A SAFETY KERNEL ARCHITECTURE[†]

Kevin G. Wika   John C. Knight

Department of Computer Science
University of Virginia
Charlottesville
Virginia 22903

Abstract

Software plays a crucial role in a large number of safety-critical systems. In spite of this, many safety-critical systems exhibit residual software errors even after extensive implementation and verification efforts. We describe a software architecture based on a safety kernel that facilitates the implementation and verification of safety-critical software. Drawing many concepts from the related notion of a security kernel, the safety kernel enforces safety policies independent of application programs and permits verification of properties of a small kernel rather than large amounts of application software.

Safety-critical software is typically custom-built for each application. This leads to consistently high development costs and limited reuse of either designs or software modules. We believe that the safety kernel architecture provides a framework for identification of generally applicable classes of safety policies. The paper examines several classes of kernel-enforced policies. The policies have been identified using criteria that consider the criticality of a policy and the effects of kernel enforcement on the simplicity and verifiability of both the application software and the safety kernel. These general policies are parameterized to enable configuration of an instance of the safety kernel. A mechanical translator is utilized to instantiate the safety kernel from the parameter information.

The concepts and design of the safety kernel architecture have been significantly influenced by our research into the development of dependable software for an experimental neurosurgical device. For example, having worked with this "real" system, we recognized that in general the safety kernel would need to coexist with a large amount of potentially unreliable software. As a result, we describe a system architecture wherein the safety kernel operates in the context of application and system software that is unreliable or at least of unknown properties.

Index Terms — software safety, software architecture, verification, security kernel, safety kernel

†. Supported in part by the National Science Foundation under grant number CCR-9213427 and in part by NASA under grant number NAG1-1123-FDP.

I 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 [4, 11]. 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 dependability because of the sheer number of tests that are required [3].

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 in a meaningful way 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 goal of the research described here is to try to deal 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.

In this paper we describe a safety kernel architecture that is being designed as part of a case study in which the software for an experimental, safety-critical system is being developed. We discuss safety policy selection and enforcement, and describe the practical application of the kernel. In the next section, we examine the general notion of a safety kernel. This is followed by a discussion of research relevant to the safety kernel. Section IV describes the safety-critical system of the case study. A description of the safety kernel architecture is presented in section V. The concept of kernel-enforced safety policies is introduced in section VI along with a discussion of the safety policies that are enforced by the safety kernel. Section VII examines implementation issues of the safety kernel and looks in detail at the mechanisms that enforce selected classes of policies. We conclude with a summary and conclusions in section VIII.

II 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 [1]. 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 [2]. 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.

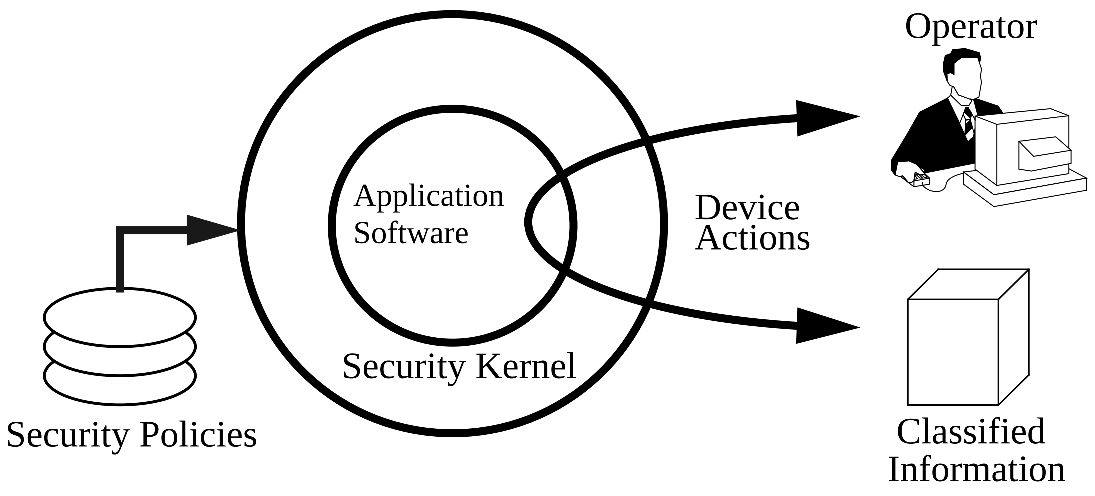

Fig. 1. Security Kernel Concept.

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 hope 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.

Although the primary objective of a safety kernel is to facilitate verification of safety-critical software, an additional concern is that safety-critical software tends to be custom built for each application. This ad hoc development practice increases the cost of software and precludes innovative concepts in one system from being exploited in others. The situation is partially attributable to the unique safety concerns of each application. However, it is also very difficult to abstract out the features that might be applicable to a range of systems without an appropriate architecture.

We believe the safety kernel concept provides the framework that is needed to identify classes of safety policies that have general utility and thereby it will facilitate reuse of software artifacts. It is obvious that some policies will be very application specific and unlikely candidates for kernel enforcement while other policies will have some general applicability. The selection of policies for enforcement in a general-purpose safety kernel architecture is based on criteria that consider both the criticality of a policy and the effects of kernel enforcement on the simplicity and verifiability of both the application software and the safety kernel.

Although the safety kernel architecture that we are developing is quite general, an *instance* of the kernel will have to be configured for a given application. A major issue in the development of the architecture is how best to parameterize it for use with a range of applications. In our design, constant and code fragment parameters are read by a mechanical translator that produces an instance of the safety kernel.

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.
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.

