Software Safety: A Formal Approach

John C. Knight
Darrell M. Kienzle

Computer Science Report No. TR-92-03
January 20, 1992

Abstract

There are many computer applications in which safety and not reliability is the overriding concern. Reduced, altered, or no functionality of such systems is acceptable as long as no harm is done. This report is concerned with the role of software in such systems and the definition of what it will mean for software to be viewed as safe. A *precise* definition of what software safety means is essential before any attempt can be made to achieve it. Without this definition, it is not possible to determine whether a specific software entity is safe. Informal, intuitive notions of software safety must be rejected if for no other reason than to protect the legal interests of the software engineer.

Software must be viewed as merely one of many components that make up a system. In the overall system context, software is no different from any of the other components of which the system is composed. Viewing software as a system component, a definition of software safety based on the establishment of precise specifications for the software's response to its own failure and to the failure of other components is presented. The definitions presented here define software to be safe if it complies with these specifications. A consequence of the definition is that the software engineer is freed from responsibilities other than the correct implementation of certain parts of the software specifications. This facilitates placement of responsibility in the event that an accident does occur. A case study is presented, demonstrating the feasibility of these definitions through the creation of formal software safety specifications for a software-controlled, safety-critical surgical device.

Acknowledgments

We would like to acknowledge Kevin Wika, Rogers Ritter, George Gillies, and Mike Lawson for providing access to the Magnetic Stereotaxis System, and for aiding in the system safety analysis. Particular thanks to Kevin for all of his assistance with understanding the prototype control software of the MSS and its safety ramifications.

This work was funded in part by NASA grant # NAG-1-1123.

Table of Contents

List of Figures

Chapter 1 - Introduction

A variety of issues are forcing many engineers building critical applications to consider the benefits of replacing mechanical and analog equipment with digital systems. For example, digital emergency shutdown systems are being developed for nuclear power plants [1] and digital flight control systems are either in use or are being planned for many commercial and military aircraft [2]. As more critical systems become dependent on computer control, the likelihood of a software fault causing physical damage or endangering human life increases.

Compounding the problem is the fact that mechanical and analog systems have demonstrated extraordinary reliability. Mechanical and electrical engineers have devoted considerable effort to determining the failure characteristics of the components they use. Combined with appropriate models, this information leads to useful predictions of failure rates thereby enabling effective designs [3, 4]. Any digital system must be similarly reliable to be considered an acceptable replacement. For digital systems that include software, the software also must achieve this level of reliability. In this paper the meaning of safety as it applies to software is discussed.

In everyday use, the term "safety" is subjective. Each observer has his[1] own definition of what constitutes unsafe behavior. Informally though, most people agree that a system is safe if it does no harm even if it provides less than complete functionality. Thus, operation in which service subsequent to a failure is incomplete but where no harm is done is generally viewed as safe. This notion of safety is very important and is clearly different

---

[1]For simplicity, throughout this report the male pronoun is used in the generic sense to mean either male or female.

from the notion of reliability. In an informal sense, safety is a "subset" of reliability and might, as a result, be easier to achieve than reliability. Also in an informal sense, safety is vastly more important that reliability in many systems. For example, in most cases it is merely an inconvenience if a car fails to start although the car would be viewed as unreliable. It matters a great deal if a car fails to stop since then the car is unsafe. The distinguishing factor in these cases is the hazards associated with each type of failure.

The possibility of erroneous software causing harm is very real, and this situation has lead to the appearance of papers in the literature discussing the safety issues raised by software [5-13]. Similarly, several standards that purport to address the development of safe software have been developed. In the informal sense just mentioned, the notion of safety might be applied to software with the interpretation that software is safe if it does no harm without concern for the provision of complete functionality. Again, it might be the case that developing safe software is simpler than developing correct software.

Much of what has been written about software safety has been within a framework of systems engineering rather than software engineering [6, 8, 11]. The rationale behind this approach is that safety cannot be assured without examining the software within its systems context. Although this is a valid and important argument, it tends to obscure the demarcation between systems and software engineering. This demarcation is important since they are separate fields whose practitioners have different skills. In addition, this approach does not permit a clear assignment of responsibility to be made. The software engineer is faced with a confusing set of concepts and questions that do not permit him to be certain of exactly what his role is. More importantly, the software engineer is unable to be certain of whether the software he has engineered will be viewed as "safe" by the community. To deal with this situation, the general approach developed to date needs to

be formalized to include a more precise definition of the role of software engineering in safety-critical computer applications.

System-safety engineers have developed a precise terminology and an extensive body of theory to support system-safety analysis. This terminology and theory permit precise, formal analysis of safety at the systems level. In particular, there is a formal notion of what constitutes a safe system. It is not sufficient to view software safety in anything less than a similarly formal way. A precise framework for discussion has to be developed and agreed upon. Such a framework must permit an unambiguous assessment of whether a particular software entity is safe or not in a formal sense. In a world in which public exposure to computerized devices is large and increasing, it is essential that developers and regulators know exactly what the term *safe software* means. Developers must know what is required of them to permit their development activities to be judged as competent (software) engineering. Regulators need to be able to specify what constitutes safe software in order to protect the public. Finally, a precise framework permits the assignment of responsibility in the event that a software entity is deployed that is unsafe according to the framework.

Here we propose precisely such a framework by providing formal definitions of software safety and related terms. The fundamental separation between software and systems engineering concerns is stressed. This separation offers benefits to practitioners in both fields. It permits systems engineers to free themselves from software details and treat software as they would any other component in building a safe system. It permits software engineers to free themselves from system concerns and to concentrate on software issues. In effect, this separation permits the role of the software engineer in the development of safety-critical systems to be defined.

A case study that tests the viability of these definitions is also presented. This study focuses on the determination and specification of the software safety requirements for a software-controlled, safety-critical surgical device. The results of this study demonstrate that software safety requirements can be determined at the level of systems analysis, without software engineering involvement. Furthermore, an exact process for determining the software safety requirements is documented.

Throughout this report the terms systems engineer and software engineer are used to mean those undertaking the systems engineering and software engineering tasks, respectively. Naturally, many individuals might be involved in each task or a single suitably-trained individual may perform both tasks. It is essential, however, that these two domains be kept logically separate no matter what the associated staffing is. In particular, the requisite documentation must be developed in all cases even if a single individual is performing both functions.

Chapter 2 - Software Safety

2.1 System Safety

The field of systems engineering has distanced itself from the subjectivity associated with safety by formalizing the everyday notion [3, 4]. This has been done by introducing the concepts of *hazard*, *risk*, and *acceptable level of risk*. A hazard is a situation that could lead to harm and therefore must be avoided. The hazards associated with a system are determined in a process called *hazard analysis*. Risk is a combination of the probability that a hazard will occur and the severity of its consequences. In a probabilistic sense, the risk associated with a hazard can be thought of as the expected loss. System safety is concerned with keeping the total risk to an acceptable level[2].

In a formal sense, a system is considered safe if it can be demonstrated that the system does not surpass the acceptable level of risk for any identified hazard. If a system performs a harmful action, but the action was not identified as hazardous behavior, the system is still acting safely in the sense of the systems-engineering formalization. Similarly, if the probability of a hazard is sufficiently low as to make the risk acceptable, the system is safe despite the fact that the hazard can still occur. These discrepancies from the everyday notion of safety are necessary by-products of the formalization of an informal concept.

Despite the formalization, the determination of system safety remains dependent on human judgment. All hazardous behavior must be determined, the probability that such behavior will result in an accident must be correctly estimated, and the risk calculated. Finally, an acceptable level of risk must be determined for the system as a whole, and each

---

[2]For a more detailed summary of these concepts see the work of Leveson [8] .

behavior whose associated risk exceeds the acceptable level must be identified and dealt with. If any of these steps is not performed correctly, a system properly assessed as safe in the formal sense might be unsafe in the conventional sense.

2.2 Software Safety

For software to be judged safe, some authors have suggested that the system that it controls must be judged safe. In fact, defining software safety in terms of system safety has been proposed as the correct approach to software safety [5, 6, 11]. We argue that this approach is fundamentally wrong and potentially harmful. With software safety defined by this equivalence, any assessment of software safety is indirect since the software is safe if and only if the system is safe. Since the software and hence the safety of the software is the responsibility of the software engineer, this approach to assessment implies that the software engineer be capable of assessing system safety. This is an unacceptable situation because the software engineer is, by definition, not qualified to determine whether a system is safe within the framework of safety defined by systems engineering.

To make any progress, the notion of software safety must be formalized to at least the extent that safety has been formalized in systems engineering. Equating software safety with systems safety is not the way to achieve this formalization. As an example of the difficulty that the equivalence leads to, consider the case of a control system that commanded the undercarriage of an aircraft to be raised while the aircraft was on the ground [8]. This has been cited as a software safety failure. It almost certainly was not. The deficiency is more likely to be an omission of this special case from the specifications. The reason that those who cite it claim that it is a software safety violation is because they understand the hazard and feel confident in specifying that such action should not occur. It is a coincidence that they understand this hazard.

In general, the hazards associated with a system will not be understood in detail by the software engineer. Consider a second (hypothetical) example in which a full-authority digital flight control system commands a B-747 to flare on final approach at an air speed of 128 knots, a height of 180 feet, a head wind of 15 knots, flaps set to 30 degrees, and with a 14% fuel reserve. Does flaring the aircraft under these circumstances constitute a hazard? Unlike the landing-gear example, most software engineers are not capable of making that determination. It is not a coincidence that they do not understand this hazard.

The fundamental difficulty here lies in the placement of responsibility. The responsibility for identifying the hazards, assessing the risks, and thereby defining system safety, the only safety that really matters, has to lie with those that have detailed knowledge of the application. Software engineers do not have the required application expertise necessary to define safety in this manner. They are not qualified to make the potentially life-threatening decisions involved in determining acceptable levels of risk, and they cannot be required to have a complete understanding of every application domain for which they are asked to build safe software. Equating software safety and system safety is as inappropriate as equating power-supply safety and system safety, and for the same reasons.

How then can a theory of software safety be developed? Such a theory has to have at least the formality of the theory of safety defined by system engineers. Yet it must not suffer from the unacceptable disadvantages that arise when software and system safety are considered equivalent. Further, the theory must not depend on the software engineer having any specific detailed knowledge of the system being developed. Such a theory is developed in the remainder of this chapter.

2.3 The Role of Software in a System

As has been noted many times, in isolation, software is never unsafe. But, in practice, software is never used in isolation. Software is always used within a system, and it is merely one of many components of a complete system. It is a component of a complete system in the same sense that entities such as computer hardware, sensors, actuators, power supplies, packaging, and even human operators are each merely components of a system.

An important part of hazard analysis is the performance of *component hazard analyses*. Such an analysis attempts to identify those system hazards that a component could effect individually. As this form of analysis considers components only in isolation, hazards caused by device interactions cannot be detected. Proponents of software safety have suggested that a *software hazard analysis* be performed to assess the possible software hazards [6]. This report postulates that the notion of a *software hazard analysis* is entirely without merit.

Components can affect system safety in one of two ways: either an individual component can be unsafe, or several components can interact in an unsafe manner. Components can be classified according to whether or not they can individually cause the system to be unsafe. Component hazard analyses attempt to identify those components that could individually pose a threat to system safety. Such an analysis is meaningless for components that cannot directly cause hazards. In isolation, computing hardware and software are inherently safe, and thus they must fall into the class of components for which a component hazard analysis is meaningless.

Such components can only be analyzed in their systems context. These components cannot be considered unsafe in isolation because, when isolated, these entities are separated from the notion of hazard. It is only in the context of the complete system that the various hazards have meaning. None of the components changes and suddenly becomes "unsafe" when incorporated into a system. Any design deficiency in a component is present when the component is isolated just as when it is part of a system. A deficiency in a component can only lead to a hazard, however, when the component is part of a system because it is the system that defines the hazard. For such components, hazard is a system concept, not a component concept. In particular, the notion of hazard is not a software concept.

As an example of this idea, consider software developed for an automatic emergency shutdown system for a nuclear power plant. This software is perfectly safe when being used with a reactor simulator, say for operator training. The reason the software can be considered safe in this case is that it is isolated from the physical system, and thus there are essentially no hazards. However, if such software were operating as part of a power-generating reactor system, the system and therefore the software is capable of unsafe behavior because of the hazards that the system defines. In this example, the two systems define two different sets of hazards and therefore two different sets of safety concerns.

That software safety depends on how the software is used seems to present a dilemma for the software engineer. On the one hand, he is aware that a software defect can lead to a system hazard and therefore that great care must be exercised in building software for safety-critical systems. On the other hand, it appears that the context of use rather than the software itself determines whether software is safe.

This dilemma is illusory, since it is the context of use that defines the software *requirements specifications* from which the software is built. The specifications contain non-functional requirements, such as timing and *reliability* requirements, as well as functional requirements, and, in practice, *all* requirements have to be met. The software engineer is expected to analyze the requirements specifications and implement them using appropriate software-engineering methods.

Examples such as the nuclear reactor shutdown system above are misleading. In that example, there were, in principle, two different pieces of software involved, *not* one as the wording implies. The software used for operator training could be different from the software used to control a production reactor because the specification for the operator-training system need not include non-functional reliability requirements for the shutdown system. In practice, of course, it is likely that the same software would be used for both although it need not be. In more conventional terms, the difference between the two pieces of software lies in the different requirements specifications that they have to meet. Although the functional requirements might be identical, the non-functional requirements in the form of the reliability with which certain functions must be implemented are entirely different.

In summary, software is a component of a system just like the other components from which the system is built. As a result of the system design, functional and non-functional requirements are imposed on all of the components including the software. For safety-critical systems, these requirements will include reliability requirements on the software just as they do for the other components. Such requirements will be derived by the systems engineer from the formal considerations of *system* safety including the system hazard analysis and risk assessment.

2.4 A Theory of Software Safety

Since software is a system component, the development of a theory of software safety must enforce separation of the theory from systems safety yet permit smooth integration with it. Without separation, we risk needlessly complicating and weakening our conclusions about software with external systems concerns. Without integration, we risk reaching conclusions about software that do not contribute to the safety of the associated systems. The development of a theory of software safety must begin with an examination of the systems-engineering process that determines the safety of a system.

As part of a complex system design process, the systems engineer develops component specifications from the system requirements. Each component is assigned functionality that contributes to the desired system functionality. For the purposes of analyzing software safety issues, assume for the moment that these specifications are entirely free of safety concerns. They will be referred to as the *intrinsic functionality specifications*:

Definition: *Intrinsic Functionality Specifications*

> The intrinsic functionality of a component is the required functionality of the component without regard to safety.

Informally, the intrinsic functionality of a component is what the component is supposed to do during normal operation. Taken together, the intrinsic functionalities of all the components implement the desired system functionality.

As well as specifying the intrinsic functionality of the components and being reasonably confident that the system will provide the desired normal service when built, the systems engineer must consider the safety aspects of the system. This involves examining

the possible hazardous states that the system can enter, and determining how each component might contribute to the system entering a hazardous state as a result of that component's failure. The systems engineer performs a probabilistic analysis based on the known failure rates of the components, and, using techniques such as system fault trees [3, 4], tries to ensure that the system-wide risks are kept below an acceptable level.

It is extremely important to note that this system-level analysis is impossible unless certain assumptions are made about component failure modes. For example, a system fault tree might analyze a state in which a power supply fails and its output is no longer available. For a power supply, it is very unlikely that an explosion of the power supply would be considered. Thus, the assumption is being made that removal of output power is a reasonable failure mode for the power supply but that explosion is not. To ensure this failure mode, power supplies sometimes employ an overvoltage protection circuit (sometimes called a "crowbar") that deliberately short-circuits a power supply and thereby blows a fuse if an overvoltage condition arises. Note that this requires circuitry over and above that needed to provide the required intrinsic functionality. A second example of this approach from the area of computing hardware design is the concept of fail-stop computers [14]. With a certain probability, a fail-stop computer either works correctly or stops.

Assumptions about the failure modes of a component actually impose an additional set of specifications on the component. Essentially, for the system safety analysis to be valid, a component must fail according to the assumptions. If it does not, nothing precise can be said about the subsequent behavior of the system. These specifications will be referred to as the *failure interface specifications* for the component:

Definition: *Failure Interface Specifications*

> The failure interface for a component is the required functionality that must be provided with a certain probability in the event that the component is unable to provide its intrinsic functionality.

The failure interface for a component states essentially what the component must do with a high degree of assurance. A component may fail in any manner whatsoever internally but, in the event that it is unable to provide its intrinsic functionality, the interface that it provides to other components in the system has to be its failure interface. The failure interface might be inert and safe thereby having what is generally called a fail-safe characteristic. It might also define limited functionality, essentially that functionality that must be present to ensure continued safe system operation or shutdown.

Simply removing the service provided by a component is not an adequate failure interface if this leads to cessation of a required system service. For many systems, removal of service is a primary hazard. For example, it is not safe to remove power from a heart pacemaker under any circumstances. From the user's perspective, a pacemaker is only safe as long as it is generating appropriate stimulation pulses. The provisions that have to be made to cope with component failure within a system are, therefore, very dependent on the specific component and what it is doing. A pacemaker might invoke a backup, fixed-rate pulse generator in the event that its primary demand-driven circuit fails. Similarly, an avionics system might put an aircraft into level flight at a safe altitude if normal flight control functions cannot be maintained.

The concept of hazardous states introduced by Leveson et al. [5] is just a special case of failure interface specifications. A hazardous state by that definition is a state that must be avoided. Hazardous states are not a sufficiently powerful concept for an adequate treatment of software safety because they are restricted to the notion of state. A state is a

static concept whereas what is required is a vehicle for specifying minimal functionality. This minimal functionality might be no functionality, it might be freedom from certain states, or it might be some critical subset of the intrinsic functionality.

The correct implementation of failure-interface specifications for a component does not guarantee safe operation after the component fails. It merely ensures the *possibility* of safe operation. The failure-interface specifications ensure that other non-failed components have a well-defined state from which to proceed. Subsequent safe operation depends on the remaining components functioning in such a way that hazards do not arise within the new state. The components that remain fully functional might have to provide a different service after a failure than they did before in order to accommodate the new state of the system.

Requiring a specific response to the failure of a component by those that remain operational imposes yet further specifications on each component, i.e., exactly what the component is to do in the event that another component fails and presents its failure interface. These specifications will be referred to as the *recovery functionality specifications* for the component:

Definition: *Recovery Functionality Specifications*

> The recovery functionality of a component is the functionality required of the component in the event that one or more other components in the system fail.

It is precisely the recovery functionality of a system's software component that is being used when the software checks the operation of another component or responds to a failure indication of another component. But software is not the only system component capable of operating in this way. For example, actuators are often designed not to respond to commands outside their normal operating range. From a practical point of view, some

components will have no recovery functionality. For example, a sensor can do nothing productive when other components fail in an embedded control system. Conversely, the software can often do a great deal.

Any component in a safety-critical system is expected to comply with all three of the sets of specifications just defined. This leads to the definition of *component specifications*:

Definition: *Component Specifications*

> The specifications for a component consist of the intrinsic specifications, the failure interface specifications, and the recovery functionality specifications.

In a safety-critical system, hazards will be avoided if the system safety analysis is correct and each component meets the second and third of the sets of specifications. This leads to the definition of the *component safety specifications*:

Definition: *Component Safety Specifications*

> The safety specifications of a component consist of the failure interface specifications and the recovery functionality specifications.

The goal of the component safety specifications is to provide the necessary framework to support the safety analysis performed by the systems engineer. Provided all the components of a system comply with their individual safety specifications, the entire system can be deemed safe in this formal sense.

With respect to component failure, software is no different from any other component. It might be possible for software to detect its own failure and present a prescribed failure interface to the rest of the system subsequent to its failure. If other components have appropriate recovery functionality, safe system operation (which might be no operation) can continue. Similarly, software can provide recovery functionality in

order to cope with the failure of other components and thereby avoid system hazards. In practice, the software in most non-trivial systems will contain such functionality.

Finally, software safety can be defined. Since software is a system component, *software safety* is defined as:

Definition: *Software Safety*

> Software is safe if it complies with its component safety specifications, i.e., its failure interface specifications and its recovery functionality specifications.

The reason that software is incorrectly viewed as somehow different from other system components in some safety discussions is because it is perhaps the most complex system component, and because it is uniquely capable of providing the functionality necessary to cope with the failure of many other components. However, in the formal sense needed to enable a theory of software safety to be developed, it is essential that software be viewed properly as a system component.

