A particular safety policy is *kernel-enforced* by a specific safety kernel if the two conditions identified by Rushby are true. Policies that do not meet these conditions are *non-*

kernel-enforced. A particular safety policy is neither inherently kernel-enforced nor inherently non-kernel-enforced. The kernel enforcement status depends on the safety policy in question and must be determined based on an analysis of the kernel responsible for enforcement. To illustrate this point, we note that at one extreme the complete absence of a kernel obviously makes all policies non-kernel-enforced. At the other extreme, the entire functionality of an application could be placed into a safety kernel thereby making all policies kernel-enforced by definition. Of course, this latter choice would run counter to the objective of having a relatively small, simple kernel.

Assuming a safety kernel that mediates access to devices, examples of kernel-enforced policies are the following:

> *Device input from computer*:
> If device A is on, then device B should not be turned on.
>
> *Device operation*:
> Over some time period T, the fraction of time that device C is on must not exceed K.

These are policies that ultimately depend only on kernel operations, i.e., the kernel can enforce them without regard to application circumstances, and they are “for all” policies, i.e., they are applied irrespective of the order and type of kernel operations invoked by the application. Thus they satisfy the two conditions.

Examples of policies that would be non-kernel-enforced, except in the extreme case where a large portion of application functionality is implemented in the safety kernel, include the following:

> *An arithmetic calculation policy*:
> The result of an application-specific algorithm must never yield a result greater than X.
>
> *A highly application specific policy*:
> A particular flight control system must set engine thrust correctly based on parameters such as air speed, altitude, and fuel efficiency.

The first of these two examples can be described by a second-order assertion. However, the policy cannot be readily enforced by the safety kernel because kernel operations are not essential for the implementation of the policy. The policy says essentially that some part of the application’s internal computation has to satisfy some assertion. Even if the safety kernel provided an operation to perform the calculation, it could not guarantee that the operation would be neither replaced nor used improperly.

The policy in the second example would be expressed as a first-order assertion. The safety kernel could ensure that the thrust requested of the engine would satisfy some reasonableness check, but it would not be able to enforce the policy without its own version of the software to perform either a reversal check or replicate the calculation.

4.2.2 Weakened Safety Policies

In general, safety policies are selected to be enforced by the kernel because such enforcement provides significant benefits. However, the benefits must be balanced by considerations such as the cost of kernel implementation and quality assurance, kernel performance overhead, and the substantial effort that would result from trying to incorporate a complex policy into the safety kernel. In some cases, there are safety policies for which it would be beneficial for the safety kernel to enforce part of the policy but where enforcement of the complete policy as it is derived from the application specification is inappropriate.

An example of a policy of this type for the MSS is the following:

> Every 0.5 s the location of the seed must be determined. The seed must not move more than 1.0 mm or faster than 2.0 mm/s with respect to a coordinate system fixed to the markers.

This policy addresses the need for carefully controlled movement of the seed and relies on the vision system for input. In principle, the safety kernel could incorporate the entire vision system and actually compute the seed position to be used in enforcing this policy. However, there are two problems with this approach. The first problem is that the vision system is quite complex. Thus it would be difficult to incorporate it into the safety kernel and do so in a manner that would ensure it would not interfere with the operation of the rest of the safety kernel. The second problem is that operation of the vision system by the safety kernel requires resources that are not available. In order to track the seed with the desired frequency, the application software needs exclusive use of the cameras, x-ray sources, and image capture hardware. In addition it uses a significant portion of the available processor resources. Therefore, it is not feasible to require that the safety kernel repeat these calculations that are already being performed by the application software.

Some safety policies that cannot be kernel-enforced in their strictest form can be kernel-enforced if *weakened*. A weakened safety policy is defined as follows:

> A weakened safety policy is a kernel-enforced policy for which *part* of the enforcement responsibility is shifted from the safety kernel to the application software.

Although the application software is given partial enforcement responsibility, it is held accountable by the safety kernel. As an example, in the enforcement of a weakened safety policy, the application software might perform some complex calculation and then be required to report the results to the safety kernel which would then complete the enforcement of the safety policy.

Returning to the example above. Since the application software is locating the seed already, reporting the position does not add significant burden, yet the notification of the position enables the safety kernel to ensure that the check is occurring. In its weakened form the example from above becomes

> Every 0.5 s the location of the seed must be *reported* to the safety kernel by the application software. The seed must not move more than 1.0 mm or

faster than 2.0 mm/s with respect to a coordinate system fixed to the markers.

The advantages of enforcing weakened policies are that the relative simplicity of the safety kernel is maintained while at the same time, by requiring notification, the application software must either provide proper notification or make an erroneous notification. The disadvantages of the weakened policies are that assurance of the original policy requires verification of the application notification, the application software processing the data is more prone to interference by other software than it would be if it were executed in the safety kernel, and erroneous application software could generate notifications that would satisfy the weakened policy but not the actual safety policy.

Justification for the use of weakened safety policies is based on the following:

- In a safety-critical system, it is assumed that the application software is not malicious (although it may be faulty). Therefore, some assurance can be gained by "trusting" the application software with certain aspects of policy enforcement. This would not be a reasonable assumption in a security system.
- The actual policy that is enforced is the same whether it is a weakened policy or not. There is a division of responsibility but otherwise the operations performed in enforcing a safety policy can be identical. The level of isolation is better for code incorporated inside the safety kernel, but for cases where this is not feasible, the weakened safety policy provides a measure of assurance that the operations required to enforce a policy are being performed.

4.3 Kernel Enforced Policies

Since safety policies are neither inherently kernel-enforced nor inherently non-kernel-enforced, it is necessary to select policies for kernel-enforcement. Safety policies are selected to be enforced by the kernel because, as discussed in the introduction, kernel enforcement of policies provides significant benefits to a safety-critical system. Essentially, the relative simplicity of the safety kernel has the potential to improve the dependability of enforcement of selected critical safety policies. However, selection of a policy or class of policies for enforcement impacts the design, implementation, and verification of the safety kernel. The impact must be evaluated with respect to the pragmatic concerns of quality assurance, cost, and functional performance discussed in the introduction.

We have selected seven of the policy classes identified in Section 4.1 for enforcement by the safety kernel prototype. It is important to point out that not all of the policies in a selected class will be kernel-enforced by the prototype. Some of the policies will be enforced in a weakened form. Others, for example those that would add significant complexity to the verification of the safety kernel, will be non-kernel-enforced.

With one exception, the classes of policies enforced by the prototype originate near the top of the canonical fault tree shown in Fig. 6. The exception is the class sensor input that is included because sensor data is critical to safety kernel enforcement of the safety policies. Not coincidentally, these seven classes are the ones that are most closely associated with operation of the application devices. Since devices actually cause mishaps, these policies have the most direct impact on system safety. Enforcing policies from the other classes would in general not benefit system safety as substantially and would increase complexity, thereby adversely impacting cost and quality assurance. It should be stressed that another set of policy classes could have been chosen for kernel enforcement. The set above addresses the requirements of the two case studies and likely a range of other systems, but there are no precise rules for selection of policies for kernel enforcement.

The rest of this section looks at the classes of kernel-enforced safety policies. Within the classes, types of safety policies have been identified. These are described and examples are provided from the MSS and UVAR. The descriptions of the classes are intended to provide more detailed information on the safety policies that are enforced by the prototype safety kernel.

4.3.1 System Operation

The policies in this class are concerned with the interaction of devices and the effects of their operation on the physical environment. The safety kernel enforces policies in this class using information obtained from sensors that observe the state of the physical system. The enforced safety policies determine the expected state of the system and compare the expected state and the observed system state. The policies enforced at a given time depend on the commands issued to devices.

> ***MSS**: The seed must always be within 2 mm of its expected position as determined from the coil current-time profiles and a model of the seed movement through the brain as a function of magnetic impulse.*
>
> ***UVAR**: The reactor water inlet temperature must not exceed 105˚ F.*

4.3.2 Device Operation

A critical role of the safety kernel is dealing with actual and apparent failures of application devices. An actual failure occurs when a device has been commanded correctly, but fails to execute the command as specified. An apparent failure occurs when a device operates correctly, but the application software fails to act or acts incorrectly. Both types of failures are detected through an inconsistency in the observed and predicted state of a device. Therefore, there will be a component of the safety kernel that has the task of periodically sensing the state of a device and comparing this state to the expected state.

> ***MSS**: The state of the x-ray source must match the most recent state commanded.*
>
> ***UVAR**: The flow in the primary cooling system must be greater than 3,400 liters/min (900 gals/min).*

4.3.3 Device Failure

Incorrect device operation can be caused either by a failure in the device itself or by an erroneous request from the software. Data from sensors observing a device cannot distinguish between these two sources of failure. Therefore, detection of the failure of a device itself must be based on status signals that come directly from the device.

***MSS**: If the servoamplifier that serves a coil detects an internal failure, it causes a status line to go low.*

4.3.4 Device Input From Computer

This class of safety policies dictates the manner in which devices may be operated by the application software. There are a number of different types of policies within this class. These policies define requirements that must be met before an application software request for device action can be effected. The safety policy that is enforced at a given time depends on the command being requested and the expected state of the system. The policy types in this class include the following:

- *Command history restrictions*
  Policies of this type describe the ordering of commands or require that some action must be performed in order to satisfy a request for device action.

  ***MSS**: Before a coil can be charged, the load current command must be successfully executed.*

  ***MSS**: Before the current in an x-ray source can be turned on the voltage must be on.*

- *Timing requirements*
  This type of policy places limits on the timing of commands (e.g., the time between commands) or on timing aspects of the device operation (e.g., the total operating time of a device).

  ***MSS**: An x-ray device must be in the “off” state for 0.2 s before the invocation of an “on” command.*

  ***MSS**: The total x-ray dose during an operation must be less than 100 millirem.*

- *Parameter checks*
  Restrictions on command parameters (e.g., range checks) are specified by policies of this type.

  ***MSS**: The current requested of the servoamplifier current controllers must be less than 100 A.*

- *Operational state conditions*
  For a given command, policies of this type document restrictions that depend on the physical state of a device or the system.

  ***UVAR**: Before the safety rods can be lifted the source range must be indicating at least 2 counts per second.*

4.3.5 Application Software Error

The basic model of the safety kernel is that it enforces policies that ultimately depend on calls to safety kernel operations. In certain cases it is desirable to extend this interface up into the application to enforce policies relating to actions of the application software.

For example, the calculation of output values for a device might have associated with it an effective reversal check. In that case, a safety policy might be that the output values should never be sent to the device without the reversal check being applied. If it is not feasible for the safety kernel to perform the reversal check, then an alternative is to require the application to *notify* the safety kernel when the check has been performed. This is an example of a weakened safety policy. Much like the device control policies described above, application software activity policies specify required sequences of actions, timing constraints on those actions, or particular checks or activities to be performed when an action occurs.

> ***MSS**: Prior to setting the currents for the servoamplifier current controllers, a reversal check must be executed to ensure that the requested currents provide the desired force. The results of the reversal check must be reported to the safety kernel.*

> ***MSS**: The safety kernel must be notified that a seed movement has been requested each time the coils are charged.*

4.3.6 Sensor Input

Because it monitors the system and devices, the safety kernel has access to data from sensors. These safety policies describe how erroneous sensor data is detected (e.g., using reasonableness checks or by comparing the observations from redundant sensors).

> ***UVAR**: The reactor pool temperatures measured by two temperature sensors must be within 3˚ F.*

> ***MSS**: The current values reported by the current controllers and the independent sensor must differ by less than 5.0 A.*

> ***MSS**: A coil current sensor reading outside of the range -100 A to + 100 A indicates a faulty sensor.*

4.3.7 Failure Response

Detection of an error in the operation of application software, support software or system devices, requires a response that can ensure that the system remains in or is returned to a safe state. Typically a continuum of recovery procedures will be available. For example, in a system where recovery was achieved by shutting devices off, a severe recovery policy might call for disconnection of power to all devices, whereas a less severe policy would be more sophisticated and would effect a more orderly shutdown of the system. For a given error, the appropriate response will be selected based on the present system state.

> ***MSS**: If an x-ray source fails so that it cannot be activated and the coils are in a discharged state, the x-ray sources and coils must be deactivated.*

> ***MSS**: If the x-ray sources are on when they are expected to be off, the power to the x-ray sources must be interrupted and the coils discharged.*

> ***UVAR**: The control rods must be scrammed if a sensor indicates a power greater than 125% of maximum power.*

It is instructive to compare these classes of policies with the policies that are enforced by a *security* kernel. A security kernel enforces policies concerned with device input from software. It does so by accepting requests from application programs and then either fulfilling or rejecting the request based on the status of both the requester and the desired information. Rejection of a request is the failure response policy for a security kernel. A security kernel does not need to be concerned with the system operation, device operation, or device failure classes because these are all concerned with policies detecting the failure of one or more devices or with erroneous operation of the devices by the application software. Other than the basic issue of whether an information device is operational, a security kernel has no responsibility for operating a device. The class of policies related to application software error are also not a concern with a security kernel. Given the unlimited variety of application programs that can execute on a security kernel, it is not feasible to offer any services that enforce policies within these programs. On the other hand, a particular *safety* kernel is likely to serve a small number of application programs (typically one) and therefore can beneficially provide for enforcement of policies that regulate the operation of application software. Finally, as noted above, the failure response for a security kernel is rejection of service as opposed to a potentially complicated safety failure response in which system devices may need to be restored to a safe state.

4.4 Kernel-Enforced Policy Evaluation

The safety kernel prototype that we have developed enforces safety policies from the seven classes described above. What specific safety policies are kernel enforced for the MSS and UVAR? What does kernel enforcement contribute to the overall safety of these two applications? In this section, the safety policies for the case studies are examined to evaluate the role of the safety kernel in these systems.

4.4.1 MSS Safety Policies

Appendix A documents the safety policies for the MSS including an indication of whether a policy is enforced by the safety kernel or the application software or whether it is enforced as a weakened safety policy. Of the 45 safety policies listed, 16 are kernel-enforced, 10 are kernel-enforced as weakened policies, and 19 are enforced by the application software. This data makes it clear that the safety kernel does not ensure system safety. On the other hand, it enforces a critical set of policies that are closely related to the operation of the application devices. To assess the overall impact of the safety kernel with the MSS, the policies that are kernel-enforced, weakened, and non-kernel-enforced are examined below.

The kernel-enforced policies come from all but one of the seven classes of kernel-enforced policies with application software error being the only class not represented. The policies that are enforced are primarily concerned with system and device monitoring, restricting control of the devices, and responding to failures.

The weakened safety policies come from the classes *system operation, input from computer*, and *application software error.* The two weakened, system operation policies deal with monitoring the movement of the seed. As discussed previously, the policies are weakened because it is not practical to incorporate the entire vision system into the safety kernel. The weakened policies from the other classes are concerned with ensuring that critical computations have been performed or that actions are sequenced in a particular manner. With all of the weakened policies, the application software is responsible for the computation and must report the results to the safety kernel. Reporting of results is required either on a periodic basis or prior to executing a device control command.

The policies that are non-kernel-enforced are concerned primarily with input to the application software and come from the classes that are designated as non-kernel-enforced. Enforcing these policies with the safety kernel would add unnecessary complexity to the safety kernel and they can be enforced effectively by the application software.

4.4.2 UVAR Safety Policies

All of the 47 safety policies listed for the University of Virginia reactor would be enforced by the safety kernel. In general, these policies require some action based on the status of a sensor(s) and are the same type as those that are enforced by either a hardware or software shutdown system. Therefore, the significance of the safety kernel role in this system should be clear. The variety of safety policies will increase as the software for this system is developed and policies are added, particularly in the classes of device input from software, software error, and failure response. As policies of this type are added there will be policies that are not enforced by the safety kernel.

4.5 Conclusion

This chapter presents the results of an experiment in which safety policies from two safety-critical applications were analyzed to answer questions concerning derivation, classification, and enforcement of these policies. Analysis of the policy data showed that safety policies can be characterized based on their place of origin on a canonical fault tree. This characterization was used to partition the policy data for the two applications. This classification system is especially suitable for identifying policies for enforcement by the safety kernel; thereby contributing to the definition of its role.

Classes of safety policies were selected for kernel-enforcement using criteria based on the ensuing benefits and the consideration of several pragmatic issues including quality assurance, cost, and functional performance. Using these criteria, we identified seven classes of policies that will be enforced by a prototype of the safety kernel. Most of these classes are directly related to the operation of devices and therefore are significant policies for system safety. Evaluation of these classes with respect to the policy data showed that the safety kernel will enforce all of the policies identified for the UVAR and will enforce in full or in a weakened form over half of the policies identified for the MSS.

5 Reliable Policy Enforcement

The focus of this chapter is an analysis of the requirements for reliable enforcement of safety policies by the safety kernel. In particular, when striving for reliable policy enforcement, what are the issues and alternatives concerning the design of the system in which the safety kernel is a component? In security systems, reliable security policy enforcement has typically required a system design in which the enforcement kernel is implemented as a system kernel. Safety-critical systems have different requirements, however, so this design may not be necessary or appropriate in the safety context.

Reliability is the probability that a particular device will function as required in a specified environment for a particular period of time [46]. Therefore, reliable policy enforcement requires that the safety kernel enforce safety policies and do so with an acceptable probability of functioning for a required period of time. In this chapter, the general requirements for reliable policy enforcement are examined. The actual system design that is developed to meet the requirements depends on the characteristics of the kernel-enforced policies. For example, a set of policies might require a system design where reliability, i.e., continuous service, is essential. On the other hand, policies for systems such as the MSS and UVAR require a system design where safety is the primary concern. We focus on safety systems of this type and develop techniques and a system design for reliable safety kernel policy enforcement.

5.1 Reliable Safety Policy Enforcement

A logical picture of the safety kernel in the system context is shown in Fig. 8. In addition to the safety kernel, the other components in the system include the application devices, the application software, and the computing platform. The computing platform includes any software and hardware that provide support services to the safety kernel. The application software includes the control software and other processes outside of the computing platform that may be active. Operating in this context, the safety kernel must be able to enforce a set of safety policies. Most importantly, it must be able to do so in spite of failures of these components

To identify the requirements for reliable kernel enforcement of safety policies, it is necessary to analyze the safety kernel in the system context to determine in what ways the enforcement of a policy can fail. Fault trees have been used to systematize this failure analysis. In the failure analysis, the root failure is hazardous operation of the system resulting from violation of kernel-enforced policies. Note that violation of individual kernel-enforced policies is not considered to be a reliable-enforcement failure unless the failures lead to hazardous system operation. As shown in Fig. 9, either of the following two high-level failures could result in this root failure:

1. The safety kernel acting in the system context fails to enforce a safety policy.

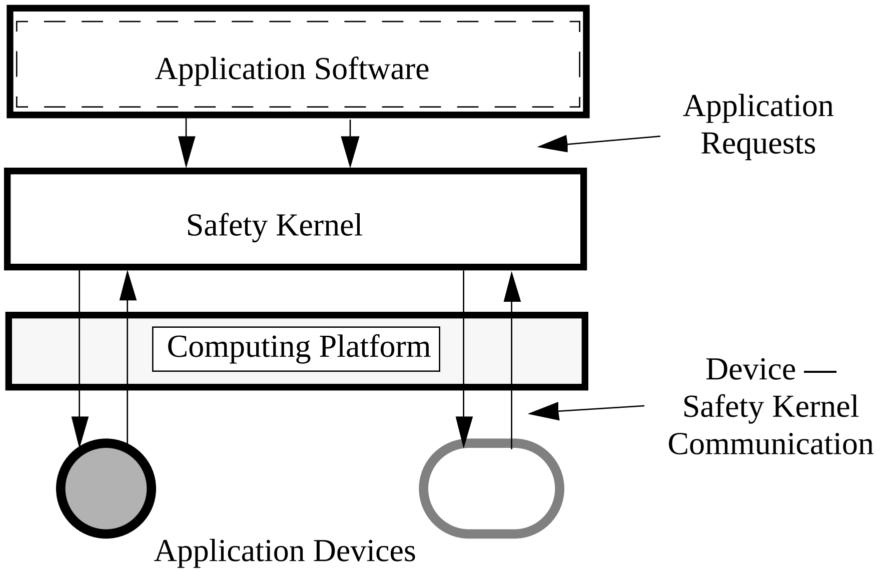

Fig. 8. Logical view of safety kernel in system context.

2. Some entity other than the safety kernel is able to control the devices in a manner that violates a safety policy.

These potential failures imply two high-level requirements that must be met to ensure that policies are enforced:

1. *Reliable safety kernel operation*
   Executing on the computing platform, the safety kernel must be able to interact with the application software and devices to enforce safety policies.
2. *Exclusive control*
   At all times during operation of the system, no other entity must be able to control a device in a manner that would violate any kernel-enforced policy.

Provided that the requirements of exclusive control and reliable operation are met, safety policies will be reliably enforced by the safety kernel. The following sections look at these requirements in more detail.

5.1.1 Exclusive Control Requirements

Exclusive control of application devices is a requirement that is similar to the security kernel requirement for *completeness* [2]. Completeness in the security context means that a security kernel mediates all access to information, i.e., it has exclusive access to the information. Exclusive access is essential in security systems where any unauthorized access to information has the potential for communication. In safety systems, access to information is not a problem in itself. Rather, exclusive *control* of devices is the concern. Access to a device is not a concern as long as the safety kernel has sufficient control over the operation of the device to enforce the specified safety policies. The conditions associated with suffi-

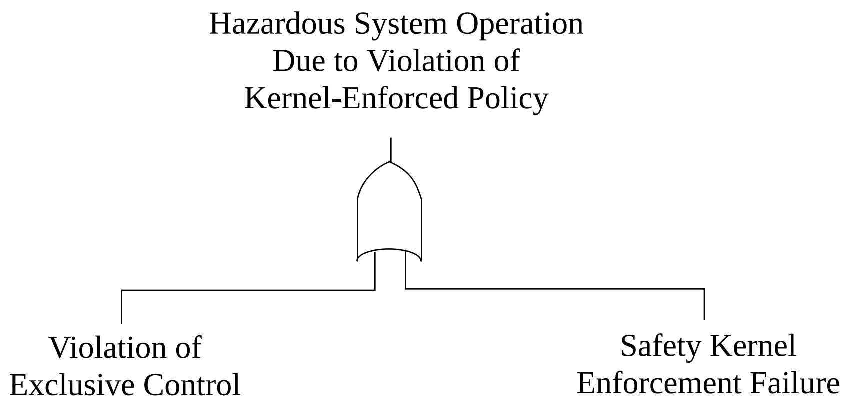

Fig. 9. Top-level fault tree for system with safety kernel.

cient control are determined based on the requirements of an application and the nature of the application devices. For some applications the safety kernel might need to ensure exclusive control by having exclusive access to devices, but for many applications it can employ techniques other than complete restriction of access. Techniques for ensuring exclusive control will be examined in Section 5.4.

5.1.2 Reliable Safety Kernel Operation Requirements

In order to enforce safety policies reliably, the safety kernel itself must meet certain dependability requirements. In addition, because the safety kernel both interacts with and depends on other components in the system, additional dependability requirements are placed on the components and on the system design that defines their interaction. Five requirements have been identified for reliable safety kernel operation. The fault tree in Fig. 10 shows the failures that the requirements address. The first two requirements, *safety kernel correctness* and *data integrity* are directly analogous to requirements that have been established for security kernels [2]. The third and fourth requirements are *dependable support services* and *dependable computing services*. Both of these are essential for security kernels, but are often assumed since security kernels have typically been built with very little underlying software. Support services in this case are defined to be those software and hardware services that are *invoked* by the safety kernel. Computing services are defined to be the basic computing platform services essential to execution and management of processes. The fifth requirement which is an issue in security systems, but critical for the safety kernel is *dependable computing resources*. Denial of service can be a problem with a security kernel, but it is essential that a safety kernel be able to act in a timely manner. Computing resources such as memory and the processor are necessary for timely operation.

These five requirements for reliable safety kernel operation are examined in more detail below. In Section 5.5, techniques are presented for meeting these requirements.

- *Safety kernel correctness*
  Central to the reliable operation of the safety kernel is the correctness of the execut-

able safety kernel. Correct, in this case, is defined as fulfilling the safety kernel obligations for enforcement of safety policies. Note that correctness does not imply that policies are necessarily enforced since enforcement of most safety policies depends on other system components. The actual achievement and measurement of safety kernel correctness might not feasible, but, from a conceptual point of view, it is a critical requirement for reliable policy enforcement. The issues associated with verification of the safety kernel are addressed in Chapter 7.

- *Data integrity*
  The safety kernel is a collection of executable instructions, configuration data, run-time state, and hardware that is carefully constructed to enforce a set of policies. In order for the safety kernel to operate as specified this information must not be corrupted. In the security realm, protection of data is provided through *isolation* of the kernel from other components in the system that might be able to effect some alteration of kernel data. In fact, in the security context the requirement is actually called isolation. However, the essence of the requirement is that the kernel *data integrity* must be guaranteed, where data is defined to be the static instruction and configuration data that constitutes the kernel software and the dynamic run-time state information. Data integrity also includes the instructions and data of support services.

- *Dependable support services*
  The safety kernel must invoke operations that are provided by the support software and hardware. Support software is defined to be any software that is utilized by the safety kernel but is not considered to be a part of the kernel at the source code level (e.g., the system libraries). This class does not include fundamental computing services not called by the safety kernel, such as memory management.

- *Dependable computing services*
  All of the entities executing on a computing platform depend on basic computing services. These services are not invoked as the support services above are, but are

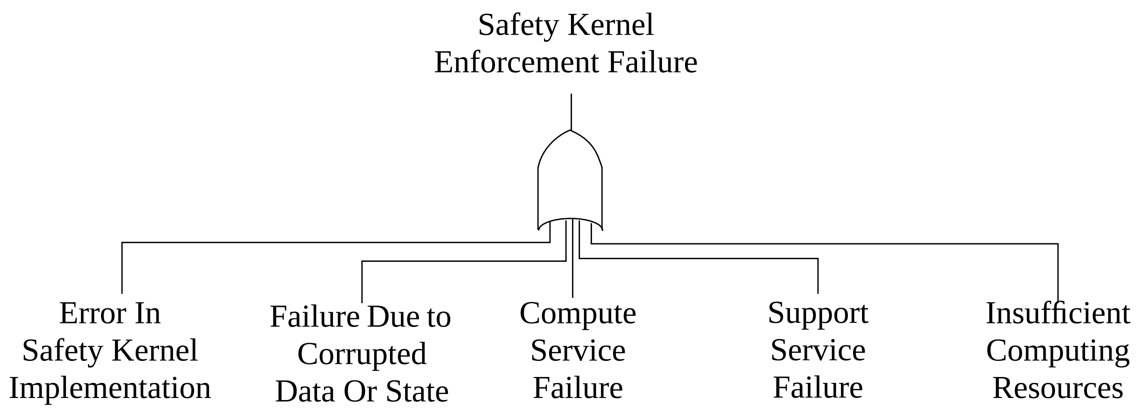
Safety Kernel Enforcement Failure
Error In Safety Kernel Implementation
Failure Due to Corrupted Data Or State
Compute Service Failure
Support Service Failure
Insufficient Computing Resources

Fig. 10. Fault tree for reliable safety kernel operation.

a part of the computing platform services that manage the details of executing processes. If one of these services fails in a manner that the system cannot tolerate, this can interfere with policy enforcement.

- *Dependable computing resources*
  A safety kernel will typically enforce a set of safety policies that require timely action. This has implications for the dependability of critical resources. The safety kernel must be able to either acquire critical resources in order to provide timely support for an application or possess the ability to detect and respond to a lack of computing resources. It must be able to do so from the beginning of operation up until the point that the system is in a stable, safe state.

The argument that this comprises the complete set of requirements is as follows. Referring to Fig. 8, if the kernel is operating reliably, it performs specified actions that are communicated to the application devices via the support services. To accomplish this, the safety kernel implementation must first of all implement the safety policies correctly. Second, the requests of the safety kernel (e.g., for communication with devices or mathematical operations) must be carried out by the support services as specified. If these first two requirements were met, and an ideal computer platform supported it, the safety kernel would perform the actions specified for enforcement of safety policies. An ideal platform would ensure the data integrity of the safety kernel and support services. It would also provide the basic computing services required for execution of the safety kernel. Finally, the ideal platform would provide sufficient resources for operation of the safety kernel.

Note, that ensuring reliable safety kernel operation does not require that device *actions* conform to the safety policies for a device. The safety kernel can only control a device via the device interface and has no control over whether directives to a device are correctly executed. In particular, the safety kernel is powerless to deal with device failures for which a response has not been anticipated and appropriate provision has not been made in the device interface. Therefore, the concern is that the requests that arrive at a device conform to and implement the required safety policies.

5.2 Safety Kernel Prototype Dependability Requirements

The requirements outlined above define the areas that must be addressed to ensure reliable policy enforcement by the safety kernel. The manner in which the requirements can be met depends on the nature of the policies that must be enforced. For example, policies that require reliable (i.e., continuous) system operation imply dependability requirements very different from those required when policies are oriented to safety.

The systems that we have been targeting for the safety kernel prototype are those such as the MSS and UVAR where the primary concern is with safety. In systems of this type, the concern is that no harm be done. Although desirable, the delivery of a useful service is not critical. This has very important ramifications for the system design because it is acceptable for a component or the system to fail as long as activity after the failure can be managed to avoid hazardous operation. In the case study systems, there are established routines for responding to failures and bringing the system to a safe state (scrammed for the UVAR and coils and X-ray sources off for the MSS). The components and the system will both leverage off of this characteristic.

5.3 Meeting Prototype Dependability Requirements

This section examines the general options available for meeting the requirements described above. Issues in choosing between options are also considered. Specific techniques for meeting the reliability requirements are discussed in Sections 5.4 and 5.5.

In order for a component to fail and interfere with policy enforcement, the component must first fail and then the failure must not be detected or addressed by other system components. This is the scenario that is depicted in Fig. 12. The figure suggests that there exist two options for a component to operate safely. The first is for the component to provide its specified functionality without failing. The second is to ensure that if the component fails it will do so in a manner that can be detected and managed by other components. To permit this second option, a component must fail with acceptable failure semantics [12,25]. Thus a reliability requirement can be satisfied either by correct functionality or correct failure, where "correct" means that the behavior conforms to the specification.

The above discussion implies that the two basic options for meeting dependability requirements are *prevention* and *detection-and-response*. This is the case with all of the requirements that will be examined in this chapter. The viability of detection-and-response for meeting reliability requirements depends on a means of first detecting and then responding to a failure. Issues in selection between these options will be discussed below.

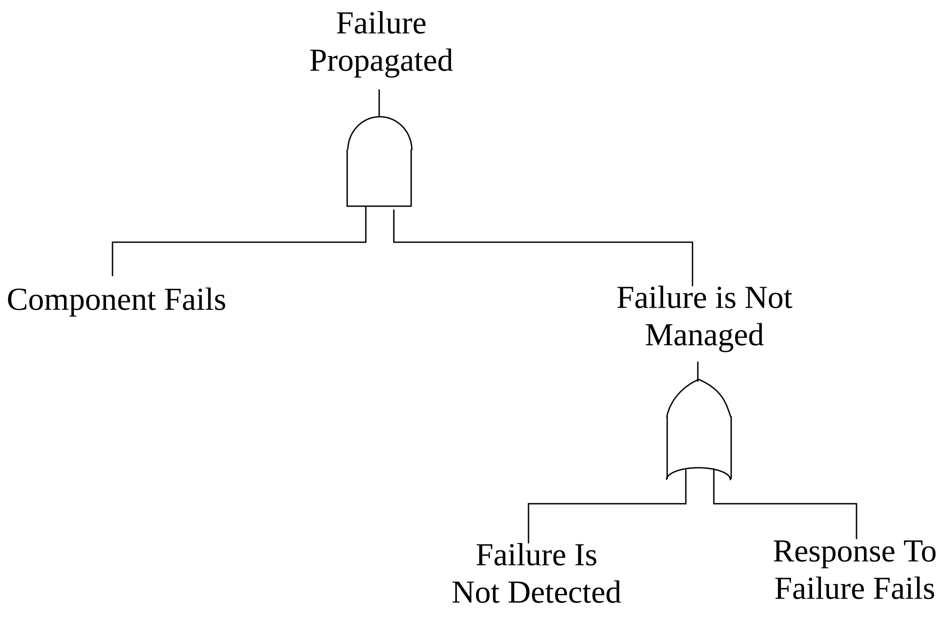

Fig. 11. Fault tree for component failure.

Although, the options are presented as an either-or choice, the reality is that both prevention and detection-and-response will often be employed. The argument for doing this is that the probability of failure when independent methods are used is the product of their individual failure probabilities. The result is that even two unacceptable failure probabilities might result in an acceptably low failure probability when combined in this manner. Additionally, in a system where safety is the primary concern (i.e., a system where the probability of reaching a safe state in the event of a failure is high), the probability of the safety kernel failing to detect and respond to a failure is approximately the probability of failing to detect the failure.

In the event that prevention and detection-and-response methods are not independent or that the degree of independence cannot be assessed, the decrease in the failure probability will not be as significant as in the case described above. However, assuming that the detection-and-response method only fails by not detecting a failure, the resulting probability of failure will be no worse than the probability of failure of prevention alone. If the detection-and-response method can cause a failure even when no actual failure has occurred, then the system fault tree would need to be modified to account for this additional failure mode.

In developing both prevention and detection-and-response techniques we will make the assumption that, although system components might be faulty, they are not malicious. This is a very important difference between safety and security systems. The lack of malicious components permits probabilistic arguments concerning component failure that would not be appropriate in a security system.

In many cases requirements could be satisfied by more than one technique or design. The pragmatic issues of quality assurance, cost, and functional performance provide additional criteria needed to evaluate design choices. These issues and their impact on the system design are discussed below.

- Quality assurance
  For a given reliability requirement, there will typically be several different techniques or designs that could address the requirement. A critical factor in evaluating alternatives is the feasibility and practicality of demonstrating reliability with respect to the requirement. Toward this end, preference will be given to designs and techniques that permit automation of development processes, reduce or simplify the software to be verified, facilitate reuse and generally facilitate quality assurance. For example, consider the problem of ensuring the integrity of data received from a complex file system. One alternative is to demonstrate that the file system will never corrupt data. Another alternative is to detect corruption utilizing redundant information in the data. Assuming that the software required to detect corruption is simpler than the file system, the second alternative is potentially a better choice.

- Cost
  In addition, to quality assurance, design choices must be evaluated with respect to cost. Issues include the effort required to develop a system, the generality of the system, the reliability provided by the design, and the reliability required by an application or class of applications.

- Functional performance
  A goal of the safety kernel is that it should be compatible with a range of safety-critical systems. This has implications for the functional performance of the safety kernel in regards to its compatibility with various application architectures and the possible performance overhead it might impose.

5.4 Ensuring Exclusive Control

Section 5.1.1 identified the requirements for exclusive safety-kernel control of the application devices in a system. This section looks at options for meeting these requirements. The specific technique or combination of techniques that will be used with an application depends on factors such as the characteristics of the devices and the requirements for device operation. The techniques that are examined here have been employed in other systems and so are not presented here as novel techniques. However, their application to the set of reliability requirements associated with the safety kernel is novel.