One concern with the safety kernel concept is that it might not be feasible to develop a kernel with significant general applicability. Although this is not consistent with at least some of the experience to date with security kernels nor with our initial results, this would not invalidate the first four benefits and would only partially negate the benefit related to reuse of the safety kernel. In this case, the level of reuse would shift from the level of safety

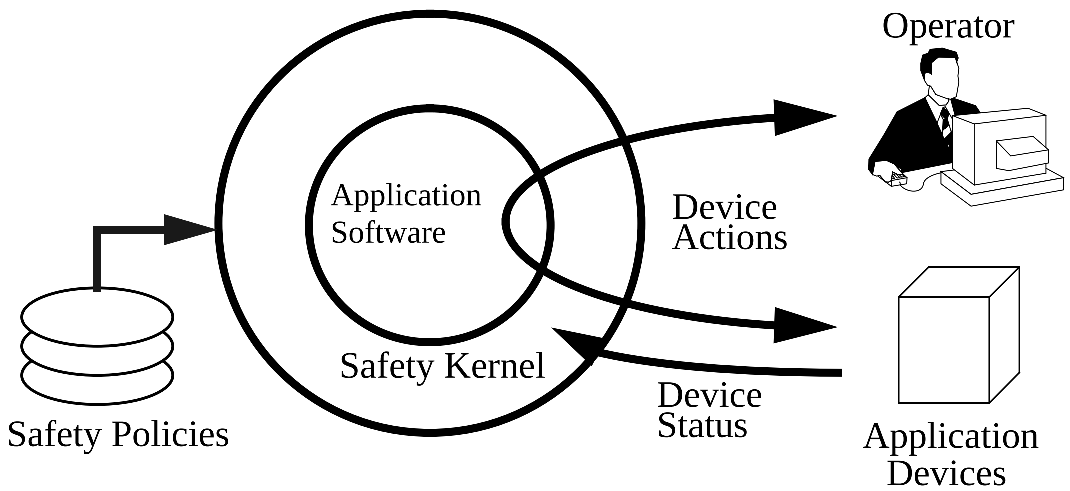

Fig. 2. Safety Kernel Concept.

kernel modules and mechanisms to the level of a design and process for instantiation of a safety kernel. This is precisely the level of reuse that occurs with security kernels. A security kernel is tailored for the processor it executes on, 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 would have solutions configured for each unique instance. In spite of this application dependence, security kernels are utilized for precisely the reasons shown above.

III BACKGROUND

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, were almost impossible to verify. Over time, however, the security kernel and other concepts have been developed that have made the development of secure systems more general, repeatable, and more amenable to verification [1].

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 methods that have some of the features of a safety kernel. In particular all utilize a relatively small, critical component to provide support for software safety.

Leveson, et al. [6] first used the term “safety kernel” to identify a concept based on a centralized location for a set of safety mechanisms. The mechanisms are used to effect error detection and recovery policies established for a given system. This safety kernel is more like an operating system kernel than a security kernel. It is a collection of potentially useful routines that are either invoked by the application or by watchdog timers. There is no notion of enforcing policies, and, because the role of the kernel is that of a monitor, it is not in a position to ensure that policies will be met. Thus the term kernel, as used in that work, was not directly analogous to its use in the security context.

A report by the NATO ad hoc Working Group on Munition Related Safety Critical Computing Systems [8] mentions a safety kernel that it defines as follows:

> 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.

This safety kernel concept is similar in function to the one proposed by Leveson, et al, serving as a monitor with no clear means of actually enforcing safety policies for a system.

Neumann [7] 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 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. A safety hierarchy could also incorporate criticality with the most critical properties enforced at the lowest levels.

Rushby has made the strongest theoretical argument for the development of a safety kernel [9]. In the process, he has more clearly defined the role of a safety kernel. 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. He observes 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. For a security kernel, *P* might be “no read up.” This is a “for all” policy that ultimately depends on kernel operations and is therefore enforced by a security kernel. 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 provide. Enforcement of positive behaviors is much more doubtful because it is difficult to ensure the proper use of functions that are provided. Rushby proposes a design for a safety kernel based on a separation kernel that restricts and monitors communication between modules to enforce policies such as those specifying valid sequences of operations. The design also includes resource managers that are responsible for the safe operation of devices.

Each of the safety kernels 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 because of the implementation and verification benefits. Rushby has identified the essence of the role of a safety kernel as an enforcer of safety policies. The logical progression from this abstract, theoretical description of a safety kernel is to evaluate the merits of a safety kernel through the development of the concepts introduced here and the instantiation of the safety kernel with a real safety-critical system. This evaluation is the subject of the remainder of this paper.

IV A SAFETY-CRITICAL APPLICATION

The safety kernel is being developed in the context of a case study with the *The Magnetic Stereotaxis System* (MSS). This is an investigational device for performing human neurosurgery being developed in a joint effort between the Department of Physics at the University of Virginia and the Department of Neurosurgery at the University of Iowa [10]. As a real, safety-critical application, the MSS forces complex, practical issues of software safety to be considered. In addition, it also serves as a target for implementation of a prototype of the safety kernel.

The MSS operates by manipulating a small permanent magnet (known as a “seed”) within the brain using an externally applied magnetic field. By varying the magnitude and gradient of the external field, the seed can be moved along a non-linear path and positioned at a site requiring therapy, e.g., a tumor. The device can be used for hyperthermia by radio-frequency heating of the seed from an external source or for chemotherapy by using the seed to deliver drugs to a site within the brain. 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 preliminary animal trials using the prototype is expected to begin in the near future.

Fig. 3 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 super-conducting electromagnets. Under the direction of the computer, power supplies and cur-rent controllers regulate the electric current in the electromagnets, thereby producing the magnetic field that acts on the seed. Along each axis perpendicular to the patient’s body, an