A system is made up of a set of interacting components of which software is but one. Each component, including the software, may be capable of dealing with its own failure by containing facilities to detect its own failure and presenting a well-defined interface to the remainder of the system. In addition, each component, including the software, might be designed to detect and cope with the failure of other system components.

2.5 Software Failure

Consider software that is part of a system for which hazards have been identified and risks estimated. There are only two events that can lead to a hazard that might be viewed as the responsibility of the software. The first is for the software safety

specifications to define the correct action but for the software not to act as specified. The second is for the software to act as specified but for the safety specifications to define the wrong action. Typically, the former would be regarded as an implementation error and the latter as a specification error. Clearly, the responsibility for the former lies with the software engineer. His expertise is precisely that of developing an implementation from a set of specifications. But what about the latter? We maintain that, formally, the latter is the responsibility of the systems engineer. It is he who defines what the software is to do, i.e., defines the software's component specifications and thereby the software's safety specifications. Thus, as noted in the context of system safety above, software that performs as specified cannot be viewed as having failed.

This view does not preclude the software engineer from contributing to the development of the component specifications. The software engineer has the responsibility of informing the systems engineer about what functionality is appropriate for the software. In fact, the software engineer is uniquely qualified to determine what can be implemented and verified efficiently in software. If he notices what appears to be an anomaly in the specifications because of his *informal* knowledge of the application domain, he is free to question the specifications. Further, if the specifications are written in a formal notation, the software engineer is free to perform any analysis that might detect defects in the specifications. The software engineer does not have the authority to act unilaterally to implement anything other than exactly what is finally specified. As noted above, the software engineer is untrained and, by definition, unqualified to make the decisions associated with such acts.

To clarify the assignment of responsibility, the following definition is introduced:

Definition: *Software Safety Failure*

> A software safety failure occurs whenever the software component of a safety-critical system does not comply with its safety specifications.

With this definition in hand, it is clear that the closest workable approximation of perfect software safety involves close cooperation between the software engineer and systems engineer. The proper form of communication between these two parties is the software specifications, and in particular the software safety specifications. In past definitions of software safety, the specifications were never mentioned. In particular, failures that resulted from faithful implementation of errant specifications were sometimes labeled software safety failures. This is clearly unacceptable to the software engineering community.

2.6 Achieving Safe Software

The purpose of establishing the detailed framework of definitions above was to permit a precise definition of safe software to be produced within a formal framework. Such a framework must permit a clear statement of what it means for software to be safe for a specific system. Without such a definition, there is no hope of being able to build safe software because it would be impossible to determine what the goals of the software were. The definitions that have been developed here provide precisely the basis that is needed for such a determination. It is not sufficient to seek software that is "safer" or "more reliable" in some sense because what is achieved might not be adequate. With no formal definition of safety, it is not possible to state that a given software entity is safe. It is also pointless to develop software safety standards such as the United Kingdom standard 00-55 [15, 16] since that prescribes a method for achieving something that is not itself defined.

A fortunate consequence of having the formal framework of definitions is that it defines the role of the software engineer. No branch of engineering is perfect [17] and so although one might know what software safety means, it might not be achievable. The next step is to determine how to achieve it.

By the definition presented here, software is safe if it complies with its safety specifications. In a formal sense, the software engineer's responsibility is to implement the software and demonstrate that the safety specifications developed by the systems engineer are implemented correctly. Once it is developed, showing that software is safe according to this definition is, therefore, an exercise in *verification*. Informally, however, the software engineer is encouraged to analyze the specifications, report anything that appears to be a deficiency, and generally be on the lookout for any defect in the system that could detract from safe operation. Recall, however, that this is not a formal responsibility of the software engineer because he is not qualified to accept this responsibility.

There are many techniques for software verification including formal proof, inspection, testing, and static analysis. Typically, several of these techniques are applied to the verification of any given system. In the verification of software safety, proportionately more resources might be used and more diverse methods might be applied than is usual in non-safety-critical systems. Anderson and Witty [18] present some simple design techniques for developing safe software that can be verified easily.

Software fault tree analysis [5, 7] has been described as a technique "that is useful for partially verifying the safety aspects of software" [7]. Though a correct statement, this description of the role of software fault trees has been misinterpreted by many practitioners. Software fault tree analysis is a technique for establishing certain properties of software in a formal manner. Software fault tree analysis is limited to verifying safety

properties involving an undesired output not being generated. Such properties cover only a small subset of the specifications usually defined for safe systems. Safety specifications also include requirements for proactive actions in the event that something goes awry. Showing that something will happen under appropriate circumstances is beyond the capabilities of software fault trees, and will require other verification techniques.

Those familiar with the use of fault trees in *systems* safety analysis should not think that the techniques are similar based on the similarity on their names. System fault tree analysis is a general technique for documenting the various hazard conditions in a system and how they might arise. It is a stochastic technique in which the probabilities of hazardous events arising can be estimated. Software fault tree analysis has no probabilistic component and is related only distantly to system fault tree analysis through the common use of logical operators within the tree structure. System fault tree analysis is applied before the system is constructed and helps the development of an adequate design. By contrast, software fault tree analysis is applied after the software is constructed and helps achieve verification.

Software fault tree analysis is a verification technique and must be viewed as such. Showing that software has met required safety properties is a verification problem, and all techniques that can contribute need to be applied. The role of the software engineer is clear in this case. It is his responsibility to show that the software safety specifications have been implemented correctly.

In summary, achieving safe software requires that safety specifications be prepared, that the software be carefully implemented, and that compliance of the software with its safety specifications be verified. The software is safe to the extent that this verification is successful. However, the system of which the software is a part might not be safe because

the safety specifications might be defective in a manner that is unknown to the software engineer. The required verification might be performed with any available software verification technique of which software fault trees is but one.

Chapter 3 - Research Directions

3.1 Overview

The theoretical definitions developed up to this point are intellectually appealing, but unless they can be shown to be applicable to actual safety-critical systems, their usefulness must be considered suspect. A criticism that has been leveled at much of the previous theoretical work in software engineering is that it is merely an ivory tower creation that has no applicability and thus no real benefit to software engineering practitioners. Even when theories are shown to be applicable to real-world systems, the actual procedures for applying the theory to other systems are often omitted or extremely vague. This offers little to the practitioner struggling with the task of building safe software.

It is quite difficult for researchers to find actual software-controlled, safety-critical systems upon which to test theoretical hypotheses. However, precisely such a system is currently being developed by the Department of Physics of the University of Virginia. This device is the Magnetic Stereotaxis System, hereafter referred to as the MSS. The MSS is an experimental, high-energy, software-controlled medical device with clear safety implications. Sometimes referred to as the "Video Tumor Fighter", after one of its many uses, this system utilizes magnetism to manipulate surgical implants within the brain. Although presently in the developmental stages, the MSS has the potential to drastically change the manner in which neurosurgery is performed.

Access to this device enables the collection of experimental data to either corroborate or refute our theoretical approach to software safety. A series of experiments have been planned to test the feasibility of the theoretical definitions of software safety

presented here. The balance of this report presents the results of the first such experiment. This experiment is intended to demonstrate the feasibility of capturing all of the system safety ramifications on the software in a software safety specification. Furthermore, the experiment is intended to determine processes for performing that task. The end result of this experiment, the MSS *software safety specification*, will serve as a precise basis for future experiments in creating, assessing, and maintaining safe software.

3.2 Experimental Design

The experiment consists of a system safety analysis of the MSS, with special attention paid to the safety ramifications of the software component. The first step in the system safety analysis is a hazard analysis performed by the systems engineers in order to ascertain what should constitute a system hazard. From these hazards, system fault trees are created to determine what combinations of component failures can lead to a system hazard. These fault trees permit the determination of software safety requirements, from which a formal software safety specification is derived.

An important result of this experiment is a detailed description of the processes used. Such a description permits systems engineers to emulate our processes in order to evaluate the safety of other software-controlled systems. The systems engineer thus is able to derive software safety specifications from the systems safety analysis in a rigorous manner. This can be accomplished without requiring either software engineering expertise or new, specialized software safety techniques. Instead, the processes presented here rely on existing systems safety techniques amended slightly to handle the special properties of a software component. With these processes, the systems engineer is able to enhance the safety of the system by enabling the software engineer to contribute towards system safety within a controlled framework.

Certain aspects of the system safety analysis will not be discussed in this report. In particular, system hazards that are determined to have no software involvement are omitted. Similarly, component hazard analyses for the various subsystems are also excluded. One final difference between the processes documented here and the required system safety analysis is the lack of failure probabilities in the former. Because the system design has yet to be finalized, individual component failure probabilities are not presently available. As a result, the process has been modified to exclude failures that require multiple independent devices failures as sufficiently improbable, but to include all failures that depend on the failure of a single device. Computing hardware failures have been omitted, under the assumption that arbitrarily reliable hardware can be built, but all software failures are included, as no meaningful probabilistic bounds can be placed on these. In addition, device failures brought about by devices actually failing are distinguished from the apparently similar cases in which software commands the devices to act incorrectly.

3.3 Basis for Evaluation

In order for this experiment to provide any useful answers, it is necessary first to clarify the questions that are being asked of it. This crucial step serves to guide the experiment and provides a basis for assessing the results. One cannot expect to find answers to all of these questions. Answers to but a few would constitute a successful experiment. Furthermore, insight gained into any of the questions, even if not constituting a direct answer, would be extremely valuable.

It is critical to keep in mind that a single experiment cannot possibly prove universal results. At best one can hope for a proof of feasibility or a negative result. This

experiment is intended to determine whether, for a single particular system, the definitions we propose here are viable. A negative result would not necessarily relegate these definitions to worthlessness, though it would rule out any universal applicability. From a single experiment, it cannot be determined whether the definitions are globally applicable, though some insight into this possibility might be gained. Particularly, there promise to be large classes of systems, for which applicability could be demonstrated by a single experiment. In addition the knowledge gained from this experiment might permit the definitions to be refined so as to be more widely applicable. Future experiments should permit further refinement until strong arguments can be made about their universality.

3.4 Testing the Theoretical Definitions

In all previous work concerning software safety, software safety has been tightly coupled with systems safety. Our fundamental assertion is that this is not necessary. Software safety can be separated from systems safety in a manner that isolates all of the system safety ramifications of the software within the software safety specifications. The safety specifications provide a means of translating the systems engineers concerns into requirements that the software engineer is qualified to work with. Thus the primary questions to be answered by this experiment are:

Question: *Can software safety be separated from systems safety?*

Question: *Can a software safety specification adequately capture the safety requirements imposed on the software by the system?*

Because the theoretical definitions of software safety are based on the premise that software can be treated as a component by the systems engineer, this premise must be subjected to scrutiny. In particular, the systems engineer's ability to model software

properties at the system level must be questioned. If this proves feasible, system safety techniques could be applied to software-controlled systems, just as they are to any conventional system, rather than necessitating the development and application of specialized software safety techniques.

This experiment focuses on the aspects of the system safety analysis that deal with software requirements. Special care is taken to treat software as a black box, dealing only with required behavior, and ignoring any implementation-specific knowledge. The result of the system safety analysis is a formal software safety specification, specifically stating the software failures that could lead to system safety being compromised. These processes depend on the assumption that software is not too complex to be treated as a component during systems safety analyses.

Question: *Can software be treated as a component in system safety analyses? Is software too complex to be treated as a component?*

The definitions we developed make a distinction between those elements of the software safety requirements that are part of the failure interface specification, and those that are taken from the recovery functionality specification. Although these categories are theoretically distinct, this experiment tests whether this separation is useful in practice. It is also aimed at uncovering any other categorizations that are useful in practice, but might not have been apparent at the theoretical level.

Question: *Are the categorizations of software safety requirements presented useful? Do other alternative categorizations present themselves?*

3.5 Software Failure Modes

Component failure modes are a well-understood systems engineering concept. The failure modes of a given component are the manners in which the component can fail. These are usually limited in number and classifiable according to their effects. This concept has not been extended to software safety in large part because of the belief that software failure modes are practically innumerable. This report argues that the concept of component failure modes can be extended to software.

Previous work in software safety analysis has made the tacit assumption that software is too complex to be treated as a system-level component [8]; we challenge that assumption based on the following reasoning. Any observed device failure mode can be either caused by the device failing, or software instructing that the device act inappropriately. All device actions that the software could command are also actions that a failed device could perform without having been commanded. Thus, software failures are only distinguishable by their effects on other devices. Therefore, the software failure modes can be determined from the possible software-initiated device failures.

The complexity normally attributed to software failures stems from the many factors that can cause a failure, rather than the effects of the failure. However, system safety analysis is only concerned with the failure modes, i.e. the effects of failure, not the causes. The only reason that software failure modes are any more numerous than other device failure modes is that software can effect numerous devices simultaneously. Thus in determining the software requirements, the systems engineer need only consider the failure modes of the other components that the software could initiate. The complete set of software failure modes can be enumerated as all possible combinations of individual

device failures. Although sound in theory, the experiment will test the usefulness of software failure modes in practice.

Question: *Are software failure modes a viable concept ?*
Can software failures be classified solely according to observable effects?

The definitions presented in this report attempt to maintain a separation between systems engineering and software engineering concerns. Primarily, software engineering tasks should not require any systems engineering expertise, and systems engineering tasks should require as little knowledge of software engineering as possible. It has been suggested that software engineering expertise is required in order to understand the impact that software failures can have on systems safety [6, 8]. Software failure modes present an alternative in which software engineering expertise is not required during the system safety analysis. If possible, this must be tested experimentally.

Question: *Is software engineering expertise required in order to understand software failure modes?*

3.6 Maintaining a High Degree of Rigor

The development of safety critical software requires a degree of rigor not necessary in other software engineering tasks. The safety critical software community has begun to advocate formal specifications and other formal methods as means of increasing the safety of such software [19]. In fact, the U.K. Ministry of Defense has actually mandated the use of formal methods on safety critical software [15]. Although clearly a needed step, this is not necessarily sufficient. If an unsafe software specification is implemented perfectly, the resultant device will be unsafe. Although it is impossible to verify a system safety analysis formally, a rigorous approach is possible. When determining software safety specifications a high degree of rigor must be enforced wherever possible.

Although not strictly formal, rigorous methods must suffice when no formal alternatives present themselves. As in any branch of engineering, mathematical constructs cannot be proven to model real-world entities exactly. Some sort of informal, convincing argument will have to be made. While it is a goal of this work to isolate the informality that is necessarily present, it is equally important not to settle for a less rigorous approach when a more rigorous alternative can be found.

Question: *Can the gap between system safety analyses and formal software specifications be bridged rigorously?*

Rather than creating an informal specification, this experiment demonstrates a process that results in a formal safety specification. This permits all informality to be eliminated by the systems engineer before the software engineer is involved. Although a desirable property, this degree of formality depends on the systems engineer being able to communicate formally the software safety requirements to the software engineer. A common argument against formal specifications is that they are incomprehensible to all but a few mathematicians [20]. If this were the case, the systems engineer would be incapable of formally specifying the software's safety requirements. This experiment attempts gauge the truth of that criticism.

Question: *Are formal specifications a suitable means of communicating software safety requirements?*

The question of whether to include an informal, or *natural language* specification, with the formal safety specification has been considered. The primary argument against this is that it might not accurately reflect the contents of the formal specification. Alternately, a natural language transliteration of the formal safety specification would

make the specification more accessible. This experiment should offer some insight into resolving this dilemma.

Question: *Does the inclusion of a natural language specification make the formal specification more accessible or does it weaken it unnecessarily ?*

3.7 Determining the Correct Process

The previous chapter proposes a definition for software safety, but presents no procedures for creating safe software. The problem of building safe software-controlled systems has been decomposed into the two smaller problems of creating safety specifications, and correctly implementing those specifications. It is a purpose of this experiment to uncover processes for a practical application of the theory. Whether a single, general process exists, or whether each system will require a customized process is an open question. If a single process does exist, it would necessarily be quite general. It is worthwhile to attempt to generalize as much as possible the processes that we have used, as it is only in their general form that such processes might be applicable elsewhere. Furthermore, these processes must be made as rigorous as possible in order to convince skeptical audiences such as regulatory agencies.

Question: *Does a single, universally applicable, process for determining software safety requirements exist?*

Previous discussions on software safety have suggested applying systems safety analysis techniques directly to software [5, 6]. In contrast, the processes demonstrated here use systems safety analysis techniques to analyze the system. Before applying these techniques to software controlled systems, the systems engineer must be aware of the special properties of software. When the presence of a software component invalidates or limits the applicability of a given technique, the systems engineer must be made aware of

this fact. Furthermore, when the software component endows a technique with additional power, this should be taken advantage of.

Question: *Are existing systems safety techniques applicable to systems containing a software component? What amendments are necessary?*

Finally, it is important that the process presented here be contrasted to those presented elsewhere [6, 8]. Because we argue that our definition of software safety is universal, it should encompass all other software safety techniques. If one process can be shown to reveal problems that another processes failed to uncover, the various processes could prove to be a valuable means of assessing one another.

Question: *Can our definitions be rigorously demonstrated to encompass existing techniques?*

3.8 Other Issues Raised

Besides testing whether building safe software is possible, the experiments documented here should shed some light on the costs involved in building safe software. This could aid in making decisions as to whether the cost of building safe software exceeded the cost of building equivalent mechanical systems. There are a spectrum of factors that might influence the techniques used, besides the technical nature of the system. These include: cost, cost of failure, available expertise, resources, modifiability, time restrictions, and regulatory scrutiny. As a result, there could conceivably be a spectrum of techniques ranging from high cost, high assurance to low cost, low assurance. If means of measuring the degree of software safety and the cost of building safe software existed, systems engineers would be able to better appreciate the cost involved with assigning safety-critical responsibilities to software.

A problem that manifests itself with software-controlled systems is a willingness to trust the software too much. An example is the Therac 25 incidents, in which inexpensive safety features that would have limited the safety ramifications of the software were omitted, with tragic results [21]. Although programmable general purpose computers are often much easier and cheaper to use than special purpose hardware, there is a danger associated with relying too much on the software to provide safety features. Because of the complexity of software, it is often far more expensive and difficult to verify safety-critical software than to provide equivalent mechanical safety devices. However, a computing subsystem is capable of far more intricate behavior than any mechanical system, and as a result can provide more versatile safety recovery.

This experiment attempts to highlight the tradeoffs involved in using software to provide system safety. This includes the cost of physical devices, the cost of verifying software properties, and the enhanced device safety provided by additional safeguards. Raising awareness of the difficulties and cost of software safety could prevent the mistake of trusting the software too much from being made.

Question: *What metrics for software safety can be found?*

Question: *What metrics for the cost of building safe software can be found?*

Our definitions rely on the separation of software safety specification from software implementation. Whether a complete separation is advisable is subject to question. As an example, a system design that requires that the software component perform an intractable or unsolvable calculation should not be considered, if there is an alternative that eliminates this possibility. Similarly, if one possible approach leads to software that is simple to prove while another is extremely costly to prove, this point should be raised during the design of the system and specification of the software. For this

reason, it might be advisable to have software engineering expertise present during the system safety analysis, if only to assess the relative costs of various software approaches.

Question: *To what degree should implementation concerns influence the specification.*

One unique aspect of this experiment is the fact that prototype software is already completed. It is instructive to compare the results of the case study with the existing system at every phase. Because the original software was built with a naive view towards software safety, a very rough measure of the efficacy of these processes can be obtained, Comparing the results of this experiment to the "raw" results, might provide some insight into which steps offer the largest "payoff" in terms of potential problems uncovered.

Question: *How much "safer" is the software created through these processes than that created under the "naive" approach.*

Some of the software safety literature discusses the notion of *defense in depth* [6, 8]. Although this is a valuable systems engineering technique that offers the potential for greatly increased safety measures, it relies on the independence of failures. The independence of failures is easy to ensure when dealing with physical device failures. It is not nearly as straightforward for software. As demonstrated by Knight and Leveson [22], independence of failures can not be assumed when multiple versions of software are developed from the same specification. Therefore, defense in depth permits separate hardware and software means to guard against the same error, but is of questionable usefulness when applied to multiple software guards.

Although no studies have been done on software using slightly different specifications, it is reasonable to assume that the probability of coincident failures drops as the similarity in specifications falls. If radically different specifications can provide guards

against the same failure, this could be exploited, but extreme caution must be used in assuring that the specifications are sufficiently diverse.