As shown in Fig. 12, the obvious first condition for an exclusive control failure is that some entity other than the safety kernel must gain access to and be able to control a device. The second condition is that the safety kernel must either be unable to detect the unauthorized access or fail in its response. This indicates that there are two options available for ensuring exclusive control. The first option is to prevent any control, e.g., by preventing access to the device. The second option for ensuring exclusive control, is for the safety kernel to be able to detect control by another entity and restore the control that is required by the safety policies.

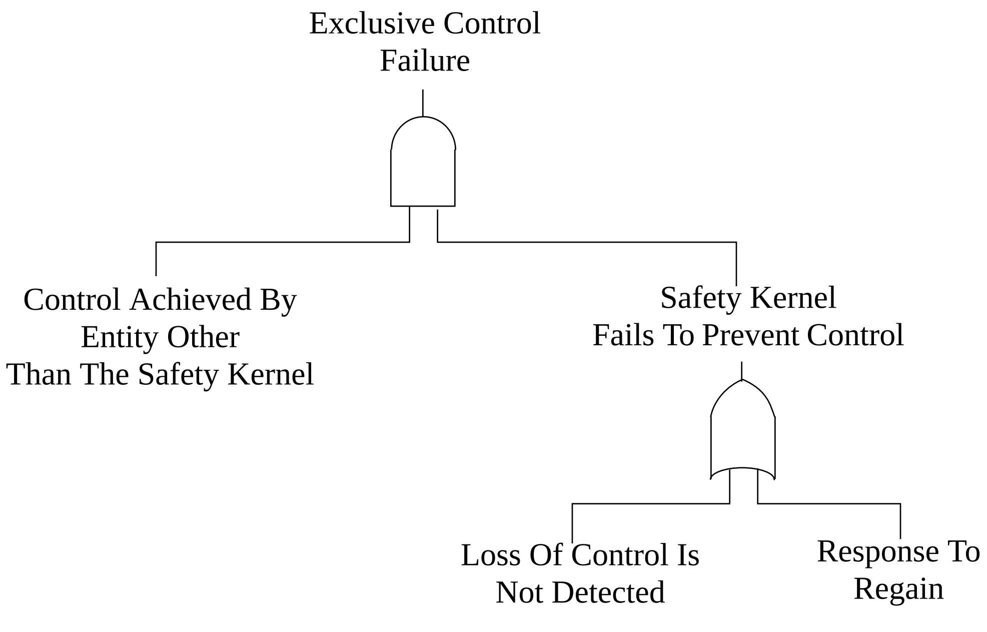
Exclusive Control Failure
Control Achieved By Entity Other Than The Safety Kernel
Safety Kernel Fails To Prevent Control
Loss Of Control Is Not Detected
Response To Regain

Fig. 12. Fault tree for exclusive control failure.

The security kernel approach to ensuring exclusive control is to provide the only means of accessing an information device. To realize this, a security kernel is implemented as a system kernel. A similar approach could also be employed with the safety kernel. However, development of a system-kernel design can be expensive and for many safety-critical systems is not necessary. It is not necessary because of the distinction between exclusive control and exclusive access and the assumption that the safety kernel is not dealing with malicious entities.

A means of ensuring exclusive control is through the use of *authentication techniques*. For example, the file system protection offered by UNIX is an authentication technique that could be used to restrict access to device drivers. Unfortunately, although this would prevent direct access, it does not help with access resulting from failures in the computing platform. A more effective authentication technique utilizes capabilities [47]. A capability is typically a bit pattern that is sufficiently long to make it improbable to be arrived at randomly. For an authentication technique, the capability serves as a key to a device. When a request is made to a device, the key must be included in the request for the request to be fulfilled. With a copy of this key, the safety kernel can control the device, but it is highly improbable that any other entity would be able to achieve control.

Two additional techniques for providing exclusive control are based on the concept of *closed-loop control* of devices. Essentially, they utilize a communication sequence to and from the device to enable the safety kernel to evaluate the state of the device and identify unauthorized control. The first technique of this type is a control prevention technique known as *command acknowledgment.* In this scheme, a command is first sent to a device. The device must echo the command, and then wait for safety kernel acknowledgment prior to executing the command. By withholding acknowledgment, the safety kernel can prevent execution of commands it did not initiate. The command acknowledgment interaction is designed so that although some entity might be able to access a device it would be improbable (assuming that the entity is not malicious) that it could both send a reasonable command and a correct acknowledgment to cause the device to act.

The second closed-loop control technique is a detection-and-response method that takes advantage of the fact that externally an exclusive control failure is indistinguishable from a failure of the device itself [25]. In either case, the operation of the device deviates from what has been directed by the safety kernel. Therefore, *independent sensors*, which are typically employed to detect and respond to device failures, can also be used to ensure exclusive control requirements. With this technique, an exclusive control failure is detected as unexpected device behavior and a response is carried out to restore control of the device.

The independent sensor technique is an indirect method for ensuring that requests executed by a device are valid. As a result, it is limited by the state observation delay and by the need to have an effective response to meet the requirements for controlling the device. Both of these factors restrict the number of situations for which this technique is appropriate. For example, in a system like a missile launcher it is not feasible to detect and then respond to an exclusive control failure that results in launching of the missile. In this situation, a more restrictive technique such as command acknowledgment would be essential.

5.5 Ensuring Safety Kernel Reliability

In order to meet the requirements outlined in Section 5.1.2, it is necessary to analyze each requirement to determine the means by which components in a system could compromise reliable operation of the kernel. This section will look at the requirements of data integrity, dependable support services, dependable computing services, and dependable resources. The other requirement, safety kernel correctness, is addressed in Chapter 7.

5.5.1 Data Integrity

This section examines issues and techniques for ensuring the integrity of instructions and data of the safety kernel, support services, and basic computing services. During operation, the data resides primarily in memory, but the integrity of static instructions and data from secondary storage is also a concern. The first concern with data integrity is the initialization of the data in memory. The second concern is data integrity during operation.

Initialization

The data that is used in the operation of the safety kernel must be that which has been developed to enforce the safety policies. This data includes the executable representation and configuration data. The storage and handling of the executable representation needs to be shown to not fail or some means of error detection needs to be employed (e.g., a checksum on the loaded instructions and data). The integrity of configuration data can be most easily ensured through the inclusion of redundant information in the data to facilitate error detection.

Operation

During operation, the primary data integrity concern is that some entity in the computer system will be able to access and alter memory that is critical to policy enforcement. Although, some sort of fault tolerant data structures [48] might be effective for detecting corruption of this type, there is no means for detecting corruption of all of the data because much of this data is never accessed directly by the safety kernel (e.g., process state data).

A primary means of ensuring data integrity is to employ memory protection to isolate data and thereby eliminate many sources of corruption. In many systems, instructions are protected in memory designated as read only. For the rest of the data, there are several design alternatives that can potentially provide for protection. Two common choices are the use of supervisor mode or utilization of the protection provided by the memory management system (e.g., virtual memory). These two alternatives are explored below.

Implementing the safety kernel with supervisor status would provides protection from user processes, but can involve potentially expensive modification of the system kernel. However, some systems, for example the micro-kernel-oriented Chorus system, provide support for moving a process into the system kernel [7]. An additional concern, particularly in a monolithic system kernel is the lack of protection within the system kernel address space. The use of a microkernel operating system [7,23] or an object oriented system [10] would provide protection within the system kernel. The other protection option is to implement the safety kernel as a user-level process and utilize virtual memory for protection. The protection provided by virtual memory is in theory just as good as that available within the system kernel. Both, in fact, are contingent on the correct operation of computing services.

An additional protection issue is dealing with the failure of computing platform memory. Since the safety kernel cannot ensure detection of a failure of this type, it is essential that underlying hardware and software address this problem.

Protection cannot guarantee data integrity because corruption can be caused by failures occurring inside of a protection boundary or by failures that occur when memory access has been provided to external entities. There is no certain means to detect corruption caused by either of these sources. As a result, it will be necessary to verify that failures of this type will not occur.

5.5.2 Dependable support services

In addition to the requirement for data integrity discussed above, it is necessary that the support services invoked by the safety kernel produce correct results in a timely manner. Options for meeting these requirements are discussed below:

- *Correct support service functionality*
  The feasibility of detecting a failure in which information is incorrectly manipulated depends on the nature of the operation. For example, when an operation is used to communicate information there are many ways of incorporating redundant information that enables the detection of a corrupted message. On the other hand, in operations where information is being generated (e.g., data from a sensor, a timer or dynamic memory allocation) there is usually no certain means of detecting a failure since there is no initial knowledge of the information. In general, the reliability of software of this type must be demonstrated by verification, by comparison with another independent source that has access to the same or correlated information, or by a reasonableness test based on extreme or expected values. Other operations apply a function to input data to produce a result. Operations of this type must either be verified to have an acceptably low failure probability or must have some sort of check (e.g., reversal or replication) to detect a failure.

  An additional issue with communication services, is whether a request is delivered to the specified destination, particularly a device. Failed communication to a device appears the same as a device failing to execute a request, and so closed-loop control can be used for detecting failed support service communication.

- *Timely support services*
  An option for meeting this requirement is to demonstrate that a support service will always be timely. This is a viable alternative for some support services. However, although this method might effectively eliminate design faults, it does not deal with hardware degradation failures.

  Another approach is to detect delayed operation by requiring the safety kernel to generate periodic “heartbeats” to indicate its liveness. For example, during critical operation a device might require receipt of a valid safety kernel message every T

seconds. If a message did not arrive in time, the device would transition to a safe operational state. A related application of this technique extends the safety kernel to include a component known as the *safety kernel watchdog*. This component is equipped with a watchdog timer and the resources to either bring the system to a safe state or to maintain some minimum level of operation. The mode of operation of the kernel watchdog is to direct the system to this basic, safe state unless timely messages are received.

5.5.3 Dependable Computing Services

There is no certain way to detect failures of the basic computing services that permit operation of the safety kernel. Therefore, it is essential that these services either function as required or fail in a manner that can be handled by the safety kernel. Because of the presence of the safety kernel watchdog, fail-stop is an acceptable failure semantics for the basic computing services. Failures that result in corruption of the safety kernel data or altered operation of the safety kernel are not acceptable. Some of the basic computing services can be eliminated. For example, by locking safety kernel pages into memory, swapping of the safety kernel memory pages is no longer required and as a result is not a source of failure.

5.5.4 Dependable Computing Resources

As an active enforcer of safety policies, the safety kernel must be able to either acquire essential resources to enforce safety policies or it must be able to detect and respond to a lack of resources. The determination of what constitutes essential resources depends on the system and safety kernel design, but possibilities include processor resources, memory, and secondary storage. From the policy enforcement perspective the concern with a lack of resources is that the safety kernel might not be able to act in a timely fashion. This presumes that resource shortages (e.g., a lack of memory) will not lead to an uncontrolled failure because the shortage can either be detected or because it has no other impact than to slow down the operation of the kernel (e.g., when there is competition for the processor).

Resource shortages can occur as a result of competition for resources or because of a failure in the element of the computing platform managing the resource. Competition for resources can be managed to ensure that the safety kernel receives sufficient resources. For example, to manage processor resources, a real-time operating system could provide preemptive, priority-based scheduling that would permit the safety kernel to be given priority access to the processor. Failures that affect resource availability must be detected and an alternate resource must be available to permit timely operation. The safety kernel watchdog described above for dealing with delayed support services can also be used to detect and respond to a resource shortage.

5.6 Prototype System Design

Using the techniques described above, a range of system designs could be developed that would meet the requirements for systems such as the MSS and UVAR. In this section we look at one design option. The design that is presented is similar to the design that will be used with the prototype for the MSS. However, some of the features will not be incor-

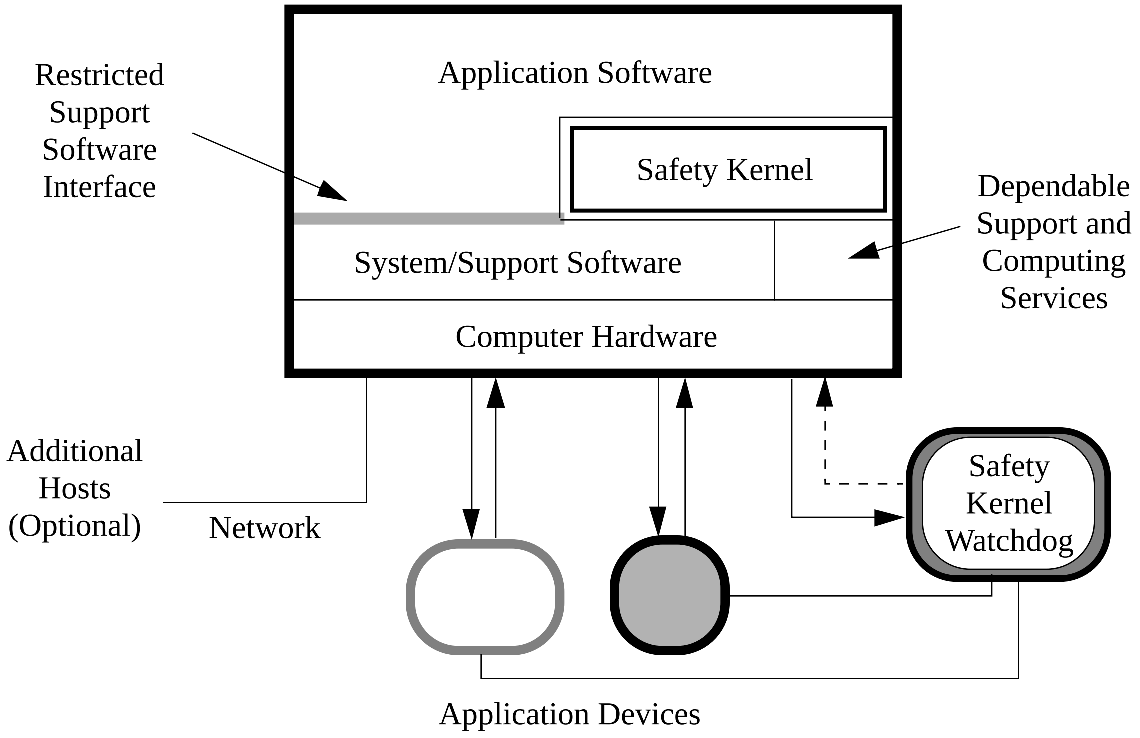

Fig. 13. Safety kernel system design.

porated in the prototype at this time. Implementation and verification of a system design that addresses all of the reliability requirements is left for future work. The features of the system design (shown in Fig. 13) include the following:

- Safety kernel as a user-level process.
- Application software and safety kernel communication via network.
- Command authentication and error detection encoding for device communication.
- Closed-loop device control using independent sensors.
- Safety kernel watchdog.
- Restriction of application software resource usage.
- Microkernel architecture for system/support software.
- A core of dependable support services and basic computing services.
- Incorporation of redundant information into configuration data.
- Error detection analysis of safety kernel executable following loading into memory.
- Support for real-time operation of the safety kernel.

The rest of this section evaluates the system design with respect to each of the requirements identified for reliable policy enforcement:

1. *Exclusive control*
   Exclusive control is provided for by a combination of techniques including authentication using capabilities and independent sensors. In the event that the safety kernel cannot regain control communicating via the computing platform, the safety kernel watchdog provides an alternate means of communicating with the devices.

2. *Safety kernel correctness*
   The system design has little impact on meeting safety kernel correctness requirements.

3. *Data integrity*
   Safety kernel memory protection in this architecture is provided by the virtual memory system of the computing platform. Assurance of protection for the safety kernel requires verification that the virtual memory will function as specified or fail with acceptable failure semantics.

   Support services and computing services that are inside of the protection or have access to safety kernel or support service data will need to be verified to not corrupt that data. Correct initialization of executable safety kernel and configuration data is verified by error detecting software.

4. *Dependable support services*
   Where feasible, error detection will be used to monitor the operation of support services (e.g., in communication with devices). Other operations will either need to be reliable or appropriate techniques will need to be developed on a per-operation basis to detect and cope with operation failures. The microkernel organization of the system software facilitates the implementation and verification of these services by providing “firewalls” that isolate the elements of the system software. This enables one module of the system to be implemented and verified to operate as specified with high probability without needing to worry about other less trustworthy modules failing and causing a failure in the critical module.

   One of the significant implications of the safety kernel watchdog in this design is that the support software and hardware do not need to be verified to be reliable. Instead they need to be verified to either operate as specified or stop if a failure is detected. This presents a potentially easier implementation and verification task than if the support software needed to achieve high reliability. For example, this could be achieved in the hardware with a dual redundant system that compared the results of each operation and stopped if there was a discrepancy [51].

5. *Dependable computing services*
   The basic computing services of the computing platform must either function as specified or be fail-stop [31]. The memory pages of the kernel will be locked in place to obviate the need for reliable swapping.

6. *Dependable resources*
   The management of storage and processor resources used by the application software provides assurance that competition will not prevent the safety kernel from acquiring the resources required for operation. The safety kernel watchdog provides for detection and response to resource failures.

Evaluating the system design with respect to the pragmatic concerns, the most significant quality assurance and cost concerns are with the need for dependable support services and computing services. However, it is important to note that the support services that are required are primarily a function of the safety kernel functionality and not the system design. In the same way a set of basic computing services are required regardless of the system design. This design is sufficiently flexible that it could be compatible with a range of applications and it also supports a parameterized safety kernel implementation.

5.7 Conclusion

This chapter has identified a set of requirements for reliable enforcement of kernel-enforced safety policies. It should be noted that the requirements for reliable policy enforcement by the safety kernel are no more stringent than those that would apply for reliable policy enforcement by the application software. It is reasonable to infer, therefore, that the safety kernel does not complicate the reliability requirements for a software system.

Techniques for meeting the reliability requirements have been examined in this chapter. These techniques make use of the fact that the requirement with the target systems is safety rather than reliability. The implication is that components can operate dependably by either correct function or correct failure. Additional flexibility for meeting the requirements for reliable policy enforcement results from the assumption that components are not malicious. Failure prevention and failure detection-and-response techniques have been identified to address the requirements. In many cases, failure detection is a viable alternative that provides significant cost and quality assurance benefits. In other cases, detection-and-response is not feasible or practical, and a component will need to be verified to operate dependably. A system design has been developed for the MSS and UVAR using the various techniques described.

6 Safety Kernel Implementation

To this point we have examined kernel enforcement of safety policies and have established a set of requirements for reliable enforcement of these policies. A logical next step is to consider alternatives for implementation. In looking at alternatives, the goal is to identify an implementation strategy that provides both reliability and cost benefits.

One implementation alternative is to “build from scratch.” This approach is certainly acceptable for developing a safety kernel based on the concepts discussed in previous chapters. However, “building from scratch” is typically expensive and more importantly does not promote the transfer of innovative and effective concepts and artifacts from one system to another. The transfer of information of this type can positively impact both the cost and reliability of the system [43]. This transfer of information is known as reuse and is an important area of research in the field of software engineering.

We are motivated to exploit reuse because of the potential cost and reliability benefits. The existence of general classes of safety policies and experience with a variety of applications has lead us to believe that reuse techniques are feasible in the implementation of the safety kernel. Therefore we have investigated the possibility that the safety kernel architecture permits the use of a general framework that can be exploited to support abstraction of general knowledge, system design, safety policies, and software artifacts.

In the development of a software system, many different forms of reuse can be employed including knowledge, specification, design, and code reuse. For example, with the safety kernel, alternatives include reuse of a canonical design, a parameterized application generator, or a complete implementation. The level of reuse that is appropriate depends on the requirements and characteristics of the system for which reuse will be employed. To identify a form of reuse and thus an implementation strategy applicable to a range of systems, information from the two case studies and other applications was used to develop general requirements and characteristics.

Fig. 14 represents in an abstract manner the vision that we are pursuing for implementation. A software safety specification details the software safety requirements for an application. This is used as the input to some as yet unspecified process that produces an implementation of the safety kernel from the specification data. The process will utilize some form of reuse and in an ideal situation will be automated. To derive the most benefit in the area of reliability, the implementation strategy must consider factors in addition to reuse. For example, the process input data that is derived from the software safety specification strategy should be in a format that permits verification analysis. It is also critical that the design and implementation of the components and the system facilitate the demonstration of reliability.

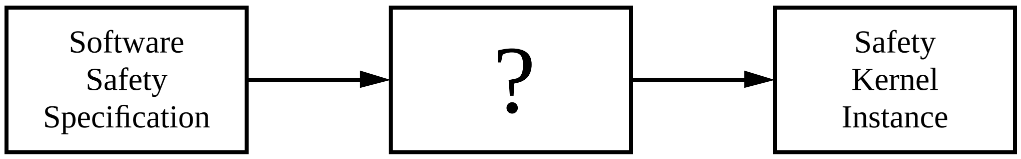

Fig. 14. Concept for safety kernel implementation.

6.1 Requirements Analysis

In this section we document the general requirements for the safety kernel. These requirements can be derived from analysis of two areas. The first area is the functional requirements and these are largely determined by the safety policies. The second area is that of interaction with system components. By characterizing the safety kernel's interactions with other components, the general requirements of the context in which it operates can be established. In the process of examining the interactions and characteristics, models can be developed that represent an abstraction of the context within which the safety kernel must operate.

In looking at the requirements for policy enforcement and for interaction with other system components, the question to be addressed is what is the degree of generality of the information and functions required by different applications. Questions of this type are addressed by the process of domain analysis. For the purposes of establishing an appropriate level of reuse, we have analyzed the information and functionality requirements with respect to the following descriptions. The descriptions characterize the generality of the requirement and correspond to points on a continuum of relative generality. Recognize that this continuum is defined based on knowledge of systems which are potential target applications for the safety kernel.

1. General Value
   The information or function is one that is general to the target applications.
2. General type
   The type of the information or function is general to the target applications. The actual value is application-specific.
3. Application-specific type
   Both the type and value of the information or function are application-specific.

6.1.1 Policy Enforcement Requirements

In looking at the classes of safety policies it is apparent that they can be grouped according to the means by which policy enforcement is invoked.

- *Interlock policies*
  These policies are invoked as a result of an application software request for some

action. In this set are policies from the classes device input from software and application software error. These policies specify conditions that must be met in order for a request to be executed and are expressed as a function of the command, the state of a device, the state of operation of a system, the parameters of a command, timing requirements of device operation, etc. Policies of this form are general to the applications we have examined, but the exact policies are unique to their specific applications. Therefore, this is a situation where the type of the policy is expected to be consistent, but the actual policies will not be general.

- *Monitoring policies*
  These policies are invoked to enforce policies that check the operational state of the system. Policies from the classes system operation, device operation, device failure and some from software error comprise the monitoring policies. To enforce a policy of this type, the safety kernel must be able to make a prediction of the expected state. This prediction will be based on the commands that have been sent to the device or devices and on a model of the operation of the device or system.

  Monitoring policies can be invoked on a periodic or aperiodic basis. Periodic monitoring activities are performed according to a static schedule. The schedule that is used at a particular time is determined by the state of both the device and the system. Periodic monitoring policies require the safety kernel to incorporate a mechanism for scheduling these monitoring activities. Aperiodic monitoring activities are scheduled relative to some event. For example, a check to ensure that an x-ray source was on for no more than a certain period would need to occur the required period of time after the source was activated. To enforce policies of this type, the safety kernel might maintain a timer queue that permits these aperiodic events to be scheduled.

  Ensuring schedulability in systems with aperiodic events is a general problem in real-time systems [28]. We will address this problem by requiring that aperiodic events be limited or that they occur at times when their invocation will not result in scheduling conflicts. The interlock policies can be used to regulate the occurrence of aperiodic events. In systems where safety is the primary concern, another approach is to detect situations in which events cannot be scheduled and then invoke a failure response. The frequency with which monitoring activities can be carried out depends on factors such as the timing characteristics of the computing platform (e.g., the context switch time and the timer resolution), the number of monitoring activities that need to occur, and the frequency of requests from the application software.

- *Failure response policies*
  Failure response policies are invoked in response to failures detected either by the safety kernel or by the application software. In some cases there may be several responses to a particular failure. The responses might range on a continuum from the least to the most drastic. In a case such as this, the safety kernel failure response mechanism will need to maintain a record of the failure responses that have been attempted in order to select the appropriate response for a specific history of failures

and responses. The failure response state of a device or system can also impact the policies that are applied to control of devices. It will be assumed that the most drastic failure response will always be able to respond to a failure.

6.1.2 System Requirements

Beyond the requirements of policy enforcement, the safety kernel is also constrained by the requirements and characteristics of its interaction with other components in the system (see Fig. 15). In the process of examining the interactions and characteristics, models have been developed that represent an abstraction of the context within which that the safety kernel must operate. Models have been developed to characterize the following:

- *Application software*
  What will the application software require of the safety kernel and how will the two interact? What is the interface between them?
- *Device control and operation*
  How are devices controlled by software? How can the operation of a device be characterized to facilitate expression of policies that relate to device operation?
- *System component architecture and organization*
  How are the components in a system organized and what is the position of the safety kernel? How should the interaction of these components be modeled to permit the expression and enforcement of a range of safety policies that relate to operation of multiple devices in a system?

The models that have been developed in these areas provide the context not only for operation of the safety kernel, but also for the expression of the safety policies. Fig. 15 shows the inputs and outputs of the safety kernel. I_A and O_A are respectively the inputs to

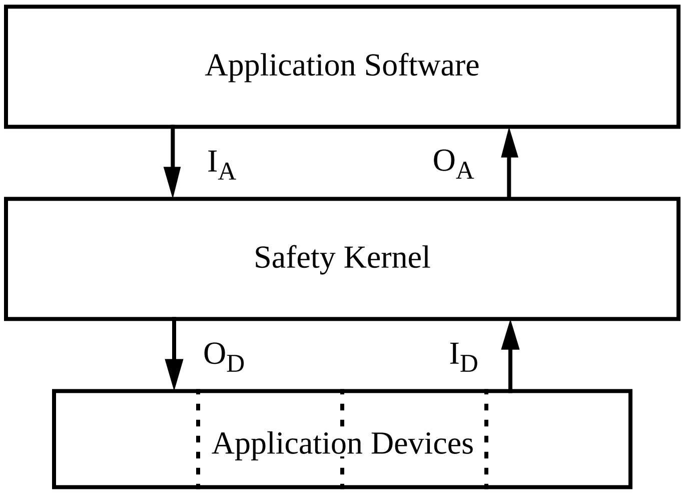

Fig. 15. Safety kernel in system context.

the safety kernel from the application software and the outputs to the application software from the safety kernel. Similarly, I_D and O_D represent data exchanged between the safety kernel and application devices. Safety policies describe requirements that are placed on inputs and outputs of the safety kernel. Therefore, a safety policy is expressed as a function of these inputs and outputs. As a result, characterizing these is a critical aspect of defining the safety kernel context. The remainder of this section look at each of the areas itemized above and examines the models that have been developed.

Application Software Model

The primary interaction between the application software and the safety kernel consists of application-software requests made via the safety-kernel interface. The safety-kernel interface must provide parameterized commands for both control of devices and enforcement of application software error safety policies. These commands must also provide a means for the application software to acquire necessary system and safety kernel state information. The interface might also include a set of requests that the application software can call to invoke safety kernel failure response routines. The interface required by a given application will be tailored for the devices in the system and the safety policies that need to be enforced.

The requests to the safety kernel from the application software are asynchronous. These requests must be serviced in a timely manner along with the scheduled system monitoring activities. In particular, it is essential to be able to ensure that scheduled operations will occur within a specified deadline relative to their scheduled time and that with some assumptions about arrival patterns that requests will be responded to within a specified period of time.

Device Model

An application device is a physical entity that is able to act on other devices or elements of its environment. The safety kernel communicates with the device via a specified set of control commands and parameters. A device has a physically observable state known here as the *operational state*. Included in the operational state are observables that are directly tied to the operation of the device. For example, the current in a coil would be part of the operational state for a current controller. The operational state can be observed by the device itself or by independent sensors. The operational state of a device is determined by the initial device state, commands received by a device, the timing of commands, and the interaction between the device and its environment. As a part of the device model, we assume that under normal operation, the operational state of the device can be predicted based on this same information for the device.

The interlock policies to be enforced at a given time depend, among other things, on the specific command being called, any parameters, and the command history. The monitoring policies enforced at a given time depend on the expected state of the device. To attempt to characterize the state of a device, the device model characterizes a device as a finite state machine. The states for the device model are known as *modes*. The mode of a device is determined by the initial state of a device and the commands that have been received by the device. Transitions from one mode to another are specified as safety policies and occur as a result of the successful execution of commands.

In many cases modeling the device as a finite state machine will not be adequate. For example, the operational state of a device may depend not only on the command history, but also on parameters included with the commands. Therefore, a device can have associated with it a set of *control parameters* (e.g., the current setting of a servo amplifier) that record values used to control a device and predict the operational state. These control parameters can be updated when a command is successfully executed. The combination of control parameters, mode, operational state, timing data, parameters and command information are used to express safety policies for a device.

System Model

From the perspective of the safety kernel, a system is composed of interacting devices. These devices can be grouped into subsystems and the entire collection of devices comprise the system. In addition to devices, a system has some physical environment that the devices act on. Associated with both the devices and the environment are physically measurable quantities that must be observed to track the operation of the devices and the response of the environment to the actions of the devices. Therefore, devices, subsystems, an environment and measurable quantities are the essential elements in the safety kernel's abstraction of the system.

A system has an operational state which consists of the observable elements of the physical system state. Examples of system observables for the MSS include the seed position and the field produced by superpositioning of the fields from the individual coils. To perform monitoring of the system, the operational state of the system is predicted based on the command states, control parameters, and timing variables of all of the devices in a system.

A system can also have a mode associated with it. This mode can either be a result of the states of the system devices or it can be used to define an operating mode which restricts the operation of system devices. Such restrictions are expressed as policies that are a function of the system mode.

6.2 Implementation Strategy

It is intuitively clear, but the previous section provides convincing evidence, that no single implementation of the safety kernel will be able to serve more than a few (likely one) applications. Furthermore, given the wide variety of safety policies it is not practical to develop a comprehensive software library where modules could be mixed and matched to implement a safety kernel for a given application. What then is an appropriate reuse strategy for implementing the safety kernel?

6.2.1 Reuse Strategy

The observations of the previous section concerning the functionality requirements of the safety kernel and the characterization of component interactions point to the existence of common *types* of functionality and to general features of the context. Therefore, the implementation strategy that has been developed incorporates the basic types of information that have been identified. Application-specific functionality and features are specified through parameterization of the various models and policy frameworks.

Reuse in this implementation strategy occurs at many levels. First, design reuse is employed by encapsulating a standard design into a translator. The design reflects the high-level requirements associated with enforcement of the three types of safety policies and the essential model of the system that views the system as including the application software and a collection of devices. The translator reuses certain enforcement and support mechanisms, so code reuse is also employed. Specification reuse is promoted by a uniform format for specification of system configuration and safety policies. Finally, the use of consistent abstractions and notations across applications supports the reuse of safety policies and the knowledge and techniques used in their development. All of these different forms of reuse are realized using the translator and the special-purpose programming language that is used to specify the system configuration and safety policies.

The overview of the safety kernel implementation concept is shown in Fig. 16. From the software safety specification, a policy specification describing the system, devices, and safety policies is developed. The process used to develop the policy specification from the software safety specification is as yet informal and not automated. An area of future work is to refine the transformation and/or the nature of the system safety specification and the safety policy specification so that the transformation is as straightforward as possible and preferably automated. The translator uses the safety policy specification in combination with built-in mechanisms and context to produce an implementation of the safety kernel.

Adoption of this translator approach results from our observations that although the specific devices, policies, etc. of applications would not be consistent, the characteristics of the devices, policies, etc. would be. A kernel-enforced, safety policy expresses some condition that is a function of the inputs to the safety kernel (commands, parameters, observations of system and device state), outputs from the safety kernel (requests to devices), and state information maintained by the safety kernel (command histories, device operation histories). Taken together all of these things form a context for the expression of safety policies. The safety kernel context is composed of the following elements:

- Common control mechanisms
  Control mechanisms include support services of which the translator has an internal

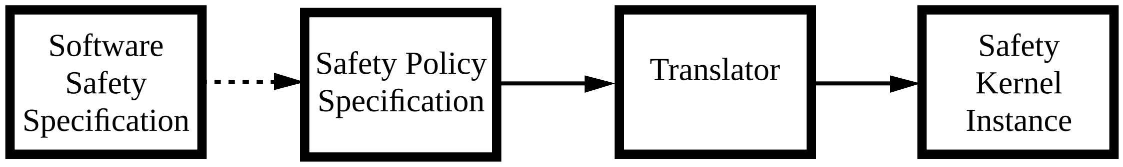

Fig. 16. Translator-based implementation of the safety kernel.

knowledge and incorporates into an instance of the safety kernel. The safety kernel depends on some support services, such as a scheduler, that are used in each instance of the safety kernel. However, the most important control mechanisms are those that carry out the processes that have been established for enforcement of safety policies. For example, when a request is made by the application software, an established process is followed to enforce interlock policies. Steps in this process include updating timing information, capturing relevant operational state, determining the mode of the device/system, checking if the command is valid for the mode, performing the request if all policies have been satisfied, and finally updating the mode and other state information. Similar well-defined processes exist for monitoring devices and responding to failures.

- Common variables
  There are common variables that are defined for the safety kernel for use in expression of safety policies. Examples include the present mode for a device, the total time spent in a mode, the time a mode was entered, the time of the last command, and the operational state vector for a device.

- System and device specific configuration information
  This information is specific to each application, but the type of information is common across applications and has been identified. Therefore, providing this information is a matter of developing the specific type of information for a given system. Essentially, the translator expects certain data to be provided and uses this to build the system context. Information of this type includes the number of devices in a system, the modes for a device, the commands for a device and device control parameters.

Within this context the safety policies for a system are specified. Details of the system context and the types of policies that are enforced will be presented in Section 6.3.

6.2.2 Parameterization

The specification of context information and of the safety policies can be looked at as parameterization of a general safety kernel framework that is provided by the translator. An obvious question, is what is the nature of this parametric information? For the purposes of simplifying safety kernel verification and implementation, ideally, the parameters would be as simple as possible. The simplest type of parameter is a *static parameter.* A static parameter is information that is used to select between available options or that is used as a value to control the actions of an operation. Examples of static parameters include numerical values and character strings. To be employed effectively, static parameterization would require that any required operations and options be built into the translator. Experience with the models above shows that this is unrealistic. The models demonstrate that although there exist common operations, the diversity of application devices and safety policies makes it infeasible to provide a basic set of operations that would be complete and general.

A more expressive parameterization alternative is to utilize a combination of static and *executable parameters*. Executable parameters specify operations that are to be executed by the safety kernel. An executable parameter would take the form of lines of software or

Failure Mode	Safety Kernel Response	Effect
Failure status returned	Detected by safety kernel, failure response invoked	Safe operation is maintained
Incorrect functionality	Failure is not detected	Policy is not enforced as specified
Corruption of safety kernel	Failure is not detected	Result is unpredictable
Operation does not complete	Detected by safety kernel watchdog	System is transitioned to shutdown state

Table 1: Executable parameter failure mode analysis

the name of a procedure. Use of an executable parameter is appropriate when the general function and context of an operation is known but the specifics of the operation are not. For example, with a request to a device it is likely that checks would be applied to any parameters each time the request was made. However, it is unlikely that a static parameter could be used to select from a set of checks that performed even a fraction of the possible checks required by even a few applications. On the other hand, executable parameters, possibly modified by static parameters could be used to specify the desired checks.