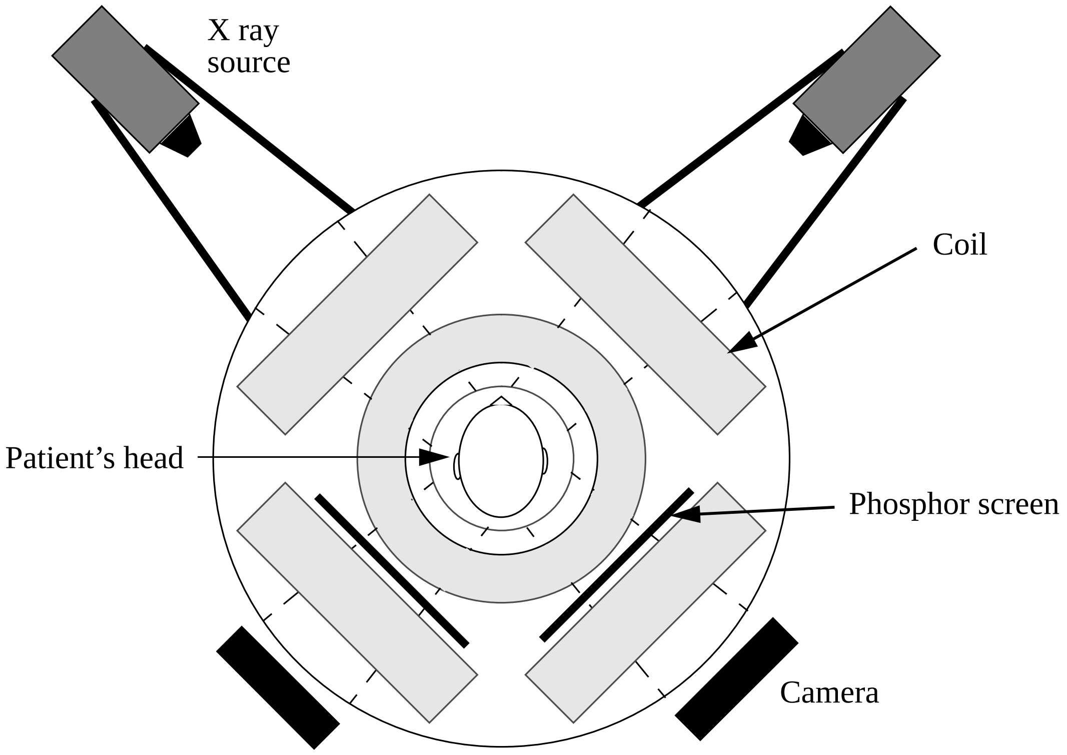

Fig. 3. View Of The MSS From Above Patient’s Head.

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.
- 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. Our safety kernel architecture is being prototyped with the MSS and will attempt to address all causes of patient injury. Throughout the rest of this paper, examples from the MSS will be used for illustration.

V SAFETY KERNEL ARCHITECTURE

The first problem that has to be addressed in the design of a safety kernel is to determine exactly how it will interact with a large amount of what is, by definition, "untrusted" software. As our understanding of this interaction has evolved, so has the role of the safety kernel in the overall system architecture. Our initial concept was, upon reflection, impossibly naive. We expect that the third (current) concept will satisfy the diverse goals. This section describes the three system architectures that have been considered.

Minimal System Architecture

Initially, we took the position that the kernel had to execute directly on the bare hardware. It seemed obvious that this would be essential in order to achieve the necessary control. Similarly, in order to limit complexity, we also took the position that a single processor architecture would be required. The complexity introduced by the concurrent operation and communication of a distributed or otherwise parallel system would obviously be

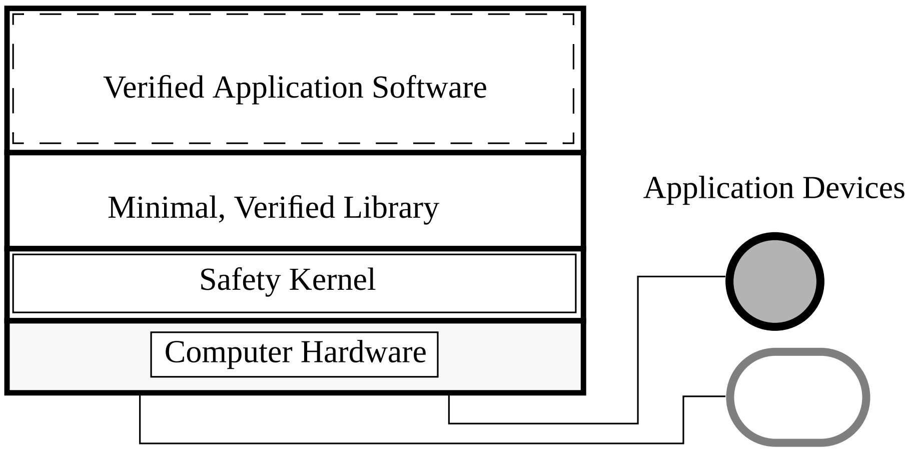

Fig. 4. Minimal System Architecture.

unacceptable. Our original concept was of an application organized as a single program executing on a platform provided by a simple safety kernel executing on bare hardware. Fig. 4 shows the simple initial architecture that we assumed.

In practice, such an organization is neither practical nor necessary. We discovered that the approach was not practical when we began the design of a kernel that was bound by our initial assumptions. The kernel was to be used by the MSS and the functional requirements of the application dictated that the restricted design could not support the application. Like many systems of its type, the MSS has computing resource requirements that cannot easily be met by a single computer. These requirements derive mainly from the need to generate elaborate bit-mapped displays that depend on extensive computation both to produce the necessary data and to manage the display hardware. We were forced to consider a distributed implementation as a result.

Partitioned System Architecture

The replacement for the minimal system architecture that we hypothesized initially was the partitioned system architecture shown in Fig. 5. This is a distributed architecture in which the application software is partitioned so that the application control software is executed on top of the safety kernel and the rest of the application is executed on other computers. This arrangement provides flexibility in the configuration of the hardware for a system. It also enables much of the complicated application software and support software to be executed on hosts other than the safety kernel host. As a result, the safety kernel can still be a relatively simple structure that is implemented on the bare hardware. The one architectural feature added to the safety kernel for this configuration is the remote safety kernel elements that deal with issues of liveness and communication (see Fig. 5).

The primary concerns with this architecture are the essential operating system functionality that even a simple safety kernel would need to provide. In particular, memory management, processor control, file system services and network communication are needed. Although this would not be prohibitive for some systems, it is a significant undertaking and most importantly, there is some question as to whether it is really necessary.

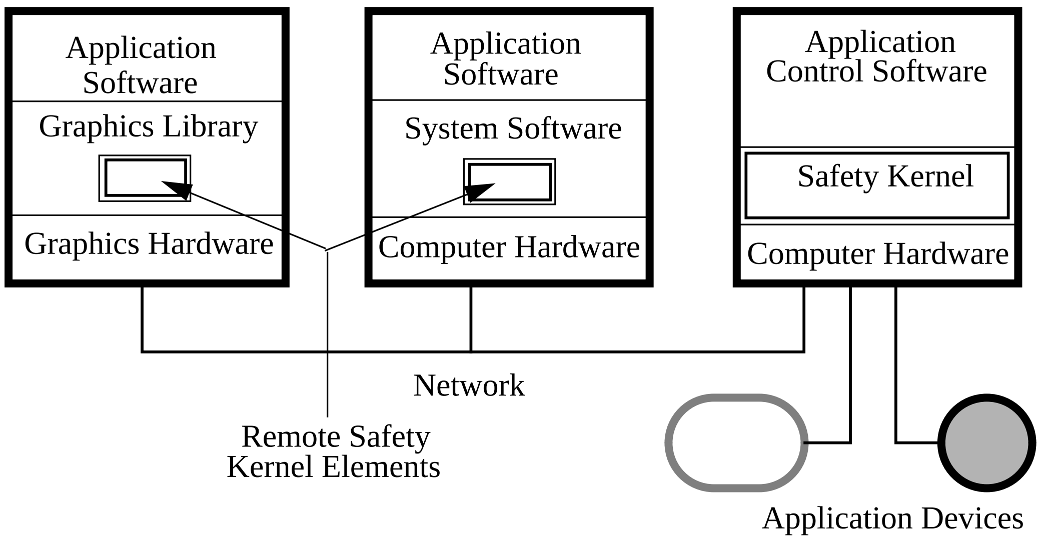

Fig. 5. Partitioned System Architecture.