Question: *Is there a manner in which defense in depth can formally contribute towards software safety?*

3.9 Summary

This chapter raises questions that should be addressed by the experiment being performed. This forms a precise basis for evaluating the success of the experiment, as all of these questions are of significant importance. Although one cannot hope for complete answers to all of them, if some light is shed on some of these questions, the experiment can be deemed a success. Furthermore, the documents resulting from this experiment will provide a basis for future experiments exploring other aspects of software safety. Some of these questions might not be fully answered until after these other experiments are performed.

Chapter 4 - Magnetic Stereotaxis System Overview

4.1 Magnetic Stereotaxis

Stereotaxis is a neurological technique used for biopsy, delivering hypothermia to brain tumors, and treating various disorders. Conventional stereotaxis is limited by the need for a direct path between the surface of the skull and the tumor to be operated on. Such a path is often blocked by critically important or easily damaged brain tissue. Establishment of a direct path through these regions can result in serious injury to the patient, possibly even death. For this reason, conventional stereotaxis techniques often cannot be used where they might otherwise be valuable. Magnetic stereotaxis utilizes magnetism to overcome the limitations of conventional stereotaxis.

Researchers at the University of Virginia and the University of Washington are involved in ascertaining the viability of magnetic stereotaxis [23]. Magnetic stereotaxis circumvents the limitations of conventional stereotaxis by permitting an indirect path to be traced between the surface of the skull and the target location. The technique uses magnets positioned exterior to the cranium to move a magnetic dipole (hereafter referred to as the seed) within the interior of the brain.

One application of magnetic stereotaxis technology is in the treatment of deep-seated brain tumors. After being maneuvered to the site of the tumor the seed can be heated by the application of low frequency radio (RF) waves to the skull. Magnetic stereotaxis also offers a potential solution to the problem of drug delivery to interior regions of the brain. The brain is capable of blocking virtually all drugs from reaching the interior regions. For many diseases, such as Parkinson's disease, drugs are available, but there is no suitable means of delivering them. The solution in the past has been to either

withhold the drugs entirely, or use dangerously high doses in attempt to penetrate the brain's natural drug barrier.

Using magnetic stereotaxis, the seed could tow a catheter directly to the location where the drugs are required. This catheter could be filled with the needed drug which would then be released over time. If the catheter were attached with heat-sensitive glue, the seed could be heated in order to release the catheter. If magnetic stereotaxis is demonstrated to be workable, future research will involve finding a means of using it to perform biopsies. At present, a prototype magnetic stereotaxis system (or MSS) is under construction so as to better test the feasibility of magnetic stereotaxis.

4.2 The Prototype Magnetic Stereotaxis System

A prototype magnetic stereotaxis system is currently in development. The MSS utilizes superconductive coils, as these are the only magnets capable of generating the magnetic fields required to move the seed. The seed is tracked using bi-planar fluoroscopy. The fluoroscope images are analyzed by a computer that determines the seed's position relative to skull markers that have been placed on the skull. A computer displays an image of the seed superimposed onto a set of pre-operative magnetic resonance (MR) scans that serve as the neurosurgeon's "road map" [24].

The magnetic stereotaxis system would not be feasible without computer controls. The process of transforming a desired seed movement into a required magnetic gradient, and then testing various field configurations until a suitable approximation is found, is a computation intensive task. In addition, the process of determining the seed's location and providing the operator with appropriate feedback necessitates a computer interface.

The prototype magnetic stereotaxis system has successfully demonstrated the feasibility of magnetic stereotaxis. As a feasibility study, it was designed with functionality as the primary goal, and safety a secondary concern. In constructing the physical components, proper manufacturing safety practices were followed, which minimizes the possibility of a component failing in a catastrophic manner. However, patient safety issues have not yet been fully considered.

The prototype software was built without a proper system safety analysis having been performed. As a result, the software was built without a correct understanding of all of its safety consequences. Although the software engineers involved were very conscientious in making the software as correct as they knew how, this is not equivalent to making the software safe. Even software that is perfectly correct can contribute to a system hazard, if the software interacts with the system in a manner not anticipated by the software engineer, or if a component failure leads to an unanticipated operating environment.

Although systems engineers would not even consider the possibility of pressing a physical prototype model into active service without an extensive redesign stressing the safety requirements, the same cannot be said of software. Software safety is simply not well enough understood. The remainder of this report demonstrates means for determining the system safety consequences of the software and suggests how such software can be constructed with safety as a primary concern.

4.3 Magnetic Stereotaxis System Operation

The magnetic stereotaxis system can be divided into the following subsystems: the magnetic manipulation subsystem, the X-ray imaging subsystem, the RF heating subsystem,

and the computing subsystem. In addition, although not strictly subsystems, the seed and the operator are also integral system components. The operation of the system basically conforms to the simple closed loop depicted in Figure 4.1.

The imaging subsystem captures an image of the seed and three skull markers. The computing subsystem locates the seed and markers on the image, and superimposes the seed on a display of the preoperative MR images. The operator then acts on the information displayed, commanding either a movement of the seed or heating of the seed. In response to this, the computing subsystem directs either the manipulation or heating subsystem in carrying out the command. After the command is completed, the imaging subsystem captures an updated image and the entire process repeats until the operation is completed. The only deviation from the simple closed loop is that, during seed movement, additional images are taken to permit the operator to track the progress of the seed.

In some aspects, the system can be considered to be a real-time system. Some of

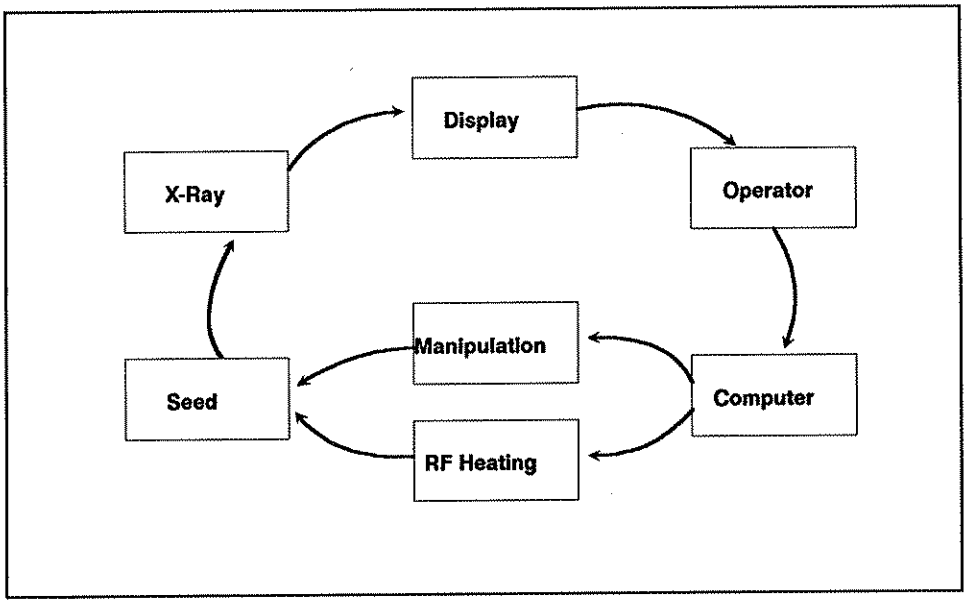


Figure 4.1 - Basic Operation of the MSS

the computer operations, especially those involving control of the other subsystems must be completed within hard deadlines. Fortunately, the more complex aspects of image formulation and current calculations are not restricted by hard deadlines. Although responsiveness in these actions is needed, it is not safety critical. This alleviates some of the more difficult challenges of safety-critical real-time software by not requiring that complex calculations be proven to be bounded in execution time. Nonetheless, enough real-time considerations are present to make the processes discussed applicable to more demanding real-time systems.

It is worthwhile to consider the purpose and design of the various subsystems in more detail. It is particularly important to understand the failure modes of each subsystem. The failure modes of a subsystem constitute all of the manners in which the subsystem can fail. Although there might be many different causes for a particular failure mode, only the effects of the failure need be considered. These failures often either cause or contribute to a system hazard. Thus, before further system safety analyses can be performed, the failure modes for each subsystem must be accounted for. Many of the software safety requirements will be derived from the failure modes of the various subsystems as part of the software's recovery functionality specifications.

The X-ray Imaging Subsystem

The imaging subsystem consists of the X-ray source, the biplanar fluoroscope, and the image capture system in the computer. This subsystem is responsible for generating two images of the seed and skull markers, one from each imaging plane. These images are received by the image capture system, where they become available to the computing subsystem. The two imaging parameters provided to the subsystem are the x-ray beam intensity and duration. A signal from the computing subsystem activates the x-ray which is

responsible for maintaining a beam of the desired intensity for the period specified, and then shutting the beam off. During this time, the computing subsystem must also signal the image capture hardware to capture the images received.

The imaging subsystem's failure modes can be divided into those caused by failure of the x-ray source and those affecting the image reception. The X-ray has three failure modes that need to be considered:

- Turning on without having been commanded.
- Remaining on for a duration other than that requested.
- Operating at an intensity other than that requested.

Although the subsystem is designed to minimize the possibility of these occurrences, they need to be considered as the modes in which the device might fail. The impact of the component failure modes on the safety of the entire system is the subject of the system-wide hazard analysis.

The image reception function is determined to have failed whenever the images provided to the computing subsystem do not accurately reflect the relative positions of the seed and markers. The failure modes can be categorized as follows:

- Images containing an object that is not physically present.
- Images not containing an object that is physically present.
- Images containing objects that are misplaced or distorted.
- Images whose signal is masked by noise.

As the software specifications must state the properties of an incorrect image, they must also contain definitions for recognizing the objects on the images. An important

division of component failure modes is into groups of those failures that can be detected and those that can't. Failures that can be detected should be precisely dealt with in the specification. Failures that cannot be detected can sometimes be protected against by other means, but often must be considered as an accepted risk. In either case, it is the systems engineer, not the software engineer, who is responsible for such a decision.

The RF Heating Subsystem

The heating subsystem is responsible for radiating the seed with radio-frequency energy, for the purpose of heating the seed. The heating subsystem is commanded on and off by signals from the computing subsystem. The RF frequency is fixed, but the intensity is contained in the command to switch the device on. The failure modes of the heating subsystem are:

- Switching on without having been commanded.
- Switching off without having been commanded.
- Failing to switch off after having been commanded.
- Failing to switch on after having been commanded.
- Generating RF waves of an intensity other than that requested.
- Generating RF waves of a frequency other than that requested.

This final failure mode has been determined to be sufficiently improbable so as to have no impact on the safety of the system, and thus will not be considered further.

The Manipulation Subsystem

The manipulation subsystem consists of six superconductive coils arranged as if on the faces of a cube. The coils are embedded in a supporting structure, misleadingly

referred to as the "helmet", in which the patient's head is placed [24]. The manipulation subsystem encompasses the power supplies needed to charge and discharge the coils, and the liquid helium cooling needed to maintain the superconductivity of the coils.

The failure modes of the manipulation subsystem fall into two groups: those which pose direct hazards, and those which affect system operation. The failure modes that pose direct hazards involve the release of large amounts of either electrical energy, thermal energy, cryogenic fluids, or toxins. These are analyzed as part of a component hazard analysis, and play no further part in this experiment. These hazards will eventually impose software safety requirements in terms of monitoring for and responding to hazardous situations. At present, these analyses have not been performed, and thus the software safety requirements have not been determined or included here.

The failure modes that effect system operation consist of any arbitrary combination of one or more coil failures in a manner that alters the magnetic forces on the seed. The coil failures are:

- Charging more rapidly or more slowly than anticipated.
- Charging to a level other than that commanded.
- Charging without having been commanded.
- Discharging without having been commanded.
- Discharging more rapidly or more slowly than anticipated.

These failure modes have the net effect of causing a magnetic field on the seed that is inconsistent with that required. In further analyses, this composite effect will be treated as the single failure mode of this subsystem.

The Computing Subsystem

The computing subsystem is responsible for managing the operator interface and controlling the other subsystems. The internal causes of failure of the computing subsystem are extremely complicated, but can only manifest themselves in terms of their effects on other subsystems. As such, the failure modes of the computing subsystem can be grouped into:

- Presenting the operator with incorrect data.
- Commanding the other subsystems incorrectly.
- Complete loss of service.

All of these failures can appear as the failure of another subsystem. If one or more devices faithfully execute errant commands from the computing subsystem, the fault lies with the computing subsystem, even though it might initially appear as a device failure. Similarly, an operator judgment error that is caused by the operator being presented with incorrect data is actually a computing subsystem failure.

The computing subsystem failure modes consist of all possible combinations of other subsystem failure modes that could be effected by the computing subsystem. Thus it is possible to enumerate all of the computing subsystem's failure modes. Rather than enumerating them, only those which have safety ramifications will be considered here. These will be determined from the system safety analysis procedures. The result will be a software safety specification that addresses all of the software failure modes that must be eliminated in the interest of system safety.

The Operator Subsystem

The final "subsystem" to consider is the human operator. It is important to examine the failure modes of the operator, as these, just as those of any other component, could impact the safety of the system. Such an approach offers insight as to how other components can be made to prevent operator failures from propagating through the system. It is conceivable that at some point in the future, the operator could be replaced by an expert system, thus strengthening the argument of viewing the operator as a subsystem, even though such a system might have different failure modes than a human.

The operator is responsible for assessing the situation as described by the computing subsystem's display of the MR images, the seed position, and other status information. Based on this assessment, the operator must choose a course of action that best accomplishes the purposes of the operation without exposing the patient to needless risk. This course of action is communicated to the computing subsystem in terms of heating and movement commands.

Because the probability of human judgment errors defy analysis, and because there is no way to formally determine the probability that such an error could be detected by the computing subsystem, assigning failure probabilities to the operator must be performed carefully. Conventional systems engineering wisdom requires that very high failure probabilities be assigned to the operator, particularly when required to notice something out of the ordinary or to respond correctly under duress [25]. Furthermore, it is reasonable to assume that presenting the operator with incorrect data will increase drastically the probability of an operator error. Therefore, ensuring that all data presented to the operator is correct is extremely important in minimizing the possibility of operator error.

Determining the failure modes of the operator is not a straightforward task. It might seem reasonable to assume that the operator is infallible, and that the device, as a medical tool, should simply carry out the operator's instructions. Such an assumption would be a mistake. A human operator is never infallible, and the computing subsystem provides an excellent means of screening operator commands. Questionable commands can either be rejected or require explicit confirmation of intent. There will necessarily remain cases in which the expertise of the operator will have to be relied on, but to not attempt to minimize these would be negligent. The question of what constitutes inadvisable input must be put to domain experts - in this case neurosurgeons. The applicability of this technique will be governed by the system engineer's ability to capture this domain expertise in a software-manipulable form.

4.4 Safety Implications of the MSS

The safety implications of the magnetic stereotaxis system are numerous. The MSS poses obvious hazards to the patients, and other less obvious hazards to the environment and any people in the vicinity (including the patient). For ease of discussion, those hazards that affect only the patient will be termed patient-specific, and those that can affect others will be labeled general hazards. The patient-specific hazards are all treatment-specific. These involve either being overexposed to magnetism, X-rays, or RF waves, or being harmed by unsafe seed movement or heating. Several of these involve the computing subsystem, and it is these that are addressed by the balance of this experiment.

The general hazards involve the release of radiation, dangerous chemicals, or large amounts of energy into the immediate vicinity [25]. The MSS has a number of such high-energy threats. The forces between each pair of coils can exceed twenty thousand pounds.

The cryogenic fluids used to maintain superconductivity pose a leakage threat. The controllers used to charge and discharge the coils are extremely high-energy devices. If discharged improperly these could cause injury, damage equipment, or even cause a fire. As mentioned previously, while these hazards cannot be caused by the computing subsystem, they can be detected and have their effects mitigated by software means. Although important parts of the software safety specification, these requirements are not covered in this study due to lack of sufficient information.

It is critical to remember that the device in question is a medical tool. Its safety is somewhat a function of the expertise and judgment of its operator. Asking the question "is it safe?" of the MSS is similar to asking the same question of a well-understood medical device such as a scalpel. The safety of a scalpel depends entirely on the skill and judgment of the operator. Furthermore, to make the scalpel safe by blunting it would render it useless. Similarly, there will always be an element of risk in the MSS. This is a direct consequence of the relative importances of ensuring safety and accomplishing the purpose of the device.

4.5 The Food and Drug Administration

One major concern in determining the safety of any medical device is FDA approval. Initially, FDA approval for tests on terminally-ill patients is a relatively simple matter. However, before the device can be put in widespread use it must receive FDA approval for general use, which is always a very difficult process. This is further complicated by the fact that the FDA has recently become aware of the difficulties in assessing the safety of software controlled devices, and has revised its requirements for certification of such devices. The Therac 25 incidents demonstrated the potential for harm that can result from unsafe software and directed a great deal of scrutiny at the FDA's

approval processes [26]. The FDA was placed in an untenable situation, where they were being asked to ensure the safety of computer-controlled medical devices, but were unable to exercise the safe option of disallowing the use of such devices. They did what any rational agency facing such a decision ought to do: they deferred.

The new FDA requirements are intentionally vague. They require adherence to good engineering practices, leaving the research community to define what these are. There is no FDA approved methodology for ensuring software safety. They simply require that a great deal of effort in the area of software safety be demonstrated, banking on this effort to reveal any possible safety problems. The attitude at the FDA is that any procedure can be used as long as convincing arguments for device safety can be made [27]. This is perhaps not a bad idea as a rigorous argument for safety should be able to convince an arbitrarily skeptical audience.

Chapter 5 - Experimental Process

5.1 Overview

This chapter outlines the bulk of the processes performed in order to derive the formal software safety specifications for the MSS. In order, the steps performed in this experiment were:

- Determination of Subsystem Failure Modes.
- System Hazard Analysis.
- System Fault Tree Construction.
- Derivation of the Software Safety Requirements.
- Creation of the Formal Software Safety Specification.

The complete list of subsystem failure modes were presented in Chapter 2. The completed system fault trees along with the associated software safety requirements are included in entirety in Appendix A. Appendix C consists of the formal software safety specifications for the MSS.

This chapter provides annotation for those documents, giving an explanation of the methods used, the rationale behind the decisions made, and indicating areas worthy of special attention. Note that the system safety analysis described here has been restricted to only those aspects that have ramifications on the software safety requirements. In addition, the results of the component failure modes analysis and the hazard analysis are given without detailing the procedures used, descriptions of which can be found elsewhere [3, 28].

5.2 The System Safety Analysis Process

Hazard Analysis

Systems engineering has formalized the notion of system safety by restricting safety assessments to only those hazards that have been identified. The Hazard analysis procedure attempts to uncover all of the potentially harmful systems hazards. For the MSS, this process was performed with the aid of the MSS systems engineers. Numerous component-level hazards and several important system-level hazards were uncovered. The system-level hazards from which the software safety requirements were derived are:

- Patient Injured by Exposure to X-ray.
- Patient Injured by RF Heating.
- Patient Injured by Seed Movement.

All of these hazards are operational hazards, being specific to patient treatment. The more general hazards involving the release of large amounts of energy into the environment are not pursued here.

Note that this list constitutes the systems engineers' perceptions of the system hazards but not necessarily the actual hazards. Because the formal definition of system safety is based on this assessment, a mistake here could undermine the usefulness of all further safety analyses. However, this is the accepted systems practice, and it is the intention of this project to demonstrate how software engineers can contribute towards systems safety, not redefine it. Nevertheless, this is an important process that must be performed by qualified systems engineers, not software engineers.

Computing Hardware Failures

A computing subsystem failure can consist of either a hardware or software failure. In order to keep the complexity of the system safety analysis at a manageable level, it was decided to operate under the assumption that the computing hardware is arbitrarily reliable. Furthermore, it is assumed that the computing hardware is free from design defects. These assumptions are not completely unrealistic. Building arbitrarily reliable computing hardware can be accomplished through the application of redundancy schemes [29], and computing hardware built using formal methods has been proven to be free from design defects [30]. Although building such hardware is not necessarily cost-effective, it permits the software safety analysis to be unimpeded by computing-hardware reliability concerns.

Were this approach not taken, the safety analysis would be extremely complicated. The number of possible computing hardware failures is astronomical, and many of these are difficult or impossible to detect. As would be expected, the failure modes of computing hardware are identical to those of the software. Because of this, a single hardware failure could conceivably invalidate all of the safety measures built into the software. For a system containing a computing hardware component to be judged safe, its safety-critical failure modes must be determined, and made sufficiently improbable. Increased reliability provides blanket coverage of all possible safety concerns and is far easier than attempting to isolate the safety implications on the computing hardware.