Special requirements must be applied to the use of executable parameters due to their potential for interfering with operation of the safety kernel. The executable parameters that are incorporated into the safety kernel will need to exhibit certain desirable properties. The one property that must be assured is that the executable parameter must not be able to interfere with the operation of the rest of the safety kernel. For example, it would be essential that an executable parameter not be able to corrupt the memory of other components of the safety kernel and thereby lead to erroneous operation of those components. Another desirable property, is that an executable parameter operate as specified or fail in a specified manner. This is a verification issue and cannot be addressed in the design of the safety kernel. Another desirable, although not essential, property is that the executable parameter should be guaranteed to complete in a specified time. This property is not essential because either the external safety kernel watchdog or an internal watchdog timer is available to maintain system safety in the event that operations are not timely. Table 1 summarizes the executable parameter failures modes and the effect that the failures have on the safety kernel. Where possible, executable parameter reuse will be employed for common operations to utilize components that demonstrate desirable properties.

Some policies will require that the safety kernel execute complex executable parameters that cannot be shown to exhibit the properties essential for incorporation into the safety kernel. Policies of this type can be enforced in a weakened form where responsibility for execution of part of the parameter is given to the application software. Another means of dealing with such parameters would be to execute them in an isolated safety kernel server, e.g., as a separate process. In this environment, the possibility of the operation producing incorrect results is still present, but failures resulting in corruption of the state would not affect safety kernel operation. The cost of this approach is additional communication overhead.

6.3 Safety Kernel Framework

This section looks in detail at the elements of the safety kernel context discussed previously. In addition, the formal parameters that are used to specify safety policies are also described. Together the context and the policy specification parameters form a framework for the safety kernel. The safety policy specification for an instance of the safety kernel is incorporated into this framework to produce a safety kernel for a particular application. As a part of this section, the language for the safety policy specification is described. The grammar for this language is presented in Appendix B and example policy specifications for the MSS and UVAR are presented in Appendix C.

6.3.1 Machine Abstraction

To permit safety policies to be specified and enforced, the safety kernel has its own representation of a device based largely on the device model described above. Each instance of a device has a command interface that provides the means of controlling the device. To model the operation of a device modes are also employed and transitions between these modes can be specified for each command. Because of this finite-state-machine aspect the abstraction for a device is called a *machine*. However, as described for the device model, the finite state machine model is not sufficient and therefore control parameters are also included in the model. Each machine has its own monitoring facilities and also has a mechanism for reporting and responding to failures. The machine abstraction is used for all application devices.

Another issue to be considered is that the devices in a system can be grouped into subsystems and into a complete system as shown in Fig. 17. Therefore, some model is also required to represent subsystems and systems. In looking at this problem, we observed that the interface for a system is similar to that for a device with a set of commands provided to control the state of the system. In addition, systems such as the MSS and UVAR have system states or modes that impact the operation of the component devices. Working with the safety policies, we have observed that interlock, monitoring, and failure response policies are also required at the system level. Therefore, the same abstraction that is used for devices is applied to subsystems and systems. Only a few minor additions to the machine abstraction are required to support the hierarchical organization and interaction of the machines.

The parameters for a machine are discussed below. The first parameters examined are those that are used to establish the context for expression of safety policies. The parameters for each of the types of policies are examined subsequently, followed by an examination of the built-in control mechanisms.

6.3.2 Built-in Context

Because a consistent abstraction is used for all devices and for subsystems and systems, there are certain variables associated with the abstraction that are predefined. In other

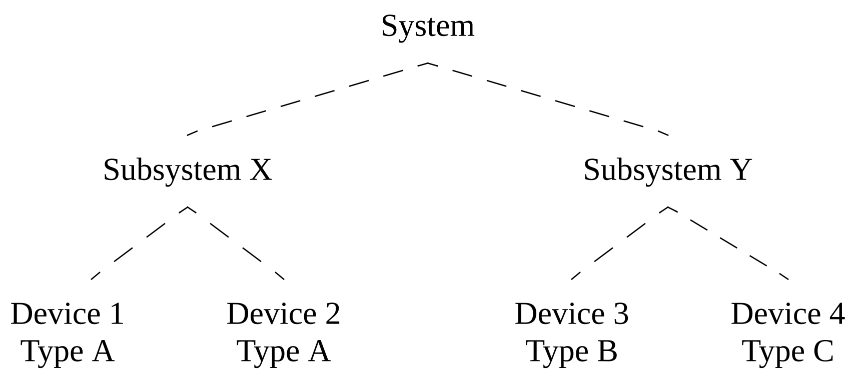

Fig. 17. Example safety kernel machine abstraction hierarchy.

words, these variables are a part of the built-in context. These variables, which can be used for the expression of safety policies, include the following:

- *Mode entry time*
  This is the time that the present mode was entered.
- *Time of last command*
  This is the time of the last successful execution of one of the device control commands. In the situation where a command causes a change in mode, this time will be the same as the *mode entry time*.
- *Time in mode*
  This is the time that the device abstraction has been in the present mode. It is calculated by subtracting the *mode entry time* from the present time.
- *Time since last command*
  This quantity is the present time minus the *time of last command* and indicates the time since the last successfully executed command.
- *Present operational state*
  This is a state vector made up of operational state variables (observable quantities) that are monitored for a device. This vector is updated by the safety kernel whenever an interlock or monitoring policy might require it.
- *Operational state at last command*
  A state vector identical to the one above, this vector contains the operational state that was recorded at the time of the most recent successfully executed command.
- *Total mode times*
  For each mode a record of the total time spent in that mode is maintained.

6.3.3 Application-Specific Context Data

Each machine has associated with it a set of parameters that are application-specific. The type of information included in the machine description is described in this section.

The machine at the very top of the hierarchy, i.e., the one that encompasses the complete system has some information that applies to all of the system:

- *Schedule descriptions*
  The translator supports specification of monitoring activities that are performed on a scheduled basis. Because the scheduling is coordinated at the system level, rather than for each individual machine, schedules are specified only at the system machine level. A schedule consists of a schedule name, times of scheduled events, an estimated activity completion time, and an activity completion deadline. The event times are specified relative to a frame which is discussed below.

  Support for aperiodic monitoring activities will be provided in a future version of the translator. These activities would be specified relative to particular events. However, for purposes of analyzing the feasibility of scheduling monitoring activities and application requests, the specification would include information concerning these aperiodic events, e.g., their maximum frequency.

  If the schedule descriptions for the periodic activities are used in combination with information about the schedules that can be active at any time, static analysis can be used to determine whether activities will be completed within deadlines. An area for future work is to incorporate information describing the computing platform timing characteristics such that the translator would be able to identify situations in which it was not feasible to perform the specified monitoring activities.

- *Frame length*
  Because the monitoring is expected to be periodic, a convenient method for specifying events is relative to a schedule frame that has duration *frame length*.

- *Watchdog schedule*
  The value of this variable must be one of the defined schedule names. It specifies which schedule must be used to send “heartbeats” to the safety kernel watchdog.

For machines that are not at the leaves of the hierarchy (i.e., those that serve to group other devices or subsystems) information is required to describe the subsystem. That information is the following:

- *Specification files*
  For every machine that has other machines as children the specification data files for each type of child must be specified.

- *Listing of component machines*
  The name and machine type must be specified for each child machine.

The parameters for a device are numerous. This information serves to provide a context for expression of safety policies and also to specify how the safety kernel will actually direct the device to act as requested.

- *Machine type*
  This gives the type name for the machine that is being specified. This name is used by a child or parent to identify the machine.
- *Modes*
  As discussed previously, modes are used by the machine abstraction to describe the state of a device or system. The present mode for a device is determined by command history and the initial mode.
- *Initial mode*
  This simply specifies the starting mode when a machine is first instantiated.
- *Interface commands*
  As a part of the safety kernel interface to the application, a machine has an interface of parameterized commands. Included in this interface may be some commands that are provided for the purpose of responding to failures.
- *Action procedures*
  Associated with each command is an executable parameter that determines the actions of a machine for that interface command.
- *Control parameters*
  In many cases modeling the device as a finite state machine will not be adequate. For example, the operational state of a device may depend not only on the command history, but also on parameters included with the commands. Therefore, a device can have associated with it a set of control parameters that record values used to control a device and predict the operational state.
- *Control parameter update procedures*
  The control parameters can be updated when a command is successfully executed. These control parameter updates are specified as an executable parameter for a given interface command.
- *Operational state variables*
  These are the observables that make up the operational state of a machine.
- *Operational state acquisition procedure*
  This procedure is essential to acquire the operational state information for the machine. This procedure is called when any of the interlock and monitoring policies are enforced and may also be called for failure response policies.
- *Constant values*
  These values are defined for a machine and are used to parameterize the safety policies.

- *Executable parameter definitions*
  The definitions of executable parameters can be specified in the specification file.

Some of the information that appears in the policy specification is not essential to the context, but is required to support generation of legal source code and to facilitate the expression of the other elements of the specification.

- *Include files*
  The declarations for types or executable parameters that appear outside of the specification can be incorporated by specifying a header file.

- *Base classes*
  This variable specifies a class that is used as a base class for the particular machine abstraction.

- *Base class constructor calls*
  When a base class is used, this information defines the call that will be made when the base class is instantiated.

- *Constructor parameters*
  This information identifies the type and name of parameters that are to be included in the instantiation of the machine.

- *Class declarations*
  Any types or variables that are to be local to a machine can be specified using this parameter.

- *Global declarations*
  This parameter specifies any types or variables that are to be global to the machine and any machines lower in the hierarchy.

- *Initialization procedure*
  If there are operations to perform upon instantiation of a safety kernel machine, they are encapsulated in this procedure. Actions could include communication with devices and setting of state variables associated with the device.

6.3.4 Policy Specification

This sections looks at the type of information that is used to specify the three types of policies.

Interlock Policies

Interlock policies place conditions on the commands that can be executed by a machine. The conditions depend on factors including the present mode of the machine, the mode of a parent machine, the value of control parameters, and the present operational state. A set of interlock policies is specified for each command in the command interface for a machine. Within this set, policies can be specified for each mode of the machine. If there is a parent machine then the mode of the parent machine can also be used to select policies.

```
INTERLOCK:                    activate
     MODE:                    off
          PARENT_MODE:        disabled
          NEW_MODE:           SYSTEM_DISABLED
          END:
          PARENT_MODE:        enabled
          NEW_MODE:           on
          CONDITION:          check_power(); NO_POWER
          END:
     END:

     MODE:                    on
     NEW_MODE:                ALREADY_ON
     END:
END:
```

Fig. 18. Example interlock policy specification.

Fig. 18 shows an example interlock policy specification for the command `activate` for a simple two-state device. The device itself is part of a system that has modes of `enabled` and `disabled`. The actual interlock policies for a particular combination of command, mode, and parent mode are specified in two parts. The first part is the `NEW_MODE` which indicates whether there is a transition specified for this command from the particular state. This specification of `NEW_MODE` must appear for every combination of command, mode, and parent mode. If there is no transition then a failure response is indicated. In the example, for the mode `off` and parent mode `disabled`, the presence of the failure response name `SYSTEM_DISABLED` indicates that the command is not valid in this state. However, when the parent mode is `enabled`, then the new mode is `on`, which is a valid mode for the device. The second part of the interlock policies is a condition that must be satisfied before the command can be executed. In this case the condition is specified as the executable parameter `check_power` and if the condition is not satisfied then the failure response named `NO_POWER` is invoked. A condition is not required for each combination of command, mode, and parent mode.

Conditions are specified as executable parameters and can utilize any of the context information described above. The types of policies that can be specified as a condition were discussed in Section 4.3 and include mode-command restrictions, timing requirements, parameter checks, operational state conditions, and total duration restrictions.

Monitoring Policies

Monitoring policies specify checks that are to be applied to the operational state of a device or system. A monitoring policy may also be applied to an internal state of the machine abstraction as a way of monitoring the activity of the application software. Fig. 19 shows an example specification of monitoring policies for the same two-state device used

```
MONITOR:                    on
     SCHEDULE:              DEVICE_ON
     CONDITION:             check_on(); SHOULD_BE_ON
     END:
END:

MONITOR:                    off
     SCHEDULE:              DEVICE_OFF
     CONDITION:             check_off(); SHOULD_BE_OFF
     END:
END:
```

Fig. 19. Example monitoring policy data.

in Fig. 18. A monitoring policy is specified for each of the modes `on` and `off`. The state could have been further subdivided by also including the parent mode, but in this case the parent mode does not affect the monitoring policy, so it was not considered. For each mode a schedule name is given that refers to a defined schedule. One or more conditions can be provided for each mode. These conditions are composed of an executable parameter that performs the check on the operational state and the name of a failure response that is invoked if the check finds an error. The condition can be a check of device failure indicators, of the consistency between expected and observed states, or of the consistency of observed states.

Failure Response Policies

A failure response is provided for each of the failures that can be detected by the safety kernel. The failure responses are described by executable parameters. As with the other policies a particular set of failure response policies is associated with each machine. Failure response policies are invoked either from within the safety kernel or by the application software. When a failure response has been executed by a machine, if there is a parent machine, the failure response call is propagated upward.

Fig. 19 shows example data for three failure response policies. Each policy is identified by the name of the error condition to which it is responding. In this case one failure response is specified for each error condition name, but it is possible to select a failure response based on the mode and parent mode. The response actions are encapsulated in an executable parameter. As with other executable parameters used in the specification, the name of a failure response is provided in case an error is detected in the process of carrying out the failure response.

6.3.5 Built-in Control Mechanisms

To be of any use, all of the context and policy information described above must be incorporated into the operation of the safety kernel. The specification information is used by the following four mechanisms to control the operation of the safety kernel:

- Failure response
- Command execution
- Interlock policy enforcement
- Monitoring policy enforcement

These mechanisms are not static mechanisms into which the parametric information is placed directly. Rather, the mechanisms define a process for performing each of the major safety policy activities. The translator combines its knowledge of these mechanisms with the specification to produce software procedures that carry out the processes defined by the mechanisms. Each of the mechanisms is described below. The failure response mechanism is discussed first because all of the other mechanisms rely on it.

Failure Response Mechanism

An important feature of the safety kernel use of executable parameters is the uniform manner in which they are treated with respect to failures. Every executable parameter has associated with it an error condition name. In addition, every executable parameter is expected to return a status which indicates either that the parameter executed successfully and no policy violation was detected or that either an error occurred or a policy violation was detected. If the status indicates a failure, then the specified error condition name is included in a call to an error handling mechanism associated with the machine. The error handling mechanism is responsible for invoking the failure response specified for the particular error condition. If no error is detected then execution continues.

```
ERROR_CONDITION:      SHOULD_BE_ON
     RESPONSE:        should_be_on(); RESPONSE_FAILED
END:

ERROR_CONDITION:      SHOULD_BE_OFF
     RESPONSE:        should_be_off(); RESPONSE_FAILED
END:

ERROR_CONDITION:      RESPONSE_FAILED
     RESPONSE:        pull_the_plug(); PANIC
END:
```

Fig. 20. Example failure response policy data.

Failure response policies are invoked by a call to the error reporting facility of a machine. The error condition being reported is a parameter to the facility. This condition is used to select the specified failure response. Each failure response also has associated with it an error condition. Therefore, if an error is detected in carrying out the response, an error is reported with the error condition. This process can continue until the ultimate response available is called. The system will need to be designed such that this last response will succeed with high probability. When the failure response for a machine completes, then if there is a parent machine, the error reporting facility for the parent is called, with the original error condition. This propagation is performed because, although a lower-level machine may be able to respond to a failure, the ultimate response is likely to depend on the state of all the machines in the system. The parent machine handles the failure response just as has been described above.

Command Execution Mechanism

When one of the interface commands provided by a machine is called, a precise set of steps is followed that leads to either the execution of the command or a policy violation and resulting error report. The steps are as follows:

1. Time variables such as the total time in state are updated so that they may be used within safety policies.
2. The operational state of the machine is acquired so that it can be used by the operations enforcing policies. The specified error condition is reported if the operation fails.
3. The interlock policies are enforced for the particular command. Enforcement of this policy involves several steps that are described below. If any of the interlock policies fails, then control is returned to the application software with a status identifying the particular failure.
4. Once the interlock policies have been passed, then the command to perform the requested action is performed. Its return status is checked and once again the error condition is reported if an error is detected.
5. If the command is successfully executed, then the state information is updated. This information includes the present command mode and various time variables. The value of *operational state at last command* is also updated with the value of *present operational state*. A new monitoring schedule is also selected if the mode has changed as a result of successful execution of the command.
6. The final step is to update any control parameters declared for the machine.

Interlock Policy Mechanism

As indicated above, the interlock policies are enforced each time an interface command is called. The following steps are performed to enforce an interlock policy:

1. If there is a parent machine then its present mode is acquired.
2. Using the present mode of the machine and the present parent mode (if any), the specified polices are selected.

3. Check the specified value for `NEW_MODE`. If the value is a valid mode then continue with the conditions, otherwise report the error with the specified error condition and return control to the interface command execution mechanism described above.
4. Invoke the executable parameters that implement the policy conditions. If all complete successfully, then return to command execution, otherwise report the error with the specified error condition.

Monitor Policy Mechanism

Monitoring is performed periodically at times specified by a schedule. The schedule that is used for a machine at any given time depends on the present mode of the machine and the present mode of the parent machine if there is one. The monitoring schedule is set when the safety kernel is activated and is updated whenever the machine or its parent change modes. The following steps are performed when a monitoring policy is enforced:

1. If there is a parent machine then its present mode is acquired.
2. Time variables such as the total time in state are updated so that they may be used within safety policies.
3. The operational state of the machine is acquired so that it can be used by the operations enforcing policies.
4. Using the present mode of the machine and the present parent mode (if any), the specified polices are selected.
5. The monitoring conditions for the policy are applied. If no errors are detected then operation continues. Otherwise, the error is reported using the error condition associated with the policy condition that failed.

As mentioned previously, support for aperiodic monitoring activities will be provided in a future version of the translator. These activities will be scheduled relative to machine interface commands. The actual enforcement of the policy will be carried out as described above.

The periodic monitoring activities are scheduled statically by the safety kernel. These activities are assumed to have a higher priority than application requests. The result is that application requests will sometimes be delayed waiting for completion of monitoring activities. With knowledge of the maximum number and duration of periodic and aperiodic monitoring activities the worst-case delay can be computed. Areas for future work include permitting specification of scheduling priorities and incorporating a timer into the scheduler for detection of scheduling delays.

All of the information presented in Section 6.3 could be expressed much more precisely in a formal language. This is an area for future work.

6.4 Translator Implementation

A translator has been developed that processes the safety policy specification described above to produce a source code representation of the safety kernel. The specification data for each type of machine is located in a separate file. Each line in the specification is started with an identifier that indicates the type of the data. Thus, the specification has the appearance of a form that has been filled in with the specific context and policy data. The translator looks at the identifier at the beginning of each line to determine what actions will be required to parse the line.

The translator functions as a typical compiler with parsing and a code generation phases. In the parsing phase, the specification is read and stored in an internal representation. The internal representation consists of a class for each specification file, i.e., machine. Within each class there are other classes that hold the data corresponding to schedules, interface commands, modes, failure responses, and executable parameter definitions. The parser was constructed without the use of a tool such as yacc because of the prototypical nature of the translator and the relative simplicity of the input data. The actual parsing of file input is relatively straightforward. In the code generation phase, C++ header and source files that represent the safety kernel are produced. One C++ class, contained in one header and one source file, is produced for each machine. Software that is generated includes the class definition, procedures for executing commands, procedures that enforce the interlock policies, a procedure for the monitoring policy, and a procedure that is called to report errors.

The translation of a specification file is handled by the same translator regardless of where the machine falls in the system hierarchy. The output software is affected by the position in the hierarchy, but only in a few well-defined ways.

6.5 Conclusion

A reuse-oriented implementation strategy has been developed for the safety kernel because of the potential reliability and cost benefits. An evaluation of whether these benefits can be realized will require use of the translator with a number of applications. In a later chapter we begin this effort by evaluating the translator with the MSS and UVAR. In addition to the general reliability and cost benefits, the translator also provides the following benefits:

- The safety kernel framework provides a standard for expression of safety policies that facilitates specification reuse. Therefore, the knowledge used to identify and express policies for one application should be readily transferrable to others.
- The translator and its special-purpose programming can be reused and therefore the effort required to develop a reliable translator can be applied once and then amortized over a number of applications.
- As more is learned about policy enforcement in safety-critical systems, the translator can be refined to permit other systems to benefit from the advances in understanding and technology.

7 Verification and Analysis

One of the requirements for reliable enforcement of kernel-enforced safety policies is the correctness of the safety kernel implementation. Correct in this case means that the safety kernel is implemented to perform the actions specified for enforcement of the safety policies. The question that is addressed in this chapter is, “What are the issues in assuring that the executable version of the safety kernel actually performs as specified?”.

The process of showing that a software system complies with its requirements is known as software verification. As shown in Fig. 21, an executable representation of the safety kernel is developed from the system safety specification. As the arrow in the figure shows, verification is performed to demonstrate conformance between the specification and the executable version. Because the development process is typically complex, direct verification that the executable safety kernel meets the requirements of the safety specification is not feasible with many verification techniques. As a result, verification techniques are typically applied to the input and output documents of the various stages of development. An overall verification is achieved by demonstrating conformance between each of the individual stages.

What does it mean for two artifacts to “conform” or to demonstrate that a system “complies” with its requirements? A software specification documents a set of properties that must be true for its software implementation. Demonstrating conformance is the process of ensuring that the specification properties are also true of the implementation. The same can be said for any two corresponding artifacts. Demonstrating conformance does not mean that the implementation is “correct” in the informal sense of “doing what it is supposed to do” because the specification could be either erroneous (e.g., inconsistent) or incomplete (i.e., lacking essential properties).

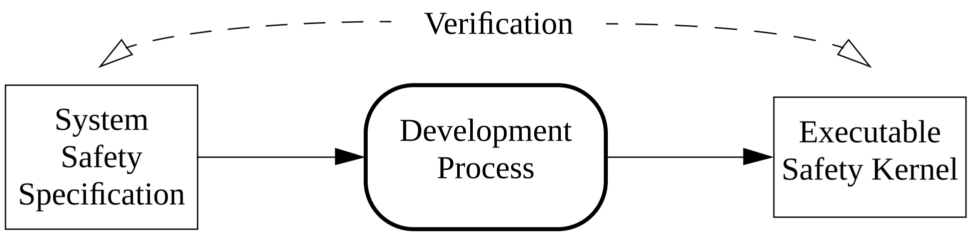

Fig. 21. Verification between specification and executable representation.

Many techniques can be employed to perform verification. Each has its strengths and all have limitations. Formal verification encompasses a collection of techniques that are used to develop a mathematical proof that two artifacts have equivalent properties. These techniques are useful for proving certain properties. However, they are not appropriate for all properties and often are not feasible for practical, complex systems. Verification techniques also include approaches that are not strictly formal. The most common example is testing. Testing is a technique that can perform direct verification between the safety specification and the executable program. Testing has many limitations, particularly for measuring system properties such as reliability [9,36,37]. However, testing is still a critical component of verification and is included in this analysis. Verification does not ensure that the safety kernel will operate as required by the system. In particular, if the specification is flawed, system operation can be erroneous. Techniques for analyzing the safety policy specification are being developed to demonstrate properties of the specification.

The rest of this section looks at high-level issues in the verification and analysis of the safety kernel. The three areas that are examined are formal verification, specification analysis, and testing. Later in the chapter we describe some verification and analysis techniques that are presently being investigated. A more thorough analysis and development of verification techniques for the safety kernel is an area for future work.

Formal Verification

We would like to be able to develop a mathematical proof that the executable safety kernel will have the properties specified in the software safety specification. In this section, the high-level issues in performing such a verification are examined.

The process developed as part of this research to produce an executable safety kernel from the system safety specification is shown in Fig. 22. As was discussed in Chapter 6, the system safety specification is used as the basis for development of the safety policy specification which is then processed by the safety kernel translator. The process proceeds from the policy specification through the translator and a C++ compiler to an executable version of the safety kernel. The arrows labeled "verification" between successive stages in the figure represent the individual verification steps that are required to perform the overall formal verification. Employing this verification process with the safety kernel, it would first be necessary to verify between the system safety specification and the safety policy specification, then between the policy specification and the C++ source code, and finally between the source code and the executable representation. In practice the process might be further decomposed, but this picture is adequate for a high-level discussion. The following items look at the three stages of verification depicted in Fig. 22.

1. *Safety specification* $\leftrightarrow$ *policy specification*
   Verification between the safety specification and the policy specification is a process that is being investigated. At this point in time, the definition of the policy specification is much farther along than the definition of the safety specification which is not yet suitable for any formal verification techniques. Basic concepts and guidelines are being developed for the safety specification and as appropriate formal notations are chosen, verification of properties between the two specifications will become feasible.

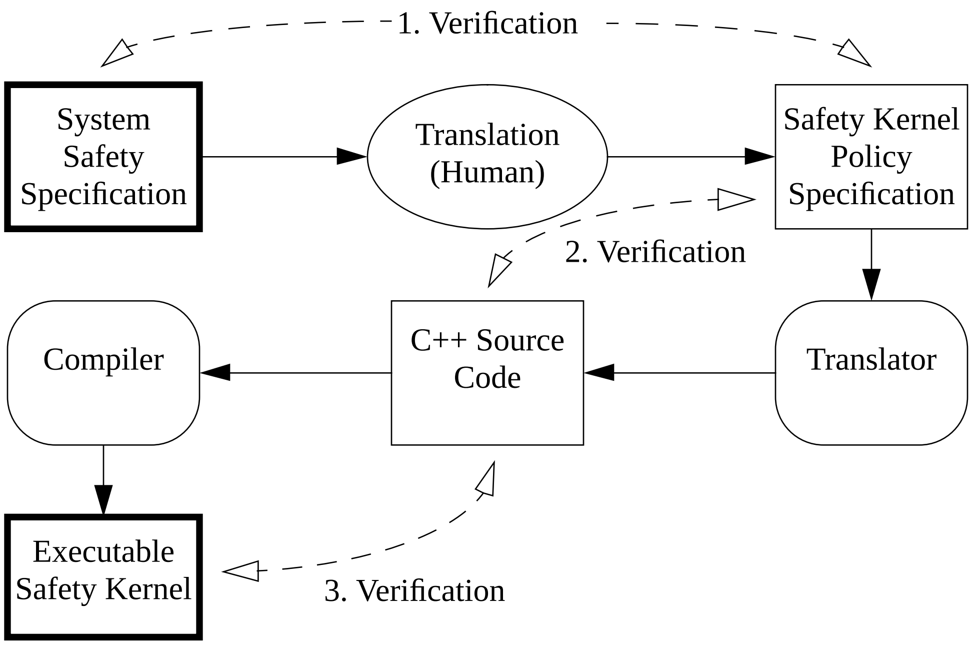

Fig. 22. Development of safety kernel executable from safety specification.

Verification between the safety specification and the policy specification could be obviated if an automatic process could be developed for deriving the policy specification directly from the safety specification. Presently, however, there is sufficient human insight required for this process (e.g., to develop weakened safety policies) that the feasibility of its automation is questionable. An area of future investigation is whether the system safety specification could be tailored to be compatible with the policy specification for purposes of facilitating partial or full automatic translation and verification.

2. *Policy specification* ↔ *source code*
Verifying that in general, the generated source code meets the policy specification requires verification of the translator. Verification of the translator is a problem very much like verification of a compiler. For now we will assume that with standard techniques and sufficient effort, the translator could be shown to implement the safety policy specification correctly. By successfully employing the translator with more than one application, this verification effort can be amortized. A thorough analysis of verification of the translator is an area for future work.

For the purposes of verifying the translator in general and a particular instance of policy specification and source code, an important consideration is that the translator and thus its output and the policy specification input can be configured to facilitate the process of verification. For example, the fact that the generated source code has a regular format, might permit it to be mechanically analyzed. Such an analysis could produce a set of source code descriptions that could then be compared with a similar set of descriptions generated for the safety policy specification. The result would be a reversal check for assessment of the effectiveness of the translator. This and other techniques for verification at this level will be investigated in ongoing work.

3. *Source code ↔ executable representation*
   A compiler is typically reused to generate executable programs for many applications. As a result, instead of focusing on verification between a specific source code input and its executable representation, emphasis is placed on demonstrating that in general the compiler will produce an executable that conforms to its source code. Demonstrating the general conformance between an executable and source code is the focus of research into compiler verification. Although an interesting area of investigation and important to the operation of an instance of the safety kernel, it is not critical to the evaluation of the concept. Therefore, we assume that standard techniques will be used to deal with this problem and that the compiler performs the translation correctly.

Specification Analysis

If the assumption is made that the translator and compiler work correctly, then the determining factor in whether the safety kernel operates "correctly" is the safety policy specification. The verification of this specification with respect to the system specification is described above. However, even with this verification completed, it is possible that the safety kernel would not operate as required by the system due to errors in the specification. Therefore, in addition to verification there is a concern with being able to detect errors by analyzing specifications. This analysis cannot validate that the specification reflects the system requirements, but it can demonstrate that the specification exhibits desirable properties such as internal completeness and consistency. A specification analysis technique and preliminary results are presented in Section 7.1.

Testing

Testing is a technique for direct verification between the specification and the executable safety kernel and, therefore, corresponds to the arrow on Fig. 21. As mentioned previously testing has definite limitations that must be understood. In particular, the only conclusion that can safely be made after testing has been performed is that the system functioned correctly on the inputs that were tested (note that even this conclusion depends on the potentially suspect assumption of perfect error detection [3]). In a practical system where the number of input combinations is huge, even extended testing will seldom test more than a small fraction of this number. Therefore, testing at the system level cannot be relied on as a verification technique in the sense of verification as a proof.

In spite of this limitation, testing of the safety kernel in the system context is an important part of verification for the purpose of developing "confidence" in a software system. With automated testing techniques and sufficient computing resources, it is possible, for example, to exercise as many combinations of inputs as would be encountered by all of the deployed systems for a particular application. Although this does not prove that there are no errors, it is imminently doable and provides a check on the other verification techniques. A system for performing this type of testing is described in Section 7.2.

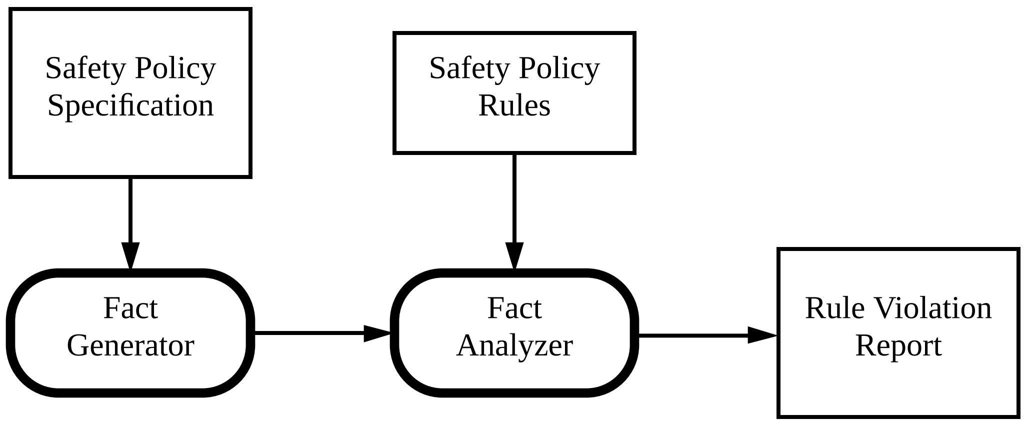

Fig. 23. Safety policy specification analysis tool.

Testing can also be employed in a manner that is equivalent to a proof [18]. Mathematical verification is used to demonstrate that certain properties, i.e., theorems, are true for two development artifacts. We are investigating the potential for demonstrating certain properties using testing. The technique which is described in Section 7.3 is used to demonstrate properties by exhaustively testing the required inputs. To facilitate this effort, a concept known as specification limitation is used to limit the input space.

7.1 Safety Policy Specification Analysis

The safety policy specification determines what safety policies are implemented by the safety kernel. Therefore, it is critical that this specification not only correspond to the safety specification, but that it also possess important internal properties such as completeness and consistency.

We have developed a prototype tool for static analysis of the policy specification. The concept for this tool is shown in Fig. 23. The policy specification is passed into a parser which generates a set of facts represented in Prolog that describe the system configuration and safety policies. The Prolog facts serve as the input to a Prolog interpreter and are evaluated with respect to a set of general rules that have been established for the safety policies.

Example interlock policy data first referred to in Chapter 6 is shown in Fig. 24. The facts that the generator would produce for this example specification are shown in Fig. 25. To analyze these facts, a set of rules has been developed which documents essential specification properties. The properties are expressed in Prolog and range from simple checks that modes have been previously declared to more sophisticated analyses of potential state transition errors. Examples of analyses that have been performed include the following:

- Every command used in an interlock policy must be declared as a command.
- Every command that is declared must have an interlock policy specified for it.

```
INTERLOCK:                  activate
     MODE:                  off
          PARENT_MODE:      disabled
          NEW_MODE:         SYSTEM_DISABLED
          END:
          PARENT_MODE:      enabled
          NEW_MODE:         on
          CONDITION:        check_power(); NO_POWER
          END:
     END:

     MODE:                  on
     NEW_MODE:              ALREADY_ON
     END:
END:
```

Fig. 24. Example interlock policy data.

- Either no parent modes or all parent modes must be included as a part of an interlock policy for a specific mode.
- A `NEW_MODE` transition must be specified for each combination of mode and parent mode.
- The value specified for `NEW_MODE` must be either a declared mode or error condition name.

```
mode(‘on’).
mode(‘off’).
parent_mode(‘enabled’).
parent_mode(‘disabled’).
valid_state(‘on’, ‘enabled’).
interlock(‘activate’).
transition(‘activate’, ‘off’, ‘disabled’,
     ‘SYSTEM_DISABLED’).
transition(‘activate’, ‘off’, ‘enabled’, ‘on’).
transition(‘activate’, ‘on’, ‘no_modes’, ‘ALREADY_ON’).
condition(‘activate’, ‘off’, ‘enabled’, ‘check_power’).
parent_transition(‘enabled’, ‘disabled’).
```

Fig. 25. Facts for example interlock policy data.

- Every interlock condition must have an error condition name associated with it.
- A transition by the parent of a machine must not result in a state that would otherwise be precluded by the interlock policies of the machine.

Additional properties of this type have been identified for the monitoring and failure response policies. If the executable parameters were accompanied by predicate logic descriptions, it would also be possible to develop properties related to the interlock and monitoring conditions.

7.2 Automated System Testing

What role should system testing play in the overall software verification effort? With an efficient testing arrangement, a large number of inputs can be tested even if they do not represent a significant fraction of the input space. If, for example, the number of test inputs could approach the total number of inputs that would be seen during the lifetime of all of the instances of an application system, this could promote confidence in the software system. Admittedly, "confidence" is an informal notion, but until feasible formal verification techniques are available, this type of confidence will be important in the verification of any complex system. We view testing of this type to be a "defense-in-depth" strategy that should be used in combination with other more formal techniques (e.g., testing for verification of properties described in Section 7.3).