Even in systems where extensive effort could be given to the implementation and verification of the essential operating functionality as part of a safety kernel, the complexity would be such that the original goal of localizing the verification effort in a small simple kernel would be lost. It is very unlikely that the implementation of such services could ever be verified satisfactorily.

Revised Partitioned System Architecture

The third and final architecture is similar to the partitioned architecture except that the safety kernel now operates on top of an existing operating system (see Fig. 6). It is assumed that this operating system might fail and so the safety kernel must be able to deal with such failures. In addition, the safety kernel must interact via an unreliable network with system and application hardware of an unknown nature. Provided the safety kernel can be designed to respond successfully to support software failures, this architecture has the benefit of a simple safety kernel that does not need to provide system functionality and is much less hardware dependent than in the previous architectures.

Satisfactory operation of the safety kernel on top of the system software relies on the assumption that there will be no malicious attempts to compromise safety, e.g., by bypassing the kernel. A security kernel could not be implemented in this manner because it could be easily circumvented. A plausible design for the safety kernel would be to have it run as a user-level server process. Such a design would isolate the safety kernel from application software failures and permit it to monitor the application devices independently. This arrangement would be transparent to the application software in that the interface to the kernel would be identical (except for initially establishing communication) to what it would be if the safety kernel were implemented directly on the bare hardware.

It appears initially that the architecture shown in Fig. 6 is unable to provide the kind of protection that is sought from a safety kernel. Since there is a significant piece of untrusted software beneath the kernel, surely it is possible for that software to fail and prohibit the kernel from enforcing safety policies. This is indeed plausible but only under quite limited circumstances. Most failures of the software beneath the kernel can be dealt with using techniques such as coding to detect data corruption, liveness checks and time-out mechanisms to ensure timely direction of devices, and independent sensors for closed loop control of devices. The issue of dealing with untrusted support software is discussed further in section VII.

The one architectural feature added to this system is the *safety kernel monitor* running on a minimal separate hardware unit. The safety kernel monitor possesses the minimum functionality required to restore the system to a safe state. Its operating mode is to execute this functionality automatically unless it receives timely safety kernel "heartbeats" to indicate that the safety kernel has not been deactivated by a failure of the system software.

For some safety-critical systems it might be essential to utilize a design where the safety kernel is constructed on the bare hardware (see Fig. 5). This could be true for systems where system software needed to be highly reliable or where the safety kernel monitor might be sufficiently complex that it is more cost effective to incorporate that functionality into the lowest level of a system kernel. However, even if this approach were taken, the architectural configuration of a safety kernel monitoring the activities of custom built system software is still applicable because of the complexity required to do processor and memory management and to provide file system and network services. In this case the safety kernel monitor would be within the kernel, but it would still serve to ensure the liveness of the safety kernel on top of the rest of the system software.

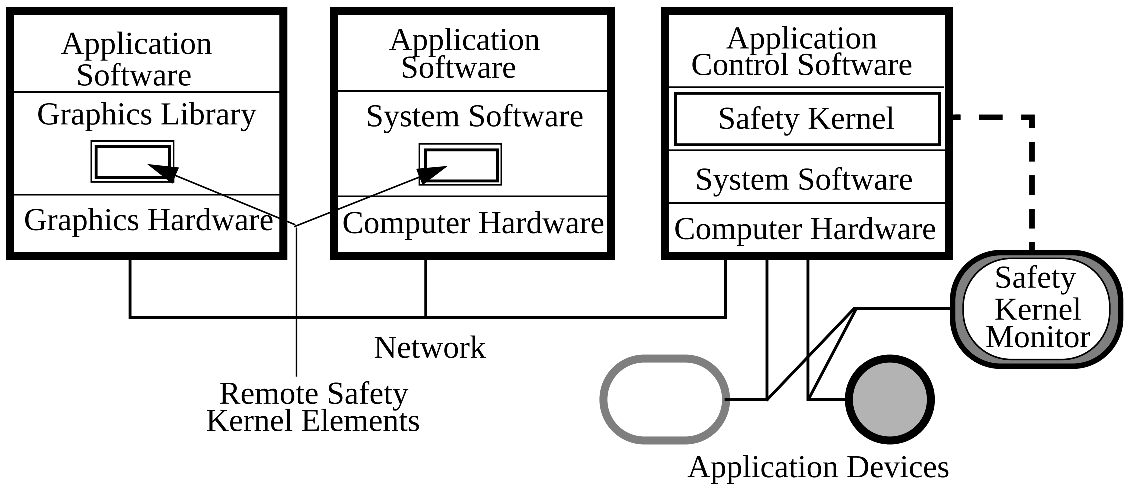

Fig. 6. Revised Partitioned Aystem Architecture.

VI SAFETY POLICIES

What form should useful safety policies take and how are they determined? We examine that issue in this section and give examples showing how the specific policies were selected for our particular kernel design and how they are applied to our case study application. An overall taxonomy of safety policies breaks them down into areas based on enforceability, generality characteristics, and functionality. We examine these three areas in turn.

Safety Policy Enforceability

Safety policies fall into two classes. Informally, the distinction between the two classes derives from the fact that certain policies can be enforced by the kernel without interaction with the application and others cannot. The first class, referred to as *kernel-enforced* policies, consists of those policies that meet the following two conditions identified by Rushby:

1. The policy of interest at the system level must ultimately depend on operations at the kernel level precluding the application from circumventing policy enforcement.
2. The policy must be a "for all" policy that holds for any combination of kernel operations.

As Rushby noted, the safety kernel enforces policies in the first class by regulating the operations that it provides to an application. These kernel-enforced policies are ensured by the safety kernel independent of the implementation of the application software. This is analogous to a security kernel where, in principle, verification of the security kernel ensures the security of the system. Examples of this type of policy are the following:

An interlock policy:
Device A and device B should never be on at the same time.

A cumulative device activity policy:
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 this with no regard to application circumstances, and that 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 necessary conditions.