System Fault Trees

Fault trees are a systems engineering technique for determining combinations of component failures and system events that could lead to system level hazards. A fault tree

provides a stochastic model for assessing system risks based on the component failure probabilities. If the risk associated with the hazard is unacceptably high, a solution to the problem can usually be derived from the fault tree. System safety analyses are typically iterative in nature. Each analysis suggests system amendments which, if implemented, invalidate the analysis. The fault tree analysis of the MSS required numerous iterations: only the final results are presented in Appendix A.

Fault trees are usually represented using the familiar notation of AND and OR gates. An example of this representation appears in Figure 5.1. This particular example states that the only way for the patient to be injured by RF heating is if either the seed is heated incorrectly, or the brain tissue is damaged by direct RF heating. Each of these possibilities is in turn explored in further detail. This process of refinement continues until meaningful probabilities can be assigned to each node. As this graphical representation quickly becomes unwieldy, the complete fault trees in Appendix A are provided in an equivalent textual notation.

By its very nature, the development of system fault trees is an informal process.

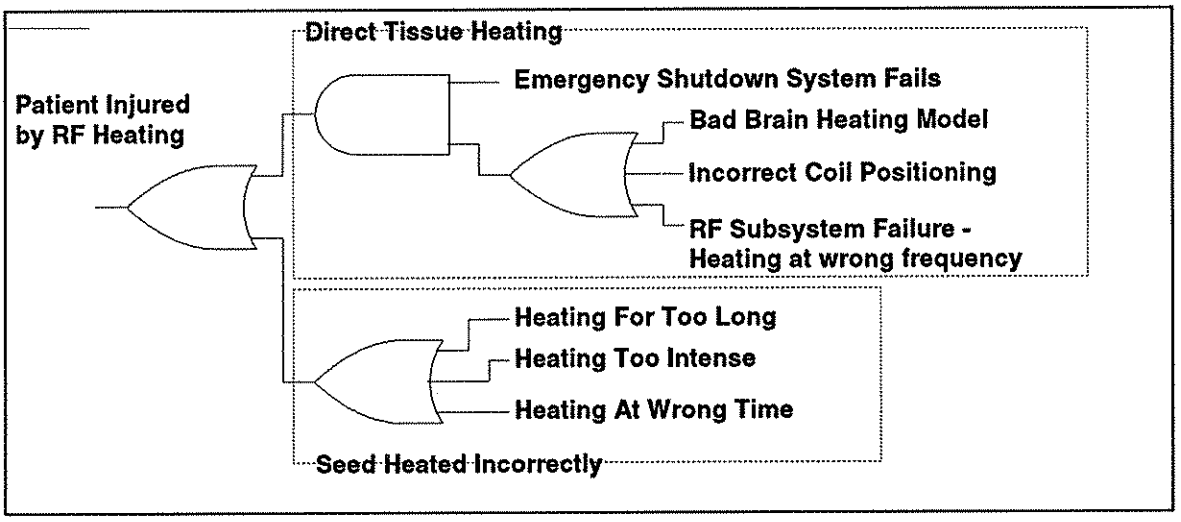


Figure 5.1 - A Sample System Fault Tree

Rigor can be introduced by adhering to well-understood constructs and by carefully considering all possible device interactions. Nevertheless, a fault tree analysis of safety cannot be proven to be either complete or correct. Systems engineers understand that they are responsible for ensuring that the fault trees are as accurate as possible. The fault trees presented here were created with the aid of the systems engineers. Although they cannot be proven correct, nobody involved in the process could find fault with them.

Note that these are systems engineering fault trees, and are in no way related to software fault trees. A system fault tree is stochastic in nature, utilizing the probabilities of component failures and other events to determine the risks associated with a system level hazard. Software fault tree analysis is an informal software verification technique loosely related to determining weakest preconditions [31].

Failure Probabilities

Although fault tree analysis is primarily concerned with failure probabilities, the fault trees presented in Appendix A contain no such information. This is due largely to the fact that the device failure probabilities are not currently available for the production version of the system. This does not pose a problem if handled carefully. The following assumptions were determined to permit fault tree analysis to be performed without probabilities:

- Any single device failure is too probable to be ignored.
- Concurrent failure of independent devices is sufficiently improbable to be ignored.
- Software failures are assigned arbitrarily high failure probabilities.
- Software failures that are part of the safety specification are arbitrarily improbable.
- Computing hardware failures are arbitrarily improbable.

As long as the production model design of the MSS enforces these assumptions, the safety analysis performed here will hold. If the software implementation can be proven to disallow all software safety failures, the system will be considered safe in the formal systems safety sense.

Fault Tree Node Categories

In a completed system fault tree the leaf nodes represent atomic events and failures that can be assigned probabilities. Because the purpose of this experiment is not to assess safety, but to derive software safety requirements, this restriction must be relaxed. Instead, leaf nodes will be restricted to certain well-defined categories. These categories are:

- The root node of another subtree.
- An individual device failure.
- A specific software failure.
- An accepted risk.

Allowing leaf nodes to represent the root of another tree merely permits fault tree decomposition into more manageable subtrees. Individual device failures form the basis for standard systems fault trees. These nodes must detail the particular failure mode and the probability assigned to that failure.

The leaf nodes detailing software failure modes are used to determine the software safety specifications. If the software failure can be assigned a probability of 1 without causing the hazard in question to exceed the acceptable level of risk, it need not be included in the software safety specifications. Conversely, it can be included in the safety

specification, and be assigned an arbitrarily small probability of failure. In this manner, these nodes form the basis of the software safety specification.

Finally, leaf nodes can represent a risk that is accepted by the systems engineer. These consist of risks that cannot be eliminated without severely impairing the usefulness of the device, and for which the systems engineer has assumed responsibility. Examples from the MSS include:

- Mathematical approximations of the magnetic fields.
- Reliance on operator judgment.
- Experimental results concerning direct brain tissue heating.
- The model of seed movement through brain tissue.

5.3 Understanding the MSS Fault Trees

MSS Emergency Stop Subsystem

As originally devised, the MSS posed a number of different hazards, but contained no measures for ensuring system safety either by preventing these hazards, or mitigating their effect. As a result, Figure 5.2 represents a typical part of the original system fault tree, highlighting the safety problems of that system. Any single failure could lead directly

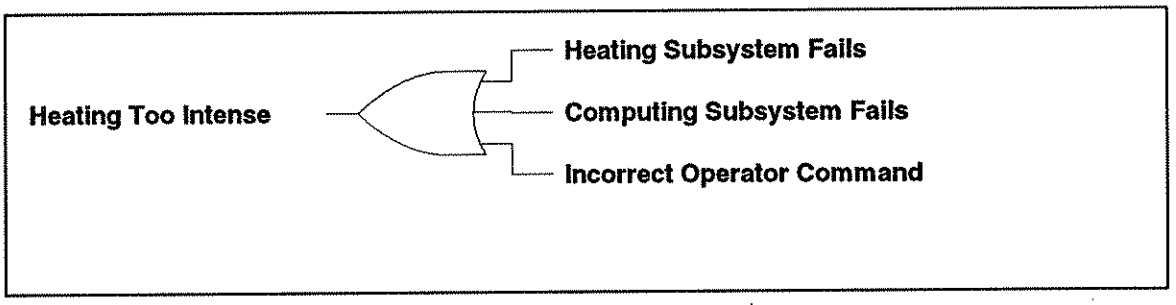


Figure 5.2 - Typical Section of the Original System Fault Tree

to a device hazard. Although the physical device and even the controlling software could be made arbitrarily reliable, there is no way to make the operator more reliable. Safeguards had to be added to make the system safer.

The simplest form of safeguard is an emergency stop subsystem. Such a system monitors device actions and halts the system in a safe manner if unsafe operation is detected. By physically separating the emergency stop subsystem from the rest of the system, independence of failures can be assured. This allows the failure probabilities of the system to be bounded by that of the emergency stop subsystem. This is precisely the approach used in commercial nuclear reactors [13]. The inclusion of an extremely reliable shutdown system permits the controlling system to be built with considerably less stringent safety requirements. The addition of such a subsystem results in the subtree shown in Figure 5.3.

In principle, a correctly implemented emergency stop subsystem must be able to detect safety violations and take preventative action, and it must be independent of the causes of these violations. Any emergency stop subsystem added to the MSS cannot fully satisfy these criteria, as it must depend partly on the expertise of the operator to detect a hazardous situation. In particular, the operator cannot be relied upon to detect his own mistakes. Furthermore, if a hazard is caused by a failure in the imaging subsystem, there is

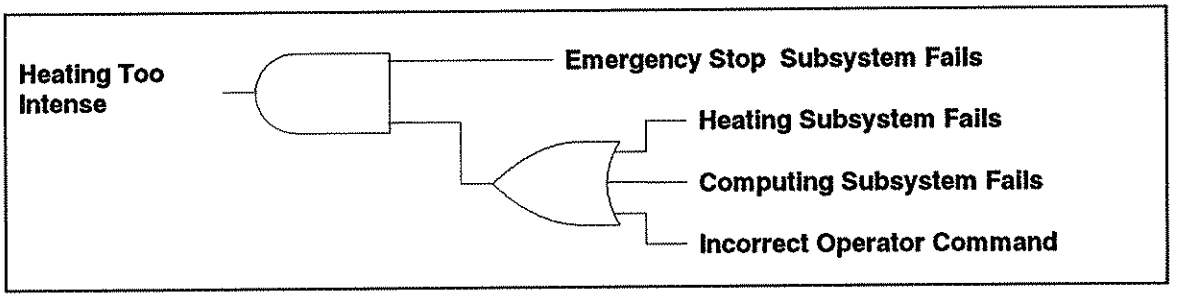


Figure 5.3 - Fault Tree After the Addition of an Emergency Stop System

no means of detecting the seed position independent of the imaging subsystem. Though the emergency stop subsystem cannot be relied upon to catch all safety violations, it is nevertheless an important precaution, and must be made as reliable as possible. Furthermore, the emergency stop system serves as a basis for additional safety measures, which can be relied upon to detect most subsystem failures.

Additional Safety Measures

The MSS safety analysis indicates the need for other safety measures. After considering numerous alternatives, it was decided that additional hardware and software functionality was necessary. The hardware took the form of sensors able to monitor the operation of each subsystem, and the software consisted of command checks to filter out operator commands that are obviously incorrect. Sensors are particularly important, as without them a single device failure could lead directly to a system hazard. By providing the computing subsystem with feedback regarding each device's actions, software can detect subsystem failures and invoke the emergency stop system as required.

The addition of sensors leads to additional software functionality requirements. The software is responsible for monitoring the sensors and responding as necessary to any detected problems. After several incorrect attempts, a subtree that reflected all of the updated failure scenarios was found. This subtree is presented in Figure 5.4. As this figure demonstrates, there are two basic classes of failure, either of which can lead to a system hazard. These are input errors, in which a hazardous command is entered by the operator and accepted by the computing subsystem, and execution errors in which the primary service and the secondary monitoring service fail coincidentally.

When the operator commands a hazardous action, the cause can be either an operator judgment error, or a correct conclusion drawn from invalid data provided by the

computing subsystem. The hazardous command is accepted by the computing subsystem when either no means of detecting the invalid input is available, or such checks exist but are implemented incorrectly.

The primary device service is considered to have failed if either the device itself fails or the computing subsystem errantly commands equivalent device behavior. A secondary monitoring service failure requires that either the sensor fail, or the computing subsystem fail to monitor the sensor or respond correctly to a hazardous condition. If, for example, the hazard involved the heater heating too intensely, the monitoring software need only halt the system if the sensor value exceeds that commanded. However, it is impossible for the software to distinguish a device failure from a sensor failure. Thus, if the sensor reports a value lower than that requested, either the device or the sensor must have failed. If the device failed, it must be shut down. If the sensor failed, the device could actually be operating at any level. In either case, the only safe course of action is to halt the system.

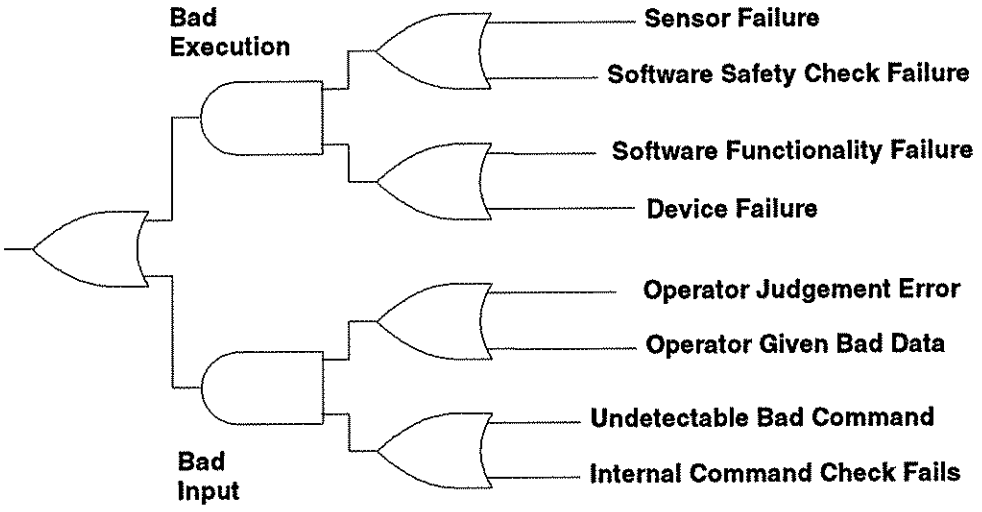


Figure 5.4 - Generic Device Failure Template

Figure 5.4 represents a generic template that was applied to each device failure mode in creating the completed fault trees. Where a branch of the subtree was not applicable to a particular device failure mode, that branch was omitted or had a probability of zero assigned to it. Examples of this are the deletion of the *bad input* subtree for failures that do not rely on user input, and omission of the *software functionality failure* branch for failure modes that cannot be software-initiated. Finally, in cases where a failure mode involves a device failing to switch off, an alternative means of turning the device off is made available to the monitoring software, generally by cutting power to the device.

Operator Presented Bad Data Subtree

The *Operator Presented Bad Data* subtree is of particular interest to the software engineer. As this subtree forms part of a number of fault trees, a failure of this kind could contribute to any number of hazards. The actions that the operator will take when presented with incorrect information cannot be predicted. Thus there is no way to determine which types of failures will cause the operator to initiate an action hazardous to the patient. The systems engineer has no choice but to require that all data presented to the operator be deemed safety-critical.

It is impossible to ensure that the software-managed displays convey the actual state of the system accurately. The displays can only be correct to the extent that the software is able to determine the state of the system. The software can be provided incorrect data by the devices responsible for reporting the system state. Although the software will often be able to detect such problems, there are cases that cannot be detected. In these cases, the possibility of incorrect data being presented to the operator must be considered an accepted risk.

In developing this subtree, it was deemed necessary to distinguish between the causes and effects of software failures. The systems safety analysis was concerned only with the effects of software failures and their ramifications on system safety. The factors that could cause a particular software failure mode were excluded from this analysis. In truth, they were included in the analysis until a late date, when it was discovered that this was entirely unnecessary. The systems engineer is only concerned with determining the software safety requirements. The software engineer must perform his own analyses to ensure that all possible causes of each safety-critical software failure are identified and prevented.

Thus the *Operator Presented Bad Data* subtree attempts to highlight the subsystem failures that could lead to incorrect data being presented. When a device or operator error can be detected, a means of detection is presented. Those failures that cannot be detected are classified as risks accepted by the systems engineers. When the device information is correct, exact requirements for presenting this information to the operator must be provided. It was found to be far easier to define what constitutes correct data presentation, than attempt to define all of the types of bad data that had to be avoided.

It is unfortunate that there is no small subset of the data presentation activity that can be labeled as safety-critical. Nevertheless, the relatively high probability of operator failure requires that a great deal of attention be paid to this process. This subtree clarifies the responsibilities of the software engineer. By isolating the subsystem failures that can cause incorrect data, the specifications for the software can be made more explicit in terms of what the software must do to tolerate the failure of other subsystems.

5.4 Deriving Software Safety Requirements

The system fault trees were created for the purpose of deriving software safety requirements. Although they cannot be extracted formally, a rigorous manner for determining the software safety requirements directly from the system fault trees was discovered. Using this method permitted a point-by-point correspondence to be demonstrated between the nodes of the fault trees and the software safety specifications. This correspondence lends a great deal of strength to the argument that the requirements listed are in fact those that are needed.

Each leaf node of the fault tree that represents a device failure was used to derive an entry in the software's recovery functionality specification. These contain a means of detecting whether the device has failed, and instructions for dealing safely with a detected device failure. Alternately, where the device failure cannot be detected, restrictions are placed on normal operation in order to preclude the undetectable device failure from causing a hazard.

Each leaf node representing a software failure actually represents a software failure that contains an unacceptable level of risk. As a result, each of these software failures had be disallowed as part of the software failure interface specification. Each such node in the tree required the creation an entry in the failure interface specification. These entries in the specification either take the form of a software property that must be preserved or, equivalently, a means of detecting and safely tolerating the software failure.

Figure 5.5 presents an example of precisely such a derivation. This corresponds to a single subtree contributing to the *Seed Heated Incorrectly* hazard. Note that the specifications are given in English rather than a formal specification language. This

stresses that the fault trees can be used to derive specifications in any particular notation, and that the introduction of formality constitutes an entirely separate step. The complete software safety requirements for the MSS are provided in Appendix A, embedded in the fault trees from which they were derived.

5.5 Creating the Formal Specifications

The Use of Formal Specifications

Although the requirements embedded in the fault trees constitute a valid safety

RF Heating Too Intense

Sensor Failure
If the value read from the sensor differs from that written to the RF source, one of the devices has presumably failed. Turn the RF source off. As the source itself may have failed, it cannot be relied upon to switch itself off - remove power from the RF source.

Software Safety Check Failure
If at any time the value read from the sensor is greater than the maximum permissible value, turn the RF source off.

Software Functionality Failure
At no time may the value written to the heater exceed that commanded by the user.

Device Failure
Indistinguishable from Sensor Failure - use the same specification.

Operator Judgment Error
An accepted risk, that cannot be further analyzed.

Operator Given Bad Data
The root node of a subtree presented elsewhere.

Undetectable Bad Command
An accepted risk, that cannot be further analyzed.

Internal Command Check Fails
If the value input by the user exceeds the maximum permissible, reject the input. If the value exceeds the reasonable level, require user confirmation of this intent. Properly implement any other rules provided by the domain experts.

Figure 5.5 - Derived Software Safety Requirements

specification, certain properties desirable of such a specification are noticeably absent. As mentioned earlier, the purpose of having a software safety specification is to precisely define the requirements that the software must adhere to in order to contribute meaningfully towards system safety. The specification that is present in the fault trees does not satisfy this requirement as it is both incomplete and ambiguous.

The specification is incomplete in the sense that it is too "high-level" a specification, using terms that are not defined, and relying on the reader's intuition to provide the necessary details. It is ambiguous in that, being written in a natural language, it relies on the reader's interpretation agreeing with the intentions of the specifier. The fact that English lacks formally defined semantics makes it unsuitable for the communication of formal properties. Although the English that is used in the fault trees was carefully chosen to minimize ambiguity, that cannot be guaranteed. Thus the format of this specification is completely inappropriate.

The solution to this problem that was opted for was to provide the software safety requirements in a formal specification language. These languages rely on precise mathematical notations to preclude ambiguity, and require that all "high-level" constructs be entirely defined from the language primitives, ensuring at least partial completeness. Although such languages cannot guarantee that a specification will be correct in any meaningful sense, they serve to eliminate the two greatest sources of errors [32].

Formal specifications offer other benefits, some of which are particularly applicable to software safety concerns. Formal specifications provide the exactness required for assessing the correctness of the implemented software. Formal specifications also eliminate the possibility of miscommunication between the systems engineers and the software engineers. The potential for assignment of legal responsibility should cause

software engineers to be particularly concerned that the tasks they are assigned are precisely defined.

A formal specification of software safety requirements also provides the systems engineer with the capability to model the safety properties of the software, without having to wait until the software is constructed, and without resorting to the "run it and find out" school of software engineering. The latter approach is particularly unsuitable to software safety issues, where the discovery of a software failure could involve causing the hazards that it is supposed to be avoiding.