The confidence gained from testing a system is tied to the fraction of the input space that has been tested. This problem must be dealt with by both reducing the input space and increasing the number of tests cases that are executed. We are investigating reduction of the size of the input space using a concept known as *specification limitation* that is described in Section 7.3. To increase the number of test cases executed, an automated test harness is being developed that is described below.

Testing has been conducted using the test harness depicted in Fig. 26. This test harness permits automated testing of the safety kernel operating with control systems such as the UVAR and the MSS [24]. In this system, the safety kernel is executing along with the application software which in this case consists of an operator display and a control program. The testing of the application in this system is supervised by the test driver which is responsible for maintaining a model of the application world, for generating operator requests, for gathering information essential to error detection, and for performing the error detection operations.

The operator display receives commands from the test harness that are identical to those that would be entered by a human operator. A relatively small addition, known as a *pseudo user* has been made to the operator display that accepts these commands. As occurs outside of the test harness, the operator display processes the operator commands making requests to the control program as necessary. The control program in turn acts on the requests and communicates with the safety kernel to effect device actions. In the test system, communication between the control program and the safety kernel is mediated by the test harness command modifier. In this position, the command modifier is able to keep a record of all commands that have been issued to the safety kernel. Furthermore, it is able to delete, substitute for, or modify the parameters of requests on their way from the control program to the safety kernel. This enables the test harness to simulate failures of the application software. This technique provides much better control of application failures than could be achieved with other techniques such as software fault injection.

The safety kernel controls a set of devices that are simulated by the test harness. The devices utilize the same interface as the actual application devices. The state of a device is determined by the simulator and is used to provide feedback to the safety kernel device monitors. The test harness uses the device state to detect errors in safety kernel operation and to determine the effect that a device has on the other elements of a system. The device simulators also provide a set of commands to the test harness that can be used to induce various types of device failures.

The operation of systems such as the MSS and the UVAR are tied to the real world and therefore to real time. In the test harness, this can greatly limit the number of test cases that can be executed. In some cases, it is feasible to separate the system from the real world time reference and instead rely on virtual time as provided by a source such as the test harness.

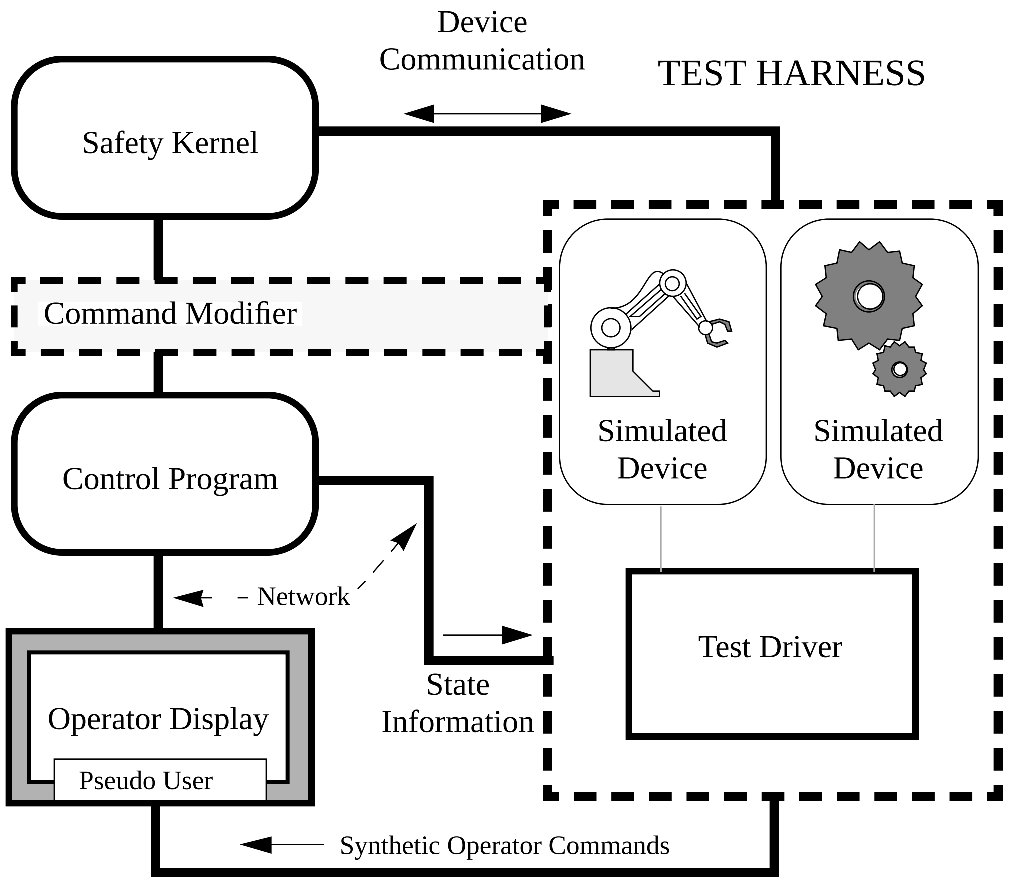

Fig. 26. Test harness for system testing.

In making this change, the concern is that the operation of the software and other components is not fundamentally altered. One way to increase the rate of operation safely is to adjust the passage of time so that the idle time between operations is reduced, but so that the computing operations and device operations are otherwise unaffected.

For example, from the perspective of the computer, the MSS operates at a very slow pace. The current controllers change current at a rate of a few amperes per second and the current requested of a current controller is updated no more than four times a second by the control program. At this rate, the safety kernel is not likely to receive more than 30 requests per second for device actions. The other safety kernel activity is monitoring of devices, and measurements that are described in the next chapter indicate that the monitoring activities require only a small fraction of the safety kernel operating time. Therefore, speeding up the operation of the MSS by a factor of three to five would be reasonable. Factoring in the elimination of the planning required by a human operator and continuous operation of the test system, a number of inputs equivalent to those generated by the physical system operating for one year could be produced by the test harness in approximately one week.

Another alternative for speeding up the testing is to accelerate the occurrence of failures. In a physical system, the occurrence of device failures is a relatively rare event. By inducing device failures in the simulated devices, it should be possible in a short period of time to generate failures that would require significantly more time (if ever) to be manifested in the real system. Since preventing and responding to failures is the concern of the safety kernel this technique permits extensive testing that would be difficult, expensive, or infeasible in the physical system.

Preliminary results of testing of the MSS safety kernel prototype are described in Section 8.2.3 in regard to the feasibility evaluation of the prototype. Extensive testing with the test harness is an area for future work.

7.3 Testing for Verification of Properties

If the size of the input space can be reduced to a point where exhaustive testing can be employed, then testing would be effective for proving selected properties. We are dealing with the input space problem in two ways:

- *Property identification*
  The first step in utilizing testing for verification of properties is careful identification of desired properties. The goal is to identify useful, albeit possibly narrow properties, that have input spaces that can be exercised exhaustively. For example, it would be impossible to demonstrate a property such as system reliability which would require exhaustive testing all of the functionality of the safety kernel. On the other hand, testing for a property such as correct operation of an executable parameter might be feasible.

- *Specification limitation*
  The concept of *specification limitation* or input limitation is that some software inputs can be restricted in a manner that does not impact the operation of the system, but greatly reduces the total number of possibilities for a particular input. For exam-

ple, in a system like the MSS, it would be acceptable to round the sensor input from a superconducting coil to the nearest ampere. This would not interfere with policy enforcement, but depending on the number of bits of sensor input could significantly reduce the number of inputs to be considered. A greater impact would be seen by restricting a value such as the requested current which is represented as a floating point value. This would reduce the number of inputs from the number of distinct computer floating point values between the minimum and maximum current to the number of integral values in this range.

We have used testing to verify the two properties shown below. These properties certainly do not imply safety kernel correctness, but they contribute to the overall verification and are properties that would be difficult to establish with other techniques. For example, it is likely that random, system testing would not test these properties exhaustively. Formal verification between the policy specification and the executable safety kernel would be non-trivial.

- For any combination of machine mode and parent machine mode, a command will not be executed if an interlock policy does not specify a transition to a valid mode.
- For any combination of machine mode and parent machine mode, a command will be executed if an interlock policy specifies a transition to a valid mode. This assumes that any conditions specified with executable parameters are met.

Using the specification-based test system shown in Fig. 27, both of these properties have been tested exhaustively. For the MSS safety kernel, a system machine and an X-ray source machine were tested in this system. For each combination of modes for these two machines, each of the commands for the two machines were invoked. Each command was determined to be valid or invalid for that mode combination based on analysis of the specification. Invalid commands were expected to return a specific error code and valid commands were expected to complete successfully.

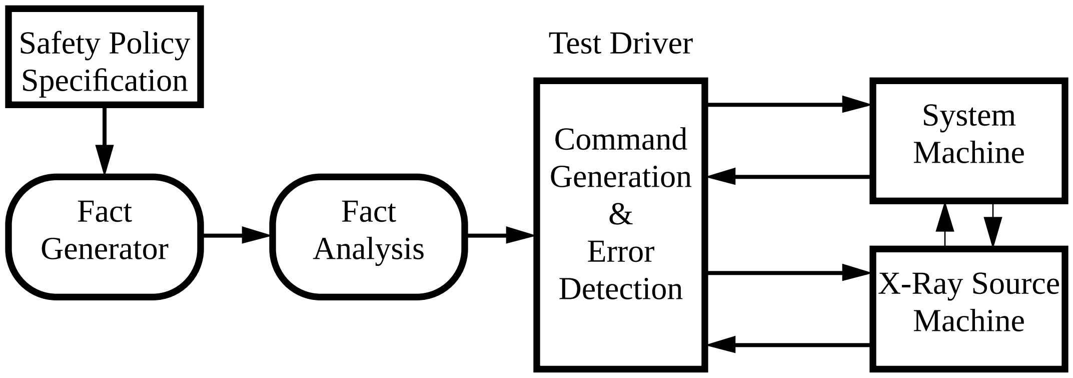

Fig. 27. Testing for verification of properties.

8 Safety Kernel Prototype

In this chapter the prototypes that have been developed for the two case studies are examined. An operational safety kernel prototype has been implemented with the MSS using the implementation strategy described in the previous chapter. With the UVAR the prototype is of the safety policy specification for the reactor safety kernel. Development of the MSS safety kernel prototype has facilitated additional evaluation and refinement in all areas, but particularly with the implementation strategy. The UVAR prototype specification was developed to evaluate the applicability of the implementation strategy to an additional safety-critical application.

The following section examines the structure of the MSS safety kernel prototype including the policy specification, the system design, and the integration of the safety kernel with the MSS application software. Section 8.2 discusses the evaluation of the safety kernel in the areas of the impact on the application software, performance, and operational feasibility. The last section describes the UVAR safety kernel prototype.

8.1 The MSS Safety Kernel Prototype

8.1.1 Safety Policy specification

An overview of the MSS safety kernel prototype as it is described by the policy specification for the MSS safety kernel is presented below. The actual safety policy specification appears in Appendix C.

The MSS safety kernel prototype has been decomposed into three types of machines as shown in Fig. 28. At the top level is the system machine that serves to group the other machines in the safety kernel. The system has its own set of commands, modes, and policies that are described in this section. Below the system machine are machines which correspond to the MSS x-ray sources and the servoamplifiers. One type of machine is specified for each type of device. The machine for an x-ray source is instantiated twice in the safety kernel, once for each of the X and Y imaging axes. Similarly, the servoamplifier machine is instantiated six times in the safety kernel, once for each of the six servoamplifiers that control the current in one of the helmet coils. Both of these types of machines are also described below.

In structuring the machines for the safety kernel, an additional layer for the group of servoamplifiers and one for the two x-ray sources could have been added. However, the interaction between devices in these groups is not that significant and the system is small enough for the simpler grouping. For a larger system, further decomposition might be very useful if not essential.

System Machine

The system machine is configured with six modes of operation. These modes are used to establish higher-level operating goals for the system and to restrict operation of the devices. In addition, the modes are used to specify policies for monitoring the system. The modes for the system are described below along with a description of how each mode affects system monitoring and the operation of the system devices.

- *Inactive*
  In the inactive mode, the only actions that are permitted for the devices are the emergency shutdown operations. Otherwise, other activities such as turning the X ray sources on or controlling the current in a servoamplifier are precluded. No monitoring takes place at the system level in this mode.

- *Vision enabled*
  This mode permits operation of the x-ray sources, but not of the servoamplifiers. No system-level monitoring takes place in this mode.

- *Currents idling*
  In this mode both the x-ray sources and the servoamplifiers can be operational. However, the currents in the coils are expected to produce a force that is below the threshold for movement, so the position of the seed is not monitored in this mode. Checks of the magnetic force on entry to this mode and continual monitoring of the coils ensures that the force will not change. This assumes, of course, that the seed does not move in the magnetic field due to movement of the patient's head. The position of the patient's head will be fixed, so this is a reasonable assumption.

- *Moving seed*
  In this mode it is expected that both x-ray sources and the servoamplifiers will be operating. Furthermore, it is assumed that the current produced in the coils by the servoamplifiers will result in a force above the threshold of movement for the seed.

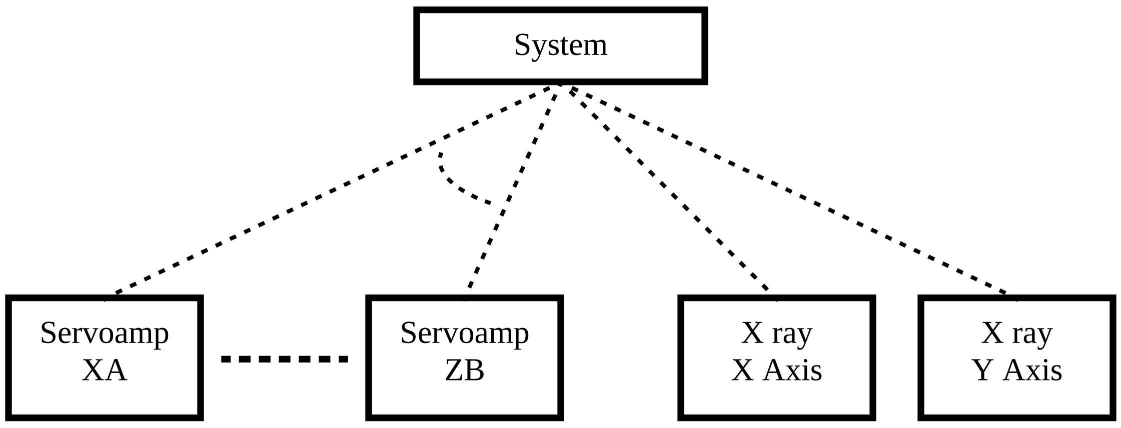

Fig. 28. Organization of the MSS safety kernel prototype.

Therefore, in this mode the position of the seed must be periodically reported to the safety kernel by the application software. In order to transition from this mode to a mode such as currents idling, a check must be performed which shows that the magnetic force is sufficiently below the threshold for movement.

- *Vision calibration*
  In the vision calibration mode, the x-ray sources can be active. No system-level monitoring is performed and transitions from this mode to any except inactive are precluded.

- *Shutdown*
  This is the mode that is entered when a failure that prevents continued safe operation is detected. The only interface commands that can operate in this mode are those that are used to respond to failures.

Changes in the modes are effected via the command interface to the system machine. These commands are relatively simple and therefore are not discussed here. One command that is of interest though is the command `register_seed_position`. This command is used by the application software to report the seed position that has been computed by the vision system. During the mode `moving seed`, this position must be reported at least once every 0.5 s.

At the system level, failure responses are specified for error conditions originating both within the system machine and from the device machines. The response that is performed is a function of the error condition and the mode of the system.

Servoamplifier Machine

A servoamplifier machine enforces policies concerning the operation of the MSS servoamplifier current controllers. A servoamplifier machine has the following four modes of operation:

- Inhibited
  In this mode the only commands that can be executed are those that read sensor values and the failure response commands. It is expected that the coil currents are zero and the servoamplifiers are monitored to ensure that this is the case.

- Operating
  In this mode, the physical servoamplifiers are operational and target current requests can be sent to the servoamplifiers. The currents are monitored to verify that they are as expected.

- Failure adjust
  This is a failure response mode in which policies designed to prevent excessive current change rates are not enforced. Rapid changes in current can cause a superconducting coil to switch to a resistive conducting mode, i.e., quench. An example of a policy of this type is one that restricts the maximum change between the present current and the new target current. This mode would be used when it is critical to bring the coils to a current (likely 0) without the normal restrictions on how rapidly

that change can be effected. The commands used by the application to control a servoamplifier are not executed in this mode.

- Shutdown
  When in this mode, the servoamplifier machine does not permit any operations to be performed other than the reset command. It is expected that the servoamplifiers are inhibited and therefore the currents are monitored to ensure that they are zero.

The command interface to a servoamplifier machine includes methods for setting the current, reading the current sensor, and for responding to failures detected by the safety kernel. Failure responses range in severity from error return values and warning messages to setting the servoamplifier requested current to zero, which has the potential for quenching the coils but is also the quickest way to reduce the current in the coils.

X-ray Source Machine

The interface to an MSS x-ray source is provided by a safety kernel x-ray source machine. This machine has the following four modes:

- Voltage off, current off
  In this mode the voltage and current for the source can be set. The source is expected to be off and monitoring is performed to ensure this.

- Voltage on, current off
  This is the intermediate mode between the one above and the next mode in which the source is active. In this mode the voltage may not be adjusted, but the current may be. Monitoring is performed to verify that the source is off.

- Voltage on, current on
  In this mode the source is expected to be producing X rays. Monitoring checks not only that the source is on as expected, but that the single and accumulated dose are below prescribed limits. In this case the dose is related to time and so the checks performed check the time that the source has been on.

- Disabled
  This mode is transitioned to when a failure has occurred and the source should not be operated by the application software. The source is expected to be off in this state.

The command interface for the x-ray source machine includes commands to get and set both the current and the voltage and also to turn both the current and the voltage off and on. Additional commands are provided to perform an emergency disable and to reset the machine after it has been disabled. The most severe failure response options for the x-ray sources are to turn them off using either the normal control interface or by using an alternate device which disconnects power to the x-ray sources.

8.1.2 System Design

Chapter 5 documented the requirements for reliable enforcement of safety policies and outlined a system design for applications such as the MSS and UVAR. This design has been used as the basis for the system design used with the MSS safety kernel prototype. However, some of the features are not included because either they are well understood and incorporating them would not address critical questions or because the techniques represent areas of research in their own right. The development of a prototype system and safety kernel that addresses all of the reliability requirements is beyond the scope of this research. The prototype system design is discussed below followed by a discussion of the features that have not been included in the prototype.

The system architecture of the prototype of the MSS is shown in Fig. 29. The design includes the following features originally listed in Section 5.6:

- Safety kernel as a user-level process.
- Application software and safety kernel communication via network.
- Closed-loop, device control using independent sensors.
- Safety kernel watchdog.

The following features are not incorporated in the system design:

- *Command authentication and error detection encoding for device communication*
  These techniques have been widely used and therefore we will make a preliminary assumption that their application to this situation would be relatively straightforward.
- *Restriction of application software resource usage*
  This is another area where methods exist that could be employed.
- *A core of dependable support services and basic computing services*
  The prototype does not address the dependability of the computing services. We

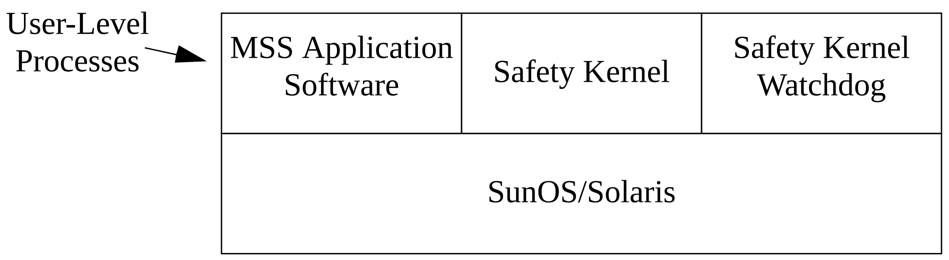

Fig. 29. MSS safety kernel prototype system architecture.

will assume that traditional verification techniques could be employed. Comprehensive treatment of this problem is an area for future research.

- *Microkernel architecture for system/support software*
  This feature supports the verification of support services and is not necessary given that the support services are not being verified.
- *Incorporation of redundant information into configuration data*
  The safety kernel does not read any configuration data at run-time.
- *Error detection analysis of safety kernel executable following loading into memory*
  We will assume that existing techniques could be applied to perform this analysis.
- *Support for real-time operation of the safety kernel*
  This issue could be addressed by selecting a computing platform with support for real-time operation. At the system-design level this is not a critical research issue. We have attempted to mitigate concerns in this area by utilizing a processor that has more than enough resources for timely safety kernel operation.

The application software, safety kernel prototype, and safety kernel watchdog are all user-level processes compiled for SunOS 4.1.x. This choice is motivated by system availability and the existence of software libraries useful for prototyping rather than by any particular characteristics of SunOS. The processes communicate via the network using a library implemented on top of TCP/IP. The safety kernel watchdog is simulated by a software process that produces an error message anytime that its timer expires. This process will not actually have access to the devices, since it is assumed that a hardware watchdog would provide the necessary functionality and actually implementing this would not be particularly instructive.

8.1.3 Integration of Safety Kernel with MSS Application Software

The system design for the safety kernel requires the MSS control program and the safety kernel to communicate via interprocess communication. The network communication in this case is implemented in a library that is built on top of TCP/IP. Using this library, each of the three machines in the MSS safety kernel have been coupled with a local base class and an external interface class that handle the packing, unpacking, and communication of messages. The interface with the machines uses remote procedure call semantics, so, except for the latency, the operation is just as if a local procedure call were being made.

The safety kernel has been integrated with the MSS control program as shown in Fig. 30. The figure depicts the uses relationship between the objects of the control program and the safety kernel. The external network interface for the safety kernel servoamplifier class is integrated as a base class for the coil class in the control program. The system machine of the safety kernel is accessed by both the seed object and the vision system. The x-ray source machines are called by the vision system.

8.2 Evaluation of MSS Safety Kernel Prototype

The evaluation of the MSS safety kernel prototype focuses on three areas: impact on the application software, performance, and operational feasibility. Although targeted at a particular instance of the safety kernel each of these areas has implications for the safety kernel and system designs that have been developed.

8.2.1 Impact on the application software

An anticipated benefit of the use of the safety kernel is the simplification of the application software. The most obvious simplification is that by performing the device and system monitoring, the safety kernel completely relieves the application software of this responsibility. This is particularly significant because the monitoring must occur on a scheduled basis implying that the monitoring must be carried out, at least, logically in parallel with other activities. It is likely therefore that some sort of separate, independent monitoring entity would be required if the safety kernel were not employed.

Since the safety kernel is partially being retrofitted to the MSS application software, some of the safety policy enforcement is already being accomplished in the application software. Examples include limit checks on parameters and conditions on devices states. Therefore, the full benefits of simplifying the application software have not been realized.

The application software for the MSS is reasonably large and complex. A rough metric is the number of lines of source code.

- Control program: ~20,000 lines

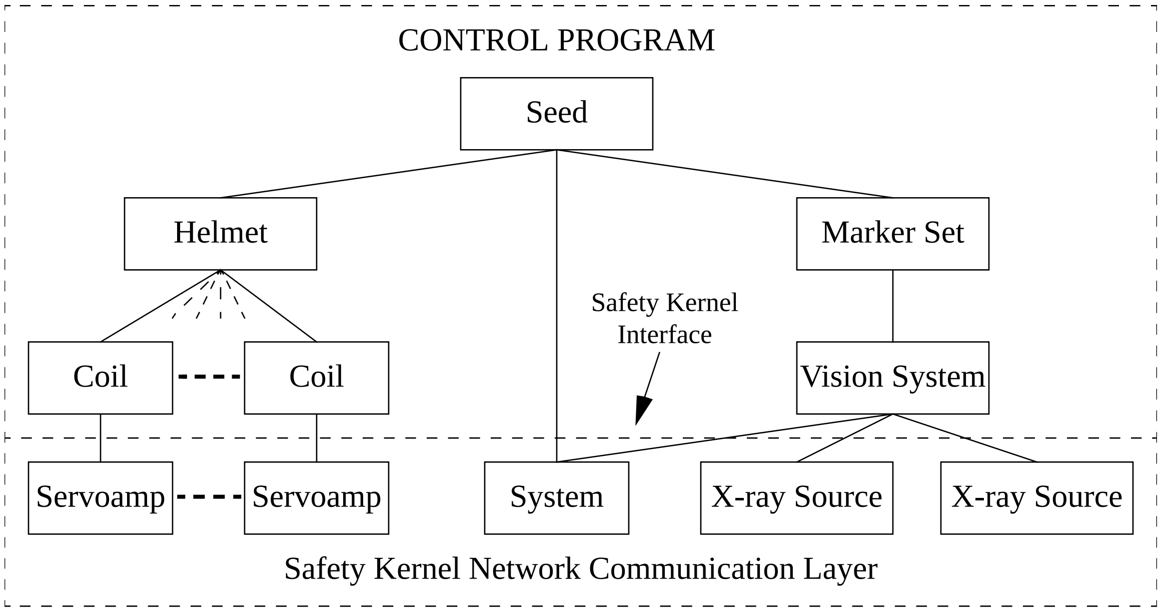

Fig. 30. Integration of the safety kernel with the MSS control program.

- Operator display: ~7,000 lines
- Libraries (not including X, system, etc.): ~40,000 lines

As more functionality is added, the quantity and complexity of the software will continue to increase. The basic safety policies for the system will not change significantly and therefore, the safety kernel should maintain its present complexity. In addition, if the reasonable assumption is made that the system safety requirements are well established, the safety kernel should be much more static than the application software. The size of the safety kernel is presently 6500 lines. The generated code of the safety kernel is quite simple and regular with six types of procedures accounting for approximately 80% of the lines of source code. Possibly a better measure of the safety kernel complexity is the length of the safety policy specification which is less than 3000 lines.

If the application software were responsible for enforcement of all of the safety policies, its complexity and fluid nature would necessitate significant verification effort. The use of a safety kernel obviates the need for verification of a critical set of safety policies for the application software. Since maintenance (continued development) will persist throughout the life of an application, the significance of being able to enforce safety policies with a relatively static component should not be underestimated.

8.2.2 Performance

An issue in evaluating the feasibility of the safety kernel concept is the performance overhead associated with its use. With our implementation of the prototype there are three main sources of overhead. The first is the network communication required for the application software to communicate with the safety kernel. For some applications, this source of overhead may lengthen the response time enough that this design would not be appropriate. In this situation, an alternative design, such as incorporating the safety kernel into the system kernel would need to be considered. The latency added by network communication will be measured.

The second source of overhead is associated with the enforcement of policies when a request is made to the safety kernel by the application software. This overhead is not expected to be significant. In addition, some of the policies that a safety kernel will enforce would otherwise be enforced by the application software. Therefore, assuming that the policies must be enforced by either the application software or by the safety kernel, their enforcement should have little additional impact on the overall performance of a software system employing a safety kernel.

The third source of overhead comes from the monitoring of devices. However, as with the policies above, since the monitoring must be performed either by the application software or by the safety kernel, the overall system performance will be unchanged by giving this responsibility to the safety kernel. In fact, since this monitoring can occur in parallel with the operation of the application software, the overall system performance could even improve. One concern with the monitoring, is that it not significantly impact the ability of the safety kernel to respond to application software requests in a timely manner. The portion of time available for satisfying requests will be measured.

As indicated above, most of the policy enforcement activities would need to be performed by the application software if they were not the responsibility of the safety kernel. Therefore, if overhead is defined as extra cost incurred in performing essential services, there is little overhead with the safety kernel prototype except for that involved in network communication. Given that lower overhead communication options exist, this is not a fundamental problem. Therefore, although the evaluation of performance is important to understanding the characteristics of this particular kernel implementation, it is essentially irrelevant to any discussion of performance pertaining to the general safety kernel concept.

Regardless of any of the above discussion, the real issue is: Can the safety kernel perform all of its required actions in a manner that meets the requirements of the MSS? A direct measurement has been made to address this question. The measurement is of the number of application software requests that the safety kernel can fulfill while performing its required monitoring activities.

Methods and Results

Measurement of the network communication overhead has been performed using the control program to generate a large number of requests to the safety kernel. The safety kernel was configured to immediately respond to each request that it received meaning that it did not enforce any safety policies. This was necessary to remove the effect of policy enforcement overhead. The monitoring policies were not enforced for the same reason. The wall-clock time required to satisfy a set of requests was measured.

The performance measurements were performed with two different physical configurations of the two programs. The first configuration had both programs executing on the same computer, a SPARCStation 20. Except for background system tasks, there were no other active processes. Measurements in this configuration consistently resulted in excess of 500 empty requests per second (less than 2 ms per request) being processed by the safety kernel.

In the second configuration, the two programs were executing on different machines but on the same subnet. Once again, an effort was made to minimize all other activity. The control program was executed on a Sun SparcServer 10/514 while the kernel was executed on a SPARCStation 20. In this case the performance was slightly better with an excess of 550 requests per second being carried out. Other configurations were not evaluated, since a safety-critical application would typically be executed on a dedicated network to reduce the interference from other network traffic. This is definitely the case with the two case studies.

The overhead associated with the enforcement of monitoring policies was measured by operating the safety kernel with just monitoring active, i.e., no requests from the application software. Under these conditions, the amount of time spent idle was measured by determining the wall-clock time that was spent in the network select call which is entered only when no monitoring activities are scheduled to be invoked. The total time of observation was also measured. The difference between these values is the amount of time that was spent in monitoring activities, scheduling, and other supporting activities. During the measurement, the six servoamplifiers were monitored four times per second and the two x-ray sources were monitored 20 times per second. In each case monitoring required acquiring the present operational state from a sensor and then comparing this state to the expected state. Running the safety kernel on a SPARCStation 20, the idle time on repeated observations conducted for at least 60 s exceeded 99% of the observation time.

The measurements described above are addressing the question of how well the safety kernel can respond to application software requests. Although the above measurements help to answer questions of safety kernel performance the actual request throughput can be measured directly. To measure this value, 10,000 sets of requests to set the current to a particular value and then to read the current sensor were sent to the safety kernel by the control program. Once again, both programs were executing on a SPARCStation 20. With monitoring as described above, approximately 175 of these request pairs or 350 individual requests were processed per second by the safety kernel. The throughput was roughly the same with monitoring inactive which is not surprising given the minimal resources required for monitoring operations. During the measurement, the safety kernel utilized approximately 55% of the processor time with the balance going to the program generating the requests.

The MSS is expected to require no more than 50 requests per second to be fulfilled by the safety kernel. Therefore, based on the measurements above, the safety kernel as implemented should require less than 10% of the processor time and should very comfortably meet the performance requirements of the MSS. With the UVAR, the requests are likely to be even fewer, however, the monitoring activities will be significantly increased. Given the efficiency with which the monitoring activities are performed, there is good reason to believe that the safety kernel design and system architecture will also meet the performance requirements of the UVAR control system.

8.2.3 Operational Feasibility

The prototype MSS safety kernel has been operated in a test harness like the one described in the previous chapter. The customization of the test harness for the MSS is described later in this section. The goal of exercising the safety kernel in this context is to evaluate basic feasibility questions that need to be addressed for this research. The questions most relevant to this research are of the following type:

- Does the safety kernel provide the functionality required for control of the devices?
- What are the issues with operating the safety kernel in the system context?
- Do the monitoring policies detect and respond to device failures?

Answering questions of this type requires incorporating the safety kernel into the system as described above, but the goal is concept and design evaluation rather than dependability assessment. Evaluation will eventually need to be performed in the context of real system hardware, but it is not presently accessible to us. With its simulated devices, the test harness actually provides greater flexibility than the real system particularly with regard to inducing device failures.

Test Harness

Testing of the safety kernel is conducted using a test harness based on the one described in Chapter 7. The version of the test harness that has been developed for the MSS [24] is shown in Fig. 31. In this system, the devices for the MSS are six servoamplifier current controllers and two fluoroscopic imaging systems. Each of the two imaging systems are simulated as two separate components: an x-ray source and an image generator. The image generator that has been developed produces synthetic fluoroscopic images using knowledge of the physical arrangement of the imaging system components, the markers affixed to the patient's skull, and the seed within the patient's head. These digital images are used by the vision system of the control program to track the seed.

The test driver model of the MSS world consists of the application devices, the markers on the patient's head, the seed inside the head, and three sets of MR images. All of these entities have a position in a world coordinate system and some of them are free to move within this coordinate system. The test harness is responsible for controlling this movement, keeping track of the positions, and transforming between the coordinate systems.

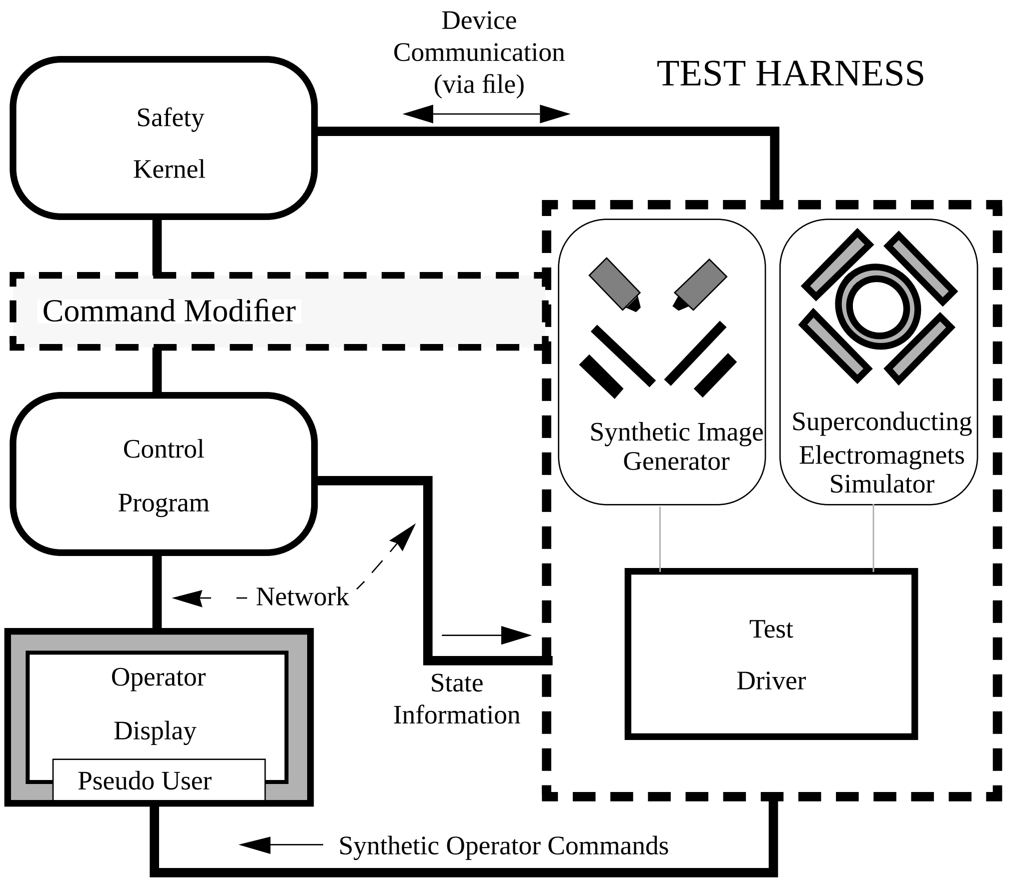

Fig. 31. MSS software system with test harness.