The second class, referred to as *application-enforced* policies, is made up of those policies that do not meet one or both of these conditions. These policies are supported rather than enforced by the kernel. To achieve enforcement of the policy, it is necessary to ensure by some form of verification that the application is using the kernel facilities correctly. This differs significantly from the security kernel analogy because the role of the kernel in this case is to support an application in achieving *its* goal in a non-malicious environment. Examples of this type of policy are 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 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. This is because the kernel is not required to act with this internal operation as it would be if the policy was enforcing some constraint on an output over which the kernel had absolute control. Extensive verification would be necessary to demonstrate exclusive and correct usage of the operation. Alternatively, the safety kernel could provide a check on the result, but once again it would need to be shown that the check was invoked for each iteration of the calculation.

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

Safety Policy Characteristics

A safety policy is neither inherently kernel-enforced nor inherently application-enforced. To see why this is so, suppose that all of an application were moved into and became part of the safety kernel, and that the one command was "start". In that case, all of the policies would be kernel-enforced because there would be nothing but the kernel. This would certainly be counterproductive, however.

The determination of which policies will be enforced by the safety kernel and which will be enforced by the application is based on maximizing the benefits (as described in section II) of a kernel architecture. For example, enforcing a particular policy might provide support for software safety, but result in undue loss of generality and increased complexity of verification. Definition of the set of safety policies provided by the kernel described in this paper was based on the following considerations:

- Is it important to enforce this policy independent of the application?
- Is the policy applicable to a range of safety-critical systems?
- How would kernel enforcement of this policy impact the simplicity and verifiability of the safety kernel?
- How would kernel enforcement of this policy impact the simplicity and verifiability of the application?

Safety Policy Functionality

Safety policies are intended to enforce safety. Necessary policy functionality, therefore, depends to a large extent on the kinds of things that have to be done to make a system safe. To the extent possible, the selection of policies for a general, reusable kernel should be based on a comprehensive precise notion of safety that satisfies a wide range of system needs.

Returning briefly to the analogy with security, we note that overall system-level security goals derive from some functional requirement that the system's developer has. For example, it is common for a system to grant different access rights to different classes of users. Achieving the desired access restrictions is sufficiently important that extensive verification of the kernel is undertaken along with a proof that the desired security requirement holds.

There is no formal notion of safety that has the same degree of rigor as that found in the statement of security. This precludes a comprehensive mathematical attack on the problem of safety in the same manner that is undertaken with security. Nevertheless, it is possible to develop elements of overall system-level safety that can be analyzed in much the same way as is done with security. This amounts to showing that a system has certain desirable properties which one would like to see in a safety-critical system but prevents the development of any kind of proof of total "safety".

To permit a systematic analysis of the functionality of safety policies, we have divided the required functionality up into four areas. These four areas are:

1. Control of peripheral devices.
2. Application software activity.
3. Device failure detection.
4. Response to failure.

The first two areas are policies that regulate actions of the application and are invoked when the application software makes a request to the safety kernel. The third class of policy relates to the role of the safety kernel as a monitor of devices. The fourth class of policy is required to respond to the failure of either the application software, support software, or a device.

Peripheral Device Control

The first set of the safety policies dictate the manner in which devices may be operated by the application. Below we list general categories within this area of functionality. We note also that all policies within these categories are kernel-enforceable. Each category is accompanied by an example of a policy instantiated for the MSS (for the sake of brevity, we have omitted the general statement of the policies that the kernel would provide):

- State-command restrictions.
  When a request is made for a device action, it is possible that the safety of the system will depend on the present state of the device. Therefore, it is important

to be able to specify policies that dictate when an action may be executed as a function of the command state of the device.

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

- State-command timing requirements.
  For a given device state there can be requirements concerning the minimum and maximum times that the device can safely be in that state. The timing might also depend on the command that is being requested. For example, for a device in the off state, it would be reasonable to require a minimum time in the state when the requested command is on. The safety kernel will provide support for real-time systems by ensuring that either the application devices are controlled in accordance with timing deadlines or that actions are taken to respond to missed deadlines.

  MSS example: An X-Ray device must be in the "off" state for 0.2 s before the invocation of an "on" command.

- Multiple-device, state-command restrictions (i.e., interlocks).
  The safety of a device action can depend on the states of other devices. Some policies will specify combinations of device states where actions are precluded or permitted.

  MSS example: The two X-Ray sources must not be active simultaneously.

- Parameter checks.
  The parameters for a command must satisfy checks that are determined by the particular command, device state, and possibly the states of other devices.

  MSS example: The current requested of the electromagnet power supplies must be less than 100 Amps.

- Operational state conditions.
  For a given command and device command state, there may be restrictions that depend on the operational state of the device. For example, a device in an initialization phase might not be able to be operated until it had achieved its operational temperature.

  MSS example: The current in a coil must be less than 15 Amps when a request is made to change the polarity of the coil.

- Total duration/dose restrictions.
  The safety of some systems will depend on a device not exceeding its total active time or some function thereof.

  MSS example: The total X-Ray dose during an operation must be less than 100 millirem.

- Device control requirements.

Some higher-level, device control properties can be ensured by kernel control of devices (e.g., duration control for a device).

MSS example: The X-Ray sources will always be activated for a period of 0.05s.

Application Software Activity

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. For example, the calculation of output values for a device might have associated with it an effective reversal check. A safety policy might be that the output values should never be sent to the device without the reversal check being applied.

Much like the device control policies described above, application software activity policies will specify required sequences of actions, timing constraints on those actions, or particular checks or activities to be performed when an action occurs. For example, a policy might dictate that actions A and B must be performed prior to the execution of kernel activity X. If A and B are provided by the kernel, then this policy would be kernel-enforced. If A and B are application software operations then the safety kernel could require that it be notified of their invocation, but it could not actually guarantee that they had been executed. The problem is that the safety kernel can require notification, but there is no way for it to enforce that requirement. For example, the application software could easily notify the kernel that an operation had been performed without actually completing the action. This type of policy would require verification of the application software to ensure that notification occurred once and only once for a given action. A policy that required the loading of configuration data prior to operation of application devices, is an example of a potentially useful property requiring some verification of the application software.

In the example above, if actions A, B, and X could all be performed in one action, then the policy could be a device control policy enforced as part of a single kernel operation. This has the advantage of obviating the need for ensuring notification, since the actions would be performed each time X was invoked.

MSS example: Prior to charging of the electromagnets, a reversal check must be executed to ensure that the requested currents provide the desired force.