The traditional software lifecycle, in which validation is used to determine the correctness of the specification is not appropriate for applications in which there is no margin for error. Finally, formal specification languages have proven themselves to be extremely cost-effective, as they do not permit specification decisions to be deferred until the implementation or verification phase. This is a common and expensive practice that is precluded by the use of formal specification languages.

The Formal Software Safety Specifications of the MSS

The formal software safety specifications for the MSS that were created as part of this experiment are presented in Appendix C. These are written in the formal specification language, Z [33, 34, 35, 36]. Appendix B contains a brief introduction to the features of the Z language that are present in the specifications.

The specifications presented here are not perfect. The low-level device interfaces are not included. The model of user-interface activities has been drastically over-simplified. Key concepts such as time granularity and screen colors have yet to be fully explored. The command checks on user-input have not been specified. The models of

seed movement through brain tissue and tissue heating over time have been omitted. In reality, these specifications are a very rough first draft. Because the actual system specifications are in a state of change, these cannot be finalized at this time.

Nevertheless, the specifications provided here precisely demonstrate the problems facing the software specifier. Precise specification of software safety properties *is* a complex procedure. Yet without such a specification, the systems engineer must rely on vague notions of what the software should do, and the software engineer must guess the intentions of the systems engineer in creating the software. A formal specification such as that presented here provides a precise basis for communication between the two parties. Furthermore, the formal specification performs the valuable task of ensuring that all specification issues are dealt with in a precise manner during the specification phase.

The remainder of this section will serve as a commentary for Appendix C, highlighting the areas of particular interest, and explaining some of the decisions made.

Command Queues and Timing Functions

In order to permit specification of timing constraints, the concept of time must be formalized. It was decided that modeling time as a natural number was fairly simple, yet offered the flexibility required. A time value represents the number of milliseconds elapsed since the start of the surgical operation. It was determined that this was a precise enough time granularity for all safety purposes.

It was determined that functions over time were the correct way in which to model device operation, thus the X-ray intensity, heating intensity, and coil currents could be modeled as functions over time. It was quickly found that the actual device state is a meaningless concept, as it cannot be known to the software. Instead, two separate

functions were provided for each device: one representing the required device activity and the other representing the sensor feedback. It was thus trivial to specify that when these functions disagree a shutdown must be initiated.

User commands do not lend themselves to being modeled as functions over time. Instead, it was determined that a sequence containing user commands, including a time stamp indicating when the command was issued, was the best model possible. This is referred to in the specification as the user command queue. The user command queue contains user commands that affect each of the devices. It was decided that the abstraction of separate user command queues for each device was a more flexible choice. Thus separate user command queues are provided for each device, and a correspondence between these and the single user command queue is specified.

Each entry in the user command queues for each device translate to one or more commands to the actual device. For example, a timed heating command must actually be implemented as a start command followed by a separate stop command to the heating subsystem. Thus another level of device queues, containing the actual device commands was required. Each entry in the user command queue requires one or more entries in the device command queue. Furthermore, all commands that initiate a device action must correspond to user commands. However, device commands to terminate an action can either correspond to a user command, or to a detected safety violation requiring device shutdown. Thus the functions that model the devices required behavior are derived from the device command queue, but the command queue can be affected by the contents of the device behavior function.

The Display of Safety-Critical Information

The software safety requirements contain entries such as "the position of the seed on the screen must coincide with the position of the seed as reported by the imaging subsystem". Although this is intuitively clear, it must be precisely formulated. Thus a model of the screen, including models of the various windows, must be specified. The screen is represented as a table of pixels, where each pixel must assume some color. The dimensions of the screen must be given as the dimensions of the table.

The image on the screen is determined from the contents and positions of an ordered sequence of windows. Each window consists of a table containing its image, its location and size, and its status on the screen. The screen image is specified at each point as the contents of the uppermost window containing that point, appropriately shifted according to the origin of the window. The three windows containing the image of the seed are specified as functions over time, in order to permit seed movement to be visually tracked. At each time, the contents of the image window must correctly display the position of the seed as determined at that time relative to an image of the patient's skull. Furthermore, the appropriate cross-sectional image to display is determined by the depth of the seed relative to the view selected.

In addition to the position of the seed, it is a safety requirement that the software provide the correct feedback to operator commands. Thus as the operator selects a movement command, an arrow must be drawn on the seed presenting the direction of movement. When a timed heat command is selected a ring is used to present the estimated extent of the projected heating. As a heating command progresses, a circle representing the estimated extent of the actual heating is maintained. All of these visual

displays are superimposed by a single function which also scales the images to fit the current window size.

Specification of Seed Movement

The specification of seed movement uses the estimations for magnetic fields and seed movement through brain tissue provided by the systems engineers [37]. In keeping with the spirit of specifications, rather than providing an algorithm for determining the coil currents based on desired seed movement, the specification states the criteria that such an algorithm must meet. These take the form of a mathematical acceptance test, projecting the seed movement caused by the provided coil currents and determining whether the projected seed movement is approximately the same as that commanded. This permits the implementor to use any implementation method from iterated estimations to table lookups.

Because the forces required for seed movement are dependent on the location of the seed, the seed movement specification must include all of the image detection specification. Like the coil current determination, the specification for the image detection elements does not state how the objects are to be extracted from the images, but merely what the characteristics of a correct solution are. Thus instead of providing blob-finding routines, the specification contains the precise definition of a "blob" and asserts that a correct algorithm will detect all blobs present in the image.

This specification also relies on three separate coordinate systems provided by the systems engineers [24]. These are the image coordinate system, the absolute coordinate system, and the relative coordinate system. The image coordinate system is used to measure object locations within the images. Because the images can be distorted, fixed

"helmet markers" are included in the images. By observing the effects of the distortion on these markers, the absolute locations of the seed and skull markers can be determined.

The absolute coordinate system is actually relative to the coils. Thus the coils and imaging hardware all have fixed absolute locations. For ease of calculations, the axes of this coordinate system are defined by the centers of each coil. The relative coordinate system is used to locate the seed relative to the cranium. Thus the skull markers have fixed relative coordinates, and can be used in a manner similar to the helmet markers to derive a mapping from absolute to relative coordinates. Finally, one additional coordinate system, the unified coordinate system (UCS), is used in displaying the preoperative MR images. This coordinate system is defined by the resolution of these images, and can be determined by having the operator identify the skull markers on the displayed MR images.

5.6 Summary

The purpose of this experiment was to demonstrate the feasibility of defining software safety in terms of a specification of safety requirements and an implementation faithful to that specification. To make these definitions useful, they had to be shown to integrate smoothly with existing systems engineering techniques. This experiment resulted in the discovery of amendments to well-understood systems engineering techniques which allowed the software safety requirements to be determined in a rigorous manner. These requirements were then presented in a concise and unambiguous format that can be used to model the software safety properties, provide a basis for formal implementation and verification techniques, and can even serve as a legal contract between the systems and software engineer. Stressing a rigorous approach to software safety enabled potential safety problems to be uncovered during system specification rather than during software verification, or worse still, system operation.

Chapter 6 - Conclusions & Prospectus

6.1 Testing the Theoretical Definitions

Software safety can never be fully separated from systems safety. Systems safety defines the meaning of software safety for any particular system. However the process of creating safe software can be separated from systems safety concerns. The study of software safety has two separate components: the systems engineering component, concerned with determining the software safety specifications from the system safety analysis, and the software engineering component, concerned with creating and verifying the software as safe. The latter component is isolated from system safety concerns. The software safety specification enables this isolation by defining the system safety concerns in software engineering terms. The software engineer can use any available software engineering means to make the software safe.

Because the software safety specifications are driven by the system safety requirements, they are not necessarily arbitrary. Classes of physical systems will have similar software safety requirements. Because of this, the study of software safety implementation issues should not be performed in isolation, but should be driven by the safety requirements of actual systems. Specific software architectures can be tailored towards the safety requirements of particular classes of systems. The MSS, being a fail-safe system, suggests certain software architectures as particularly effective. These will be explored in future experiments on the MSS.

Specifying Software Safety

The process of specifying the MSS software safety requirements demonstrated that all of the system safety ramifications that were uncovered could be captured in the

specification. From a theoretical standpoint, any property that can be implemented can also be specified. Thus, any safety related property that is desired of the software can be specified as part of the software safety specification. If a property cannot be specified, it cannot possibly be implemented.

It has been suggested that software safety, like software robustness, cannot possibly be specified. Software robustness is the property that software should do something reasonable when it encounters an unanticipated situation. This is only possible when "reasonable" can be exactly specified. Similarly, software safety has been suggested to be that the software do something safe when it encounters an unanticipated situation [9]. Clearly, for this to have meaning, an exact definition of "safe" is required. To accept that software safety cannot be specified would constitute an admission that safe software cannot be built.

Software Complexity

In this particular experiment, the software component was not too complex to be dealt with as a component in the system safety analysis. In the general case, this should continue to hold true. Because the systems engineer is only concerned with individual software properties, these are usually far less complex than the functional properties of the software component. Thus, the software safety specifications are already decomposed by properties.

Two means of software decomposition are immediately apparent. One is to decompose the software into layers of services, much as complex software projects are structured. This should not be a systems engineer's concern. Decomposing the software specification in this manner needlessly introduces software implementation details. The other means of software decomposition is to break the software into modules, each

concerned with some particular aspect. Although this is certainly possible, it offers no benefits over the existing method of decomposing the software by properties.

The formal specification of the MSS illustrates that even the specification of a single property can be quite involved. In these instances, it is useful to decompose the specification in layers, so as to permit complexity to be controlled and common sections of the specifications to be reused. This does not impose restrictions on the decomposition of the implementation, although reuse of specifications and related code promises to make implementation and verification much easier [38].

Categorizing Safety Requirements

The theoretical categorizations of failure interface specifications and recovery functionailty specifications proved to be very useful during the MSS safety analysis. It was observed that quite often a software failure could emulate a device failure, or a device failure could emulate a software failure. As a result, there is a great deal of overlap between the events addressed by the failure interface specification and the recovery functionality specification. During the MSS safety analysis, these categorizations provided a useful checklist for determining whether all possible causes of a failure had been considered.

The MSS safety analysis process revealed several other possible categorizations of safety requirements. One such categorization is the division of safety requirements into active failures and passive failures. Under this scheme, active failures are those that are caused by the software initiating an incorrect action. Passive failures are those that are due to the software's inactivity. These categorizations are of particular interest to the implementor, as they determine if a simple guard on an output will suffice, or whether more elaborate software measures are required.

Another useful categorization is the separation of software services into sporadic and periodic events. This is very useful for the purpose of modeling or prototyping timing constraints. In addition, processor load requirements, availability, and scheduling algorithms can be analyzed. Neither of these additional categorizations require that the theoretical framework be changed. From a theoretical standpoint, the original categorizations are still the most elegant. This does not make the additional categorizations any less useful, nor does it rule out the possibility that other useful categorizations exist. It will be a primary goal of future experiments to seek these out.

Summary

The process of creating the MSS safety specifications demonstrated the usefulness of the theoretical definitions. The theoretical categorizations proved useful in revealing the powerful technique of software failure modes. It was determined that software safety can, in practice, be decomposed into the problems of creating a safe specification, and properly implementing the specification. Most importantly, the software safety specifications are software properties expressed in software engineering terms and can be attacked using general-purpose software implementation and verification techniques. Nevertheless, software safety requirements are not arbitrary, and future software safety research should be driven by the safety needs of actual systems.

6.2 Software Failure Modes

The MSS safety analysis corroborates the assertion that software failure modes are a viable concept. They are determined in a straightforward manner and easily manipulated. However, because other approaches to software safety are informal, there is no way to determine formally whether the use of software failure modes is any more or less

safe than alternative approaches. During the MSS safety analysis, no software failures were found that could not be determined by enumerating the software failure modes. Future experiments in software safety specification will further test the viability of this approach.

In creating the MSS software safety specification, software engineering expertise was not a prerequisite for determining the software failure modes. A knowledge of how the software can effect other devices and a basic understanding of the special properties of software are all that were required. From this basis, the software failure modes are easily enumerated - they are simply all possible combinations of software effects on other devices. The one difficult area is with determining beforehand the effects of providing the operator with incorrect data. In this area, expertise in the psychology of user interfaces would be more beneficial than software engineering knowledge.

6.3 Maintaining a High Degree of Rigor

System safety analyses are used to determine whether a given system satisfies a specific definition of system safety. If the system is deemed unsafe, a solution to the offending problem can often be derived from the safety analyses. These same analyses can be performed on systems containing a software component. In order to determine the safety-critical software properties and the associated levels of assurance required, all software failures are initially assigned arbitrarily high failure probabilities. Those properties that must be reduced in probability in order to assure system safety then become part of the software safety requirements. From these individual requirements, a formal specification of software safety specifications can be derived. The process of translating the informally-stated software safety requirements into a formal specification is

necessarily informal. At present, finding rigorous techniques for deriving formal specifications is an area of active research.

The Suitability of Formal Specifications

The MSS specification experiment did not answer the question of whether formal specification languages are a suitable means of communicating software safety specifications to the degree desired. This was primarily due to a lack of formal specifications expertise. The systems engineers did not write the formal software safety specifications as was originally hoped. The systems engineers were able to read and understand the formal specifications, but whether they would be able to write them effectively was not tested. The process of creating the formal specifications illuminated a number of potential safety problems, that went uncovered during previous steps. In particular, the formal specification of the image detection processes raised a number of potential problems that were not addressed in the existing prototype software.

Z, the formal specification language that was chosen, turned out to have several undesirable properties. Although excellent for expressing functional properties, Z has no built-in provisions for non-functional properties, in particular timing properties. Although constructs were created for the purpose of specifying timing constraints, they were very clumsy when compared to the timing constructs present in other, less general-purpose, languages [12, 39, 40]. Z also has no simple means for defining low-level device interfaces, although these could also be constructed. Future work will experiment with the use of multiple specification languages to specify different properties.

Natural Language Accompaniment

A natural language accompaniment to the formal software safety specification of the MSS was determined to be necessary. Because a Z specification can only specify relationships between abstract entities, a separate explanation of how to relate these to physical entities was needed. The Z specifications of device interfaces, the user input and output, and the notion of time had to be related to their real world counterparts.

Additional natural language accompaniment was used in a manner very similar to comments in an implementation language. Comments can be very useful for explaining structural decisions, describing potential pitfalls, and pointing out non-intuitive aspects of the specification. The comments should not replace understanding the body of the specification, as the comments cannot possibly capture the precision of the specification. Natural language accompaniments play an important role, in terms of both documenting the specification and describing its relation to real-world entities. However, the use of natural language accompaniment to a formal specification has to be very strictly controlled. They cannot replace reading and fully understanding the specification, or provide a translation of the specification, as that would defeat the purpose of having a formal specification.

Summary

Safety-critical applications require a degree of rigor beyond that typically encountered in software engineering. System safety analysis, although inherently informal, requires an extremely rigorous approach. This experiment has demonstrated that it is possible to rigorously derive a formal software safety specification from the system safety analysis. In this manner, the software engineer can be sure that his efforts in achieving

high assurance will contribute meaningfully towards system safety, and the systems engineer is able to precisely control how the software interacts with the remainder of the system. The applicability of this approach is directly dependent on the ability of the systems engineer to formally communicate the software safety requirements. Thus, future work will involve determining the properties of a formal specification language that are necessary for specifying safety properties.

6.4 Determining the Correct Process

Just as no single systems safety technique is applicable to all physical systems, no single technique can be used to analyze all systems with a software component. Nevertheless, the basic process outlined above is universally applicable. By amending systems techniques to deal with the special properties of software, software safety requirements can be rigorously derived. Systems engineering presently uses safety analysis techniques in order to determine whether a system, as specified, is safe. If this proves not to be the case, component requirements are altered or amended as necessary. This is a process that is accepted by systems engineering as the best available. The MSS experiment has demonstrated that this same basic process is applicable to systems with a software component, as long as caution is exercised.

Special Software Properties

This experiment demonstrates conclusively that existing systems safety techniques can be applied to systems containing software, in order to obtain the software safety requirements. In doing so, it is critical to understand the properties of software that are different from other system components:

- Software failures are design, not degradation failures.

- Software failures can appear as other device failures.
- Software failures are distinguished by their effects on other components.
- Different software failures are not necessarily independent.
- Multiple software components do not necessarily fail independently.
- Meaningful software failure probabilities are extremely difficult to assign.

Comparisons to Other Approaches

Because systems safety analyses are informal processes, one process cannot be formally demonstrated to be superior to another. A rigorous argument is presented as to why the process described here is superior to those that existed previously. Other processes attempt to demonstrate informally that the software cannot cause a system hazard, where systems hazards are a vague notion. The process introduced here attempts to formally demonstrate that software cannot cause a system hazard, where the systems hazards that can be effected by software are precisely detailed.

The only manner in which the other processes could uncover a problem that this process misses is if a software engineer were to find a system hazard that the systems engineer missed. However, the software engineer has only a vague concept of what a systems hazard is, and is working from the perspective of the implementation details. The systems engineer has a precise knowledge of what constitutes a systems hazard, and is working at the level of component requirements. It is not unreasonable to postulate that it is unlikely that the software engineer, handicapped as he is, will detect mistakes that the systems engineer has missed.

Summary

The process described here demonstrates how systems engineering techniques can be applied to systems containing a software element. With a basic understanding of the characteristics of software that differentiate it from all other components, existing systems engineering techniques can be amended to handle these differences. From a system safety analysis, software safety requirements can be derived directly. In this manner, the manners in which software can contribute to systems hazards can be determined by the systems engineer and communicated to the software engineer. This approach is clearly superior to those in which the software engineer attempts to determine the software's safety ramifications from the perspective of the implementation, after the code has been written.

6.5 Other Issues Raised

Measuring Software Safety

Because the software safety requirements encapsulate the safety ramifications on the software, assessing software safety is simply a special case of assessing reliability. Whereas reliability is concerned with the correct implementation of the specification, safety is concerned with a critical subset of the specification. Measuring software reliability, and thus software safety, is an open problem that this experiment cannot hope to answer.

Physical systems reliability is measured in terms of failure probabilities. Because the analysis of physical devices is concerned with degradation faults, the probability of failure is a meaningful concept. Software engineering has borrowed this measure, but because software failures are design faults, the notion of probability of failure is far less

meaningful. Nevertheless, in order to provide the systems engineers with a basis for systems safety analyses, software engineering must provide meaningful failure probabilities for safety-critical software properties.

This experiment was limited to software properties that are either verified or unverified. The safety critical properties must be verified, whereas other properties can be left unverified. The methods used for verification are left undefined, but it is assumed that methods exist for achieving arbitrarily high levels of assurance. This assumption is not necessarily valid. Given the complexity of the hardware-software platform and the complexity of the control software, formally proving any non-trivial property could be impossible. Nevertheless, this is a goal that must be targeted in order to make software safety a useful contributor to systems safety.

Cost of Software Safety

The cost of developing safe software will depend most heavily on the degree of assurance required by the systems safety analysis. The cost of safety can be defined as the difference between the cost of building software that is demonstrably safe, and the cost of unproved software. Because the degree of rigor used in the unproved software is subject to wide variation, this can be very difficult to measure.

The cost of developing safe software will also depend on the nature of the software properties to be proven. As a simple example, the software controlling a fail-safe system might be orders of magnitude cheaper to verify than software that must always deliver the correct answer. Similarly, a property with a simple acceptance test can be ensured by the inclusion of a run-time check, whereas a property for which no such test exists could require a more complete proof. Because the MSS is a fail-safe system, the results of this experiment can only be applied to determining properties of such systems.

The experiments performed on the MSS to this point have been concerned only with specification. Because the system safety analysis is not particular to software safety, it cannot really be considered as a cost of software safety. The creation of formal safety specifications is an additional cost. The creation of this specification averted a number of potential problems that might only have been discovered during verification or execution, where they could prove very costly. Thus, as in other cases where formal specifications are used, the cost of this phase is more than offset by the possible gains. The other phases of software safety - analysis, implementation, verification, and maintenance - will be examined in future experiments, and their costs will be determined at that point.

Implementation Concerns

Idealistically, one would like to imagine that implementation concerns should not effect the specification. Realistically speaking, this is impossible. Systems engineering is concerned not only with the ability to create a system, but with the ability to create a system at a reasonable cost. Thus, the systems engineers must be aware of the costs associated with software safety. This is not unique to software; systems engineer must have a grasp on the costs associated with every component in the system in order to design a safe and reliable system that can be built within a reasonable cost. Thus, the costs and difficulties of creating different classes of safe software must be understood by systems engineering. This increases the need for software safety cost metrics.