Preliminary Results

The test harness has been configured to supervise the execution of a simple MSS surgical procedure. In this procedure, the seed is started at the middle of the head and moved incrementally toward the right side of the brain. The servoamplifiers are adjusted to simulate the currents required to produce seed movement and the vision system tracks the seed periodically during a movement. In the testing arrangement, when one procedure is completed, the seed is repositioned at the center of the head and the procedure is repeated.

Testing with this system is in progress with several different types of tests. The first test is designed to exercise the safety kernel without any injected failures. The surgical procedures are performed and the safety kernel is monitored to verify that it is not enforcing safety policies when operation is in fact correct.

In the second set of tests the surgical procedures is repeated, but faults are injected at random using the command modifier. The nature of a fault is a substitution of an invalid command for the originally requested command. An invalid command is one that is precluded by the interlock policies for the present mode of the machine. It is expected that invalid commands will return an error status. The test harness will monitor the return values to detect erroneous operation.

The third type of test will execute the same surgical procedure except there will be device failures induced by the test harness. The safety kernel is expected to detect and respond to these failures within a specified period of time. Error detection will be based on whether the device failure is successfully detected.

8.3 UVAR Safety Policy Specification

The UVAR case study has been used throughout the development and evaluation of the safety kernel concept. It has provided another data point for understanding the problems of software dependability in safety-critical systems and an additional context for evaluation of research products. Eventually, the safety kernel will be developed for the reactor just as it has been for the MSS. This work is not included in this dissertation. However, to provide some additional information on the general applicability of the safety kernel translator, a preliminary safety policy specification has been developed for a safety kernel prototype for the UVAR. The specification is preliminary because the specification for the control software for this system is just being developed. As a result, not all of the safety policies have been identified. The data is also preliminary in that only enough of the specification (see Appendix C) was completed to demonstrate the feasibility of the safety kernel for the UVAR.

This section contains a high-level description of the machines of the prototype. The safety kernel for the UVAR utilizes the hierarchy of machines shown in Fig. 32. These machines correspond to the components in the system that can be manipulated to control the operation of the system. Safety rod machines 1 through 3 provide the interface to the rods that are used to moderate the nuclear chain reaction. These are the rods that are dropped into the reactor core to effect a scram of the reactor. The regulator rod machine controls the rod that is adjusted to maintain constant reactor power once the reactor has

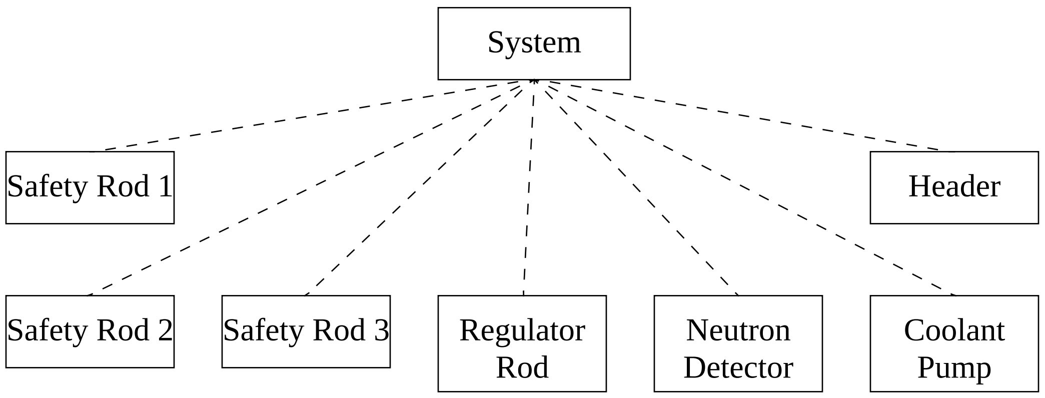

Fig. 32. UVAR safety kernel machine hierarchy.

been brought up to operating power. This rod can be adjusted manually or automatically using a simple feedback system. The header and coolant pump are components in the cooling system. The header is placed up against the bottom of the reactor core and serves as the inlet into the cooling system for water that has just passed through the core. The reactor can be operated in a 0.2 MW convective cooling mode in which case the header must be positioned away from the core or in the 2.0 MW mode in which case the header must be up against the reactor core to permit active cooling. Each of the safety kernel machines is examined later in this section.

The reactor system includes a multitude of sensors that provide data for monitoring the reactor operation and various safety conditions. With one exception, these sensors are not modeled as machines because they are not controlled by the reactor operator. Rather they provide input to the machines for the purpose of monitoring the system and individual devices. The exception is the neutron detector for the source range that can be moved to position it closer to or farther from the core. This detector is used for neutron detection at relatively low power. Moving the detector away from the core when the reactor is operating at high power slows the degradation of the detector.

Other than warnings to the operator, a scram is the sole failure response when failures are detected. It is the only one necessary because there is a very high probability that this will be an adequate response to the failure. It is invoked without intermediate failure responses because, other than causing some inconvenience, a scram does not harm the system and is the conservative choice for ensuring system safety. The safety kernel watchdog for the reactor will consist of a timer and the functionality required to effect a scram. Shutdown requirements for a power reactor will differ in that intermediate failure responses would be desirable to attempt to sustain power generation. Scramming a power reactor is also a much longer and more complicated process than it is with the UVAR. Later research on this project will address the problems presented by these requirements.

The primary role of the system machine is monitoring of the operational state of the reactor. It is the system machine that also determines when a scram is required as a failure response. The modes of the system are the following:

- *Inactive*
  In the inactive mode, the reactor is in a scram condition. The safety kernel monitors to ensure that this is the case. In this state, none of the device commands can be executed.

- *Setup*
  This mode is used for configuration of the reactor prior to start-up. Typically, configuration involves selecting either the 0.2 MW or the 2.0 MW power range for operation and then setting the devices accordingly.

- *Start-up low*
  When the reactor is to be operated at 0.2 MW of power, this mode is used for bringing the system up to this power. The operational state is monitored during this time to enforce policies applying specifically to the start-up process.

- *Start-up high*
  This mode is the same as startup_low except it is used to bring the reactor to its full power of 2.0 MW.

- *Operating low*
  The reactor is operating at 0.2 MW in this mode. In this mode the reactor power can be regulated by either manual or automatic adjustment of the regulator rod.

- *Operating high*
  This mode is similar to operating low. The difference is in the position of the header and the requirement that the pump be on in this mode.

- *Scrammed*
  Normal operation is prohibited in this mode. The only transition from this mode is to the inactive mode that occurs with the scram reset command.

The commands for the system cause the transitions between modes. The one command that results in device action is the scram command.

The machines corresponding to the specific devices are relatively simple because little functionality is required of the devices. The safety rod machines have very little state operation associated with the operation of the control rods. The only two modes are *operating* and *failed*. The regulator rod machine can be operated both manually and automatically; therefore, it has the modes *auto* and *manual*. The header has the modes *up* and *down*. The pump can be either *on* or *off*. The neutron detector has the modes *in_place* and *removed*. In addition, all of the machines have the mode *failed* which is used to indicate that a failure of the device itself has been detected. In the *failed* mode, normal device operation is precluded.

A representative set of the safety policies for the reactor have been specified using the machines and modes described above. Included in the specified safety policies are all of those that relate to direct control of the devices and selected monitoring policies. The monitoring policies are similar enough in function and specification that it was not necessary to specify all of them to demonstrate the feasibility of describing the UVAR safety kernel with the language provided by the translator.

8.4 Conclusion

The purpose for building the two prototypes described in this chapter was to evaluate the feasibility of the safety kernel concept and mechanisms with real systems. Safety policy specifications were developed for both systems to describe the systems and the policies to be enforced for each. The safety kernel translator was used to generate a source code representation of the safety kernel for the MSS which was then compiled to produce an operating safety kernel. Performance measurements of the safety kernel demonstrate that the overhead due to either network communication and enforcement of monitoring policies is minimal. The safety kernel is capable of handling more than five times the number of requests that are expected to be generated by the MSS application software. Feasibility testing of the safety kernel with the application software and simulated devices demonstrates that the safety kernel provides the functionality required for system operation and, to the extent tested, enforces the safety policies designated as being kernel-enforced.

9 Conclusions and Future Work

9.1 Conclusions

The subject of this research has been an evaluation of the feasibility of the safety kernel as a software architecture for the enforcement of safety policies. Previous work has yielded some basic enforcement kernel concepts and proposed safety kernel designs. However, many of the basic feasibility issues in employing a safety kernel with practical safety-critical systems have not been previously addressed. In the evaluation of feasibility, four major areas have been addressed.

- Policy enforcement
  To facilitate evaluation and description of the role of the safety kernel we have developed a classification system for safety policies. Rushby's original ideas on kernel-enforcement have been extended to define the issues in selection of classes for enforcement. The concept of a weakened safety policy has been introduced that facilitates enforcement of policies that might otherwise not be enforced. The seven classes of safety policies identified for enforcement by the safety kernel prototype are related to device operation and therefore are critical to system safety.

- Reliable enforcement
  The safety kernel must be able to enforce safety policies reliably and do so in spite of failures of other system components. Our analysis of potential component failures resulted in a set of requirements for reliable safety kernel policy enforcement. For systems where safety is the primary concern, we have demonstrated that the system-kernel design used with *security* kernels is neither necessary nor the most effective for meeting the reliability requirements. In safety systems, correct operation implies either correct functionality or correct failure. This permits a system design in which both error prevention and error detection and response techniques are employed to meet dependability requirements.

- Implementation
  We have developed an implementation strategy employing a special-purpose specification language and translator. To identify this level of reuse, safety policy enforcement was characterized and models were developed for the components with which the safety kernel interacts. Although the particular policies and components are application specific, the types of the policies and characteristics of the components are general. From these general characteristics, a framework consisting of a machine abstraction, built-in context information, built-in control mechanisms, and user-supplied system description information has been developed. The framework provides a context for specification of safety policies. The special-purpose specification language is used for description of the application-specific context

information and the safety policies. The translator generates a source code representation of the safety kernel from the description.

- Verification
Safety kernel verification is facilitated by the use of the translator. In principle, the translator and the compiler are reused, therefore verification should focus on their operation rather than on individual artifacts that they process. If it can be shown that the translator and the compiler are correct, then a correct policy specification results in a correct safety kernel (although a specification error could still result in undesirable operation). The feasibility of analyzing this specification, to ensure important properties, has been established. We have shown that exhaustive testing can be employed for verification of non-trivial properties. In addition, a test system has been developed that permits automated, high-volume testing.

To support research in the areas described above we have employed the MSS and UVAR as case studies. We have identified safety policies for both of these systems and grouped them according to the classes of safety policies. A system design suitable for reliable policy enforcement in systems of this type has also been developed. We have produced safety policy specifications for both applications and have demonstrated the feasibility of the translator and special-purpose specification language. The MSS safety kernel prototype has been shown to have a positive impact on the application software and its performance has been measured to be more than sufficient for the demands of the two systems. The prototype has been operated in the test harness for 1000 surgical procedures under normal operation and with erroneous commands. Static analysis has been used to demonstrate properties of the MSS safety policy specification. Finally, exhaustive testing has been employed to verify properties of the interlock policy mechanism for the MSS.

Examination of the four areas above indicates that it is feasible for the safety kernel to enforce selected safety policies for a safety-critical application. Another way of evaluating the safety kernel contribution is to consider a software control system for an application like the MSS or UVAR without a safety kernel. It is clear that basic system safety policies do not change with the presence or absence of a safety kernel. Hence, all of the safety policies would need to be enforced. The bottom line is that policy enforcement like that provided by the safety kernel is essential to system safety. We argue that providing for this enforcement in a consistent, well-defined, and possibly reusable manner is feasible and is superior to ad hoc implementation of policy enforcement functionality.

9.2 Future Work

In order to advance the safety kernel from the level of a feasible concept to a practical, reliable technology, several open issues need to be addressed. The topics listed below focus on continued development and evaluation of the safety kernel concept and on technology required for reliable policy enforcement.

- *Translation: system safety specification to safety policy specification*
The translation from system safety specification to safety kernel description is presently informal. This translation could be much more structured and ideally would

be performed mechanically. To facilitate translation, the form of both the system safety specification and the safety policy specification should be evaluated.

- *Verification*
  The verification process outlined here needs to be developed to permit a systematic demonstration of safety kernel reliability. Included in this effort will be techniques for verification of the translator and for verification between the system safety specification and the policy specification. Formal specification of the translator and an implementation tailored to facilitate verification will likely be required for the development of verification techniques. The analysis of the policy specification should be extended to include a more comprehensive set of properties. Finally, testing for verification of properties and high-volume testing need to be integrated into the verification process.

- *Implementation*
  As mentioned previously, the safety kernel translator needs to be extended to support aperiodic device monitoring. In addition, when the safety kernel is applied to systems beyond the MSS and UVAR, it is possible that the translator will need to be adapted. This could result in additional functionality including more complicated failure responses and mode updates resulting from a change in operational state. Ensuring that periodic monitoring activities, aperiodic monitoring activities, and application requests can be scheduled is an additional area for future work.

- *Evaluation*
  To assess the system design, the effectiveness and generality of the translator, and the domain of applicability of the safety kernel, the safety kernel should be employed with other applications. The first target will be a prototype of the safety kernel for the UVAR. Particularly useful information could be gained from systems with different dependability requirements, e.g., systems where reliability is the primary concern.

References

1. Addy, E. A., “A Case Study on Isolation of Safety-Critical Software,” in *Proceedings of COMPASS* 1991, Washington, D.C., pp. 75-83.

2. Ames, S. R., Jr., M. Gasser and R. R. Schell, “Security Kernel Design and Implementation: an Introduction,” *IEEE Computer* Vol. 16-7 (July 1983) pp. 14-22.

3. Ammann, P. E., S. S. Brilliant, and J. C. Knight, “The Effect of Imperfect Error Detection on Reliability Assessment via Life Testing,” *IEEE Transactions on Software Engineering* Vol. 20-2 (February 1994).

4. Anderson, T. Ed., *Safe and Secure Computing Systems* (Blackwell Scientific Publications, 1989).

5. Anderson, T. and P. A. Lee, *Fault Tolerance Principles and Practice* (Prentice Hall International, Inc., London 1981) p. 64.

6. Avizienis, A., “The N -Version Approach to Fault-Tolerant Software,” *IEEE Transactions on Software Engineering* Vol. SE-11 (1985) pp. 1491-1501.

7. Bricker, A., M. Gien, M. Guillemont, J. Lipkis, D. Orr, and M. Rozier, “Architectural Issues in Microkernel-Based Operating Systems: the CHORUS Experience,” *Computer Communications* Vol. 14-6 (July/August 1991) pp. 347-357.

8. Brilliant, S.S., Knight, J.C., and Leveson, N.G., “The Consistent Comparison Problem in N-Version software,” *IEEE Transactions on Software Engineering* Vol. 15-11 (November 1989) pp. 1481-1485.

9. Butler, R. W. and G. B. Finelli, “The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software,” *IEEE Transactions on Software Engineering* Vol. 19-1 (January 1993) pp. 3-12.

10. Campbell, R. H., N. Islam, D. Raila, and P. Madany, “Designing and Implementing CHOICES: An Object-Oriented System in C++,” *CACM* Vol. 36-9 (Sept 1993) p. 117-126.

11. Chen, L., and A. Avizienis, “N-version Programming: A Fault-Tolerance Approach to Reliability of Software Operation,” in *Digest of papers of the 8th International Symposium on Fault-Tolerant Computing,* Tolouse, France, 1978, pp. 3-9.

12. Cristian, F., “Basic Concepts and Issues in Fault-Tolerant Distributed Systems,” *Operating Systems of the 90s and Beyond* (Springer-Verlag, Berlin 1991) pp. 118-149.

13. Eckhardt, D. E, and L. D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors," *IEEE Transactions on Software Engineering,* Vol. SE-11 (1985), pp. 1511-1517.

14. Eckhardt, D. E, and L. D. Lee, "Fundamental Differences in the Reliability of N - Modular Redundancy and N -Version Programming," *The Journal of Systems and Software,* Vol. 8 (1988) pp. 313-318.

15. Fraim, L. J., "Scomp: A Solution to the Multilevel Security Problem," *IEEE Computer*, Vol. 16-7 (July 1983) pp. 26-34.

16. Garman, J. R., "The Bug Heard 'Round the World," *ACM Software Engineering Notes* Vol. 6-5 (October 1981) pp. 3-10.

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

18. Goodenough, J. B. and S. L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Transactions on Software Engineering* SE-1 (June 1975).

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

20. Higgs, J. C., "A High Integrity Software Based Turbine Governing System," in *Proceedings of Safety of Computer Control Systems* (*SAFECOMP* '83). Pergamon, Elmsford, N.Y. pp. 207-218.

21. Joseph, M.K., Architectural Issues in Fault-Tolerant, Secure Computing Systems, Ph.D. Thesis, UCLA, Los Angeles, USA, 1988.

22. Karger, P. A., et al, "A Retrospective on the VAX VMM Security Kernel," *IEEE Transactions on Software Engineering*, 17-11 (Nov. 1991) pp. 1147-1165.

23. Kirschen, D., "An Overview of the Mach Operating System," *Operating Systems Technical Committee Newsletter*, Vol. 3-2, p. 57.

24. Knight, J. C., A. G. Cass, A. M. Fernandez, and K. G. Wika, "Testing a Safety-Critical Application," Department of Computer Science, University of Virginia, Technical Report No. CS-94-08, February 1994.

25. Knight, J. C. and D. M. Kienzle.,"Safety-Critical Computer Applications: The Role of Software Engineering," Technical Report TR-92-23, Department of Computer Science, University of Virginia, 1993.

26. Knight, J.C., and Leveson, N.G., "An Empirical Study of Failure Probabilities in Multi-Version Software," *Digest of papers of the 16th International Symposium on Fault-Tolerant Computing*, Vienna, Austria, 1986, pp 165-170.

27. Knight, J. C., and N. G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," *IEEE Transactions on Software*

Engineering Vol. SE-12 (1986) pp. 96-109.

28. Kopetz, H., “Event-Triggered Versus Time-Triggered Real-Time Systems,” *Operating Systems of the 90s and Beyond* (Springer-Verlag, Berlin 1991) pp. 86-101.

29. Laprie, J. C., “The Dependability Approach to Critical Computing Systems,” in *Proceedings of the 1st European Conference On Software Engineering,* Strasbourg, France, 1987, pp.233-243.

30. Leveson, N.G., “Software Fault Tolerance in Safety-Critical Applications,” in *Proceedings of the 3rd International Conference on Fault-Tolerant Computing Systems*, Bremerhaven, Germany, 1987.

31. Leveson, N. G., “Software Safety: Why, What, and How,” *ACM Computing Surveys*, Vol. 18 (June 1986) pp. 125-163.

32. Leveson, N. G. and T. J. Shimeall, “Safety Assertions for Process-Control Systems,” in *Proceedings of 13th International Conference on Fault Tolerant Computing*, Milan, Italy, June, 1983.

33. Leveson, N. G., T. J. Shimeall, J. L. Stolzy, and J. C. Thomas, “Design for Safe Software,” in *Proceedings AIAA Space Sciences Meeting*, Reno, Nevada, 1983.

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

35. McCormick, N. J., *Reliability and Risk Analysis* (Academic Press, Inc., San Diego, CA, 1981).

36. Miller, D. R., “Making Statistical Inferences About Software Reliability,” NASA Contractor Report 4197, NASA Langley Research Center, Hampton, Virginia, USA, 1988.

37. Miller, D. R., “The role of Statistical Modeling and Inference in Software Quality Assurance,” in *Software Certification*, ed. B. de Neumann, (Elsevier Applied Science, London, UK, 1989) pp. 135-152.

38. Moffett, J. D. and J. A. McDermid, “Policies for Safety-Critical Systems: the Challenge of Formalisation,” *Fifth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, Toulouse, France, Oct. 1994.

39. NATO AC/310 Ad Hoc Working Group on Munition Related Safety Critical Computing Systems, “Safety Design Requirements and Guidelines for Munition Related Safety Critical Computing Systems,” NATO Standardization Agreement (STANAG) 4404 (Draft), March 1990.

40. Neumann, P. G., “On Hierarchical Design of Computer Systems for Critical Applications,” *IEEE Transactions on Software Engineering* Vol. SE-12 (September 1986) pp. 905-920.

41. Neumann, P.G., Editor, “Risks to the Public,” *Software Engineering Notes*.

42. Parnas, D. L., “On the Criteria to be Used in Decomposing Systems Into Modules,” *Communications of the ACM* Vol. 15 (Dec. 1972) pp. 220-225.

43. Prieto-Díaz, R., “Status Report: Software Reusability,” *IEEE Software* Vol. 10-5 (May 1993) pp. 61-66.

44. Rushby, J., “Kernels for Safety?,” in *Safe and Secure Computing Systems*, T. Anderson Ed. (Blackwell Scientific Publications, 1989) pp. 210-220.

45. Rushby, J. and B. Randell, “A Distributed Secure System,” *IEEE Computer* Vol. 16-7 (July 1983) pp. 55-67.

46. Siewiorek, D.P., and Swarz, R.S., *The Theory and Practice of Reliable System Design* (Digital Press, Bedford, MA, USA, 1982).

47. Tanenbaum, A. S., R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum, “Experiences with the Amoeba Distributed Operating System,” *CACM*, 33-12 (Dec. 1990) pp. 46-63.

48. Taylor, D. J., D. E. Morgan, and J. P. Black, “Redundancy in Data Structures: Improving Software Fault Tolerance,” *IEEE Transactions on Software Engineering* Vol. SE-6 (Nov. 1980) pp. 585-594.

49. Taylor, R. H., et. al., “Augmentation of human precision in computer-integrated surgery,” *Innovation and Technology in Biology and Medicine* Vol. 13 (1992) pp. 450-468.

50. Taylor, R. H., et al., “Taming the Bull: Safety in a Precise Surgical Robot,” in *Proceedings Fifth International Conference on Advanced Robotics*, Pisa, Italy, June 1991, pp. 865-870.

51. Toy, W. N., “Fault-Tolerant Design of Local ESS Processors,” *Proc. IEEE Vol.* 66 (Oct. 1978) pp. 1126-1145.

52. University of Virginia Reactor Safety Committee, *University of Virginia Reactor Design and Analysis Handbook*, last modified July 7, 1989.

53. Wika, K. G., “A User Interface and Control Algorithm for the Video Tumor Fighter,” Masters Thesis, University of Virginia, May 1991.

Appendix A - Case Study Safety Policies

Table 2: MSS Safety Policies

Policy Statement	Class	Enforcement Status
If the seed moves faster than 1.0 mm/s with respect to a coordinate system fixed to the markers the current in the coils must be dropped to zero and the operator notified. This check must be executed every 0.5 s when there is current in the coils.	*System Operation*	*Weakened*
The seed must always be within 2 mm of its expected position as determined from the coil current-time profiles and a model of the seed movement through the brain as a function of magnetic impulse. This check must be executed every 0.5 s when there is current in the coils.	*System Operation*	*Weakened*
The coil current observed by the independent sensor must always be within 2.0 A of the current predicted by the coil ramping model.	*Device Operation*	*Kernel*
At anytime a current controller is regulating the current in a coil, the independent sensor must be read and the value compared with the predicted current every 0.5 s.	*Device Operation*	*Kernel*
From the time a current controller is turned on, the current must be read and the value compared with the predicted current at least every 2.0 s.	*Device Operation*	*Kernel*
The x-ray sources must never be active for more than 0.1 s at any one time.	*Device Operation*	*Kernel*
The status conditions indicated by the current controller must be monitored to detect when the current controller has failed.	*Device Failure*	*Kernel*
The x-ray source status indicators must be monitored to detect source failure.	*Device Failure*	*Kernel*
The current requested of a current controller must range from -100 A to +100 A.	*Input from Computer*	*Kernel*
Except in an emergency shutdown situation the magnitude of the charging/discharging current change rate must be between 0.0 and 1.0 inclusive.	*Input from Computer*	*Kernel*
An x-ray device must be in the “off” state for 0.2 s before the invocation of an “on” command.	*Input from Computer*	*Kernel*
The total x-ray dose during an operation must be less than 1.4 R.	*Input from Computer*	*Kernel*

Table 2: MSS Safety Policies

Policy Statement	Class	Enforcement Status
Prior to the commencement of a ramping sequence, a reversal check must be executed to ensure that the requested currents provide the desired direction within 30 degrees and an impulse within 20%.	*Input from Computer*	*Weakened*
Prior to executing a seed movement, a determination must be made if there is the potential for “run-away” producing forces in the requested direction of seed movement for the requested distance.	*Input from Computer*	*Weakened*
When the helmet coordinate position of an object is computed from the two screen positions the error in the fit must be less than 2.0 pixels.	*Input from Computer*	*Weakened*
The 3D positions of the markers and seed must be within specified expected containment volumes. The containment volume for the seed will move with the seed, while the marker containment volumes will be determined when objects are identified on the x-ray images and will remain fixed. The initial position of the seed containment volume will also be established at this time.	*Input from Computer*	*Weakened*
The distances between all pairs of markers will be determined when the objects are identified on the x-ray images. Each time the seed is located the distances between all pairs of markers will be computed and must be within 2.0 mm of the original values.	*Input from Computer*	*Weakened*
When the transformation matrix between helmet and UCS coordinates is computed, the root mean square error in the fit must be less than 2.0 pixels and the error in the fit for any marker must be less than 3.0 pixels.	*Input from Computer*	*Weakened*
The calibration parameters computed for the vision system must satisfy reasonableness checks that are based on physical measurements of the vision system.	*Input from Computer*	*Application*
New current control parameters must be entered each time before a ramping sequence is initiated.	*Software Error*	*Weakened*
A seed movement must have been requested each time the coils are charged unless one of the explicit current control commands is used.	*Software Error*	*Weakened*
If while there is current in the coils, the vision system cannot locate the seed or marker(s) or one of the location or correspondence requirements is not met, the coil currents must be dropped to zero.	*Failure Response*	*Kernel*
If the current in one or more coils is found to be inconsistent with its expected value, the current in all coils must be dropped to zero.	*Failure Response*	*Kernel*
If a current controller fails, the current in all coils must be dropped to zero.	*Failure Response*	*Kernel*
If the seed has moved too fast or too far, the current in the coils must be adjusted at the maximum rate to zero.	*Failure Response*	*Kernel*
If an x-ray source fails, the current in the coils must be dropped to zero and the power to the source disconnected.	*Failure Response*	*Kernel*

Table 2: MSS Safety Policies

Policy Statement	Class	Enforcement Status
A bounding volume must be specified for each marker and the marker as identified by the user must be within this volume. An example volume would be the left half of the image set.	*Operator Input*	*Application*
In the object identification, each object must be identified once.	*Operator Input*	*Application*
In developing the transformation from a view to the UCS, the ratio of the common axis distance between pairs of markers in the reference view and the view for which the transformation is being computed must be within 10% of the weighted average of the ratio determined from the distances between all pairs of markers. The one exception would be if the common axis distance is less than 10 pixels on one of the views. In this case, for each pair, the ratio of the common axis distance to the total distance for all marker pairs must be within 0.02.	*Operator Input*	*Application*
In developing the transformation from a view to the UCS, the difference between the common axis position of the center of mass and the common axis position of the marker should be computed for both the reference view and the view for which the transformation is being computed. For each marker, these two distances should be within 2 mm.	*Operator Input*	*Application*
The position of an object on an image must not be within 10 pixels of any other object on the image.	*Sensor Input*	*Application*
Checks must exist to evaluate the quality of the image that is received from an imaging axis. These checks may occur as a result of some of the other object location checks.	*Sensor Input*	*Application*
When an object is located on an image, criteria specified for the object and for the particular location method used must be met in order for the location to be considered valid.	*Sensor Input*	*Application*
A coil current sensor is faulty if its value is outside of the range -200 to +200 A.	*Sensor Input*	*Kernel*
All files for a given patient must be coded with a unique identifier that is checked when the information from the file is read.	*Application Data*	*Application*
The vision system parameters (e.g., camera constants) must match the values for the components of the vision system.	*Application Data*	*Application*
The constants used in the control of the currents must correspond to the values for the current controllers and coils.	*Application Data*	*Application*
The number of markers, their sizes, and their position descriptions must conform to the markers used on a patient.	*Application Data*	*Application*

Table 2: MSS Safety Policies

Policy Statement	**Class**	**Enforcement Status**
For each imaging axis and each marker, the expected bounding rectangle for the marker must be specified using a set of terms such as left-half, right-half, etc. When objects are identified by the operator, the position of each object must fall within this bounding rectangle. These bounding rectangles should not be specified by looking at the actual fluoroscope images. The intent is to provide a check on the operator's interpretation of the images.	*Operator Error*	*Application*
The position of an object identified on an x-ray image must not be within 5.0 pixels of the position of any of the other objects on the image.	*Operator Error*	*Application*
A request for seed movement must not exceed a distance of 15 mm.	*Operator Error*	*Application*
If the seed is towing a catheter, a warning must be issued if a requested direction differs by more than 60 degrees from the previous direction of seed movement. A confirmation is required to proceed with the move.	*Operator Error*	*Application*
The MR images displayed must be those for the patient and must be the set selected for the particular operation.	*Operator Information*	*Application*
The MR images must be clearly labeled as to the imaging point of view.	*Operator Information*	*Application*
The seed must be represented with a color that is easily distinguished from the background MR images and all other objects represented on the display.	*Operator Information*	*Application*

Table 3: UVAR Safety Policies

Policy Statement	Class	Enforcement Status
During reactor start-up, the period, i.e., the effective “e-folding time,” must not be below 100 seconds with a single status source or below 30 seconds with corroborating information.	*System Operation*	*Kernel*
If any of the following conditions is true the control rods must be in the scram position.		
Safety channel 1 indicates a power greater than 125% (2 MW mode) or 12.5% (0.2 MW mode).	*System Operation*	*Kernel*
Safety channel 2 indicates a power greater than 125% (2 MW mode) or 12.5% (0.2 MW mode).	*System Operation*	*Kernel*
The period amplifier reads a value less than 3.5 s.	*System Operation*	*Kernel*
The scram button on the console is pressed.	*System Operation*	*Kernel*
The radiation level on the bridge exceeds 30 mR/hr. In this case the ventilation door and the reactor room personnel door should be closed automatically.	*System Operation*	*Kernel*
The primary coolant pump is turned on with the header down.	*System Operation*	*Kernel*
The primary coolant pump is turned off with the header up.	*System Operation*	*Kernel*
The flow in the primary cooling system is below 3,400 liters/min (900 gpm).	*System Operation*	*Kernel*
The scram button at the reactor room personnel door is pressed.	*System Operation*	*Kernel*
The scram button on the ground floor is pressed.	*System Operation*	*Kernel*
The reactor room truck door is open.	*System Operation*	*Kernel*
The emergency escape hatch is open.	*System Operation*	*Kernel*
The air pressure to the primary header is greater than or equal to 2 psi.	*System Operation*	*Kernel*
The reactor inlet water temperature exceeds 105˚ F.	*System Operation*	*Kernel*
The pool level falls below 19 ft., 3 1/4 in.	*System Operation*	*Kernel*
The radiation at the reactor face exceeds 2 mR/hr.	*System Operation*	*Kernel*

Table 3: UVAR Safety Policies

Policy Statement	Class	Enforcement Status
The key switch on the console is turned off.	*System Operation*	*Kernel*
The range switch is set to 2 MW with the header down.	*System Operation*	*Kernel*
Evaluation or fire alarm is active.	*System Operation*	*Kernel*
The primary coolant pump is on with the header down.	*System Operation*	*Kernel*
The following result in an intermittent tone being sounded.		
The regulating rod shifts from automatic to manual.	*System Operation*	*Kernel*
High radiation is detected on any area monitor or on either argon monitor.	*System Operation*	*Kernel*
High radiation is detected on the core gamma monitor.	*System Operation*	*Kernel*
High radiation is detected on the criticality monitor.	*System Operation*	*Kernel*
High radiation is detected on the constant air monitor.	*System Operation*	*Kernel*
Entry into the demineralizer room is detected.	*System Operation*	*Kernel*
Entry into the heat exchanger room is detected.	*System Operation*	*Kernel*
A high ΔT is measured across the reactor core.	*System Operation*	*Kernel*
High demineralizer conductivity is detected.	*System Operation*	*Kernel*
The secondary pump is de-energized.	*System Operation*	*Kernel*
The following two policies have indicators other than a scram or intermittent tone		
When the key switch is on, local alarm bells are activated by opening the heat exchanger room door or the demineralizer room door.	*System Operation*	*Kernel*
If the error signal, as displayed on the deviation meter, exceeds 7.5%, then the regulating rod is switched from automatic to manual mode and an alarm is sounded.	*System Operation*	*Kernel*

Table 3: UVAR Safety Policies

Policy Statement	Class	Enforcement Status
The rods must not be withdrawn at a rate faster than 1.5 mm/s.	*Device Operation*	*Kernel*
The position of the regulating rods must be adjusted at least once per second based on the power output of the reactor.	*Device Operation*	*Kernel*
If the regulating rod is either at its top or bottom limit, then the regulating rod is switched from automatic to manual mode and an alarm is sounded.	*Device Operation*	*Kernel*
If any of the scram conditions identified above are true then it must not be possible to withdraw the rods. The rods must remain in the scram position.	*Device Operation*	*Kernel*
Signals that indicate device status must be monitored to detect potential device failures.	*Device Failure*	*Kernel*
The source range must be indicating at least 2 cps to withdraw a safety rod.	*Input from Computer*	*Kernel*
The nuclear instrumentation must be out of test mode to withdraw a safety rod.	*Input from Computer*	*Kernel*
The reactor must be scrammed when any failure occurs that might interfere with safe operation of the reactor.	*Failure Response*	*Kernel*
Failures that do not pose an immediate threat to reactor safety, but that require possible operator intervention must result in an audible alarm being sounded.	*Failure Response*	*Kernel*
The instrumentation must respond as specified to control inputs.	*Sensor Input*	*Kernel*
The instrumentation must respond to the removal of the neutron source.	*Sensor Input*	*Kernel*
If the normal control switch is used to move the regulating rod, then the regulating rod is switched from automatic to manual mode and an alarm is sounded.	*Operator Error*	*Kernel*
If the linear power recorder is turned off, then the regulating rod is switched from automatic to manual mode and an alarm is sounded.	*Operator Error*	*Kernel*
If the switch that determines whether the mode is manual or automatic is set to manual, then the regulating rod is switched from automatic to manual mode and an alarm is sounded.	*Operator Error*	*Kernel*