MSS example: A seed movement must have been requested each time the coils are charged.

Device Failure Detection

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 neglects to perform an essential operation. Both types of failures are detected through an inconsistency in the observed and predicted state of the device. Therefore, there is a component of the safety kernel that has the exclusive task of periodically sensing the state of the devices and comparing this state to the expected state for the device. Two classes of device failure detection policies are enforced.

- State consistency:
  The operational state and the expected state for a software-controlled device should be consistent. The means of evaluating consistency and determining the expected state are defined for each particular device and for each device state.

  MSS example: The state of the X-Ray must match the most recent state commanded by the software.

  MSS example: While the coils are charged, the X-Ray sources must be activated at least once every 2.0 s.

- Observed state consistency
  To obtain statistical confidence in the operational state information for a device, there will likely be two or more sources of operational state information for a device. Since failure of these sources could lead to problems in evaluating state consistency, the safety kernel must implement policies that document unacceptable discrepancies between sources.

  MSS example: The current values reported by the current controllers and the independent sensor must differ by less than 5.0 Amps.

Response To Failure

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 (i.e., the state of the application devices). The response will be specified in safety policies that dictate the action to be taken for a particular failure and system state.

MSS example: If an X-Ray fails so that it cannot be activated and the coils are in a discharged state, then the X-Ray sources and the coils must be deactivated.

MSS example: If the X-Ray sources are on when they are expected to be off, then the power to the X-Rays must be interrupted and the coils discharged.

MSS example: If a coil fails while charged, then all of the coils must be quenched and the X-Ray sources deactivated.

VII IMPLEMENTATION ISSUES

From the previous section, we conclude that, although many particular safety policies exist, in general they fall into two classes, kernel enforced and application enforced, that they have to be considered carefully with respect to their generality and applicability, and that their functionality lies within a small number of areas. Given this, the next issue to consider is techniques for implementation. In this section, we look at the following four issues concerning implementation:

1. How will an instance of the safety kernel be instantiated for a particular application?
2. How will kernel-enforced safety policies be specified and implemented?
3. How will the safety kernel support verification of safety policies?
4. How will the safety kernel deal with unreliable support software?

Instantiation of the Safety Kernel

The kernel concept and a substantial amount of the design and implementation are intended to be generally applicable to a wide range of safety-critical systems. To permit enforcement of safety policies that are tailored to specific applications, many aspects of the kernel will be parameterized. The parameter information will be incorporated into the kernel to produce an instance for a particular application.

Fig. 7 depicts the process we are developing for instantiating the safety kernel we are building. First, a set of safety policies are derived from the software safety specification. The process of derivation for our kernel is presently informal but the prospect of extracting them from a formal specification clearly exists. The safety policies are specified in the kernel configuration data using constant information, such as an acceptable state transition or a maximum value for a parameter, and code fragments to parameterize the general mecha-

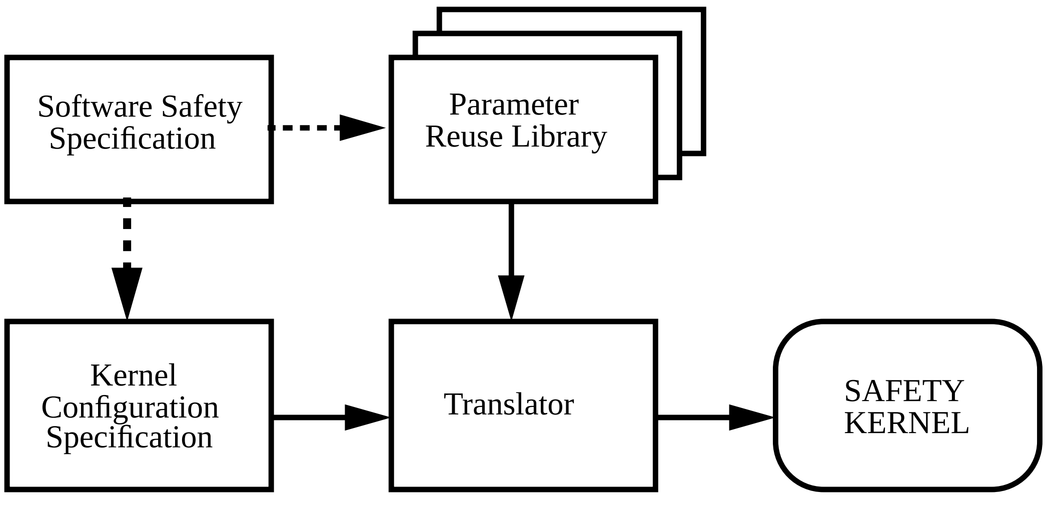
Software Safety Specification
Parameter Reuse Library
Kernel Configuration Specification
Translator
SAFETY KERNEL

Fig. 7. Instantiation Of Safety Kernel.

nisms provided. To facilitate reuse of the code fragments, a library will be included as a repository for certified code fragments that might be applicable to a range of applications. The configuration information is read in by a translator and will be mechanically incorporated into an executable safety kernel.

Specification and Implementation of Kernel-Enforced Safety Policies

With the kernel instantiation process outlined above there are two critical design issues. The first is the mechanism by which the various policies are specified. The second is the way in which the policy specifications are incorporated into an implementation of the safety kernel. These two issues are addressed below for a number of the classes of policies described in section VI.

Many of the device control policies are enforced at the time a request is made for a device action. As a result, these policies can be enforced through the use of a “filter” on the kernel operations. Included in this group are state-command restrictions, state-command timing requirements, parameter checks, and operational state condition checks. These policies are specified using a tabular format like the one shown in Fig. 8. The table has device states on the left side of the table and potential action requests across the top. The table is a representation of a finite state machine with the action requests representing the inputs to the machine. The transitions are recorded in the cells along with conditions that must be satisfied for the transition to occur. The states are the *command states* that result from action requests. Other elements of the state that might change without a direct action request are a part of the *operational state* of the device. For example, a motor might have command states on and off, but the speed of the motor could be a part of the operational state that would change without a direct command. In addition to holding the acceptable transition for a state, a cell might also contain restrictions on the operational state, parameters checks, and timing requirements. Logically, the state-action table will be utilized as follows:

1. When the safety kernel receives an action request for a device, it will

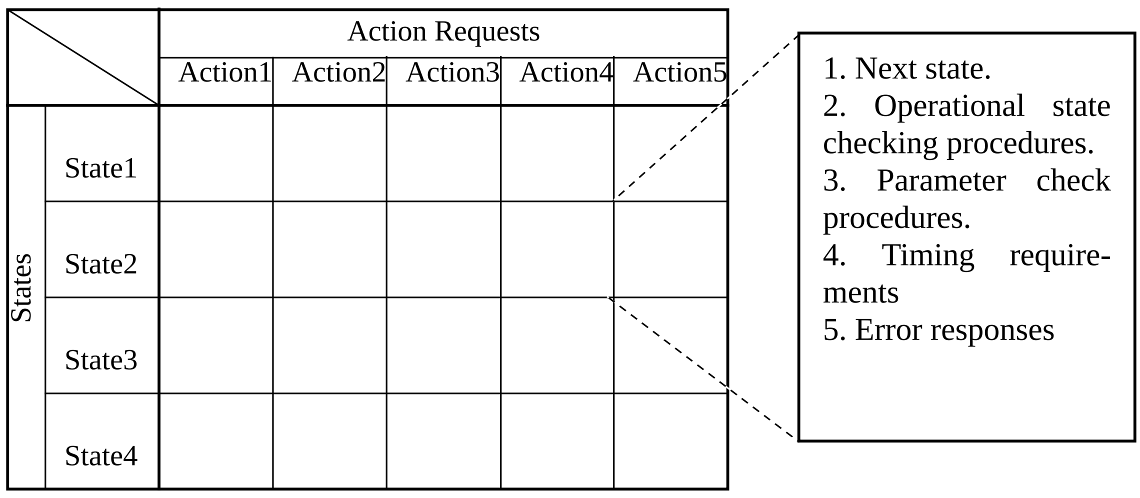

Fig. 8. Specification Of Device Control Policies.

first check to see if there is a transition specified for the device in its present state. If there is no legitimate transition, then a specified response procedure will be invoked.

2. If there is a transition, then that transition will be made only if any conditions on the operational state are met, if timing restrictions are satisfied, and if the checks on the action request parameters are satisfied. Otherwise, a specified response procedure will be called.
3. If the above are met, then the state of the table will be changed and the action request will be sent to the device.

The tabular specification is read by a translator that produces a procedure for each type of device action request. Within each procedure, actions and checks corresponding to a table cell are selected based on the present state of the device. When the checks are applied status values are produced that indicate whether the requested action may be carried out or if an error response is required. For each device the safety kernel maintains a record of the present state and the time of the most recent action request.

The specification and implementation of the application software activity policies is very similar to that for the device control polices. A tabular representation is used to specify acceptable command sequences and actions to be taken when a particular software activity occurs.

Another major set of safety policies are the device failure detection policies. These policies are primarily concerned with the monitoring of devices to ensure that the device state is consistent with the most recent command to the device. The policies for detection of device failure will be specified for each of the possible command states. The policies will determine the expected state of the device based on the present command state of the device and the time the state was entered. The policies will also dictate how the prediction should be compared to the actual device state to detect possible device failures and will designate which error response should be called in the event that an error is detected. The observed state consistency policies will be specified similarly. Because the devices will be monitored on a periodic basis, each command state will also have associated with it a time that determines how frequently the device should be monitored.

This specification of the device failure detection policies will be translated into a monitoring procedure for each device. Within the procedure the appropriate state checks will be selected depending on the command state of the device. The safety kernel will invoke these device monitoring procedures at the intervals specified for the present state of the device.

The specification and implementation of the error response policies would be quite similar to the device error detection policies. Responses would be provided for each type of error and for various combinations of the device states that would need to be considered in responding to the error. An error response is chosen based on the type of the error and the present state of the system.

A consideration with the error response policies in particular and with policies parameterized with code fragments in general is whether the kernel or the application should execute the code fragment. Code that is executed within the kernel would need to be verified for certain properties, so that it would not fail and interfere with the operation of the kernel. In many cases the best scenario would be to have the kernel identify the need to run a particular routine and then signal the application to actually execute the operation. This is especially critical in cases where the application is being reconfigured due to some device failure detected by the kernel. In this case, there is no way that the kernel could actually execute the software necessary for continued operation. Executing responses within the application software would enable the kernel to continue to monitor the application and obviate the need for the safety kernel to monitor itself. Even in this case, it still might be desirable to have the safety kernel perform the immediate response operations, and then turn continued operation over to the application. Simple checks and device error detection routines will be executed from within the kernel.

Verification of Kernel-Enforced Safety Policies

The safety policies that are enforced by the kernel are ensured independent of the implementation of an application. Thus, verification of kernel-enforced policies involves verification of the safety kernel and not the application software. That is not to say that the application will not need to be verified, but it will not be necessary to verify it to ensure enforcement of the kernel-enforced safety policies. Verification of the kernel is focused on two areas. The first area is the verification of the general mechanisms of the kernel and any tools used to produce an instance of the kernel from configuration data. The second area is the verification of the configuration data, including constant information and code fragments, that specify the policies to be enforced by an instance of the kernel.

In theory the kernel mechanisms and support tools will be sufficiently general that they can be reused with a range of applications. These elements of the kernel will be verified once and then utilized multiple times. Verification of the safety kernel mechanisms and support tools will be based on software inspections, testing, and formal verification of essential properties.

Because it must be repeated, the major concern with the verification of an instance of the safety kernel is making certain that the configuration of the safety kernel accurately reflects the requirements given in the software safety specification. Verification of an instance of the safety kernel involves demonstrating that the given parameters correctly specify the desired safety policies and that the code fragment parameters exhibit essential properties. Ideally, the verification of the use of parameters in the configuration of an instance of the kernel could be obviated by developing a precise mapping from the software safety specification to the configuration data. In lieu of this, the emphasis will be on specifying kernel configuration safety policy data in a manner (e.g., the structured tables described above) that simplifies verification with the software safety specification.

Code fragment parameters will need to be verified for properties such as bounded execution time, freedom from memory leaks, and correct implementation of the particular check or action. Since these fragments are a part of the kernel, the verification techniques will be the same ones identified above for verification of kernel mechanisms. The code fragment reuse library is intended to support reuse of these parameters and thereby reduce the verification required for an instance of the safety kernel.

Dealing with Unreliable Support Software

The safety kernel is situated between the application and a significant amount of support software. In particular, the kernel depends on an unverified operating system and network communication subsystem. Clearly, the kernel must be implemented in such a way that it can deal with failures in this support software.