Comparisons to the Naive Approach

In assessing whether the processes presented here result in software that is any safer than that created under the "naive" approach, any answer must be speculative. The old implementation was created without a safety specification. Thus by the definition of

software safety presented here, the software cannot be judged unsafe, in that it does not violate its software safety requirements. In a systems safety sense, a system controlled by this software would be very unsafe, as a single device or software failure could lead to a system hazard. The new software safety specification exists without an implementation. It is fair to say that any reasonably good implementation of this specification will result in software that makes the system orders of magnitude safer. Thus, although extremely high assurance is a desirable property in safety-critical software, it would appear that the largest "payoff" comes from the safety analysis itself. Even if no further software safety measures are taken, the software implemented from a "safe" specification will be far safer than that derived from an "unsafe" specification. Thus, the safety of the implementation seems to depend more on the safety of the specification than on the safety of the implementation, although an errant implementation could nullify the benefits of a safe specification.

Defense-in-Depth

Although defense-in-depth can be used to make software safer, it must be applied with a great deal of caution. Independence of software failures cannot be assumed, and thus it would appear to have very little benefit. Where this experiment has found it to be useful is in the application of a series of software guards, each being progressively less stringent, but easier to verify. For example, it is difficult to verify that no heating command exceeds the user commanded duration, but relatively simple to verify that no heating command exceeds some specified maximum duration. Although the latter case is not stringent enough to eliminate the hazard entirely, it serves to reduce the effects of another software error. These two failures are not entirely independent; they both depend on the availability of a monitoring function. There are classes of failures for which they are independent, however, and thus this approach is not without some merit.

Summary

This experiment was not able to discover metrics for either software safety or the cost of software safety. However, the formal software safety specification does provide a basis for assessment, and permits general purpose assessment techniques to be applied to the software safety properties. Although no metrics for cost could be found, the tradeoffs involved in assigning safety-critical functionality to software are highlighted. This experiment has also revealed that the systems engineer must have some understanding of the difficulties of software implementation, though no more so than any other system component. Finally, this experiment suggests that the largest "payoff" is not in the creation of ultra-high reliability software, but in rigorously determining the software safety requirements at the system level.

6.6 Summary

There are many computer applications in which safety and not reliability is the overriding concern. Reduced, altered, or no functionality of such systems is acceptable as long as no harm is done. This report concerns the role of software in such systems and the definition of what it will mean for software to be viewed as safe. A *precise* definition of what software safety means is essential before any attempt can be made to achieve it. Without this definition, it is not possible to determine whether a specific software entity is safe. Informal, intuitive notions of safety must be rejected as they have been from the realm of system safety if for no other reason that to protect the legal interests of the software engineer.

It is claimed that software must be viewed as merely one of many components that make up a system. In the overall system context, software is no different from any of the

other components of which the system is composed. Viewing software as a system component, we present a definition of software safety based on the establishment of precise specifications for the software's response to its own failure and to the failure of other components. A consequence of the definition is that the software engineer is freed from responsibilities other than the correct implementation of certain parts of the software specifications. This facilitates placement of responsibility in the event that an accident does occur.

The experiment on the Magnetic Stereotaxis System documented here demonstrates that this approach to software safety is practically viable. An exact process for determining software safety requirements from the system safety analysis is outlined. From these requirements, a formal software safety specification is derived. This specification permits a precise and meaningful measure of software safety to be taken. The specification will also serve as a basis for future experiments in software safety regarding the construction, verification, and maintenance of safety critical software.

Software engineering is very limited in its ability to manufacture components known to be free from defects. But perhaps the biggest difference between software and other components, is that systems engineers do not fully realize the limitations of software. Software engineering is capable of building almost any software entity, but it finds great difficulty in building something that is verifiable. Unfortunately, software engineering has failed to make this point clear to the rest of the engineering world.

Past assessments of software safety have often relied on the assurance technology at hand, rather than on the required level of assurance. Software is labeled as safe when no detectable errors are found by suspect assurance techniques. By doing so, the software engineering community has misled itself into believing that its software is safe. Rather

than label the software safe under such circumstances, the software engineer must make clear to the systems engineer that the safety requirements on the software must be restricted for it to be considered safe in a meaningful way.

The approach to software safety taken here is based on the specifications and not existing implementation techniques. If it proves to be technically infeasible to build a specific software system and assure that it is safe according to the specifications, the systems engineer will have to narrow the safety requirements on the software or knowingly build systems with components for which there is no adequate assurance of safe operation. If he chooses the former, he has understood that software safety is not the panacea that some have made it out to be. If he chooses the latter, he is accepting the risks involved. In either case, the definition of software safety must be compatible with rather than confused with systems safety. This will ensure that future research into software safety assurance methods will not be complicated unnecessarily by systems safety concerns but will contribute to system safety assurance.

References

1. Sayet, C., E. Pilaud, "An Experience of a Critical Software Development," Proceedings of the 20th International Conference on Fault Tolerant Computing, Newcastle Upon Tyne, U.K., June 1990.

2. Traverse, P., "Dependability of Digital Computers On Board Airplanes," Proceedings of the International Working Conference on Dependable Computing for Critical Applications, Santa Barbara, CA, 1989.

3. Green, A., *Safety Systems Reliability*, John Wiley & Sons, New York, 1983.

4. Thomson, J., *Engineering Safety Assessment*, John Wiley & Sons, New York, 1987.

5. Leveson, N., P. Harvey, "Analyzing Software Safety," IEEE Transactions on Software Engineering, Vol. SE-9, No. 5, Sept. 1983.

6. Leveson, N., "Software Safety in Embedded Computer Systems," Communications of the ACM, Vol. 34, No. 2, Feb. 1991.

7. Leveson, N., S. Cha, T. Shimeall, "Safety Verification of Ada Programs Using Software Fault Trees," IEEE Software, Vol. 8, No. 4, July 1991.

8. Leveson, N., "Software Safety: Why, What, and How," Computing Surveys, Vol. 18, No. 2, June 1986.

9. Jaffe, M., N. Leveson, M. Heimdahl, B. Melhart, "Software Requirements Analysis for Real-Time Process-Control Systems," IEEE Transactions on Software Engineering, Vol. 17, No. 3, March 1991.

10. Leveson, N., J. Stolzy, "Safety Analysis Using Petri Nets," IEEE Transactions on Software Engineering, Vol. SE-13, No. 3, March 1987.

11. Leveson, N., "Safety," in *Aerospace Software Engineering*, AIAA, Washington, D.C., 1991.

12. Jahanian, F., A. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," IEEE Transactions on Software Engineering, Vol. SE-12, No. 9, Sept. 1986.

13. Parnas, D., A. van Schouwen, S. Kwan, "Evaluation Standards for Safety Critical Software," Technical Report 88-220, Dept. of Computing and Information Science, Queen's University, Ontario, May 1988.

14. Schlichting, R., F. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," ACM Transactions on Computing Systems, Vol. 1, No. 3, August 1983.

15. United Kingdom Ministry of Defense, The Procurement of Safety Critical Software In Defence Equipment, Interim Defence Standard 00-55 Issue 1, London, ENGLAND, 1991.

16. Draft Standard for Software Safety Plans, P1228, Software Safety Plans Working Group, Software Engineering Standards Subcommittee, IEEE Computer Society, USA, July 1991.

17. Petroski, H., *To Engineer is Human*, St. Martin's Press, New York, 1985.

18. Anderson, T., R. Witty, Safe Programming, Bit, Vol. 18, 1978.

19. Butler, R., "NASA Langley's Research Program in Formal Methods," Proceedings of the Sixth Annual Conference on Computer Assurance, Gaithersburg, Maryland, June 1991.

20. Hall, A., "Seven Myths of Formal Methods," IEEE Software, Vol. 7, No. 5, September 1990.

21. Leveson, N., "A Case Study of the Therac 25 Incidents," Meeting on Software for Critical Systems, New Orleans, December, 1991.

22. Knight, J., N. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-version Programming," IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, Jan. 1986.

23. Howard, M., M. Grady, R. Ritter, G. Gillies, E. Quate, J. Molloy, "Magnetic Movement of a Brain Thermoceptor," Neurosurgery, Vol. 24, 1989.

24. Wika, K., "A User Interface and Control Algorithm for the Video Tumor Fighter," Masters Thesis, University of Virginia, Charlottesville, Virginia, May 1991.

25. United Kingdom Ministry of Defense, Hazard Analysis and Safety Classification of the Computer and Programmable Electronic Systems Elements of Defence Equipment, Interim Defence Standard 00-56 Issue 1, London, ENGLAND, 1991.

26. Houston, M., "Position Statement on FDA Regulation of Software Controlled Medical Devices," Forum on Standards for High Integrity Software, Gaithersburg, Maryland, June 28, 1991.

27. Houston, M., "FDA-HIMA Conference on Regulation of Software," Risks Digest, P. Neumann, Moderator, Vol. 12, No. 60, Nov. 6, 1991.

28. Roland, H., B. Moriarty, *System Safety Engineering and Management*, John Wiley & Sons, New York, 1983.

29. Hayes, J., *Computer Architecture and Organization*, McGraw-Hill, New York, 1988.

30. Hunt, W., "FM8501: A Verified Microprocessor," Ph.D. Dissertation, University of Texas, Austin, Texas, Feb. 1986.

31. Dijkstra, E., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

32. Wing, J., "A Specifier's Introduction to Formal Methods," IEEE Computer, Vol. 23, No. 9, Sept. 1990.

33. Spivey, J., *The Z Notation: A Reference Manual*, Prentice Hall International, New York, 1989.

34. Spivey, J., *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge University Press, Cambridge, 1988.

35. Ince, D., *An Introduction to Discrete Mathematics and Formal Systems Specification*, Oxford University Press, Oxford, 1988.

36. Woodcock, J., M. Loomes, *Software Engineering Mathematics*, Addison-Wesley, Reading, Massachusetts, 1988.

37. Walker, W., K. Wika, E. Quate, M. Lawson, G. Gillies, R. Ritter, J. Molloy, M. Grady, M. Howard III, "WYNNMAG: A Finite-Element Algorithm for Calculating the Magnetic Fields and Gradients of Magnetic Manipulation Coils," Technical Report No. UVA/640419/NEEP90/105, University of Virginia, Charlottesville, Virginia, March 1990.

38. Garlan, D., N. Delisle, "Formal Specifications as Reusable Frameworks," Proceedings of the Third International Symposium of VDM Europe, April 1990.

39. Harel, D. "Statecharts: A Visual Formalism for Complex Systems," Scientific Computer Programming, Vol. 8, 1987.

40. Hansen, K. , A. Ravn, H. Rischel, "Specifying and Verifying Requirements of Real Time Systems," Software Engineering Notes, Vol. 16, No. 5, Dec. 1991.

Appendix A - System Fault Trees for the MSS

This appendix contains the completed system safety fault trees for the Magnetic Stereotaxis System. The fault trees are presented in a textual notation, where indentation and change of fonts is used to denote hierarchical progression. The default combination of entries at each level is "OR". When an "AND" combination between two levels at the same entry is required, it is denoted through the use of an explicit "and" at the end of the first line.

The root of each fault tree corresponds to a single hazard. The sole exception is the *Operator Presented With Bad Data* tree which actually forms a subtree of the other trees. At the lowest level, these fault trees contain the derived software safety requirements. Each node corresponds to an individual requirement, which is listed with the node.

Patient Injured by Exposure to X-ray

X-radiation too intense

Emergency Shutdown System Fails *and*
Bad Execution

Primary Failure *and*

Software Functionality Failure

The software must not command the x-ray to be more intense than the user directed.

Device Failure

The software must observe the sensor values whenever the device should be on, and stop the device if the sensed intensity is greater than the commanded intensity.

Safety Net Failure

Sensor Failure

The software must observe the sensor values whenever the device should be on, and stop the device if the sensed intensity is lower than the commanded intensity.

Software Safety Check Failure

The software must observe the sensor values at all times, and if ever the sensed intensity exceed the maximum allowed, stop the x-ray device.

Bad Input

Operator Provides Bad Input *and*

Operator Judgement Error

Accepted risk

Operator Given Bad Data

See separate subtree

Computer Accepts Bad Input

Undetectable Bad Command

Accepted Risk

Command Check Fails

If the user commanded x-ray intensity (or product of intensity and duration) exceeds the nominal limit, force user confirmation of the command.

If the user commanded x-ray intensity (or product of intensity and duration) exceeds the maximum limit, reject the input.

If any other user provided rules are violated, question or reject the input as required.

X-radiation too long

Too many x-ray pulses

It should be noted here, that bad execution is not considered here. This is because any spurious x-ray pulses will already be considered failures under "x-radiation at wrong time" and need not be duplicated here.

Bad Input

Operator Provides Bad Input *and*

Operator Judgement Error

Accepted Risk

Operator Given Bad Data

See separate subtree

Computer Accepts Bad Input

Undetectable Bad Command

Accepted risk

Command Check Fails

If the operator requests x-ray usage that would exceed an advisable threshold, force user confirmation of the command.

If the operator requests x-ray usage that would exceed the allowable threshold, reject the input.

If any other user provided rules are violated, question or reject the input as required.

X-ray pulse too long

Emergency Shutdown System Fails *and*

Bad Execution

Primary Failure *and*

Software Functionality Failure

The software must not command the x-ray to be on for longer than the user directed.

Device Failure

The software must observe the sensor values when the device is turned on, and stop the device if the sensed intensity is greater zero after the period expires.

Safety Net Failure

Sensor Failure

The software must observe the sensor values when the device is turned on, and stop the device is the sensed intensity is zero during that period.

Software Safety Check Failure

The software must observe the x-ray sensor at all times, and stop the device if the sensor ever indicates that it has been continuously on for longer than the maximum allowed duration.

Bad Input

Operator Provides Bad Input *and*

Operator Judgement Error

Accepted risk.

Operator Given Bad Data

See separate subtree

Computer Accepts Bad Input

Undetectable Bad Command
Accepted risk.
Command Check Fails
If the operator commands an x-ray pulse that is longer than the advisable period, request confirmation of the command
If the operator commands an x-ray pulse that is longer than the maximum duration, reject the input.
If any other user provided rules are violated, question or reject the input as required.

X-radiation at wrong time

Emergency Shutdown System Fails *and*
Bad Execution

Primary Failure *and*

Software Functionality Failure
The software may only command the x-ray to start when the user has requested such a pulse.

Device Failure
The software must observe the sensor at all times, and if ever the intensity is non-zero when the user has not commanded an x-ray, shut the x-ray off.

Safety Net Failure

Sensor Failure
The software must observe the sensor at all times, and if ever the intensity is not equal to the commanded intensity, or zero if no command is pending, shut the x-ray off.

Software Safety Check Failure
The software must never permit the x-ray to be on at the same time as either the coils are charged, or the heater is on.
The software must never permit the x-ray to be on immediately after another x-ray, without some operator command in between the pulses.

Bad Input

Operator Provides Bad Input *and*

Operator Judgement Error
Accepted risk.

Operator Given Bad Data
See separate subtree

Computer Accepts Bad Input

Undetectable Bad Command
Accepted risk.

Command Check Fails

If the operator requests an x-ray while the seed is heating or moving, deny the request.
If any other user provided rules are violated, question or reject the input as required.

Patient Injured by RF Heating

Direct Tissue Heating

Emergency Shutdown System Fails *and* Bad Brain Heating Model

Accepted risk.

Incorrect Coil Positioning

The software must remind the user to ensure correct coil positioning before the first heating command is accepted.
Yet to be fully addressed.

RF Subsystem Failure - Wrong Frequency

Negligible probability.

Seed Heated Incorrectly

Heating Too Long

Emergency Shutdown System Fails *and*
Bad Execution

Primary Failure *and*

Software Functionality Failure

The software must always turn the heating off within the user-supplied duration of turning it on.

Device Failure

After turning the device off, the software must monitor the sensor to ensure that it has gone off, otherwise the plug must be pulled on the device.

Safety Net Failure

Sensor Failure

During the heating period, if the sensor is not equal to the commanded intensity, shut the device off and pull the plug.

Software Safety Check Failure

If at any time, the sensor has been continually positive for a period exceeding the maximum allowed heating time, shut the heating off, and pull the plug.

Bad Input

Operator Provides Bad Input *and*

Operator Judgement Error

Accepted risk.

Operator Given Bad Data

See separate subtree

Computer Accepts Bad Input

Undetectable Bad Command

Accepted risk.

Command Check Fails

If the operator inputs a command whose duration (or product of duration and intensity) exceed the advisable limit, request confirmation.
If the operator inputs a command whose duration (or product of duration and intensity) exceed the maximum limit, reject the input.
If any other user provided rules are violated, question or reject the input as required.

Heating Too Intense

Emergency Shutdown System Fails *and*
Bad Execution

Primary Failure ***and***

Software Functionality Failure

The software must not command the heating to be more intense than the user directed.

Device Failure

The software must observe the sensor values whenever the device should be on, and stop the device if the sensed intensity is greater than the commanded intensity.

Safety Net Failure

Sensor Failure

The software must observe the sensor values whenever the device should be on, and stop the device if the sensed intensity is lower than the commanded intensity.

Software Safety Check Failure

The software must observe the sensor values at all times, and if ever the sensed intensity exceed the maximum allowed, stop the heating device.

Bad Input

Operator Provides Bad Input ***and***

Operator Judgement Error

Accepted risk.

Operator Given Bad Data

See separate subtree

Computer Accepts Bad Input

Undetectable Bad Command

Accepted risk.

Command Check Fails

If the user commanded heating intensity (or product of intensity and duration) exceeds the nominal limit, force user confirmation of the command.

If the user commanded heating intensity (or product of intensity and duration) exceeds the maximum limit, reject the input.
If any other user provided rules are violated, question or reject the input as required.

Heating at Wrong Time

Emergency Shutdown System Fails *and*
Bad Execution

Primary Failure *and*

Software Functionality Failure

The software may only command the heating to start when the user has requested heating.

Device Failure

The software must observe the sensor at all times, and if ever the intensity is non-zero when the user has not commanded heating, shut the device off, and pull the plug.

Safety Net Failure

Sensor Failure

The software must observe the sensor at all times, and if ever the intensity is not equal to the commanded intensity, or zero if no command is pending, shut the heating off.

Software Safety Check Failure

The software must never permit the heating to be on at the same time as either the coils are charged, or the x-ray is on.
The software must never permit the heating to be on immediately after another heating event, without some operator command in between the events.

Bad Input

Operator Provides Bad Input *and*

Operator Judgement Error

Accepted risk.

Operator Given Bad Data

See separate subtree

Computer Accepts Bad Input

Undetectable Bad Command

Accepted risk.

Command Check Fails

If the operator requests heating while the seed is moving or the x-ray is on, deny the request.
If a heating command is entered immediately after a previous heating command, question the user.

If any other user provided rules are violated, question or reject the input as required.

Patient Injured by Seed Movement

Seed Moves at Wrong Time

Emergency Shutdown System Fails *and*

Bad Execution

Primary Failure *and*

Software Functionality Failure

The software may only command a change in coil currents after the operator has commanded seed movement.

Device Failure

The software must monitor the coil sensors at all times that a movement has not been requested.
If a single coil quenches, quench them all immediately.
If a single coil begins discharging, attempt to stop the discharge. If this fails, pull the plug on all the coils.
If a single coil begins to charge, attempt to stop it. If this fails, pull the plug on the coils.

Safety Net Failure

Sensor Failure

The software must monitor the coil sensors at all times that movement has been requested. If at any time, a sensor does not match the current requested, pull the plug on the coils, or quench if the discrepancy is too great.

Software Safety Check Failure

If at any time, f(coil currents) = discharge, attempt to discharge, and failing that, pull the plug on the device.
If at any time, f(coil currents) = quench, quench the coils

Bad Input

Operator Provides Bad Input *and*

Operator Judgement Error

Accepted risk.

Operator Given Bad Data

See separate subtree

Computer Accepts Bad Input

Undetectable Bad Command

Accepted risk.

Command Check Fails

If the operator enters a movement command while the seed is heating, or the x-ray is on, reject the command.
If any other user provided rules are violated, question or reject the input as required.

Incorrect Movement

Emergency Shutdown System Fails *and*

Bad Brain Model

Accepted risk.

Bad Magnetic Model

Accepted risk.