Appendix B - Safety Kernel Translator Grammar

Following is a context-free grammar for the language recognized by the prototype safety kernel translator. Non-terminals that appear in bold, italic font are taken from the grammar summary presented in *The C++ Programming Language* by Stroustrup. Not all of the C++ features are supported, so some elements such as argument declaration lists are not taken directly from the C++ grammar. Note that there are context-sensitive rules required. Thus, for example, the grammar does not indicate that a mode should be declared in order to be referenced in a policy.

```
spec → spec_list
spec_list → spec_list spec_entry | spec_entry
spec_entry →
     DEV_TYPE:        machine_type |
     DEV_CONFIG:      configuration_file |
     CHILD:           machine_type child_name |
     MODE:            mode_name |
     INIT_MODE:       mode_name |
     COMMAND:         command_name ( formal_parameter_list )  |
     ACT_PROC:        command_name: procedure_call;
                           error_condition_name |
     OP_ST_VAR:       variable_declararion |
     ST_ACQ_PROC:     acquire_state(state_vector);
                           error_condition_name |
     CNT_PAR:         variable_declaration |
     CNT_PAR_UPD:     command_name: procedure_call;
                           error_condition_name |
     SCHEDULE:        schedule_name: time_list; completion_time,
                           deadline |
     FRAME_LENGTH:    time |
     WDOG_SCHEDULE:   schedule_name |
     INCLUDE:         filename |
     CONST:           simple_type constant_name = literal |
     BASE_CLASS:      access_specifier class_name |
     CONST_PARAM:     formal_parameter |
     BASE_CONST:      procedure_call |
     INIT_PROC:       procedure_call; error_condition_name |
     RESET_PROC:      procedure_call; error_condition_name |
     DECL:            variable_declararion |
     DEV_DECL:        variable_declararion |
     PROC_DEF:        fct_body |
     INTERLOCK:       command_name interlock_body END: |
     MONITOR:         mode_name error_detect_body END: |
     ERROR_COND:     error_condition_name failure_response_body
                          END:



interlock_body →
     transition_condition |
     interlock_body interlock_mode_condition
interlock_mode_condition →
     mode_list transition_condition END: |
     mode_list interlock_parent_mode_condition END:
interlock_parent_mode_condition → parent_mode_list
     transition_condition END:
error_detect_body →
     error_detect_condition |
     parent_error_detect_condition
parent_error_detect_condition →
     parent_mode_list error_detect_condition END:
error_detect_condition →
     schedule_name condition |
     error_detect_condition condition
failure_response_body →
     response_procedure |
     mode_failure_response_list
mode_failure_response_list →
     mode_failure_response |
     mode_failure_response_list mode_failure_response
mode_failure_response →
     mode_list response_procedure END: |
     mode_list parent_mode_failure_response response_procedure
           END:
parent_mode_failure_response →
     parent_mode_list response_procedure END: |
     parent_mode_failure_response parent_mode_list
          response_procedure END:
response_procedure → RESPONSE: procedure_call;
          error_condition_name
mode_list →
     mode_entry |
     mode_list mode_entry
parent_mode_list →
     parent_mode_entry |
     parent_mode_list parent_mode_entry
mode_entry → MODE: mode_name
parent_mode_entry → PARENT_MODE: parent_mode_name
```

```
transition_condition →
      new_state |
      transition_condition condition
new_state → NEW_MODE: new_mode
new_mode →
      mode_name |
      error_condition_name
condition → CONDITION: procedure_call; error_condition_name

procedure_declaration → procedure_name ( formal_parameter_list )
procedure_call → procedure_name ( actual_parameter_list )
formal_parameter_list →
            |
      formal_parameter |
      formal_parameter_list, formal_parameter
formal_parameter →
      simple_type identifier |
      simple_type &identifier |
      simple_type *identifier
actual_parameter_list →
            |
      actual_parameter |
      actual_parameter_list, actual_parameter
actual_parameter →
      identifier |
      &identifier |
      literal
variable_declaration →
      simple_type identifier |
      simple_type *identifier

time_list →
      time_list , event_time |
      event_time
configuration_file → filename
schedule_name → identifier
machine_type → identifier
child_name → identifier
mode_name → identifier
command_name → identifier
error_condition_name → identifier
procedure_name →
      identifier |
      class_name::identifier
```

constant_name → ***identifier***
event_time → *time*
completion_time → *time*
deadline → *time*
time → ***literal***

Appendix C - Safety Policy Specifications

MSS Safety Policy Specification

System Machine

```
#include “config_macros.h”

INCLUDE:       coordinate.h
INCLUDE:       helmet.h
INCLUDE:       local_system.h

DEV_TYPE:      mss_system

DEV_CONFIG:    servoamp.fil
DEV_CONFIG:    xray.fil

CHILD:         mss_servoamp XA_servo
CHILD:         mss_servoamp XB_servo
CHILD:         mss_servoamp YA_servo
CHILD:         mss_servoamp YB_servo
CHILD:         mss_servoamp ZA_servo
CHILD:         mss_servoamp ZB_servo
CHILD:         mss_xray X_xray
CHILD:         mss_xray Y_xray

FRAME_LENGTH:  1.0

SCHEDULE:      COIL_ON: 0.0, 0.25, 0.5, 0.75; 0.02, 0.5
SCHEDULE:      COIL_IDLING: 0.0, 0.5; 0.02, 0.5
SCHEDULE:      COIL_OFF: 0.0; 0.02, 0.5
SCHEDULE:      XRAY_ON: 0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, \
                      0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, \
                      0.8, 0.85, 0.9, 0.95; 0.01, 0.05
SCHEDULE:      XRAY_OFF: 0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, \
                      0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, \
                      0.8, 0.85, 0.9, 0.95; 0.01, 0.05
SCHEDULE:      WATCHDOG: 0.0, 0.33, 0.67; 0.01, 0.5

WDOG_SCHED:    WATCHDOG

BASE_CLASS:    public local_system
CONST_PARAM:   int obj_id
CONST_PARAM:   dispatcher *disp
BASE_CONST:    local_system(obj_id, disp)


MODE:          inactive
MODE:          vision_enabled
MODE:          currents_idling
MODE:          moving_seed
MODE:          vision_calibration
MODE:          shutdown
```

```
INIT_MODE:      inactive

COMMAND:        vision_calibration_begin()
COMMAND:        vision_calibration_end()
COMMAND:        move_seed_begin(coordinate helmet_seed_pos, coordinate UCS_seed_pos)
COMMAND:        move_seed_end(double present_force)
COMMAND:        idle_currents_begin(double present_force)
COMMAND:        idle_currents_end(double present_force)
COMMAND:        enable_vision()
COMMAND:        disable_vision()
COMMAND:        emergency_shutdown()
COMMAND:        shutdown_reset()

COMMAND:        register_seed_position(coordinate helmet_seed_pos, coordinate UCS_seed_pos)

ACT_PROC:       move_seed_begin: do_begin_move_seed(helmet_seed_pos, UCS_seed_pos); \
                                      MOVE_SEED_BEGIN_FAILED
ACT_PROC:       register_seed_position: do_register_seed_position (helmet_seed_pos, \
                                      UCS_seed_pos); RUNAWAY_SEED

CNT_PAR:        coordinate most_recent_helmet_seed_position
CNT_PAR:        coordinate most_recent_UCS_seed_position
CNT_PAR:        double position_update_time

CONST:          double MAX_IDLE_PERIOD_MOVEMENT = 15.0
CONST:          double MAX_INCREMENTAL_MOVEMENT = 5.0
CONST:          double MAX_UCS_MOVEMENT = 2.0
CONST:          double THRESHOLD_FORCE = 2.0
CONST:          double MIN_UPDATE_PERIOD = 2.0

INTERLOCK:      vision_calibration_begin
  MODE:         inactive
  NEW_MODE:     vision_calibration
  END:

  MODE:         vision_enabled
  MODE:         currents_idling
  MODE:         moving_seed
  MODE:         vision_calibration
  MODE:         shutdown
  NEW_MODE:     MUST_BE_INACTIVE
  END:
END:


INTERLOCK:      vision_calibration_end
  MODE:         vision_calibration
  NEW_MODE:     inactive
  END:

  MODE:         inactive
  MODE:         vision_enabled
  MODE:         currents_idling
  MODE:         moving_seed
  MODE:         shutdown
  NEW_MODE:     NOT_IN_VISION_CALIBRATION
  END:
END:


INTERLOCK:      move_seed_begin
  MODE:         inactive
  MODE:         vision_enabled
  MODE:         currents_idling
  NEW_MODE:     moving_seed
  END:

  MODE:         moving_seed
  NEW_MODE:     ALREADY_MOVING_SEED
  END:

  MODE:         vision_calibration
  NEW_MODE:     IN_VISION_CALIBRATION
  END:

  MODE:         shutdown
  NEW_MODE:     SHUTDOWN
  END:
END:


INTERLOCK:      move_seed_end
  MODE:         moving_seed
  MODE:         currents_idling
  NEW_MODE:     inactive
  CONDITION:    servoamps_zero_currents(); CURRENTS_NOT_ZERO
  CONDITION:    servoamps_inhibited(); SERVO_AMPS_NOT_INHIBITED
  END:

  MODE:         inactive
  MODE:         vision_enabled
  MODE:         vision_calibration
  MODE:         shutdown
  NEW_MODE:     NOT_MOVING_SEED
  END:
END:


INTERLOCK:      idle_currents_begin
  MODE:         inactive
  MODE:         vision_enabled
  NEW_MODE:     currents_idling
  CONDITION:    servoamps_zero_currents(); CURRENTS_NOT_ZERO
  END:

  MODE:         moving_seed
  NEW_MODE:     currents_idling
  CONDITION:    servoamps_zero_currents(); CURRENTS_NOT_ZERO
  END:

  MODE:         currents_idling
  NEW_MODE:     ALREADY_IDLING
  END:

  MODE:         vision_calibration
  NEW_MODE:     IN_CALIBRATION
  END:

  MODE:         shutdown
  NEW_MODE:     SHUTDOWN
  END:
END:


INTERLOCK:      idle_currents_end
  MODE:         currents_idling
  NEW_MODE:     inactive
  CONDITION:    servoamps_zero_currents(); CURRENTS_NOT_ZERO
  CONDITION:    servoamps_inhibited(); SERVO_AMPS_NOT_INHIBITED
  END:
```

```
  MODE:         inactive
  MODE:         vision_enabled
  MODE:         moving_seed
  NEW_MODE:     NOT_IDLING
  END:


  MODE:         vision_calibration
  NEW_MODE:     IN_CALIBRATION
  END:

  MODE:         shutdown
  NEW_MODE:     SHUTDOWN
  END:
END:


INTERLOCK:      enable_vision
  MODE:         inactive
  NEW_MODE:     vision_enabled
  END:

  MODE:         vision_enabled
  MODE:         currents_idling
  MODE:         moving_seed
  NEW_MODE:     VISION_ALREADY_ENABLED
  END:


  MODE:         vision_calibration
  NEW_MODE:     IN_CALIBRATION
  END:

  MODE:         shutdown
  NEW_MODE:     SHUTDOWN
  END:
END:


INTERLOCK:      disable_vision
  MODE:         inactive
  NEW_MODE:     VISION_ALREADY_DISABLED
  END:

  MODE:         vision_enabled
  NEW_MODE:     inactive
  END:

  MODE:         currents_idling
  MODE:         moving_seed
  NEW_MODE:     CURRENTS_NOT_ZERO
  END:


  MODE:         vision_calibration
  NEW_MODE:     IN_CALIBRATION
  END:

  MODE:         shutdown
  NEW_MODE:     SHUTDOWN
  END:
END:
```

```
INTERLOCK:     emergency_shutdown
  MODE:        inactive
  MODE:        vision_enabled
  MODE:        currents_idling
  MODE:        moving_seed
  MODE:        vision_calibration
  MODE:        shutdown
  NEW_MODE:    shutdown
  END:
END:


INTERLOCK:     shutdown_reset
  MODE:        inactive
  MODE:        vision_enabled
  MODE:        currents_idling
  MODE:        moving_seed
  MODE:        vision_calibration
  NEW_MODE:    SHUTDOWN
  END:

  MODE:        shutdown
  NEW_MODE:    inactive
  END:
END:


MONITOR:       inactive
SCHEDULE:      NO_SCHEDULE
END:

MONITOR:       vision_enabled
SCHEDULE:      NO_SCHEDULE
END:

MONITOR:       currents_idling
SCHEDULE:      COIL_IDLING
END:

MONITOR:       moving_seed
SCHEDULE:      COIL_ON
CONDITION:     runaway(); POSITION_NOT_REGISTERED
END:

MONITOR:       vision_calibration
SCHEDULE:      NO_SCHEDULE
END:

MONITOR:       shutdown
SCHEDULE:      COIL_ON
CONDITION:     servoamps_zero_currents(); SPURIOUS_ON
END:


ERROR_COND:    CALIBRATION_BEGIN_FAILED
  RESPONSE:    announce_error(“CALIBRATION_BEGIN_FAILED”); NO_RESPONSE
END:

ERROR_COND:    MOVE_SEED_BEGIN_FAILED
  RESPONSE:    announce_error(“MOVE_SEED_BEGIN_FAILED”); NO_RESPONSE
END:

ERROR_COND:    NO_RESPONSE
END:
```

```
ERROR_COND:    MUST_BE_INACTIVE
  RESPONSE:    announce_error(“MUST_BE_INACTIVE”); NO_RESPONSE
END:

ERROR_COND:    NOT_IN_VISION_CALIBRATION
  RESPONSE:    announce_error(“NOT_IN_VISION_CALIBRATION”); NO_RESPONSE
END:

ERROR_COND:    ALREADY_MOVING_SEED
  RESPONSE:    announce_error(“ALREADY_MOVING_SEED”); NO_RESPONSE
END:

ERROR_COND:    IN_VISION_CALIBRATION
  RESPONSE:    announce_error(“IN_VISION_CALIBRATION”); NO_RESPONSE
END:

ERROR_COND:    SHUTDOWN
  RESPONSE:    announce_error(“SHUTDOWN”); NO_RESPONSE
END:

ERROR_COND:    CURRENTS_NOT_ZERO
  RESPONSE:    announce_error(“CURRENTS_NOT_ZERO”); NO_RESPONSE
END:

ERROR_COND:    SERVO_AMPS_NOT_INHIBITED
  RESPONSE:    announce_error(“SERVO_AMPS_NOT_INHIBITED”); NO_RESPONSE
END:

ERROR_COND:    NOT_MOVING_SEED
  RESPONSE:    announce_error(“NOT_MOVING_SEED”); NO_RESPONSE
END:

ERROR_COND:    FORCE_ABOVE_THRESHOLD
  RESPONSE:    announce_error(“FORCE_ABOVE_THRESHOLD”); NO_RESPONSE
END:

ERROR_COND:    ALREADY_IDLING
  RESPONSE:    announce_error(“ALREADY_IDLING”); NO_RESPONSE
END:

ERROR_COND:    IN_CALIBRATION
  RESPONSE:    announce_error(“IN_CALIBRATION”); NO_RESPONSE
END:

ERROR_COND:    NOT_IDLING
  RESPONSE:    announce_error(“NOT_IDLING”); NO_RESPONSE
END:

ERROR_COND:    VISION_ALREADY_ENABLED
  RESPONSE:    announce_error(“VISION_ALREADY_ENABLED”); NO_RESPONSE
END:

ERROR_COND:    VISION_ALREADY_DISABLED
  RESPONSE:    announce_error(“VISION_ALREADY_DISABLED”); NO_RESPONSE
END:

ERROR_COND:    SPURIOUS_ON
  RESPONSE:    announce_error(“SPURIOUS_ON”); NO_RESPONSE
END:

ERROR_COND:    POSITION_NOT_REGISTERED
  RESPONSE:    announce_error(“POSITION_NOT_REGISTERED”); NO_RESPONSE
END:

ERROR_COND:    RUNAWAY_SEED
```

```
  RESPONSE:    announce_error(“RUNAWAY_SEED”); NO_RESPONSE
END:

// mss_servoamp error conditions

ERROR_COND:    mss_servoamp::NO_RESPONSE
  RESPONSE:    announce_error(“mss_servoamp::NO_RESPONSE”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::INIT_FAILED
  RESPONSE:    announce_error(“mss_servoamp::INIT_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::RESET_FAILED
  RESPONSE:    announce_error(“mss_servoamp::RESET_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::SET_CURRENT_FAILED
  RESPONSE:    announce_error(“mss_servoamp::SET_CURRENT_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::GET_ACTUAL_CURRENT_FAILED
  RESPONSE:    announce_error(“mss_servoamp::GET_ACTUAL_CURRENT_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::GET_CURRENT_SETTINGS_FAILED
  RESPONSE:    announce_error(“mss_servoamp::GET_CURRENT_SETTINGS_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::INHIBIT_FAILED
  RESPONSE:    announce_error(“mss_servoamp::INHIBIT_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::UNINHIBIT_FAILED
  RESPONSE:    announce_error(“mss_servoamp::UNINHIBIT_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::GET_AMP_STATUS_FAILED
  RESPONSE:    announce_error(“mss_servoamp::GET_AMP_STATUS_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::READ_FAULT_FAILED
  RESPONSE:    announce_error(“mss_servoamp::READ_FAULT_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::EMERGENCY_SET_CURRENT_FAILED
  RESPONSE:    announce_error(“mss_servoamp::EMERGENCY_SET_CURRENT_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::EMERGENCY_DUMP_FAILED
  RESPONSE:    announce_error(“mss_servoamp::EMERGENCY_DUMP_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::DO_SHUTDOWN_FAILED
  RESPONSE:    announce_error(“mss_servoamp::DO_SHUTDOWN_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::FAILURE_RESET_FAILED
  RESPONSE:    announce_error(“mss_servoamp::FAILURE_RESET_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::COILS_INACTIVE
  RESPONSE:    announce_error(“mss_servoamp::COILS_INACTIVE”); NO_RESPONSE
END:
```

```
ERROR_COND:    mss_servoamp::SYSTEM_IS_SHUTDOWN
  RESPONSE:    announce_error(“mss_servoamp::SYSTEM_IS_SHUTDOWN”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::SHOULDNT_BE_OPERATING
  RESPONSE:    announce_error(“mss_servoamp::SHOULDNT_BE_OPERATING”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::NON_ZERO_CURRENT
  RESPONSE:    announce_error(“mss_servoamp::NON_ZERO_CURRENT”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::SERVOAMP_FAULT
  RESPONSE:    announce_error(“mss_servoamp::SERVOAMP_FAULT”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::SERVOAMP_SHUTDOWN
  RESPONSE:    announce_error(“mss_servoamp::SERVOAMP_SHUTDOWN”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::ALREADY_OPERATING
  RESPONSE:    announce_error(“mss_servoamp::ALREADY_OPERATING”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::IN_FAILURE_ADJUST
  RESPONSE:    announce_error(“mss_servoamp::IN_FAILURE_ADJUST”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::INHIBITED
  RESPONSE:    announce_error(“mss_servoamp::INHIBITED”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::CANT_SET_CURRENT
  RESPONSE:    announce_error(“mss_servoamp::CANT_SET_CURRENT”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::CURRENT_CHANGE_TOO_LARGE
  RESPONSE:    announce_error(“mss_servoamp::CURRENT_CHANGE_TOO_LARGE”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::STEP_TIME_TOO_SMALL
  RESPONSE:    announce_error(“mss_servoamp::STEP_TIME_TOO_SMALL”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::NOT_AT_TARGET_CURRENT
  RESPONSE:    announce_error(“mss_servoamp::NOT_AT_TARGET_CURRENT”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::CANT_RESET
  RESPONSE:    announce_error(“mss_servoamp::CANT_RESET”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::SHOULD_BE_OFF
  RESPONSE:    announce_error(“mss_servoamp::SHOULD_BE_OFF”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::SHOULD_BE_ON
  RESPONSE:    announce_error(“mss_servoamp::SHOULD_BE_ON”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::SHOULD_BE_SHUTDOWN
  RESPONSE:    announce_error(“mss_servoamp::SHOULD_BE_SHUTDOWN”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::STATE_ACQUISITION_FAILED
```

```
  RESPONSE:    announce_error(“mss_servoamp::STATE_ACQUISITION_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::NONZERO_CURRENT
  RESPONSE:    announce_error(“mss_servoamp::NONZERO_CURRENT”); NO_RESPONSE
END:


// mss_xray error conditions
ERROR_COND:    mss_xray::SET_CURRENT_FAILED
  RESPONSE:    announce_error(“set_current failed”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::INIT_FAILED
  RESPONSE:    announce_error(“INIT_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::TEST_RESET_FAILED
  RESPONSE:    announce_error(“TEST_RESET_FAILED”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::GET_CURRENT_FAILED
  RESPONSE:    announce_error(“get_current failed”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::SET_VOLTAGE_FAILED
  RESPONSE:    announce_error(“set_voltage failed”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::GET_VOLTAGE_FAILED
  RESPONSE:    announce_error(“get_voltage failed”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::CURRENT_ON_FAILED
  RESPONSE:    announce_error(“current_on failed”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::CURRENT_OFF_FAILED
  RESPONSE:    announce_error(“current_off failed”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::VOLTAGE_ON_FAILED
  RESPONSE:    announce_error(“voltage_on failed”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::VOLTAGE_OFF_FAILED
  RESPONSE:    announce_error(“voltage_off failed”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::RESET_FAILED
  RESPONSE:    announce_error(“reset failed”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::EMERGENCY_DISABLE_FAILED
  RESPONSE:    announce_error(“emergency_disable failed”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::FAILURE_RESET_FAILED
  RESPONSE:    announce_error(“failure_reset failed”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::STATE_ACQUISITION_FAILED
```

```
  RESPONSE:    announce_error(“state acquisition failed”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::SYSTEM_IS_INACTIVE
  RESPONSE:    announce_error(“system is inactive”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::CURRENT_PARAM_TOO_LOW
  RESPONSE:    announce_error(“CURRENT_PARAM_TOO_LOW”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::CURRENT_PARAM_TOO_HIGH
  RESPONSE:    announce_error(“CURRENT_PARAM_TOO_HIGH”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::NEGATIVE_CURRENT_PARAM
  RESPONSE:    announce_error(“NEGATIVE_CURRENT_PARAM”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::SYSTEM_IS_SHUTDOWN
  RESPONSE:    announce_error(“SYSTEM_IS_SHUTDOWN”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::CANT_SET_WITH_CURRENT_ON
  RESPONSE:    announce_error(“CANT_SET_WITH_CURRENT_ON”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::DISABLED
  RESPONSE:    announce_error(“DISABLED”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::NOT_DISABLED
  RESPONSE:    announce_error(“NOT_DISABLED”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::VOLTAGE_PARAM_TOO_LOW
  RESPONSE:    announce_error(“VOLTAGE_PARAM_TOO_LOW”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::VOLTAGE_PARAM_TOO_HIGH
  RESPONSE:    announce_error(“VOLTAGE_PARAM_TOO_HIGH”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::NEGATIVE_VOLTAGE_PARAM
  RESPONSE:    announce_error(“NEGATIVE_VOLTAGE_PARAM”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::VOLTAGE_NOT_ON
  RESPONSE:    announce_error(“VOLTAGE_NOT_ON”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::INSUFF_TIME_IN_STATE
  RESPONSE:    announce_error(“INSUFF_TIME_IN_STATE”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::MAX_TOTAL_ON_TIME_EXCEEDED
  RESPONSE:    announce_error(“MAX_TOTAL_ON_TIME_EXCEEDED”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::VOLTAGE_TOO_LOW
  RESPONSE:    announce_error(“VOLTAGE_TOO_LOW”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::CURRENT_TOO_LOW
  RESPONSE:    announce_error(“CURRENT_TOO_LOW”); NO_RESPONSE
```

```
END:

ERROR_COND:    mss_xray::CURRENT_ALREADY_ON
  RESPONSE:    announce_error(“CURRENT_ALREADY_ON”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::VOLTAGE_ALREADY_ON
  RESPONSE:    announce_error(“VOLTAGE_ALREADY_ON”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::CURRENT_STILL_ON
  RESPONSE:    announce_error(“CURRENT_STILL_ON”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::SYSTEM_MUST_BE_INACTIVE
  RESPONSE:    announce_error(“SYSTEM_MUST_BE_INACTIVE”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::SHOULD_BE_OFF
  RESPONSE:    announce_error(“SHOULD_BE_OFF”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::SHOULD_BE_ON
  RESPONSE:    announce_error(“SHOULD_BE_ON”); NO_RESPONSE
END:

ERROR_COND:    mss_xray::MAX_ON_TIME_EXCEEDED
  RESPONSE:    announce_error(“MAX_ON_TIME_EXCEEDED”); NO_RESPONSE
END:


PROC_DEF: int announce_error(char *message)
{
  cerr << “ERROR for device “ << device_name << “ -- “ <<
             message << “\n” << endl;
  cerr << “Present system mode: “ << present_mode << endl;

  return OK;
}
END:


PROC_DEF: int do_begin_move_seed(coordinate helmet_seed_pos, coordinate UCS_seed_pos)
{
  most_recent_helmet_seed_position = helmet_seed_pos;
  most_recent_UCS_seed_position = UCS_seed_pos;
  position_update_time = get_present_time();
  return OK;
}
END:


PROC_DEF: int do_register_seed_position(coordinate helmet_seed_pos, \
                                     coordinate UCS_seed_pos)
{
  coordinate move_dist;

  most_recent_helmet_seed_position = helmet_seed_pos;
  move_dist = UCS_seed_pos - most_recent_UCS_seed_position;
  most_recent_UCS_seed_position = UCS_seed_pos;
  position_update_time = get_present_time();

  if(move_dist.magnitude() > MAX_UCS_MOVEMENT) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int runaway()
{
  if((get_present_time() - position_update_time) > MIN_UPDATE_PERIOD) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int servoamps_zero_currents()
{
  if(XA_servo->get_coil_current() > LOW_CURRENT ||
     XB_servo->get_coil_current() > LOW_CURRENT ||
     YA_servo->get_coil_current() > LOW_CURRENT ||
     YB_servo->get_coil_current() > LOW_CURRENT ||
     ZA_servo->get_coil_current() > LOW_CURRENT ||
     ZB_servo->get_coil_current() > LOW_CURRENT) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int servoamps_inhibited()
{
  if(XA_servo->get_present_mode() != mss_servoamp::inhibited ||
     XB_servo->get_present_mode() != mss_servoamp::inhibited ||
     YA_servo->get_present_mode() != mss_servoamp::inhibited ||
     YB_servo->get_present_mode() != mss_servoamp::inhibited ||
     ZA_servo->get_present_mode() != mss_servoamp::inhibited ||
     ZB_servo->get_present_mode() != mss_servoamp::inhibited) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int sub_threshold_force(double present_force)
{
  if(present_force > THRESHOLD_FORCE) {
    return ERROR;
  }
  return OK;
}
END:
```