One way to deal with the dependence on support software is to reduce the amount of software upon which the kernel depends. This is one of the goals of the partitioned architecture. Moving all of the application software except the control software to remote hosts is intended to simplify the functionality required of the support software. As a result the safety kernel and the critical application control software depend on a far smaller amount of untrusted software. The partitioning also permits the hardware and software that control application devices to be physically isolated from much of the application software, thus reducing the possibility of harmful interaction between critical device control software and other modules.

Although reducing the amount of support software is a good first step, there are still several general classes of support software failures with which the safety kernel must be concerned including (but not limited to):

- Incorrect execution of device control commands, including spurious commands to devices.
- Failure of support software operations serving the kernel or application software.
- Failure of software on remote hosts in the partitioned architecture.
- Failure to provide timely service.
- Corruption of data.

The failure of the system software in performing device control commands, such as not turning on a device or turning one on when not required, is dealt with by the device failure policies described above. This is possible because, from the perspective of the safety kernel, failure of the operating system in this manner is *indistinguishable from a failure of the device itself*. In fact, this type of operating system failure emulates the failure of the device [5]. As a result, by being prepared to respond to device failures, the kernel is also able to respond to this class of operating system failure. If for some reason the monitoring was not sufficient, extra assurance of correct system execution of device commands would be provided by an acknowledgment system. With this approach, a device would be required to echo any request for action back to the safety kernel and wait for an acknowledgment for action.

The failure of an application initiated request to the operating system appears identical to the failure of the application itself. Therefore, the kernel requires no extra facilities to prevent this type of failure from impacting system safety. The failure of a safety kernel request for an operating system service is a problem only in the event that the failure is not detectable and the system continues to operate with corrupted data. In the worst case, detectable failures can always be dealt with by invoking the safety kernel monitor referred to previously and described in detail below.

The partitioned safety kernel must be able to deal with software having undetermined properties on the remote hosts. This software will include operating systems, communications packages, user interface libraries, etc. The concern with this software is not that it might fail to execute — the safety kernel will assume that this is likely and will be prepared to invoke a recovery mechanism if necessary. The real concern is with those actions that might adversely impact the application control software. For example, remote software could fail in a manner that would cause the application control software to be flooded with messages. Although these messages might have legitimate content, they might cause the application control software to fail. The safety kernel will deal with this by monitoring message traffic to ensure that the communication to the kernel conforms to specified safety rules. Transmission of garbled messages might also result in a failure. This is a relatively simple problem that can be dealt with by suitable redundancy in the message data.

The failure to provide timely service can result from support software domination of resources or from a fatal operating system error. In either case, there is no means for the safety kernel itself to respond to the problem, because it is not receiving the services it needs to continue operation. This class of failure renders the safety kernel and all other software either inoperative or unable to perform in a timely manner. As a result, this type of failure (and only this type of failure) must be dealt with by an external entity that is capable of detecting the failure and restoring the system to a safe state. This is the *safety kernel monitor* and it could be either purely a hardware device or a dedicated computer. It would operate as a watchdog timer that required periodic liveness signals from the safety kernel. In the event that the liveness signal was not received on time, the monitor would execute a routine to restore the system to a safe state.

Corruption of data occurs with data stored in a file system or handled by the network (it is assumed that corruption of data in memory will be detected by the hardware and will either be corrected or cause the safety kernel monitor to be invoked). Corruption of data will be dealt with by requiring that all data utilize some redundant information such that corruption can be detected with some appropriate probability.

VIII CONCLUSIONS

Software in any non-trivial safety-critical system must be assumed to contain faults that could compromise system safety. The complexity of the software, and the lack of completely effective techniques for either preventing the introduction of or removing these faults combine to produce this problem. The MSS case study provides a ready example of a complex software-controlled system. The safety kernel attempts to deal with a part of this problem by providing execution-time enforcement of several classes of safety policies in a relatively small, simple structure. This structure simplifies the implementation of the policies and facilitates their verification. Situated between the application software and the physical devices, the safety kernel enforces these policies regardless of the actions of the application software.

A partitioned architecture is utilized to enable the safety kernel to deal more easily with a large amount of untrusted software and to meet the diverse computer resource requirements of typical safety-critical systems. An early minimal architecture was rejected based on the experience with the requirements of the MSS. The partitioned architecture emphasizes the separation of the control software from the rest of the application software in order to permit a relatively simple safety kernel and support software. The safety kernel operates on top of existing, unmodified support software in this architecture because it is able to detect and respond to most system software failures. This frees the safety kernel from providing typical system services and enhances the portability of the kernel. Failures of the support software that completely inhibit the kernel are detected by the safety kernel monitor that is equipped to restore the system to a safe (perhaps inactive) state. Although some systems might require a safety kernel built on the bare hardware, we conclude that the model of the system software sandwiched between a safety kernel and a safety kernel monitor is widely applicable.

The safety kernel architecture provides a heretofore unavailable framework for the abstraction of safety policies that are applicable to a range of safety-critical systems. Using this framework several classes of general, kernel-enforced safety policies have been identified. These kernel-enforced policies ultimately depend on kernel operations and are true for all combinations of kernel operations — the two conditions identified by Rushby. Policies are not inherently kernel-enforced, but are selected for kernel enforcement based on criteria that consider the impact on the simplicity and verifiability of both the safety kernel and the application software. The relevance and utility of the identified classes has been confirmed through their application to the MSS.

Although the safety kernel is a general architecture, an instance of the kernel will be configured for a given application. The issues with configuration of the kernel are how to facilitate verification of an instance of the kernel and how best to promote its reuse by a range of applications. The safety kernel will enforce safety policies that are parameterized by constant information and code fragments. The parameterization enables the policies to be quite general and the structured format of the specifications will facilitate the analysis of the kernel configuration. The parameter information will be integrated into the kernel framework by a mechanical translator.

The feasibility of the safety kernel concept and all the design and implementation issues described here are being evaluated during the development of a prototype with an instantiation targeted at the MSS. The implementation of this prototype and the associated tools is incomplete. The MSS application software is being developed in parallel and is also incomplete at this time. This paper reports our results to date obtained with this development activity.

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.

REFERENCES

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

2. Anderson, T. Ed., *Safe and Secure Computing Systems*, Blackwell Scientific Publications, 1989.

3. 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, pp. 3-12, January 1993.

4. Garman, J. R., “The Bug Heard ‘Round the World,” ACM Software Engineering Notes Vol. 6-5, pp. 3-10, October 1981.

5. 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.

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

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

8. 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.

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

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

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