Bad Input

Operator Provides Bad Input *and*

Operator Judgement Error

Accepted risk.

Operator Given Bad Data

See separate subtree

Computer Accepts Bad Input

Undetectable Bad Command

Accepted risk.

Command Check Fails

If the operator enters a command to move the seed more than the advisable distance, request confirmation.
If the operator enters a command to move the seed more than the permitted distance, reject the command.
If any other user provided rules are violated, question or reject the input as required.

Bad Execution

Incorrect Magnetic Impulse Applied

Incorrect Field Applied

Primary Failure *and*

Software Functionality Failure
Device Failure

Safety Net Failure

Sensor Failure
Software Safety Check Failure

Field Applied for Incorrect Duration

Primary Failure *and*

Software Functionality Failure
Device Failure

Safety Net Failure

Sensor Failure
Software Safety Check Failure

Distorted Or Unstable Field

Primary Failure ***and***

Software Functionality Failure
Device Failure

Safety Net Failure

Sensor Failure
Software Safety Check Failure

Operator Presented With Bad Data

Bad Status Information

Seed Heating Status Incorrect

At any time that the screen reports that the seed is cold, it must actually be cold.

Seed Momentum Incorrect

At any time that the screen reports that the seed is not moving, it must actually be still.

More than One Seed on the Screen

Software draws two seeds

At no time should there be more than one seed image on each cranium view window.

More than one seed on image returned by fluoroscope

If the cranium image returned by the imaging subsystem contains two seeds, inform the operator of the error.

Image capture board fails to clear previous image

If the cranium image returned by the imaging subsystem contains two seeds, inform the operator of the error.

No Seed on the Screen

Software fails to draw seed

At all times in which the system is in operator input mode, there must be at least one seed image on each cranium view window.

No seed on image presented by imaging subsystem

If no seed can be located on the cranium image returned by the imaging subsystem, inform the operator of this fact.

Seed hidden behind a marker

If no seed can be located on the cranium image returned by the imaging subsystem, and the projected seed location coincides with a marker location, inform the operator of this fact and request instructions.

Seed's Relative Position to Cranium Incorrect

Operator identifies marker incorrectly

After the operator identifies the markers, if the results are inconsistent, reject them. If, after any transposition of images, the results appear inconsistent, inform the operator and specifically bring up the possibility that the markers might have been identified incorrectly.

Operator places marker incorrectly

After the operator identifies the markers, if the results are inconsistent, reject them. If after any transposition of images, the results appear inconsistent, inform the operator and specifically bring up the possibility that the markers might have been placed incorrectly.

Seed location incorrect on imaging subsystem image

If the seed location appears to be inconsisten with projections, inform the operator. If a great deal of noise is detectable on the image, inform the operator that the image is suspect.

Software draws seed in wrong location

Whenever the seed is drawn on a cranium image, its location must coincide with the location of the seed on the image returned by the imaging subsystem, within the specified degree of accuracy.

MR Image of Cranium Incorrect

Operator provides incorrect image to computing subsystem

At the start of the operation, check that all of the images contain the proper check information. Present the images to operator to verify that they appear to be correct.

Image distorted or damaged since being taken

At the start of the operation, perform checksums on each of the images to ensure that they have not been corrupted since being recorded. Use row checksums and column checksums to minimize the possibility of damaged files going unnoticed. Present each of the files to the operator for verification that they are not damaged.

Software draws image incorrectly

The cranium images displayed on the screen must match the images verified by the operator. In addition, each must contain the proper checksums and check information.

In addition, the cross-sectional images displayed must agree with the seed position as determined from the image returned by the imaging subsystem.

Feedback to Command Incorrect

When the operator is providing a command, all feedback must accurately reflect the command being entered.

Appendix B

Brief Introduction to Z

B.1 Overview

The formal specification language used to specify the MSS software safety requirements is Z, pronounced "zed". Z was chosen primarily because of its growing popularity, and thus its acceptance in the software engineering community. Z is also very well suited for the specification of limited properties, permitting the specification of safety requirements without necessitating a complete functionality specification. In addition, Z is free of implementation concerns, permitting it to be used by non-computer scientists to specify what the software is to do without being mired in implementation-specific details of how it is to be achieved.

Although Z specifications appear cryptic at first, the Z language consists of a few simple extensions to the basic mathematics of computer science. At the heart of Z are the propositional and predicate calculi and very basic set theory. The extensions to these can be grouped into those which add to the expressive power of the language, and those which are merely part of the mathematic toolkit, which is somewhat akin to a standard library. The toolkit consists of numerous useful predefined symbols which, although extensions to the basic language, are common enough to warrant a standard symbology.

B.2 Basics

From propositional calculus, Z borrows the operators: $\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg$ and primitive symbolic constants, all of which behave as expected. Predicate calculus contributes the existential and universal quantifiers $(\exists, \forall)$ as well as the general notion of a predicate.

From set theory Z inherits the concepts of set enumeration, set construction, membership, power sets, and cartesian products. From these primitives, other operations such as union, intersection, and difference are constructed. Again, there are no suprises here. In addition, the predefined sets $\mathbb{N}$ and $\mathbb{Z}$, as well as their individual elements are included. The basic arithmetic and relational operations on the integers are also predefined. Z differs from basic set theory in the concept of types. In Z, all primitive elements are assigned types, and all sets must be of homogeneous types. Although not critical, this permits type-checking to be automated, thereby making the language more robust by removing many potential mistakes.

Sets of ordered pairs are classified as relations between the pairs' element types, which must be the same for each pair. Relations are further divided into functions and subsets thereof: injective, surjective, bijective, partial, and total. The notion of relation application and the predefined function ran (for range) and dom (for domain) are also defined in the language. Additionally relation composition and inversion and included. Note that all of the classification of sets into relations and functions adds nothing to the inherent expressiveness of the language, however the additional notation contributes significantly to clarity.

Every item in Z has an associated type. Z contains a single predefined type ($\mathbb{Z}$) and two derivative types ($\mathbb{N}$ and $\mathbb{N}_1$), but permits new primitive types to be defined and more complex types to be constructed. Primitive types are defined by enumeration. Z provides the power set and cartesian product constructors to permit arbitrarily complex types to be built from the primitive types. Other type constructors, such as functions or sequences, can be built from the basic constructors, although again these add no expressiveness. For example, a function of type $X \rightarrow Y$ is identical to the

type $\mathbb{P}(X \times Y)$. These both represent a set of 2-tuples, in which the first component is of type X, and the second of type Y. The simplicity of the first definition, as well as the additional rules governing the allowed contents of the set, make the first definition superior.

B.3 Decomposition

Z specifications require a notation to facilitate decomposition in order to minimize complexity. Towards this end, Z provides axiomatic definitions, generic definitions and schemas. Axiomatic definitions represent global invariants, generic definitions permit parameterized types in specifications, and schemas are used to define a group of interrelated symbols, and the relations among them. In addition, schemas can be used to represent activities, by the defining the relations on the variables before and after the activity (pre- and post- conditions).

Axiomatic Definitions

Axiomatic definitions are used to define global variables, functions and constants, and to restrict the legal values of the variables. For example, a global variable definition might appear as:

$$limit : \mathbb{N}$$
$$\hline$$
$$limit \leq 65535$$

and a definition of a square function might be written as:

$$square : \mathbb{N} \rightarrow \mathbb{N}$$
$$\hline$$
$$\exists\, n : \mathbb{N} \bullet square(n) = n * n$$

Note that all specifications in Z take the form of a declaration section containing one or more definitions followed by an optional constraints section consisting of a list of predicates that constrain the defined elements.

Generic Definitions

Generic definitions are used to define a set of specifications that are type-independent. Generic definitions accept one or more type parameters, and produce a different resulting specification for each different set of arguments. Generic definitions are particularly useful for defining collections of any given type, such as a simple collection or the ubiquitous stack example. A generic definition is included in a specification by instantiation with specific type parameters. When such an instantiation is present in a specification, it can be likened to a macro inclusion. As a result, once instantiated, the generic definition is no different from any other schemas.

An example generic schema follows, in which a sequence of some type, S, has a function applied to each element, resulting in a sequence of the new type, T:

$$[S, T]$$

$$apply : \mathrm{seq}[S] \times (S \rightarrow T) \rightarrow \mathrm{seq}[T]$$

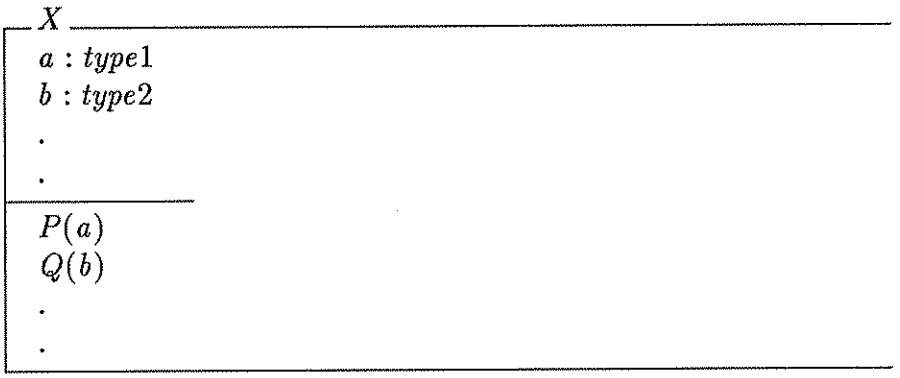
$$apply(x, f) = y \Leftrightarrow \#x = \#y \land \forall i : \mathbb{N} \mid i \in 1 \,..\, \#x \bullet y[i] = f(x[i])$$

Schemas

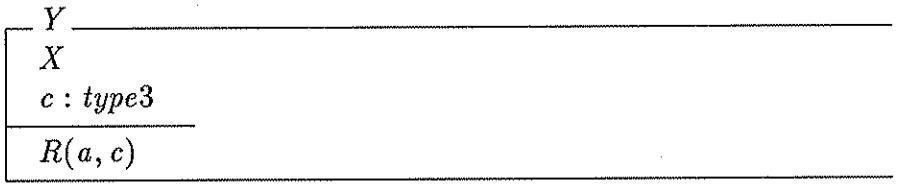
A schema permits a collection of definitions to be given an associated name, and allows rules to be imposed on the items. A schema is defined as:

$$X \mathrel{\widehat=} a : type1;\ b : type2;\ \ldots \mid P(a) \land Q(b) \land \ldots$$

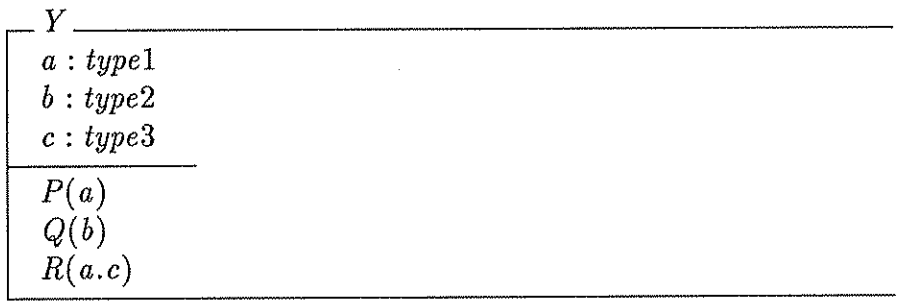
although it is semantically equivalent to, and more commonly written as:



Where any variables of any types can be defined, and P and Q are arbitrary predicates which can be replaced by any legal Z expression. Schema inclusion takes the form:



and is semantically equivalent to:



Another form of schema inclusion can be used to simplify complicated type definitions. For example, a complex number might be defined by one schema, and then used as a type definition in other schemas.

Complex

$Real : \mathbb{N}$

$Im : \mathbb{N}$

.

.

.

ComplexAdd

$\_ + \_ : complex \times complex \rightarrow complex$

$\forall x, y, z : complex \bullet$

$x + y = z \Leftrightarrow x.Real + y.Real = z.Real \land$

$x.Im + y.Im = z.Im$

Note that the "dot" notation allows one to distinguish the named components of variables given by a schema type. Note also that any relations specified to hold between the components of a complex number must be obeyed by x,y, and z, as they are implicitly appended to the complex-addition schema's predicates. Note also that the underscores in $\_ + \_$ permits inline operators to be defined.

Z does require rules for variable scope, and naming conflict resolution. Any variable name which is used in both the current schema and an included schema defaults to the current schema. If two variables in included schemas are given the same name, a conlict arises which must be resolved by explicit overriding, which is enacted through the "dot" notation demonstrated previously.

This concludes the description of basic Z functionality. The remainder of this chapter will introduce the predefined operators that are part of the mathematical toolkit and used in the formal safety specification of the MSS.

B.4 The Mathematical Toolkit

Functions

Z contains an extensive collection of predefined function classes. All of these have well-understood mathematical definitions. Although none of these add any expressive power to the language, they are valuable in terms of the conciseness they offer. Those which appear in the MSS safety specification are:

$\leftrightarrow$	*relation*
$\pfun$	*partial function*
$\fun$	*total function*

Z permits function or relation application to be expressed in a number of ways. Binary and unary operators can be defined using the inline notation, and are applied as are any other such operator. The MSS safety specifications consistently use paranthesized comma-delimited lists for other functions. In addition, the maplet operator ($\mapsto$) is used to denote a relational pairing.

Set Operations

The set operations that Z provides include some very familiar notations, such as union, intersection, subset, and set difference, and some very unfamiliar notation. The unfamiliar notation involves strange-looking symbols standing for very simple concepts. The domain restriction S $\dres$ R of a relation R to a set S relates x to y if and only if R relates x to y and x is a member of S. Similarly, the range restriction R $\rres$ T of R to a set T relates x to y if and only if R relates x to y and y is a member of T. As the names imply, these functions restrict a relation to only the set of elements in either the range or domain that one is concerned about [33].

The domain anti-restriction ($\ndres$) and range anti-restriction ($\nrres$) are the

logical complements of the restriction operators. Instead of restricting the relation to the elements explicitly included, these operators exclude all of the elements listed. These are easily defined as:

$$S \lhd R = (X \setminus S) \lhd R$$
$$R \rhd T = R \rhd (Y \setminus T)$$

where X and Y represent the universe of consideration for the domain of S and range of T.

The # operator is applied to a set (or sequence) to specify its cardinality (or length).

Non-Standard Practices

The use of tuples in the MSS safety specification is non-standard, yet consistent. A tuple is defined as a parenthesized, comma-delimited list of typed elements. Such a construct is assumed to have the same type as a schema with elements of the same types. Furthermore, tuples can appear in the range of functions. Thus if a function is defined as:
$Commands : time \rightarrow (value, interval)$
the following notation is acceptable:
$Commands(t) = (intensity, duration)$

Another non-standard practice that is used consistently in the MSS safety specification is the use of a shorthand notation to reference schema types that have only one element. In these cases, the name of the single element is often identical to the name of the schema type. For example, given the following definition of *NormalizedVector*, and an element, $F : NormalizedVector$, F is actually shorthand for $F.v$, and $F.x$ is shorthand for $F.v.x$.

NormalizedVector

$v : Vector$

$square(v.x) + square(v.y) + square(v.z) = 1$

This might present a problem when mechanical type checking means become available, and will thus be eliminated from the next draft of the MSS safety specifications.

One final deviation from standard Z present in the specifications provided here is the use of $\mathbb{R}$ to designate real numbers. Z has no predefined notion of real numbers, nor of real-number arithmetic, or trigonometric functions. These functions were used without being specified, the rationalization being that this was an exercise in software safety, not the specification of a real number system. Nevertheless, this is a deficiency which will have to be addressed in the future.

Sequences

The most important predefined complex type definition is the sequence. A sequence is defined as:

$$\mathrm{seq}[X] \mathrel{\widehat=} f : \mathbb{N} \pfun X \mid \mathrm{dom}\, f = 1..\#f$$

Thus a sequence consists of a mapping between a finite initial subset of the natural numbers and elements of some other base type. This is most easily understood as equivalent to an array in an implementation language. An important distinction, however, is the fact that specifications can reference sequence elements by their range, and thus could be best thought of an as associative array. However, the sequence is a mathematical entity, and how it is implemented is strictly an implementation issue.

Although the syntax for defining a sequences is $\{1 \mapsto t, 2 \mapsto u, 3 \mapsto v\}$, Z provides the alternative $\langle t, u, v \rangle$ for this purpose. Because sequences are actually functions, referencing takes the form of a function application. However, in order to make the specifications more understandable, the square-bracket notation is used to differentiate sequence references from function applications. To reference an element by its contents requires the use of the universal qualifiers, and also requires that the possibility of duplicate values be explicitly dealt with. Z provides several useful functions for dealing with

sequences. These are: catenation, head, last, tail, front, reversal, and filter. Catenation ($\frown$) is used to combine sequences; head and last provide the first or last element; tail and front are used to obtain sequences sans either the first or last element; reversal reverses the order of the sequence elements; and filter ($\upharpoonright$) is used in a manner similar to range restriction, to extract only the desired sequence elements. All of these fuctions differ from the similar set operations in the fact that they are often required to renumber their elements so as to not leave any gaps in the sequence. In the event that gaps are created in a sequence, *squash* can be invoked to remove them.

Tables

Although not part of the Z tool-kit, the table construct developed by Woodcock and Loomes [36] is critical to the MSS safety specification, and can be considered a primitive. The table is defined as a multidimension sequence - or a sequence of sequences of identical length:

$$[X]$$
$$Table[X] : \mathbb{P}(\text{seq}[\text{seq}[X]])$$
$$Table = \{t : \text{seq}[\text{seq}[X]] \mid \forall s1, s2 : \text{ran } t \bullet \#s1 = \#s2\}$$

$$[X]$$
$$rows, cols : Table[X] \rightarrow \mathbb{P}\,\mathbb{N}$$
$$\forall t : Table[X] \bullet$$
$$\text{rows } t = \text{dom } t$$
$$\text{cols } t = \text{dom}(\bigcup(\text{ran } t))$$

All of the table functions will be used heavily by the MSS safety specification. Although tables could be extended to three or more dimensions, this is not necessary for this particular specification.

B.5 Omissions

The Z language and the Z mathematical toolkit contains a number of other features, which are not critical to the understanding of the MSS safety specification. Where any of these are used they will be documented. Others, which are never used, will not be mentioned here. In particular, the Z specifications usually use a before and after notation which permits processes which alter schemas to be defined. Unfortunately, this basic Z capability contains no provisions for time restrictions, and thus is unsuitable for the MSS safety specification. The Z language also contains rigorous rules for defining functions with λ, and for specifying the binding of parameters to arguments in a schema, although these are not used here. Other constructs, such as the notion of transitive closure, are assumed to be well-understood, and although the Z toolkit provides rigorous definitions, these will not be provided here.