X-ray Source Machine

```
#include “config_macros.h”

DEV_TYPE:      mss_xray

INCLUDE:       general_procs.h
INCLUDE:       kevex_125.h
```

```
INCLUDE:        local_xray.h

CONST:          int MIN_I_ON_VOLTAGE = 50
CONST:          int MIN_I_ON_CURRENT = 250
CONST:          int MIN_VOLTAGE = 0
CONST:          int MAX_VOLTAGE = 125
CONST:          int MIN_X_RAY_CURRENT = 0
CONST:          int MAX_X_RAY_CURRENT = 500
CONST:          double MIN_OFF_TIME = 0.25
CONST:          double MAX_ON_TIME = 0.1
CONST:          double MAX_TOTAL_ON_TIME = 100.0


BASE_CLASS:     private kevex_125
BASE_CLASS:     private local_xray
CONST_PARAM:    char *config_file
CONST_PARAM:    int obj_id
CONST_PARAM:    dispatcher *disp
BASE_CONST:     kevex_125(config_file)
BASE_CONST:     local_xray(obj_id, disp)
INIT_PROC:      initialize_xray(); mss_xray::INIT_FAILED
RESET_PROC:     reset_xray(); mss_xray::TEST_RESET_FAILED

MODE:           Voff_Ioff
MODE:           Von_Ioff
MODE:           Von_Ion
MODE:           disabled
INIT_MODE:      Voff_Ioff

COMMAND:        set_current(short the_current)
COMMAND:        get_current(short &the_current)
COMMAND:        set_voltage(short the_voltage)
COMMAND:        get_voltage(short &the_voltage)
COMMAND:        current_on()
COMMAND:        current_off()
COMMAND:        voltage_on()
COMMAND:        voltage_off()
COMMAND:        get_xray_status()
COMMAND:        reset()
COMMAND:        emergency_disable()
COMMAND:        failure_reset()

ACT_PROC:       set_current: do_set_current(the_current); mss_xray::SET_CURRENT_FAILED
ACT_PROC:       get_current: do_get_current(the_current); mss_xray::GET_CURRENT_FAILED
ACT_PROC:       set_voltage: do_set_voltage(the_voltage); mss_xray::SET_VOLTAGE_FAILED
ACT_PROC:       get_voltage: do_get_voltage(the_voltage); mss_xray::GET_VOLTAGE_FAILED

ACT_PROC:       current_on: do_current_on(); mss_xray::CURRENT_ON_FAILED
ACT_PROC:       current_off: do_current_off(); mss_xray::CURRENT_OFF_FAILED
ACT_PROC:       voltage_on: do_voltage_on(); mss_xray::VOLTAGE_ON_FAILED
ACT_PROC:       voltage_off: do_voltage_off(); mss_xray::VOLTAGE_OFF_FAILED
ACT_PROC:       reset: do_reset(); mss_xray::RESET_FAILED
ACT_PROC:       get_xray_status: do_get_xray_status(); mss_xray::NO_RESPONSE

ACT_PROC:       emergency_disable:                                  do_emergency_disable();
mss_xray::EMERGENCY_DISABLE_FAILED
ACT_PROC:       failure_reset: do_failure_reset(); mss_xray::FAILURE_RESET_FAILED

CNT_PAR:        int current
CNT_PAR:        int voltage
CNT_PAR:        double most_recent_on_time

CNT_PAR_UPD:    set_current: update_current(the_current); mss_xray::NO_RESPONSE
CNT_PAR_UPD:    set_voltage: update_voltage(the_voltage); mss_xray::NO_RESPONSE
CNT_PAR_UPD:    current_on: set_current_on_time(); mss_xray::NO_RESPONSE
```

```
CNT_PAR_UPD:   reset: reset_I_V(); mss_xray::NO_RESPONSE
CNT_PAR_UPD:   emergency_disable: reset_I_V(); mss_xray::NO_RESPONSE


OP_ST_VAR:     int xray_sensor

ST_ACQ_PROC:   acquire_state (state_vector); mss_xray::STATE_ACQUISITION_FAILED


INTERLOCK:             set_current
  MODE:                Voff_Ioff
    PARENT_MODE:       inactive
    NEW_MODE:          mss_xray::SYSTEM_IS_INACTIVE
    END:

    PARENT_MODE:       vision_enabled
    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    NEW_MODE:          Voff_Ioff
    CONDITION:         greater_than_min(the_current,  MIN_X_RAY_CURRENT); \
                                      mss_xray::CURRENT_PARAM_TOO_LOW
    CONDITION:         less_than_max(the_current, MAX_X_RAY_CURRENT); \
                                      mss_xray::CURRENT_PARAM_TOO_HIGH
    END:

    PARENT_MODE:       vision_calibration
    NEW_MODE:          Voff_Ioff
    CONDITION:         greater_than_min(the_current, 0.0); mss_xray::NEGATIVE_CURRENT_PARAM
    CONDITION:         less_than_max(the_current, MAX_X_RAY_CURRENT); \
                                      mss_xray::CURRENT_PARAM_TOO_HIGH
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_xray::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                Von_Ioff
    PARENT_MODE:       inactive
    NEW_MODE:          SYSTEM_IS_INACTIVE
    END:

    PARENT_MODE:       vision_enabled
    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    NEW_MODE:          Von_Ioff
    CONDITION:         greater_than_min(the_current, MIN_X_RAY_CURRENT); \
                                      mss_xray::CURRENT_PARAM_TOO_LOW
    CONDITION:         less_than_max(the_current, MAX_X_RAY_CURRENT); \
                                      mss_xray::CURRENT_PARAM_TOO_HIGH
    END:

    PARENT_MODE:       vision_calibration
    NEW_MODE:          Von_Ioff
    CONDITION:         greater_than_min(the_current, 0.0); mss_xray::NEGATIVE_CURRENT_PARAM
    CONDITION:         less_than_max(the_current, MAX_X_RAY_CURRENT); \
                                      mss_xray::CURRENT_PARAM_TOO_HIGH
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_xray::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                Von_Ion
  NEW_MODE:             mss_xray::CANT_SET_WITH_CURRENT_ON
  END:

  MODE:                 disabled
  NEW_MODE:             mss_xray::DISABLED
  END:
END:


INTERLOCK:              set_voltage
  MODE:                 Voff_Ioff
    PARENT_MODE:        inactive
    NEW_MODE:           mss_xray::SYSTEM_IS_INACTIVE
    END:

    PARENT_MODE:        vision_enabled
    PARENT_MODE:        currents_idling
    PARENT_MODE:        moving_seed
    NEW_MODE:           Voff_Ioff
    CONDITION:          greater_than_min(the_voltage, MIN_VOLTAGE); \
                               mss_xray::VOLTAGE_PARAM_TOO_LOW
    CONDITION:          less_than_max(the_voltage, MAX_VOLTAGE); \
                               mss_xray::VOLTAGE_PARAM_TOO_HIGH
    END:

    PARENT_MODE:        vision_calibration
    NEW_MODE:           Voff_Ioff
    CONDITION:          greater_than_min(the_voltage, 0.0); mss_xray::NEGATIVE_VOLTAGE_PARAM
    CONDITION:          less_than_max(the_voltage,MAX_VOLTAGE);\
                               mss_xray::VOLTAGE_PARAM_TOO_HIGH
    END:

    PARENT_MODE:        shutdown
    NEW_MODE:           mss_xray::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                 Von_Ioff
    PARENT_MODE:        inactive
    NEW_MODE:           mss_xray::SYSTEM_IS_INACTIVE
    END:

    PARENT_MODE:        vision_enabled
    PARENT_MODE:        currents_idling
    PARENT_MODE:        moving_seed
    NEW_MODE:           Von_Ioff
    CONDITION:          greater_than_min(the_voltage, MIN_VOLTAGE);\
                               mss_xray::VOLTAGE_PARAM_TOO_LOW
    CONDITION:          less_than_max(the_voltage, MAX_VOLTAGE); \
                               mss_xray::VOLTAGE_PARAM_TOO_HIGH
    END:

    PARENT_MODE:        vision_calibration
    NEW_MODE:           Von_Ioff
    CONDITION:          greater_than_min(the_voltage, 0.0); mss_xray::NEGATIVE_VOLTAGE_PARAM
    CONDITION:          less_than_max(the_voltage, MAX_VOLTAGE); \
                               mss_xray::VOLTAGE_PARAM_TOO_HIGH
    END:

    PARENT_MODE:        shutdown
    NEW_MODE:           mss_xray::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                 Von_Ion
  NEW_MODE:             mss_xray::CANT_SET_WITH_CURRENT_ON
  END:

  MODE:                 disabled
  NEW_MODE:             mss_xray::DISABLED
  END:
END:


INTERLOCK:              current_on
  MODE:                 Voff_Ioff
  NEW_MODE:             mss_xray::VOLTAGE_NOT_ON
  END:

  MODE:                 Von_Ioff
    PARENT_MODE:        inactive
    NEW_MODE:           mss_xray::SYSTEM_IS_INACTIVE
    END:

    PARENT_MODE:        vision_enabled
    PARENT_MODE:        currents_idling
    PARENT_MODE:        moving_seed
    NEW_MODE:           Von_Ion
    CONDITION:          check_off_time(MIN_OFF_TIME); mss_xray::INSUFF_TIME_IN_STATE

    CONDITION:          less_than_max (total_time_in_state(Von_Ion),MAX_TOTAL_ON_TIME);\
                               mss_xray::MAX_TOTAL_ON_TIME_EXCEEDED
    CONDITION:          greater_than_min(voltage, MIN_I_ON_VOLTAGE); \
                               mss_xray::VOLTAGE_TOO_LOW
    CONDITION:          greater_than_min(current, MIN_I_ON_CURRENT); \
                                mss_xray::CURRENT_TOO_LOW
    END:

    PARENT_MODE:        vision_calibration
    NEW_MODE:           Von_Ion
    CONDITION:          greater_than_min(voltage, MIN_I_ON_VOLTAGE); \
                               mss_xray::VOLTAGE_TOO_LOW
    CONDITION:          greater_than_min(current, MIN_I_ON_CURRENT); \
                               mss_xray::CURRENT_TOO_LOW
    END:

    PARENT_MODE:        shutdown
    NEW_MODE:           mss_xray::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                 Von_Ion
  NEW_MODE:             mss_xray::CURRENT_ALREADY_ON
  END:

  MODE:                 disabled
  NEW_MODE:             mss_xray::DISABLED
  END:
END:


INTERLOCK:              current_off
  MODE:                 Voff_Ioff
    PARENT_MODE:        inactive
    NEW_MODE:           mss_xray::SYSTEM_IS_INACTIVE
    END:

    PARENT_MODE:        vision_enabled
    PARENT_MODE:        currents_idling
    PARENT_MODE:        moving_seed
    PARENT_MODE:       vision_calibration
    NEW_MODE:          Voff_Ioff
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_xray::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                Von_Ioff
    PARENT_MODE:       inactive
    NEW_MODE:          mss_xray::SYSTEM_IS_INACTIVE
    END:

    PARENT_MODE:       vision_enabled
    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    PARENT_MODE:       vision_calibration
    NEW_MODE:          Von_Ioff
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_xray::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                Von_Ion
    PARENT_MODE:       inactive
    NEW_MODE:          mss_xray::SYSTEM_IS_INACTIVE
    END:

    PARENT_MODE:       vision_enabled
    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    PARENT_MODE:       vision_calibration
    NEW_MODE:          Von_Ioff
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_xray::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                disabled
  NEW_MODE:            mss_xray::DISABLED
  END:
END:


INTERLOCK:             voltage_on
  MODE:                Voff_Ioff
    PARENT_MODE:       inactive
    NEW_MODE:          mss_xray::SYSTEM_IS_INACTIVE
    END:

    PARENT_MODE:       vision_enabled
    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    NEW_MODE:          Von_Ioff
    END:

    PARENT_MODE:       vision_calibration
    NEW_MODE:          Von_Ioff
    END:
```

```
    PARENT_MODE:       shutdown
    NEW_MODE:          mss_xray::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                Von_Ioff
  NEW_MODE:            mss_xray::VOLTAGE_ALREADY_ON
  END:

  MODE:                Von_Ion
  NEW_MODE:            mss_xray::VOLTAGE_ALREADY_ON
  END:

  MODE:                disabled
  NEW_MODE:            mss_xray::DISABLED
  END:
END:


INTERLOCK:             voltage_off
  MODE:                Voff_Ioff
  MODE:                Von_Ioff
    PARENT_MODE:       inactive
    NEW_MODE:          mss_xray::SYSTEM_IS_INACTIVE
    END:

    PARENT_MODE:       vision_enabled
    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    PARENT_MODE:       vision_calibration
    NEW_MODE:          Voff_Ioff
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_xray::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                Von_Ion
  NEW_MODE:            mss_xray::CURRENT_STILL_ON
  END:

  MODE:                disabled
  NEW_MODE:            mss_xray::DISABLED
  END:

END:


INTERLOCK:             reset
  MODE:                Voff_Ioff
  MODE:                Von_Ioff
  MODE:                Von_Ion
    PARENT_MODE:       inactive
    PARENT_MODE:       vision_enabled
    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    PARENT_MODE:       vision_calibration
    NEW_MODE:          Voff_Ioff
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_xray::SYSTEM_IS_SHUTDOWN
    END:
  END:
```

```
  MODE:                 disabled
  NEW_MODE:             mss_xray::DISABLED
  END:
END:


INTERLOCK:              emergency_disable
  MODE:                 Voff_Ioff
  MODE:                 Von_Ioff
  MODE:                 Von_Ion
  MODE:                 disabled
  NEW_MODE:             disabled
  END:
END:


INTERLOCK:              failure_reset
  MODE:                 disabled
    PARENT_MODE:        inactive
    NEW_MODE:           Voff_Ioff
    CONDITION:          off_state_consistency(present_op_state); mss_xray::XRAY_STILL_ON
    END:
    PARENT_MODE:        vision_enabled
    PARENT_MODE:        currents_idling
    PARENT_MODE:        moving_seed
    PARENT_MODE:        vision_calibration
    PARENT_MODE:        shutdown
    NEW_MODE:           mss_xray::SYSTEM_MUST_BE_INACTIVE
    END:

  END:
  MODE:                 Voff_Ioff
  MODE:                 Von_Ioff
  MODE:                 Von_Ion
  NEW_MODE:             mss_xray::NOT_DISABLED
  END:
END:


// Device monitoring policies
MONITOR:       Voff_Ioff
  SCHEDULE:    XRAY_OFF
  CONDITION:   off_state_consistency(present_op_state); mss_xray::SHOULD_BE_OFF
END:

MONITOR:       Von_Ioff
  SCHEDULE:    XRAY_OFF
  CONDITION:   off_state_consistency(present_op_state); mss_xray::SHOULD_BE_OFF
END:

MONITOR:       Von_Ion
  SCHEDULE:    XRAY_ON
  CONDITION:   on_state_consistency(present_op_state); mss_xray::SHOULD_BE_ON
  CONDITION:   less_than_max (total_time_in_state(Von_Ion), MAX_TOTAL_ON_TIME);\
                         mss_xray::MAX_TOTAL_ON_TIME_EXCEEDED
  CONDITION:   less_than_max (time_in_state, MAX_ON_TIME); mss_xray::MAX_ON_TIME_EXCEEDED
END:

MONITOR:       disabled
  SCHEDULE:    XRAY_OFF
  CONDITION:   off_state_consistency(present_op_state); mss_xray::SHOULD_BE_OFF
END:
```

```
ERROR_COND:    mss_xray::SET_CURRENT_FAILED
  RESPONSE:    announce_error(“set_current failed”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::INIT_FAILED
  RESPONSE:    announce_error(“INIT_FAILED”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::TEST_RESET_FAILED
  RESPONSE:    announce_error(“TEST_RESET_FAILED”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::GET_CURRENT_FAILED
RESPONSE:      announce_error(“get_current failed”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::SET_VOLTAGE_FAILED
RESPONSE:      announce_error(“set_voltage failed”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::GET_VOLTAGE_FAILED
RESPONSE:      announce_error(“get_voltage failed”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::CURRENT_ON_FAILED
RESPONSE:      announce_error(“current_on failed”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::CURRENT_OFF_FAILED
RESPONSE:      announce_error(“current_off failed”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::VOLTAGE_ON_FAILED
RESPONSE:      announce_error(“voltage_on failed”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::VOLTAGE_OFF_FAILED
  RESPONSE:    announce_error(“voltage_off failed”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::RESET_FAILED
  RESPONSE:    announce_error(“reset failed”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::EMERGENCY_DISABLE_FAILED
  RESPONSE:    announce_error(“emergency_disable failed”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::FAILURE_RESET_FAILED
  RESPONSE:    announce_error(“failure_reset failed”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::STATE_ACQUISITION_FAILED
  RESPONSE:    announce_error(“state acquisition failed”); mss_xray::NO_RESPONSE
END:
ERROR_COND:    mss_xray::XRAY_STILL_ON
  RESPONSE:    announce_error(“system is inactive”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::SYSTEM_IS_INACTIVE
  RESPONSE:    announce_error(“system is inactive”); mss_xray::NO_RESPONSE
END:
```

```
ERROR_COND:    mss_xray::CURRENT_PARAM_TOO_LOW
  RESPONSE:    announce_error(“CURRENT_PARAM_TOO_LOW”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::CURRENT_PARAM_TOO_HIGH
  RESPONSE:    announce_error(“CURRENT_PARAM_TOO_HIGH”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::NEGATIVE_CURRENT_PARAM
  RESPONSE:    announce_error(“NEGATIVE_CURRENT_PARAM”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::SYSTEM_IS_SHUTDOWN
  RESPONSE:    announce_error(“SYSTEM_IS_SHUTDOWN”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::CANT_SET_WITH_CURRENT_ON
  RESPONSE:    announce_error(“CANT_SET_WITH_CURRENT_ON”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::DISABLED
  RESPONSE:    announce_error(“DISABLED”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::NOT_DISABLED
  RESPONSE:    announce_error(“NOT_DISABLED”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::VOLTAGE_PARAM_TOO_LOW
  RESPONSE:    announce_error(“VOLTAGE_PARAM_TOO_LOW”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::VOLTAGE_PARAM_TOO_HIGH
  RESPONSE:    announce_error(“VOLTAGE_PARAM_TOO_HIGH”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::NEGATIVE_VOLTAGE_PARAM
  RESPONSE:    announce_error(“NEGATIVE_VOLTAGE_PARAM”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::VOLTAGE_NOT_ON
  RESPONSE:    announce_error(“VOLTAGE_NOT_ON”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::INSUFF_TIME_IN_STATE
  RESPONSE:    announce_error(“INSUFF_TIME_IN_STATE”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::MAX_TOTAL_ON_TIME_EXCEEDED
  RESPONSE:    announce_error(“MAX_TOTAL_ON_TIME_EXCEEDED”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::VOLTAGE_TOO_LOW
  RESPONSE:    announce_error(“VOLTAGE_TOO_LOW”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::CURRENT_TOO_LOW
  RESPONSE:    announce_error(“CURRENT_TOO_LOW”); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::CURRENT_ALREADY_ON
  RESPONSE:    announce_error(“CURRENT_ALREADY_ON”); mss_xray::NO_RESPONSE
END:
```

```
ERROR_COND:    mss_xray::VOLTAGE_ALREADY_ON
  RESPONSE:    announce_error("VOLTAGE_ALREADY_ON"); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::CURRENT_STILL_ON
  RESPONSE:    announce_error("CURRENT_STILL_ON"); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::SYSTEM_MUST_BE_INACTIVE
  RESPONSE:    announce_error("SYSTEM_MUST_BE_INACTIVE"); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::SHOULD_BE_OFF
  RESPONSE:    announce_error("SHOULD_BE_OFF"); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::SHOULD_BE_ON
  RESPONSE:    announce_error("SHOULD_BE_ON"); mss_xray::NO_RESPONSE
END:

ERROR_COND:    mss_xray::MAX_ON_TIME_EXCEEDED
  RESPONSE:    announce_error("MAX_ON_TIME_EXCEEDED"); mss_xray::NO_RESPONSE
END:


PROC_DEF: int acquire_state (state_vector &v)
{
  v.xray_sensor = read_sensor();
  if(v.xray_sensor == kevex_125::SENSOR_ERROR) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int initialize_xray()
{
  most_recent_on_time = get_present_time();
  current = 0;
  voltage = 0;
  return OK;
}
END:


PROC_DEF: int reset_xray()
{
  most_recent_on_time = get_present_time();
  current = 0;
  voltage = 0;
  return OK;
}
END:


PROC_DEF: int on_state_consistency(state_vector &v)
{
  if(v.xray_sensor != kevex_125::XRAY_ON) {
    return ERROR;
  }
  return OK;
}
END:
```

```
PROC_DEF: int off_state_consistency(state_vector &v)
{
  if(v.xray_sensor != kevex_125::XRAY_OFF) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int set_current_on_time()
{
  most_recent_on_time = get_present_time();
  return OK;
}
END:


PROC_DEF: int check_off_time(double min_off_time)
{
  double present = get_present_time();

  if((present - most_recent_on_time) < min_off_time) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int update_current(int the_current)
{
  current = the_current;
  return OK;
}
END:


PROC_DEF: int update_voltage(int the_voltage)
{
  voltage = the_voltage;
  return OK;
}
END:


PROC_DEF: int reset_I_V()
{
  current = 0;
  voltage = 0;
  return OK;
}
END:


PROC_DEF: int do_failure_reset()
{
  kevex_125::reset();
  return OK;
}
END:


PROC_DEF: int announce_error(char *message)
```

```
{
  cerr << “ERROR for device “ << device_name << “ -- “ <<
               message << “\n” << endl;
  cerr << “Present device mode: “ << present_mode << endl;

  return OK;
}
END:


PROC_DEF: int max_total_on_time_exceeded()
{
  announce_error(“Maximum total on time exceeded - disabled”);

  emergency_disable();
  return OK;
}
END:


PROC_DEF: int do_set_current(short the_current)
{
  if(kevex_125::set_current(the_current) != XRAY_OK) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int do_get_current(short &the_current)
{
  if(kevex_125::get_current(the_current) != XRAY_OK) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int do_set_voltage(short the_voltage)
{
  if(kevex_125::set_voltage(the_voltage) != XRAY_OK) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int do_get_voltage(short &the_voltage)
{
  if(kevex_125::get_voltage(the_voltage) != XRAY_OK) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int do_current_on()
{
  if(kevex_125::current_on() != XRAY_OK) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int do_current_off()
{
  if(kevex_125::current_off() != XRAY_OK) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int do_voltage_on()
{
  if(kevex_125::voltage_on() != XRAY_OK) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int do_voltage_off()
{
  if(kevex_125::voltage_off() != XRAY_OK) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int do_reset()
{
  if(kevex_125::reset() != XRAY_OK) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int do_get_xray_status()
{
  if(kevex_125::get_xray_status() != XRAY_OK) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int do_emergency_disable()
{
  kevex_125::current_off();
  kevex_125::voltage_off();
  // Pull the plug too

  return OK;
}
END:
```

Servoamplifier Machine

```
#include “config_macros.h”

DEV_TYPE:      mss_servoamp

INCLUDE:       general_procs.h
INCLUDE:       servoamp.h
INCLUDE:       math.h
INCLUDE:       local_servoamp.h

BASE_CLASS:    private servoamp
BASE_CLASS:    private local_servoamp
INIT_PROC:     initialize_servoamp(the_inductance); mss_servoamp::INIT_FAILED
RESET_PROC:    reset_servoamp(); mss_servoamp::RESET_FAILED

OP_ST_VAR:     double coil_current
OP_ST_VAR:     double sensor_current
OP_ST_VAR:     double set_current
OP_ST_VAR:     unsigned set_dac_value
OP_ST_VAR:     int status
OP_ST_VAR:     int fault_line
DEV_DECL:      double inductance
CONST_PARAM:   double the_inductance
CONST_PARAM:   char *config_file
CONST_PARAM:   int obj_id
CONST_PARAM:   dispatcher *disp
BASE_CONST:    servoamp(config_file)
BASE_CONST:    local_servoamp(obj_id, disp)

ST_ACQ_PROC:   acquire_state(state_vector); mss_servoamp::STATE_ACQUISITION_FAILED

COMMAND:       set_current(double I)
COMMAND:       get_actual_current(double &I)
COMMAND:       get_current_settings(double &I, unsigned &dac_value)
COMMAND:       inhibit()
COMMAND:       uninhibit()
COMMAND:       get_amp_status()
COMMAND:       read_fault()
COMMAND:       emergency_set_current(double I)
COMMAND:       emergency_dump()
COMMAND:       shutdown()
COMMAND:       failure_reset()

ACT_PROC:      set_current : do_set_current(I); mss_servoamp::SET_CURRENT_FAILED
ACT_PROC:      get_actual_current : do_get_actual_current(I); \
                                      mss_servoamp::GET_ACTUAL_CURRENT_FAILED
ACT_PROC:      get_current_settings : do_get_current_settings(I, dac_value); \
                                      mss_servoamp::GET_CURRENT_SETTINGS_FAILED
ACT_PROC:      inhibit : do_inhibit(); mss_servoamp::INHIBIT_FAILED
ACT_PROC:      uninhibit : do_uninhibit(); mss_servoamp::UNINHIBIT_FAILED
ACT_PROC:      get_amp_status : do_get_amp_status(); mss_servoamp::GET_AMP_STATUS_FAILED
ACT_PROC:      read_fault : do_read_fault(); mss_servoamp::READ_FAULT_FAILED
ACT_PROC:      emergency_set_current : do_emergency_set_current(I); \
                                      mss_servoamp::EMERGENCY_SET_CURRENT_FAILED
ACT_PROC:      emergency_dump : do_emergency_dump(); mss_servoamp::EMERGENCY_DUMP_FAILED
ACT_PROC:      shutdown : do_shutdown(); mss_servoamp::DO_SHUTDOWN_FAILED
ACT_PROC:      failure_reset : do_failure_reset(); mss_servoamp::FAILURE_RESET_FAILED

CNT_PAR:       double target_current

CNT_PAR_UPD:   set_current: update_target_current(I); mss_servoamp::NO_RESPONSE
CNT_PAR_UPD:   inhibit: update_target_current(0.0); mss_servoamp::NO_RESPONSE
```

```
CNT_PAR_UPD:   emergency_dump: update_target_current(0.0); mss_servoamp::NO_RESPONSE
CNT_PAR_UPD:   emergency_set_current: update_target_current(I); mss_servoamp::NO_RESPONSE
CNT_PAR_UPD:   failure_reset: update_target_current(0.0); mss_servoamp::NO_RESPONSE
CNT_PAR_UPD:   shutdown: update_target_current(0.0); mss_servoamp::NO_RESPONSE


MODE:          inhibited
MODE:          operating
MODE:          failure_adjust
MODE:          shutdown_mode
INIT_MODE:     inhibited

CONST:         double LOW_CURRENT = 5.0
CONST:         double EXPECTED_CURRENT_TOLERANCE = 5.0
CONST:         double MAX_CURRENT_CHANGE = 20.0
CONST:         double ENERGY_CHANGE_RATE = 6000.0
CONST:         double MAX_INHIBIT_CURRENT = 2.0
CONST:         double CURRENT_CHARGE_RATE = 10.0
CONST:         double TARGET_CURRENT_TOLERANCE = 5.0


// State-command policies
INTERLOCK:             inhibit

  MODE:                inhibited
    PARENT_MODE:       inactive
    PARENT_MODE:       vision_enabled
    PARENT_MODE:       vision_calibration
    NEW_MODE:          mss_servoamp::COILS_INACTIVE
    END:

    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    NEW_MODE:          inhibited
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_servoamp::SYSTEM_IS_SHUTDOWN
    END:
  END:


  MODE:                operating
    PARENT_MODE:       inactive
    PARENT_MODE:       vision_enabled
    PARENT_MODE:       vision_calibration
    NEW_MODE:          mss_servoamp::SHOULDNT_BE_OPERATING
    END:

    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    NEW_MODE:          inhibited
    CONDITION:         zero_current(); mss_servoamp::NON_ZERO_CURRENT
    CONDITION:         servo_faults (); mss_servoamp::SERVOAMP_FAULT
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_servoamp::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                failure_adjust
  NEW_MODE:            mss_servoamp::IN_FAILURE_ADJUST
  END:
```

```
  MODE:                shutdown_mode
  NEW_MODE:            mss_servoamp::SERVOAMP_SHUTDOWN
  END:
END:


INTERLOCK:             uninhibit

  MODE:                inhibited
    PARENT_MODE:       inactive
    PARENT_MODE:       vision_enabled
    PARENT_MODE:       vision_calibration
    NEW_MODE:          mss_servoamp::COILS_INACTIVE
    END:

    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    NEW_MODE:          operating
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_servoamp::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                operating
  NEW_MODE:            mss_servoamp::ALREADY_OPERATING
  END:

  MODE:                failure_adjust
  NEW_MODE:            mss_servoamp::IN_FAILURE_ADJUST
  END:

  MODE:                shutdown_mode
  NEW_MODE:            mss_servoamp::SERVOAMP_SHUTDOWN
  END:
END:


INTERLOCK:             set_current
  MODE:                inhibited
  NEW_MODE:            mss_servoamp::INHIBITED
  END:

  MODE:                operating
    PARENT_MODE:       inactive
    PARENT_MODE:       vision_enabled
    PARENT_MODE:       vision_calibration
    NEW_MODE:          mss_servoamp::SHOULDNT_BE_OPERATING
    END:

    PARENT_MODE:       currents_idling
    NEW_MODE:          operating
    CONDITION:         less_than_max(I, target_current); mss_servoamp::CANT_SET_CURRENT
    END:

    PARENT_MODE:       moving_seed
    NEW_MODE:          operating
    CONDITION:         less_than_max(I, MAX_CURRENT); mss_servoamp::CANT_SET_CURRENT
    CONDITION:         greater_than_min(I, MIN_CURRENT); mss_servoamp::CANT_SET_CURRENT
    CONDITION:         reasonable_current_param(I); mss_servoamp::CURRENT_CHANGE_TOO_LARGE
    CONDITION:         at_target_current(); mss_servoamp::NOT_AT_TARGET_CURRENT
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_servoamp::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                failure_adjust
  NEW_MODE:            mss_servoamp::IN_FAILURE_ADJUST
  END:

  MODE:                shutdown_mode
  NEW_MODE:            mss_servoamp::SERVOAMP_SHUTDOWN
  END:
END:


INTERLOCK:             get_actual_current
  MODE:                inhibited
    PARENT_MODE:       inactive
    PARENT_MODE:       vision_enabled
    PARENT_MODE:       vision_calibration
    NEW_MODE:          mss_servoamp::COILS_INACTIVE
    END:

    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    NEW_MODE:          inhibited
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_servoamp::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                operating
    PARENT_MODE:       inactive
    PARENT_MODE:       vision_enabled
    PARENT_MODE:       vision_calibration
    NEW_MODE:          mss_servoamp::SHOULDNT_BE_OPERATING
    END:

    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    NEW_MODE:          operating
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_servoamp::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                failure_adjust
  NEW_MODE:            failure_adjust
  END:

  MODE:                shutdown_mode
  NEW_MODE:            mss_servoamp::SERVOAMP_SHUTDOWN
  END:
END:


INTERLOCK:             get_current_settings
  MODE:                inhibited
    PARENT_MODE:       inactive
    PARENT_MODE:       vision_enabled
    PARENT_MODE:       vision_calibration
    NEW_MODE:          mss_servoamp::COILS_INACTIVE
    END:

    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    NEW_MODE:          inhibited
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_servoamp::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                operating
    PARENT_MODE:       inactive
    PARENT_MODE:       vision_enabled
    PARENT_MODE:       vision_calibration
    NEW_MODE:          mss_servoamp::SHOULDNT_BE_OPERATING
    END:

    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    NEW_MODE:          operating
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_servoamp::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                failure_adjust
  NEW_MODE:            failure_adjust
  END:

  MODE:                shutdown_mode
  NEW_MODE:            mss_servoamp::SERVOAMP_SHUTDOWN
  END:
END:


INTERLOCK:             get_amp_status
  MODE:                inhibited
    PARENT_MODE:       inactive
    PARENT_MODE:       vision_enabled
    PARENT_MODE:       vision_calibration
    NEW_MODE:          mss_servoamp::COILS_INACTIVE
    END:

    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    NEW_MODE:          inhibited
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_servoamp::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                operating
    PARENT_MODE:       inactive
    PARENT_MODE:       vision_enabled
    PARENT_MODE:       vision_calibration
    NEW_MODE:          mss_servoamp::SHOULDNT_BE_OPERATING
    END:

    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    NEW_MODE:          operating
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_servoamp::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                failure_adjust
  NEW_MODE:            failure_adjust
  END:

  MODE:                shutdown_mode
  NEW_MODE:            mss_servoamp::SERVOAMP_SHUTDOWN
  END:
END:


INTERLOCK:             read_fault
  MODE:                inhibited
    PARENT_MODE:       inactive
    PARENT_MODE:       vision_enabled
    PARENT_MODE:       vision_calibration
    NEW_MODE:          mss_servoamp::COILS_INACTIVE
    END:

    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    NEW_MODE:          inhibited
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_servoamp::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                operating
    PARENT_MODE:       inactive
    PARENT_MODE:       vision_enabled
    PARENT_MODE:       vision_calibration
    NEW_MODE:          mss_servoamp::SHOULDNT_BE_OPERATING
    END:

    PARENT_MODE:       currents_idling
    PARENT_MODE:       moving_seed
    NEW_MODE:          operating
    END:

    PARENT_MODE:       shutdown
    NEW_MODE:          mss_servoamp::SYSTEM_IS_SHUTDOWN
    END:
  END:

  MODE:                failure_adjust
  NEW_MODE:            failure_adjust
  END:

  MODE:                shutdown_mode
  NEW_MODE:            mss_servoamp::SERVOAMP_SHUTDOWN
  END:
END:


INTERLOCK:             emergency_set_current
```

```
  MODE:                 inhibited
  NEW_MODE:             mss_servoamp::INHIBITED
  END:

  MODE:                 operating
  NEW_MODE:             failure_adjust
  END:

  MODE:                 failure_adjust
  NEW_MODE:             failure_adjust
  END:

  MODE:                 shutdown_mode
  NEW_MODE:             mss_servoamp::SERVOAMP_SHUTDOWN
  END:
END:


INTERLOCK:              emergency_dump
  MODE:                 inhibited
  NEW_MODE:             mss_servoamp::INHIBITED
  END:

  MODE:                 operating
  NEW_MODE:             failure_adjust
  END:

  MODE:                 failure_adjust
  NEW_MODE:             failure_adjust
  END:

  MODE:                 shutdown_mode
  NEW_MODE:             mss_servoamp::SERVOAMP_SHUTDOWN
  END:
END:


INTERLOCK:              shutdown
  MODE:                 inhibited
  NEW_MODE:             mss_servoamp::INHIBITED
  END:

  MODE:                 operating
  NEW_MODE:             failure_adjust
  CONDITION:            zero_current(); mss_servoamp::NONZERO_CURRENT
  END:

  MODE:                 failure_adjust
  NEW_MODE:             failure_adjust
  CONDITION:            zero_current(); mss_servoamp::NONZERO_CURRENT
  END:

  MODE:                 shutdown_mode
  NEW_MODE:             mss_servoamp::SERVOAMP_SHUTDOWN
  END:
END:


INTERLOCK:              failure_reset
  MODE:                 inhibited
  NEW_MODE:             mss_servoamp::INHIBITED
  END:

  MODE:                 operating
  NEW_MODE:             mss_servoamp::CANT_RESET
  END:

  MODE:               failure_adjust
  NEW_MODE:           inhibited
  CONDITION:          zero_current(); mss_servoamp::NONZERO_CURRENT
  END:

  MODE:               shutdown_mode
  NEW_MODE:           inhibited
  END:
END:


// Device error detection policies
MONITOR:       inhibited
  SCHEDULE:    COIL_OFF
  CONDITION:   inhibit_state_consistency(present_op_state); mss_servoamp::SHOULD_BE_OFF
END:

MONITOR:       operating
  SCHEDULE:    COIL_ON
  CONDITION:   operating_state_consistency(present_op_state); mss_servoamp::SHOULD_BE_ON
END:

MONITOR:       failure_adjust
  SCHEDULE:    COIL_ON
  CONDITION:   operating_state_consistency(present_op_state); mss_servoamp::SHOULD_BE_ON
END:

MONITOR:       shutdown_mode
  SCHEDULE:    COIL_OFF
  CONDITION:   inhibit_state_consistency(present_op_state); \
                      mss_servoamp::SHOULD_BE_SHUTDOWN
END:


// Error conditions
ERROR_COND:    mss_servoamp::NO_RESPONSE
  RESPONSE:    announce_error(“mss_servoamp::NO_RESPONSE”); mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::INIT_FAILED
  RESPONSE:    announce_error(“mss_servoamp::INIT_FAILED”); mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::RESET_FAILED
  RESPONSE:    announce_error(“mss_servoamp::RESET_FAILED”); mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::SET_CURRENT_FAILED
  RESPONSE:    announce_error(“mss_servoamp::SET_CURRENT_FAILED”);
mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::GET_ACTUAL_CURRENT_FAILED
  RESPONSE:    announce_error(“mss_servoamp::GET_ACTUAL_CURRENT_FAILED”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::GET_CURRENT_SETTINGS_FAILED
  RESPONSE:    announce_error(“mss_servoamp::GET_CURRENT_SETTINGS_FAILED”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::INHIBIT_FAILED
```

```
  RESPONSE:    announce_error(“mss_servoamp::INHIBIT_FAILED”); mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::UNINHIBIT_FAILED
  RESPONSE:    announce_error(“mss_servoamp::UNINHIBIT_FAILED”); mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::GET_AMP_STATUS_FAILED
  RESPONSE:    announce_error(“mss_servoamp::GET_AMP_STATUS_FAILED”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::READ_FAULT_FAILED
  RESPONSE:    announce_error(“mss_servoamp::READ_FAULT_FAILED”);
mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::EMERGENCY_SET_CURRENT_FAILED
  RESPONSE:    announce_error(“mss_servoamp::EMERGENCY_SET_CURRENT_FAILED”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::EMERGENCY_DUMP_FAILED
  RESPONSE:    announce_error(“mss_servoamp::EMERGENCY_DUMP_FAILED”);\
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::DO_SHUTDOWN_FAILED
  RESPONSE:    announce_error(“mss_servoamp::DO_SHUTDOWN_FAILED”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::FAILURE_RESET_FAILED
  RESPONSE:    announce_error(“mss_servoamp::FAILURE_RESET_FAILED”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::COILS_INACTIVE
  RESPONSE:    announce_error(“mss_servoamp::COILS_INACTIVE”); mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::SYSTEM_IS_SHUTDOWN
  RESPONSE:    announce_error(“mss_servoamp::SYSTEM_IS_SHUTDOWN”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::SHOULDNT_BE_OPERATING
  RESPONSE: announce_error(“mss_servoamp::SHOULDNT_BE_OPERATING”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::NON_ZERO_CURRENT
  RESPONSE:    announce_error(“mss_servoamp::NON_ZERO_CURRENT”); mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::SERVOAMP_FAULT
  RESPONSE:    announce_error(“mss_servoamp::SERVOAMP_FAULT”); mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::SERVOAMP_SHUTDOWN
  RESPONSE:    announce_error(“mss_servoamp::SERVOAMP_SHUTDOWN”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::ALREADY_OPERATING
```

```
  RESPONSE:    announce_error(“mss_servoamp::ALREADY_OPERATING”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::IN_FAILURE_ADJUST
  RESPONSE:    announce_error(“mss_servoamp::IN_FAILURE_ADJUST”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::INHIBITED
  RESPONSE:    announce_error(“mss_servoamp::INHIBITED”); mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::CANT_SET_CURRENT
  RESPONSE:    announce_error(“mss_servoamp::CANT_SET_CURRENT”); mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::CURRENT_CHANGE_TOO_LARGE
  RESPONSE:    announce_error(“mss_servoamp::CURRENT_CHANGE_TOO_LARGE”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::STEP_TIME_TOO_SMALL
  RESPONSE:    announce_error(“mss_servoamp::STEP_TIME_TOO_SMALL”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::NOT_AT_TARGET_CURRENT
  RESPONSE:    announce_error(“mss_servoamp::NOT_AT_TARGET_CURRENT”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::CANT_RESET
  RESPONSE:    announce_error(“mss_servoamp::CANT_RESET”); mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::SHOULD_BE_OFF
  RESPONSE:    announce_error(“mss_servoamp::SHOULD_BE_OFF”); mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::SHOULD_BE_ON
  RESPONSE:    announce_error(“mss_servoamp::SHOULD_BE_ON”); mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::SHOULD_BE_SHUTDOWN
  RESPONSE:    announce_error(“mss_servoamp::SHOULD_BE_SHUTDOWN”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::STATE_ACQUISITION_FAILED
  RESPONSE:    announce_error(“mss_servoamp::STATE_ACQUISITION_FAILED”); \
                              mss_servoamp::NO_RESPONSE
END:

ERROR_COND:    mss_servoamp::NONZERO_CURRENT
  RESPONSE:    announce_error(“mss_servoamp::NONZERO_CURRENT”); mss_servoamp::NO_RESPONSE
END:


PROC_DEF: int announce_error(char *message)
{
  cerr << “ERROR for device “ << device_name << “ -- “ <<
               message << “\n” << endl;
  cerr << “Present device mode: “ << present_mode << endl;
  return OK;
```

```
}
END:


PROC_DEF: int acquire_state(state_vector &v)
{
  servoamp::get_actual_current(v.coil_current);

  servoamp::get_current_settings(v.set_current, v.set_dac_value);

  v.status = servoamp::get_amp_status();
  v.fault_line = servoamp::read_fault();

  return OK;
}
END:


PROC_DEF: int inhibit_state_consistency(const state_vector &v)
{
  if(fabs(v.coil_current) > MAX_INHIBIT_CURRENT) {
    cout << v.coil_current << endl;
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int operating_state_consistency (state_vector &v)
{
  if(fabs(v.coil_current - target_current) < MAX_CURRENT_CHANGE &&
     fabs(v.coil_current - expected_current()) < EXPECTED_CURRENT_TOLERANCE) {
      return OK;
  }
  cout << v.coil_current << “  “ << target_current <<
    “  “ << expected_current() << endl;
  return ERROR;
}
END:


PROC_DEF: int failure_adjust_state_consistency (state_vector &v)
{
  if(fabs(v.coil_current - target_current) < MAX_CURRENT_CHANGE &&
     fabs(v.coil_current - expected_current()) < EXPECTED_CURRENT_TOLERANCE) {
      return OK;
  }
  return ERROR;
}
END:


PROC_DEF: int update_target_current(double I)
{
  target_current = I;
  return OK;
}
END:


PROC_DEF: int at_target_current ()
 {
  if (fabs (target_current - present_op_state.coil_current) > TARGET_CURRENT_TOLERANCE) {
    cerr << “Not at target current of “ << target_current <<
```

```
      “ present: “ << present_op_state.coil_current << “\n”;
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int reasonable_current_param (double I)
{
  if (fabs(I - target_current) > MAX_CURRENT_CHANGE ||
      0.5 * inductance * fabs (sqr (I) - sqr (target_current)) > ENERGY_CHANGE_RATE)
  {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int servo_faults()
{
  if (present_op_state.fault_line != AMP_OK){
     return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int zero_current()
{
  if (fabs(present_op_state.coil_current) > LOW_CURRENT ||
       target_current != 0) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: double expected_current()
{
  double I;
  I = op_state_at_last_cmd.coil_current +
         sign (target_current - op_state_at_last_cmd.coil_current)
         * CURRENT_CHARGE_RATE * time_since_last_cmd;

  if ((fabs(I - op_state_at_last_cmd.coil_current)>
      fabs(target_current - op_state_at_last_cmd.coil_current)) ||
        (fabs(target_current - I) < TARGET_CURRENT_TOLERANCE)) {
         I = target_current;

  }
  return I;
}
END:


PROC_DEF: int do_set_current(double I)
{
  if(servoamp::set_current(I) != AMP_OK) {
    return ERROR;
  }
  cout << device_name << “ set to “ << I << endl;
  return OK;
}
END:


PROC_DEF: int do_get_actual_current(double &I)
{
  if(servoamp::get_actual_current(I) != AMP_OK) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int do_get_current_settings(double &I, unsigned &dac_value)
{
  if(servoamp::get_current_settings(I, dac_value) != AMP_OK) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int do_inhibit()
{
  servoamp::inhibit();
  return OK;
}
END:


PROC_DEF: int do_uninhibit()
{
  servoamp::uninhibit();
  return OK;
}
END:


PROC_DEF: int do_get_amp_status()
{
  if(servoamp::get_amp_status() != AMP_OK) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int do_read_fault()
{
  if(servoamp::read_fault() != AMP_OK) {
    return ERROR;
  }
  return OK;
}
END:


PROC_DEF: int do_emergency_dump()
{

  return OK;
```

```
}
END:


PROC_DEF: int do_failure_reset()
{

  return OK;
}
END:


PROC_DEF: int do_shutdown()
{

  return OK;
}
END:


PROC_DEF: int do_emergency_set_current(double I)
{
  servoamp::set_current(I);
  return OK;
}
END:


PROC_DEF: int initialize_servoamp(double the_inductance)
{
  inductance = the_inductance;
  target_current = 0.0;
  servoamp::uninhibit();
  servoamp::set_current(target_current);
  servoamp::inhibit();
  return OK;
}
END:


PROC_DEF: int reset_servoamp()
{
  target_current = 0.0;
  servoamp::uninhibit();
  if(servoamp::get_amp_status() == AMP_OK) {
    servoamp::set_current(target_current);
  }
  servoamp::inhibit();
  return OK;
}
END:
```

UVAR Safety Policy Specification

System Machine

```
#include “config_macros.h”


INCLUDES:      coordinate.h
INCLUDES:      local_system.h
```

```
DEV_TYPE:      uvar_system

DEV_CONFIG:    header_dev.fil
DEV_CONFIG:    regulator_rod.fil
DEV_CONFIG:    safety_rod.fil
DEV_CONFIG:    pump_dev.fil
DEV_CONFIG:    neutron_detector.fil

CHILD:         header_dev header
CHILD:         pump_dev pump
CHILD:         neutron_detector detector
CHILD:         regulator_rod regulator
CHILD:         safety_rod control1
CHILD:         safety_rod control2
CHILD:         safety_rod control3

FRAME_LENGTH:  2.0

SCHEDULE:      REACTOR_OFF: 0.0, 0.5, 1.0, 1.5; 0.02, 0.5
SCHEDULE:      REACTOR_ON: 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, \
                       0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9; 0.02, 0.1
SCHEDULE:      WATCHDOG: 0.0, 0.4, 0.8, 1.2, 1.6; 0.02, 0.5
WDOG_SCHED:    WATCHDOG

OP_ST_VAR:     double channel1_power
OP_ST_VAR:     double channel2_power
OP_ST_VAR:     double period_amp
OP_ST_VAR:     double bridge_radiation
OP_ST_VAR:     double reactor_face_radiation
OP_ST_VAR:     int scram_button_reactor_room
OP_ST_VAR:     int scram_button_ground_floor
OP_ST_VAR:     int reactor_room_truck_door
OP_ST_VAR:     int emergency_escape_hatch
OP_ST_VAR:     double primary_header_air_pressure
OP_ST_VAR:     double reactor_inlet_water_temp
OP_ST_VAR:     double pool_level             // multiple
OP_ST_VAR:     int fire_alarm
OP_ST_VAR:     double radiation_monitor      // multiple
OP_ST_VAR:     double argon_monitor          // multiple
OP_ST_VAR:     double core_gamma_monitor
OP_ST_VAR:     double criticality_monitor
OP_ST_VAR:     double constant_air_monitor
OP_ST_VAR:     int demin_room_door
OP_ST_VAR:     int heat_exch_room_door
OP_ST_VAR:     double core_delta_t
OP_ST_VAR:     double demin_conductivity

ST_ACQ_PROC:   acquire_state(state_vector); STATE_ACQUISITION_FAILED

MODE:          inactive
MODE:          setup
MODE:          startup_low
MODE:          startup_high
MODE:          operating_low
MODE:          operating_high
MODE:          scrammed

INIT_MODE:     inactive

COMMAND:       setup_begin()
COMMAND:       setup_end()
COMMAND:       startup_low_begin()
COMMAND:       startup_high_begin()
COMMAND:       operating_low_begin()
```

```
COMMAND:       operating_high_begin()
COMMAND:       scram()
COMMAND:       scram_reset()

ACT_PROC:      setup_begin: do_setup_begin(); SETUP_BEGIN_FAILED
ACT_PROC:      setup_end : do_setup_end(); SETUP_END_FAILED
ACT_PROC:      startup_low_begin : do_startup_low_begin(); STARTUP_LOW_BEGIN_FAILED
ACT_PROC:      startup_high_begin : do_startup_high_begin(); STARTUP_HIGH_BEGIN_FAILED
ACT_PROC:      operating_low_begin : do_operating_low_begin(); OPERATING_LOW_BEGIN_FAILED
ACT_PROC:      operating_high_begin             :             do_operating_high_begin();
OPERATING_HIGH_BEGIN_FAILED
ACT_PROC:      scram: do_scram(); SCRAM_FAILED
ACT_PROC:      scram_reset : do_scram_reset(); SCRAM_RESET_FAILED

CONST:         double MAX_IDLE_PERIOD_MOVEMENT = 15.0
CONST:         double MAX_INCREMENTAL_MOVEMENT = 5.0
CONST:         double MAX_UCS_MOVEMENT = 2.0
CONST:         double THRESHOLD_FORCE = 2.0

INTERLOCK:     setup_begin
  MODE:        inactive
  NEW_MODE:    setup
  END:

  MODE:        setup
  MODE:        startup_low
  MODE:        startup_high
  MODE:        operating_low
  MODE:        operating_high
  MODE:        scrammed
  NEW_MODE:    MUST_BE_INACTIVE
  END:
END:


INTERLOCK:     setup_end
  MODE:        setup
  NEW_MODE:    setup
  END:

  MODE:        inactive
  MODE:        startup_low
  MODE:        startup_high
  MODE:        operating_low
  MODE:        operating_high
  MODE:        scrammed
  NEW_MODE:    MUST_BE_IN_SETUP
  END:
END:


INTERLOCK:     startup_low_begin
  MODE:        setup
  NEW_MODE:    startup_low
  CONDITION:   check_setup_low(); ILLEGAL_LOW_CONFIG
END:

  MODE:        inactive
  MODE:        startup_low
  MODE:        startup_high
  MODE:        operating_low
  MODE:        operating_high
  MODE:        scrammed
  NEW_MODE:    MUST_BE_IN_SETUP
  END:
```

```
END:


INTERLOCK:     startup_high_begin
  MODE:        setup
  NEW_MODE:    startup_high
  CONDITION:   check_setup_high(); ILLEGAL_HIGH_CONFIG
  END:

  MODE:        inactive
  MODE:        startup_low
  MODE:        startup_high
  MODE:        operating_low
  MODE:        operating_high
  MODE:        scrammed
  NEW_MODE:    MUST_BE_IN_SETUP
  END:
END:


INTERLOCK:     operating_low_begin
  MODE:        startup_low
  NEW_MODE:    operating_low
  END:

  MODE:        inactive
  MODE:        setup
  MODE:        startup_high
  MODE:        operating_low
  MODE:        operating_high
  MODE:        scrammed
  NEW_MODE:    MUST_BE_IN_STARTUP_LOW
  END:
END:


INTERLOCK:     operating_high_begin
  MODE:        startup_high
  NEW_MODE:    operating_high
  END:

  MODE:        inactive
  MODE:        setup
  MODE:        startup_low
  MODE:        operating_low
  MODE:        operating_high
  MODE:        scrammed
  NEW_MODE:    MUST_BE_IN_STARTUP_HIGH
  END:
END:


INTERLOCK:     scram
  MODE:        inactive
  MODE:        setup
  MODE:        startup_low
  MODE:        startup_high
  MODE:        operating_low
  MODE:        operating_high
  MODE:        scrammed
  NEW_MODE:    scrammed
  END:
END:
```

```
INTERLOCK:     scram_reset
  MODE:        inactive
  MODE:        setup
  MODE:        startup_low
  MODE:        startup_high
  MODE:        operating_low
  MODE:        operating_high
  NEW_MODE:    MUST_BE_SCRAMMED
  END:

  MODE:        scrammed
  NEW_MODE:    scram_reset()
  END:
END:

CONST:         double MAX_POWER = 125.0

MONITOR:       inactive
SCHEDULE:      REACTOR_OFF
END:

MONITOR:       setup
SCHEDULE:      REACTOR_OFF
END:

MONITOR:       startup_low
SCHEDULE:      REACTOR_ON
END:

MONITOR:       startup_high
SCHEDULE:      REACTOR_ON
CONDITION:     runaway(); RUNAWAY_SEED
END:

MONITOR:       operating_low
SCHEDULE:      REACTOR_ON
END:

MONITOR:       operating_high
SCHEDULE:      REACTOR_ON
CONDITION:     less_than_max(present_op_state.channel1_power, MAX_POWER); \
                                      MAX_POWER_EXCEEDED
CONDITION:     less_than_max(present_op_state.bridge_radiation, MAX_BRIDGE_RADIATION); \
                                      MAX_BRIDGE_RADIATION_EXCEEDED

// Other conditions would follow for all of the operational state
// variables.
END:


MONITOR:       scrammed
SCHEDULE:      REACTOR_ON
END:

// All of the error responses will be warnings or scrams
ERROR_COND:    MAX_POWER_EXCEEDED
  RESPONSE:    do_scram(); NO_RESPONSE
END:

ERROR_COND:    MAX_BRIDGE_RADIATION_EXCEEDED
  RESPONSE:    do_scram(); NO_RESPONSE
END:


PROC_DEF: int do_scram();
```

```
{
  // invoke scram here

  return OK;
}
END:
```

Safety Rod Machine

```
#include “config_macros.h”


DEV_TYPE:      safety_rod

INCLUDES:      general_procs.h
INCLUDES:      math.h

INIT_PROC:     initialize_safety_rod()

OP_ST_VAR:     double position

ST_ACQ_PROC:   acquire_state(state_vector); safety_rod::STATE_ACQUISITION_FAILED

MODE:          operating
MODE:          failed
INIT_MODE:     operating


COMMAND:       up()
COMMAND:       down()
COMMAND:       scram()
COMMAND:       set_failed()
COMMAND:       reset()


ACT_PROC:      up : do_up; safety_rod::UP_FAILED
ACT_PROC:      down : do_down; safety_rod::DOWN_FAILED
ACT_PROC:      scram : do_scram; safety_rod::SCRAM_FAILED
ACT_PROC:      set_failed : do_set_failed(); safety_rod::SET_FAILED_FAILED
ACT_PROC:      reset : do_reset(); safety_rod::RESET_FAILED


// State-command policies
INTERLOCK:            up
  MODE:               operating
    PARENT_MODE:      inactive
    NEW_MODE:         safety_rod::COILS_INACTIVE
    END:
    PARENT_MODE:      setup
    PARENT_MODE:      startup_low
    PARENT_MODE:      startup_high
    PARENT_MODE:      operating_low
    PARENT_MODE:      operating_high
    PARENT_MODE:      scrammed
    NEW_MODE:         operating
    END:
  END:

  MODE:               failed
    NEW_MODE:         safety_rod::FAILED
  END:
END:
```

```
INTERLOCK:              down
  MODE:                 operating
    PARENT_MODE:        inactive
    NEW_MODE:           safety_rod::COILS_INACTIVE
    END:
    PARENT_MODE:        setup
    PARENT_MODE:        startup_low
    PARENT_MODE:        startup_high
    PARENT_MODE:        operating_low
    PARENT_MODE:        operating_high
    PARENT_MODE:        scrammed
    NEW_MODE:           operating
    END:
  END:

  MODE:                 failed
    NEW_MODE:           safety_rod::FAILED
  END:
END:


INTERLOCK:              scram
  MODE:                 operating
  MODE:                 failed
    NEW_MODE:           operating
  END:
END:


INTERLOCK:              set_failed
  MODE:                 operating
    PARENT_MODE:        scrammed
    PARENT_MODE:        inactive
    PARENT_MODE:        setup
    NEW_MODE:           failed
    END:

    PARENT_MODE:        startup_low
    PARENT_MODE:        startup_high
    PARENT_MODE:        operating_low
    PARENT_MODE:        operating_high
    NEW_MODE:           safety_rod::NOT_SCRAMMED
    END:
  END:

  MODE:                 failed
  NEW_MODE:             failed
  END:
END:


INTERLOCK:              reset
  MODE:                 operating
  NEW_MODE:             safety_rod::NOT_FAILED
  END:

  MODE:                 failed
  NEW_MODE:             operating
  END:
END:


MONITOR:                operating
  SCHEDULE:             REACTOR_ON
```

```
END:


MONITOR:               failed
  SCHEDULE:            NO_SCHEDULE
END:


// Error conditions
// All of the error responses will be warnings or setting failed
// Scram is triggered at the system level

ERROR_COND:            safety_rod::NO_RESPONSE
  RESPONSE:            announce_error(“safety_rod::NO_RESPONSE”); safety_rod::NO_RESPONSE
END:


PROC_DEF: int announce_error(char *message)
{
  cerr << “ERROR for device “ << device_name << “ -- “ <<
               message << “\n” << endl;
  cerr << “Present device mode: “ << present_mode << endl;
  return OK;
}
END:
```

Regulator rod Machine

```
#include “config_macros.h”


DEV_TYPE:      regulator_rod

INCLUDES:      general_procs.h
INCLUDES:      math.h

INIT_PROC:     initialize_regulator_rod()

OP_ST_VAR:     double position

ST_ACQ_PROC:   acquire_state(state_vector); regulator_rod::STATE_ACQUISITION_FAILED

MODE:          auto
MODE:          manual
MODE:          failed
INIT_MODE:     manual

COMMAND:       up()
COMMAND:       down()
COMMAND:       auto_up()
COMMAND:       auto_down()
COMMAND:       set_auto()
COMMAND:       set_manual()
COMMAND:       set_failed()
COMMAND:       reset()


ACT_PROC:      up : do_up; regulator_rod::UP_FAILED
ACT_PROC:      down : do_down(); regulator_rod::DOWN_FAILED
ACT_PROC:      auto_up : do_auto_up(); regulator_rod::AUTO_UP_FAILED
ACT_PROC:      auto_down : do_auto_down(); regulator_rod::AUTO_DOWN_FAILED
ACT_PROC:      set_auto : do_set_auto(); regulator_rod::SET_AUTO_FAILED
```

```
ACT_PROC:      set_manual : do_set_manual(); regulator_rod::SET_MANUAL_FAILED
ACT_PROC:      set_failed : do_set_failed(); regulator_rod::SET_FAILED_FAILED
ACT_PROC:      reset : do_reset(); regulator_rod::RESET_FAILED


CLASS_DECL:    double last_position


// State-command policies
INTERLOCK:             up
  MODE:                auto
    PARENT_MODE:       inactive
    NEW_MODE:          regulator_rod::COILS_INACTIVE
    END:
    PARENT_MODE:       setup
    PARENT_MODE:       startup_low
    PARENT_MODE:       startup_high
    PARENT_MODE:       operating_low
    PARENT_MODE:       operating_high
    PARENT_MODE:       scrammed
    NEW_MODE:          manual
    END:
  END:

  MODE:                manual
    PARENT_MODE:       inactive
    NEW_MODE:          regulator_rod::COILS_INACTIVE
    END:
    PARENT_MODE:       setup
    PARENT_MODE:       startup_low
    PARENT_MODE:       startup_high
    PARENT_MODE:       operating_low
    PARENT_MODE:       operating_high
    PARENT_MODE:       scrammed
    NEW_MODE:          manual
    END:
  END:

  MODE:                failed
    NEW_MODE:          regulator_rod::FAILED
  END:
END:


INTERLOCK:             down
  MODE:                auto
    PARENT_MODE:       inactive
    NEW_MODE:          regulator_rod::COILS_INACTIVE
    END:
    PARENT_MODE:       setup
    PARENT_MODE:       startup_low
    PARENT_MODE:       startup_high
    PARENT_MODE:       operating_low
    PARENT_MODE:       operating_high
    PARENT_MODE:       scrammed
    NEW_MODE:          manual
    END:
  END:

  MODE:                manual
    PARENT_MODE:       inactive
    NEW_MODE:          regulator_rod::COILS_INACTIVE
    END:
    PARENT_MODE:       setup
    PARENT_MODE:       startup_low
    PARENT_MODE:      startup_high
    PARENT_MODE:      operating_low
    PARENT_MODE:      operating_high
    PARENT_MODE:      scrammed
    NEW_MODE:         manual
    END:
  END:

  MODE:               failed
    NEW_MODE:         regulator_rod::FAILED
  END:
END:


INTERLOCK:            auto_up
  MODE:               auto
    PARENT_MODE:      inactive
    PARENT_MODE:      setup
    PARENT_MODE:      startup_low
    PARENT_MODE:      startup_high
    PARENT_MODE:      scrammed
    NEW_MODE:         regulator_rod::ILLEGAL_MODE_AUTO
    END:

    PARENT_MODE:      operating_low
    PARENT_MODE:      operating_high
    NEW_MODE:         auto
    END:
  END:

  MODE:               manual
    PARENT_MODE:      inactive
    NEW_MODE:         regulator_rod::COILS_INACTIVE
    END:
    PARENT_MODE:      setup
    PARENT_MODE:      startup_low
    PARENT_MODE:      startup_high
    PARENT_MODE:      operating_low
    PARENT_MODE:      operating_high
    PARENT_MODE:      scrammed
    NEW_MODE:         regulator_rod::IN_MANUAL_MODE
    END:
  END:

  MODE:               failed
    NEW_MODE:         regulator_rod::FAILED
  END:
END:


INTERLOCK:            auto_down
  MODE:               auto
    PARENT_MODE:      inactive
    PARENT_MODE:      setup
    PARENT_MODE:      startup_low
    PARENT_MODE:      startup_high
    PARENT_MODE:      scrammed
    NEW_MODE:         regulator_rod::ILLEGAL_MODE_AUTO
    END:

    PARENT_MODE:      operating_low
    PARENT_MODE:      operating_high
    NEW_MODE:         auto
    END:
  END:
```

```
  MODE:               manual
    PARENT_MODE:      inactive
    NEW_MODE:         regulator_rod::COILS_INACTIVE
    END:
    PARENT_MODE:      setup
    PARENT_MODE:      startup_low
    PARENT_MODE:      startup_high
    PARENT_MODE:      operating_low
    PARENT_MODE:      operating_high
    PARENT_MODE:      scrammed
    NEW_MODE:         regulator_rod::IN_MANUAL_MODE
    END:
  END:

  MODE:               failed
    NEW_MODE:         regulator_rod::FAILED
  END:
END:


INTERLOCK:            set_auto
  MODE:               auto
    PARENT_MODE:      inactive
    PARENT_MODE:      setup
    PARENT_MODE:      startup_low
    PARENT_MODE:      startup_high
    PARENT_MODE:      scrammed
    NEW_MODE:         regulator_rod::ILLEGAL_MODE_AUTO
    END:
    PARENT_MODE:      operating_low
    PARENT_MODE:      operating_high
    NEW_MODE:         regulator_rod::ALREADY_AUTO
    END:
  END:

  MODE:               manual
    PARENT_MODE:      inactive
    NEW_MODE:         regulator_rod::COILS_INACTIVE
    END:
    PARENT_MODE:      setup
    PARENT_MODE:      startup_low
    PARENT_MODE:      startup_high
    PARENT_MODE:      scrammed
    NEW_MODE:         regulator_rod::NOT_IN_OPERATING_MODE
    END:

    PARENT_MODE:      operating_low
    PARENT_MODE:      operating_high
    NEW_MODE:         auto
    END:
  END:

  MODE:               failed
    NEW_MODE:         regulator_rod::FAILED
  END:
END:


INTERLOCK:            set_manual
  MODE:               auto
    PARENT_MODE:      inactive
    PARENT_MODE:      setup
    PARENT_MODE:      startup_low
    PARENT_MODE:      startup_high
    PARENT_MODE:       scrammed
    NEW_MODE:          regulator_rod::ILLEGAL_MODE_AUTO
    END:
    PARENT_MODE:       operating_low
    PARENT_MODE:       operating_high
    NEW_MODE:          manual
    END:
  END:

  MODE:                manual
    PARENT_MODE:       inactive
    NEW_MODE:          regulator_rod::COILS_INACTIVE
    END:
    PARENT_MODE:       setup
    PARENT_MODE:       startup_low
    PARENT_MODE:       startup_high
    PARENT_MODE:       scrammed
    PARENT_MODE:       operating_low
    PARENT_MODE:       operating_high
    NEW_MODE:          regulator_rod::ALREADY_MANUAL
    END:
  END:

  MODE:                failed
    NEW_MODE:          regulator_rod::FAILED
  END:
END:


INTERLOCK:             set_failed
  MODE:                auto
  MODE:                manual
  MODE:                failed
  NEW_MODE:            failed
  END:
END:


INTERLOCK:             reset
  MODE:                auto
  MODE:                manual
  NEW_MODE:            regulator_rod::NOT_FAILED
  END:

  MODE:                failed
  NEW_MODE:            manual
  END:
END:


// Device error detection policies

MONITOR:               auto
  PARENT_MODE:         inactive
  PARENT_MODE:         setup
  PARENT_MODE:         startup_low
  PARENT_MODE:         startup_high
  PARENT_MODE:         scrammed
  SCHEDULE:            REACTOR_ON
  CONDITION:           return_ERROR(); regulator_rod::ILLEGAL_MODE_AUTO
  END:
  PARENT_MODE:         operating_low
  PARENT_MODE: o       perating_high
  CONDITION:           check_auto(); regulator_rod::AUTO_FAILURE
  END:
```

```
END:

MONITOR:               manual
  SCHEDULE:            REACTOR_ON
  CONDITION:           check_manual(); regulator_rod::MANUAL_FAILURE
END:


MONITOR:               failed
  SCHEDULE:            NO_SCHEDULE
END:


// Error conditions
// All of the error responses will be warnings or setting failed
// Scram is triggered at the system level

ERROR_COND:            regulator_rod::NO_RESPONSE
  RESPONSE:            announce_error("regulator_rod::NO_RESPONSE");
regulator_rod::NO_RESPONSE
END:


PROC_DEF: int announce_error(char *message)
{
  cerr << "ERROR for device " << device_name << " -- " <<
               message << "\n" << endl;
  cerr << "Present device mode: " << present_mode << endl;
  return OK;
}
END:
```

Pump Machine

```
 #include "config_macros.h"


DEV_TYPE:      pump_dev

INCLUDES:      general_procs.h
INCLUDES:      math.h

INIT_PROC:     initialize_pump_dev

OP_ST_VAR:     double cooling_flow

ST_ACQ_PROC:   acquire_state(state_vector); regulator_rod::STATE_ACQUISITION_FAILED

MODE:          off
MODE:          on
MODE:          failed
INIT_MODE:     off

COMMAND:       turn_off()
COMMAND:       turn_on()
COMMAND:       set_failed()
COMMAND:       reset()


ACT_PROC:      turn_on : do_turn_on; pump_dev::TURN_ON_FAILED
```

```
ACT_PROC:      turn_off : do_turn_off; pump_dev::TURN_OFF_FAILED
ACT_PROC:      set_failed : do_set_failed(); pump_dev::SET_FAILED_FAILED
ACT_PROC:      reset : do_reset(); pump_dev::RESET_FAILED


// State-command policies
INTERLOCK:             turn_off
  MODE:                off
    PARENT_MODE:       inactive
    NEW_MODE:          pump_dev::COILS_INACTIVE
    END:
    PARENT_MODE:       startup_low
    PARENT_MODE:       startup_high
    PARENT_MODE:       operating_low
    PARENT_MODE:       operating_high
    NEW_MODE:          pump_dev::ILLEGAL_MODE_OFF
    END:
    PARENT_MODE:       setup
    PARENT_MODE:       scrammed
    NEW_MODE:          regulator_rod::ALREADY_OFF
    END:
  END:

  MODE:                on
    PARENT_MODE:       inactive
    NEW_MODE:          pump_dev::COILS_INACTIVE
    END:
    PARENT_MODE:       startup_low
    PARENT_MODE:       startup_high
    PARENT_MODE:       operating_low
    PARENT_MODE:       operating_high
    NEW_MODE:          pump_dev::REACTOR_IS_OPERATING
    END:
    PARENT_MODE:       setup
    PARENT_MODE:       scrammed
    NEW_MODE:          off
    END:
  END:

  MODE:                failed
    NEW_MODE:          pump_dev::FAILED
  END:
END:


INTERLOCK:             turn_on
  MODE:                off
    PARENT_MODE:       inactive
    NEW_MODE:          pump_dev::COILS_INACTIVE
    END:
    PARENT_MODE:       startup_low
    PARENT_MODE:       startup_high
    PARENT_MODE:       operating_low
    PARENT_MODE:       operating_high
    NEW_MODE:          pump_dev::SHOULD_BE_ON
    END:
    PARENT_MODE:       setup
    PARENT_MODE:       scrammed
    NEW_MODE:          on
    END:
  END:

  MODE:                on
    PARENT_MODE:       inactive
    NEW_MODE:          pump_dev::COILS_INACTIVE
    END:
    PARENT_MODE:      startup_low
    PARENT_MODE:      startup_high
    PARENT_MODE:      operating_low
    PARENT_MODE:      operating_high
    PARENT_MODE:      setup
    PARENT_MODE:      scrammed
    NEW_MODE:         pump_dev::ALREADY_ON
    END:
  END:

  MODE:               failed
    NEW_MODE:         pump_dev::FAILED
  END:
END:


INTERLOCK:            set_failed
  MODE:               on
  MODE:               off
  MODE:               failed
  NEW_MODE:           failed
  END:
END:


INTERLOCK:            reset
  MODE:               on
  MODE:               off
  NEW_MODE:           pump_dev::NOT_FAILED
  END:

  MODE:               failed
  NEW_MODE:           off
  END:
END:


// Device error detection policies

MONITOR:              off
  PARENT_MODE:        inactive
  PARENT_MODE:        setup
  SCHEDULE:           REACTOR_OFF
  CONDITION:          return_ERROR(); pump_dev::ILLEGAL_MODE_AUTO
  END:

  PARENT_MODE:        startup_low
  PARENT_MODE:        startup_high
  PARENT_MODE:        operating_low
  PARENT_MODE:        operating_high
  SCHEDULE:           REACTOR_ON
  CONDITION:          return_ERROR(); pump_dev::ILLEGAL_MODE_OFF
  END:

  PARENT_MODE:        scrammed
  CONDITION:          check_off(); pump_dev::SHOULD_BE_OFF
  END:
END:

MONITOR:              on
  PARENT_MODE:        inactive
  SCHEDULE:           REACTOR_OFF
  CONDITION:          return_ERROR(); pump_dev::ILLEGAL_MODE_ON
  END:
```

```
  PARENT_MODE:         setup
  PARENT_MODE:         startup_low
  PARENT_MODE:         startup_high
  PARENT_MODE:         operating_low
  PARENT_MODE:         operating_high
  PARENT_MODE:         scrammed
  SCHEDULE:            REACTOR_ON
  CONDITION:           check_on(); pump_dev::SHOULD_BE_ON
  END:
END:


MONITOR:               failed
  SCHEDULE:            NO_SCHEDULE
END:


// Error conditions
// All of the error responses will be warnings or setting failed
// Scram is triggered at the system level

ERROR_COND:            mss_servoamp::NO_RESPONSE
  RESPONSE:            announce_error("mss_servoamp::NO_RESPONSE");
mss_servoamp::NO_RESPONSE
END:


PROC_DEF: int announce_error(char *message)
{
  cerr << "ERROR for device " << device_name << " -- " <<
               message << "\n" << endl;
  cerr << "Present device mode: " << present_mode << endl;
  return OK;
}
END:
```

Header Machine

```
#include "config_macros.h"


DEV_TYPE:      header_dev

INCLUDES:      general_procs.h
INCLUDES:      math.h

INIT_PROC:     initialize_header()

ST_ACQ_PROC:   acquire_state(state_vector); header_dev::STATE_ACQUISITION_FAILED

MODE:          up
MODE:          down
MODE:          failed
INIT_MODE:     operating


COMMAND:       move_up()
COMMAND:       down()
COMMAND:       set_failed()
```

```
COMMAND:       reset()

ACT_PROC:      move_up : do_move_up; header_dev::MOVE_UP_FAILED
ACT_PROC:      move_down : do_move_down; header_dev::MOVE_DOWN_FAILED
ACT_PROC:      set_failed : do_set_failed(); header_dev::SET_FAILED_FAILED
ACT_PROC:      reset : do_reset(); header_dev::RESET_FAILED


// State-command policies
INTERLOCK:            move_up
  MODE:               down
    PARENT_MODE:      setup
    NEW_MODE:         up
    END:

    PARENT_MODE:      inactive
    PARENT_MODE:      startup_low
    PARENT_MODE:      startup_high
    PARENT_MODE:      operating_low
    PARENT_MODE:      operating_high
    PARENT_MODE:      scrammed
    NEW_MODE:         header_dev::NOT_IN_SETUP
    END:
  END:

  MODE:               up
  NEW_MODE:           header_dev::ALREADY_UP
  END:

  MODE:               failed
  NEW_MODE:           header_dev::FAILED
  END:
END:


INTERLOCK:            move_down
  MODE:               up
    PARENT_MODE:      setup
    NEW_MODE:         down
    END:

    PARENT_MODE:      inactive
    PARENT_MODE:      startup_low
    PARENT_MODE:      startup_high
    PARENT_MODE:      operating_low
    PARENT_MODE:      operating_high
    PARENT_MODE:      scrammed
    NEW_MODE:         header_dev::NOT_IN_SETUP
    END:
  END:

  MODE:               down
  NEW_MODE:           header_dev::ALREADY_DOWN
  END:

  MODE:               failed
    NEW_MODE:         header_dev::FAILED
  END:
END:


INTERLOCK:            set_failed
  MODE:               operating
    PARENT_MODE:      scrammed
    PARENT_MODE:      inactive
    PARENT_MODE:       setup
    NEW_MODE:          failed
    END:

    PARENT_MODE:       startup_low
    PARENT_MODE:       startup_high
    PARENT_MODE:       operating_low
    PARENT_MODE:       operating_high
    NEW_MODE:          header_dev::NOT_SCRAMMED
    END:
  END:

  MODE:                failed
  NEW_MODE:            failed
  END:
END:


INTERLOCK:             reset
  MODE:                operating
  NEW_MODE:            header_dev::NOT_FAILED
  END:

  MODE:                failed
  NEW_MODE:            operating
  END:
END:


MONITOR:               up
  SCHEDULE:            REACTOR_ON
  CONDITION:           header_is_up(); header_dev::SHOULD_BE_UP
END:

MONITOR:               down
  SCHEDULE:            REACTOR_ON
  CONDITION:           header_is_down(); header_dev::SHOULD_BE_DOWN
END:

MONITOR:               failed
  SCHEDULE:            NO_SCHEDULE
END:


// Error conditions
// All of the error responses will be warnings or setting failed
// Scram is triggered at the system level

ERROR_COND:            header_dev::NO_RESPONSE
  RESPONSE:            announce_error(“header_dev::NO_RESPONSE”); header_dev::NO_RESPONSE
END:


PROC_DEF: int announce_error(char *message)
{
  cerr << “ERROR for device “ << device_name << “ -- “ <<
               message << “\n” << endl;
  cerr << “Present device mode: “ << present_mode << endl;
  return OK;
}
END:
```

Neutron Detector Machine

```
include “config_macros.h”


DEV_TYPE:      neutron_detector

INCLUDES:      general_procs.h
INCLUDES:      math.h

INIT_PROC:     initialize_header()

ST_ACQ_PROC:   acquire_state(state_vector); neutron_detector::STATE_ACQUISITION_FAILED

MODE:          operating
MODE:          failed
INIT_MODE:     operating


COMMAND:       forward()
COMMAND:       backward()
COMMAND:       set_failed()
COMMAND:       reset()


ACT_PROC:      forward : do_forward; neutron_detector::FORWARD_FAILED
ACT_PROC:      backward : do_backward; neutron_detector::BACKWARD_FAILED
ACT_PROC:      set_failed : set_failed(); neutron_detector::SET_FAILED_FAILED
ACT_PROC:      reset : reset(); neutron_detector::RESET_FAILED

// State-command policies
INTERLOCK:            forward
  MODE:               operating
    PARENT_MODE:      setup
    PARENT_MODE:      inactive
    PARENT_MODE:      startup_low
    PARENT_MODE:      startup_high
    PARENT_MODE:      operating_low
    PARENT_MODE:      operating_high
    PARENT_MODE:      scrammed
    NEW_MODE:         operating
    END:
  END:

  MODE:               failed
    NEW_MODE:         neutron_detector::FAILED
  END:
END:


INTERLOCK:            backward
  MODE:               operating
    PARENT_MODE:      setup
    PARENT_MODE:      inactive
    PARENT_MODE:      startup_low
    PARENT_MODE:      startup_high
    PARENT_MODE:      operating_low
    PARENT_MODE:      operating_high
    PARENT_MODE:      scrammed
    NEW_MODE:         operating
    END:
  END:
```

```
  MODE:                 failed
    NEW_MODE:           neutron_detector::FAILED
  END:
END:


INTERLOCK:              set_failed
  MODE:                 operating
    PARENT_MODE:        scrammed
    PARENT_MODE:        inactive
    PARENT_MODE:        setup
    NEW_MODE:           failed
    END:

    PARENT_MODE:        startup_low
    PARENT_MODE:        startup_high
    PARENT_MODE:        operating_low
    PARENT_MODE:        operating_high
    NEW_MODE:           neutron_detector::NOT_SCRAMMED
    END:
  END:

  MODE:                 failed
  NEW_MODE:             failed
  END:
END:


INTERLOCK:              reset
  MODE:                 operating
  NEW_MODE:             neutron_detector::NOT_FAILED
  END:

  MODE:                 failed
  NEW_MODE:             operating
  END:
END:
MONITOR:                operating
  SCHEDULE:             REACTOR_ON
END:

MONITOR:                failed
  SCHEDULE:             NO_SCHEDULE
END:


// Error conditions
// All of the error responses will be warnings or setting failed
// Scram is triggered at the system level

ERROR_COND:             neutron_detector::NO_RESPONSE
  RESPONSE:             announce_error("neutron_detector::NO_RESPONSE"); \
                               neutron_detector::NO_RESPONSE
END:


PROC_DEF: int announce_error(char *message)
{
  cerr << "ERROR for device " << device_name << " -- " <<
               message << "\n" << endl;
  cerr << "Present device mode: " << present_mode << endl;
  return OK;
}
END:
```