Appendix C

Formal Software Safety Specifications for the MSS

OrthogonalTransformation

$$M : Point \rightarrow Point$$
$$M : Vector \rightarrow Vector$$

$$\forall x, y, z : Point \bullet$$
$$Distance(x, y)/Distance(x, z) =$$
$$Distance(M(x), M(y))/Distance(M(x), M(z))$$
$$\forall v : Vector, p : Point \mid p.x = v.x \land p.y = v.y \land p.z = v.z \bullet$$
$$M(v) = M(p) - M(origin)$$

Distance

$$Distance : Point \times Point \rightarrow \mathbb{R}$$

$$Distance(a, b) = r \Leftrightarrow$$
$$square(r) = square(a.z - b.z) +$$
$$square(a.y - b.y) +$$
$$square(a.x - b.x)$$

Online

$$Online : Segment \rightarrow Point$$

$$Online(s, x) \Leftrightarrow$$
$$Normalize(s.begin - s.end) = Normalize(s.begin - x) \lor$$
$$Normalize(s.end - s.begin) = Normalize(s.begin - x)$$

Midpoint

$$Midpoint : Segment \rightarrow Point$$

$$Midpoint(s) = a \Leftrightarrow$$
$$2 * a.x = s.end.x + s.begin.x \land$$
$$2 * a.y = s.end.y + s.begin.y \land$$
$$2 * a.z = s.end.z + s.begin.z$$

Segment

$$begin : Point$$
$$end : Point$$
$$number : \mathbb{N}$$

FieldRing

$$FieldRing : \mathbb{R} \times Point \times Current \times \mathbb{R} \times \mathbb{R} \rightarrow Vector$$

$$\forall z : \mathbb{R};\ seed : Point;\ I : Current;\ turns : \mathbb{R};\ radius : \mathbb{R} \bullet$$
$$FieldRing(z, seed, I, turns, radius) = F \Leftrightarrow$$
$$\exists segments : \mathrm{seq}[Segment] \bullet$$
$$F = Sum(segments,$$
$$FieldBar(z, seed, I, 2 * pi * radius, turns)) \land$$
$$\#segments = grain + 1 \land$$
$$segments[1].begin = (radius, 0, 0) \land$$
$$segments[grain + 1].end = (radius, 0, 0) \land$$
$$\forall i : \mathbb{N} \mid i \in 1 \,..\, grain + 1$$
$$\bullet\ segments[i].number = i \land$$
$$\forall i : \mathbb{N} \mid i \in 1 \,..\, grain$$
$$\bullet\ segments[i].end = segments[i + 1].begin \land$$
$$\forall i : \mathbb{N} \mid i \in 2 \,..\, grain + 1 \bullet$$
$$segments[i].begin.x = radius * cos(i * delta) \land$$
$$segments[i].begin.y = radius * sin(i * delta) \land$$
$$segments[i].begin.z = 0$$
$$\land\ delta = 360/grain$$

FieldBar

$$FieldBar : \mathbb{R} \times Point \times Current \times \mathbb{R} \times \mathbb{N} \rightarrow (Segment \rightarrow Vector)$$

$$\begin{aligned}
&\forall z : \mathbb{R};\ seed : Point;\ I : Current;\ L : \mathbb{R};\\
&\quad turns : \mathbb{N};\ s : Segment \bullet\\
&FieldBar(z, seed, I, L, turns)(s) = F \Leftrightarrow\\
&\quad \exists R, a, x, y, z, r, t, phi : \mathbb{R};\ F : Vector \bullet\\
&\quad\quad \exists_1 pt : Point \mid Online(s, pt) \bullet\\
&\quad\quad\quad \forall pt2 : Point \mid online(s, pt2) \bullet\\
&\quad\quad\quad\quad Distance(seed, pt2) \geq Distance(seed, pt) \wedge\\
&\quad\quad R = Distance(seed, pt) \wedge\\
&\quad\quad a = max(Distance(s.begin, pt), Distance(s.end, pt)) \wedge\\
&\quad\quad x = (Mu * turns * I)/(4 * Pi * R) \wedge\\
&\quad\quad square(y) = square(a)/(square(a) + square(R)) \wedge\\
&\quad\quad square(z) = square(a - L)/\\
&\quad\quad\quad (square(a - L) + square(R)) \wedge\\
&\quad\quad B = x * (y - z) \wedge\\
&\quad\quad r = distance((seed.x, seed.y, 0), midpoint(s)) \wedge\\
&\quad\quad t = arctan(z/r) \wedge\\
&\quad\quad F.x = B * sin(t) * sin((2 * s.number - 1) * phi) \wedge\\
&\quad\quad F.y = B * sin(t) * cos((2 * s.number - 1) * phi) \wedge\\
&\quad\quad F.z = B * cos(t) \wedge\\
&\quad\quad phi = 180/grain
\end{aligned}$$

$$\begin{aligned}
&Pi : \mathbb{R}\\
&Mu : \mathbb{R}\\
&\text{---}\\
&Pi = 3.1415926535890\\
&Mu = 4 * Pi/10000000
\end{aligned}$$

field

$$field : Point \rightarrow (coil \rightarrow Vector)$$

$$\begin{aligned}
&field(seed)(c) = B \Leftrightarrow\\
&\quad c.M(B) = FieldCoil(c.M(c.position).z, c.M(seed), c.I,\\
&\quad\quad c.inner, c.outer, c.width, c.turns)
\end{aligned}$$

FieldCoil

$$FieldCoil : \mathbb{R} \times Point \times current \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{N} \rightarrow Vector$$

$$\begin{aligned}
&FieldCoil(z, seed, I, inner, outer, width, turns) = F \Leftrightarrow \\
&\quad \exists_1 c, d : \mathbb{R} \bullet \\
&\qquad c * 2 = inner + outer \wedge \\
&\qquad d * 2 = outer - inner \wedge \\
&\qquad F = FieldBand(z, seed, I, width, turns/3, c - d) \\
&\qquad\quad + FieldBand(z, seed, I, width, turns/3, c) \\
&\qquad\quad + FieldBand(z, seed, I, width, turns/3, c + d)
\end{aligned}$$

FieldBand

$$FieldBand : \mathbb{R} \times Point \times current \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow Vector$$

$$\begin{aligned}
&FieldBand(z, seed, I, width, turns, radius) = F \Leftrightarrow \\
&\quad F = FieldRing(z - width/3, seed, I, turns/3, radius) + \\
&\qquad FieldRing(z, seed, I, turn/3, radius) + \\
&\qquad FieldRing(z + width/3, seed, I, turns/3, radius)
\end{aligned}$$

coil

$$\begin{aligned}
&position : Point \\
&M : OrthogonalTransformation \\
&I : current \\
&inner : \mathbb{R} \\
&outer : \mathbb{R} \\
&width : \mathbb{R} \\
&turns : \mathbb{N}
\end{aligned}$$

coils

$coils : \text{seq}[coil]$

$$
\begin{aligned}
&\#coils = 6 \\
&\forall\, p : Point \bullet \\
&\qquad coils[1].M(p) = (p.x, p.y, p.z) \land \\
&\qquad coils[2].M(p) = (-p.z, -p.x, p.y) \land \\
&\qquad coils[3].M(p) = (p.x, -p.z, p.y) \land \\
&\qquad coils[4].M(p) = (p.z, p.x, p.y) \land \\
&\qquad coils[5].M(p) = (-p.x, p.z, p.y) \land \\
&\qquad coils[6].M(p) = (-p.x, p.y, -p.z) \land \\
&\forall\, i : \mathbb{N} \mid i \in 1 \,..\, 6 \bullet \\
&\qquad coils[i].M(coils[i].position).x = 0 \land \\
&\qquad coils[i].M(coils[i].position).y = 0 \land \\
&\qquad coils[i].M(coils[i].position).z < 0
\end{aligned}
$$

force

$force : Point \times coils \rightarrow Vector$

$delta : component \rightarrow Vector$

$force(seed, currents) = F \Leftrightarrow$

$\exists\, mu : Vector, partial : component \times component \rightarrow \mathbb{R} \bullet$

$mu = moment * normalize(field(seed)(currents)) \wedge$

$F.x = mu.x * partial(x, x) +$

$mu.y * partial(x, y) +$

$mu.z * partial(x, z) \wedge$

$F.y = mu.x * partial(y, x) +$

$mu.y * partial(y, y) +$

$mu.z * partial(y, z) \wedge$

$F.z = mu.x * partial(z, x) +$

$mu.y * partial(z, y) +$

$mu.z * partial(z, z) \wedge$

$\forall\, a, b : component \bullet$

$partial(a, b) * 2 * delta(b) =$

$field(seed + delta(b))(currents).a -$

$field(seed - delta(b))(currents).a$

$delta(a) = v \Leftrightarrow$

$\forall\, b : component \bullet$

$a = b \Rightarrow v.b = Delta \wedge$

$a \neq b \Rightarrow v.b = 0$

$component \mathrel{\widehat{=}} x, y, z$

Vector

$x, y, z : \mathbb{R}$

Point

$x, y, z : \mathbb{R}$

ComputeMove

$$ComputeMove : Point \times Point \times (Point \rightarrow Point) \rightarrow Vector$$

$$ComputeMove(seed, target, rel2abs) = v \Leftrightarrow$$
$$\quad v = rel2abs(target) - rel2abs(seed)$$

CoilCurrents

$$CoilCurrents : Point \times Vector \rightarrow coils \times interval$$

$$CoilCurrents(seed, move) = (currents, duration) \Leftrightarrow$$
$$\quad ProjectMove(seed, currents, duration)\ approx\ seed + move$$

ProjectMove

$$ProjectMove : Point \times coils \times interval \rightarrow Point$$

$$ProjectMove(seed, c, t) = target \Leftrightarrow$$
$$\quad t < 1 \land seed = target$$
$$\quad \lor\ t > 1 \land$$
$$\quad\quad \exists\, p : point \bullet$$
$$\quad\quad\quad ProjectMove(seed, currents, t - 1) = p \land$$
$$\quad\quad\quad ProjectMove(p, currents, 1) = target$$
$$\quad \lor\ t = 1 \land$$
$$\quad\quad \exists\, F, orientation : Vector \bullet$$
$$\quad\quad\quad target - seed = ApplyForce(F) \land$$
$$\quad\quad\quad F = sum(currents, force(seed, orientation)) \land$$
$$\quad\quad\quad orientation = normalize($$
$$\quad\quad\quad\quad sum(currents, field(seed)))$$

ApplyForce

$$ApplyForce : Vector \rightarrow Vector$$

$$ApplyForce(F) = move \Leftrightarrow$$
$$\quad insert\ physics\ here$$

move

$$move : image \times image \times Point \rightarrow coils \times interval$$

$$\begin{aligned}
&move(left, right, target) = (currents, duration) \Leftrightarrow \\
&\quad \exists\, seed : Point;\ skull : \mathrm{seq}[Points];\ v : Vector; \\
&\qquad trans : OrthogonalTransformation \bullet \\
&\qquad AnalyzeImage(left, right) = (seed, skull) \\
&\qquad \land\ apply(MarkersRelative, rel2abs)\ PermEq\ skull \\
&\qquad \land\ v = ComputeMove(rel2abs^{-1}(seed), target, rel2abs) \\
&\qquad \land\ CoilCurrents(seed, v) = (currents, duration)
\end{aligned}$$

AnalyzeImage

$$AnalyzeImage : image \times image \rightarrow Point \times \mathrm{seq}[Point]$$

$$\begin{aligned}
&AnalyzeImage(left, right) = (SeedAbs, SkullAbs) \Leftrightarrow \\
&\quad \exists\, SeedImage : Point;\ trans : OrthogonalTransformation; \\
&\qquad SkullImage, HelmetImage : \mathrm{seq}[Point] \bullet \\
&\qquad FindObjects(left, right) = \\
&\qquad\qquad (SkullImage, HelmetImage, SeedImage) \\
&\qquad \land\ trans(HelmetImage)\ PermEq\ HelmetFixed \\
&\qquad \land\ apply(SkullImage, trans) = SkullAbs \\
&\qquad \land\ trans(SeedImage) = SeedAbs
\end{aligned}$$

$[T]$

$$PermEq : \mathrm{seq}[T] \leftrightarrow \mathrm{seq}[T]$$

$$\begin{aligned}
&\forall\, x, y : \mathrm{seq}[T] \bullet \\
&\quad x\ PermEq\ y \Leftrightarrow \\
&\qquad \#x = 0 \land \#y = 0 \lor \\
&\qquad \exists\, i : \mathbb{N} \mid i \in 1 \,..\, \#y \bullet \\
&\qquad\quad x[1] = y[i] \land \\
&\qquad\quad tail(x)\ PermEq\ squash(y \lhd \{i\})
\end{aligned}$$

$$HelmetAbsolute : \mathrm{seq}[Point]$$
$$MarkersRelative : \mathrm{seq}[Point]$$

$$\#MarkersRelative = 3$$
$$\#HelmetAbsolute = 4$$

VectorArith

$$
\begin{array}{l}
\_ - \_ : Point \times Point \rightarrow Vector \\
\_ + \_ : Point \times Vector \rightarrow Point \\
\_ + \_ : Vector \times Point \rightarrow Point \\
\_ * \_ : Vector \times \mathbb{R} \rightarrow Vector \\
\hline
\forall p, q : Point, v, w : Vector, r : \mathbb{R} \bullet \\
\quad p + v = q \Leftrightarrow \\
\qquad p.x + v.x = q.x \land p.y + v.y = q.y \land p.z + v.z = q.z \\
\quad \land\ p - q = v \Leftrightarrow p + v = q \\
\quad \land\ v + p = q \Leftrightarrow p + v = q \\
\quad \land\ v * r = w \Leftrightarrow \\
\qquad v.x * r = w.x \land v.y * r = w.y \land v.z * r = w.z
\end{array}
$$

$[S, T]$

$$
\begin{array}{l}
apply : \mathrm{seq}[S] \times (S \rightarrow T) \rightarrow \mathrm{seq}[T] \\
\hline
apply(x, f) = y \Leftrightarrow \\
\quad \#x = \#y \land \\
\quad \forall i : \mathbb{N} \mid i \in 1 \,..\, \#x \bullet y[i] = f(x[i])
\end{array}
$$

FindObjects

$$
\begin{array}{l}
FindObjects : image \times image \rightarrow \\
\qquad\qquad \mathrm{seq}[Point] \times \mathrm{seq}[Point] \times Point \\
\hline
FindObjects(left, right) = (skull, helmet, seed) \Leftrightarrow \\
\quad \exists\, objects : \mathrm{seq}[Point] \bullet \\
\qquad apply(objects, ProjectLeft)\ PermEq \\
\qquad\qquad BlobFind(left) \land \\
\qquad apply(objects, ProjectRight)\ PermEq \\
\qquad\qquad BlobFind(right) \land \\
\qquad ReasonableSeed(seed) \land \\
\qquad ReasonableHelm(helmet) \land \\
\qquad ReasonableSkull(skull) \land \\
\qquad objects\ PermEq\ skull \frown helmet \frown \langle seed \rangle
\end{array}
$$

$[S, T]$

$$sum : \text{seq}[S] \times (S \to T) \to T$$

$$sum(s, f) = x \Leftrightarrow$$
$$s = \langle\rangle \land x = 0 \lor$$
$$s \neq \langle\rangle \land x = f(head(s)) + sum(tail(s), f)$$

NormalizedVector

$$v : Vector$$

$$square(v.x) + square(v.y) + square(v.z) = 1$$

normalize

$$normalize : Vector \to NormalizedVector$$

$$normalize(v) = n \Leftrightarrow$$
$$\exists r : \mathbb{R} \bullet$$
$$square(r) = square(v.x) + square(v.y) + square(v.z)$$
$$\land\ n.x * r = v.x \land n.y * r = v.y \land n.z * r = v.z$$

approx

$$\_approx\_ : Point \leftrightarrow Point$$

$$p\ approx\ q \Leftrightarrow$$
$$square(p.x - q.x) + square(p.y - q.y) +$$
$$square(p.z - q.z) \leq square(MaxApprox)$$

Blob

$$position : Point$$
$$radius : \mathbb{R}$$

$$radius > 0$$

BlobFind

$$
\begin{array}{l}
BlobFind : Image \leftrightarrow \mathrm{seq}[Blob] \\
IsDark : Image \times Blob \rightarrow \mathbb{N} \\
Tally : Image \rightarrow (Point \rightarrow \mathbb{N}) \\
\_IntDiv\_ : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \\
\_Encompasses\_ : Blob \leftrightarrow Point \\
\hline
BlobFind(screen) = blobs \Leftrightarrow \\
\quad \forall b : blob \mid b \in \mathrm{ran}\, blobs \bullet \\
\qquad IsDark(screen, b) \geq 75 \land \\
\quad \forall a : blob \mid a.radius \geq 2 \bullet \\
\qquad IsDark(screen, a) \geq 75 \Rightarrow \\
\qquad\quad \exists b : blob \mid b \in \mathrm{ran}\, blobs \bullet \\
\qquad\qquad b\ encompasses\ a.position \\
IsDark(screen, b) = p \Leftrightarrow \\
\quad \exists s : \mathrm{seq}[Point] \bullet \\
\qquad \forall pt : Point \bullet b\ encompasses\ (pt) \Leftrightarrow pt \in s \land \\
\qquad sum(s, tally(screen)) * 100\ IntDiv\ \#s = p \\
a\ IntDiv\ b = c \Leftrightarrow c * b \geq a \land c * (b - 1) < a \\
tally(screen)(pt) = b \Leftrightarrow \\
\quad screen[pt.x, pt.y] = on \land b = 1 \lor \\
\quad screen[pt.x, pt.y] = off \land b = 0 \\
b\ encompasses\ pt \Leftrightarrow distance(pt, b.position) \leq b.radius
\end{array}
$$

$$
CommandType \triangleq \{StartHeat, StopHeat, TimedHeat, StartMove, StopMove, Shutdown, Reset\}
$$

Command

$type : CommandType$
$stamp : time$
$HeatIntensity : \mathbb{N}$
$HeatDuration : interval$
$MoveTarget : point$

$stamp > 0$
$type = TimedHeat \Rightarrow$
$\quad HeatDuration > 0 \land$
$\quad HeatIntensity = FixedHeatingIntensity$
$type = StartHeat \Rightarrow$
$\quad HeatIntensity = FixedHeatingFrequency$
$type = StartMove \Rightarrow$
$\quad MoveTarget \neq origin$

CommandQueue

$q : \text{seq}[Command]$

$\forall i, j \in \text{dom}\, q \bullet$
$\quad q[i].stamp = q[j].stamp \Rightarrow i = j$

CommandQueues

$UserCommands : CommandQueue$
$HeatCommands : CommandQueue$
$MoveCommands : CommandQueue$

$UserCommands[1].type = reset$
$\forall u : Command \mid u \in \text{ran}\, UserCommands \bullet$
$\quad u.type \in \{StartHeat, StopHeat, TimedHeat, Reset\} \Rightarrow$
$\quad\quad \exists h : Command \mid h \in \text{ran}\, HeatCommands \bullet h = u \land$
$\quad u.type \in \{StartMove, StopMove, Reset\} \Rightarrow$
$\quad\quad \exists m : Command \mid m \in \text{ran}\, MoveCommands \bullet m = u$
$\forall h : Command \mid h \in \text{ran}\, HeatCommands \bullet$
$\quad \exists u : Command \mid u \in \text{ran}\, UserCommands \bullet h = u$
$\forall m : Command \mid m \in \text{ran}\, MoveCommands \bullet$
$\quad \exists u : Command \mid u \in \text{ran}\, UserCommands \bullet m = u$

FluoroImagePair

$$
\begin{array}{l}
stamp : time \\
left : FluoroImage \\
right : FluoroImage
\end{array}
$$

FlouroImageQueue

$$
\begin{array}{l}
fluoro : \mathrm{seq}[FluoroImagePair] \\
\hline
fluoro[1].stamp = UserCommands[1].stamp + InitialImage \\
\forall i : \mathbb{N} \mid i \in \mathrm{dom}\, MoveCommands \\
\qquad \wedge\ MoveCommands[i].type = StartMove \bullet \\
\quad start = MoveCommands[i].stamp \wedge \\
\quad target = MoveCommands[i].target \wedge \\
\quad insert(fluoro, start) \wedge \\
\quad move(fluoro[start].left, flouro[start].right, target) = \\
\qquad\qquad (currents, duration) \wedge \\
\quad stop = min(MoveCommands[i+1].time, start + duration) \wedge \\
\quad images = truncate((stop - start) * ImagesPerSec) \wedge \\
\quad \forall j : \mathbb{N} \mid j \in 1 \,..\, images \bullet \\
\qquad insert(fluoro, start + j/ImagesPerSec) \\
\quad \wedge\ insert(fluoro, stop)
\end{array}
$$

$[T]$

$$
\begin{array}{l}
insert : \mathrm{seq}[T] \rightarrow time \\
\hline
insert(s, t) \Leftrightarrow \\
\quad \exists x : T \mid x \in \mathrm{ran}\, s \bullet x.time = t
\end{array}
$$

SeedPosition

$$
\begin{array}{l}
SeedPosition : time \rightarrow point \\
\hline
SeedPosition(t) = pt \Leftrightarrow \\
\quad \exists_1 i : \mathbb{N} \mid i \in \mathrm{dom}\, images \bullet \\
\qquad images[i].stamp \leq t \wedge \\
\qquad images[i+1].stamp > t \wedge \\
\qquad pt = LocateSeed(images[i].left, images[i].right)
\end{array}
$$

currents

$current : \mathbb{N}$

$\_approx\_ : current \leftrightarrow current$

$current \leq UpperCurrentLimit$

$x \ approx \ y \Leftrightarrow$
$\quad max((x - y), (y - x)) \leq CurrentDelta$

CoilCurrentValues

$CoilCurrent : \text{seq} : [current]$

$\_approx\_ : CoilCurrent \leftrightarrow CoilCurrent$

$\#CoilCurrent = 6$

$x \ approx \ y \Leftrightarrow$
$\quad \forall i : \mathbb{N} \mid i \in 1 \,..\, 6 \bullet$
$\quad\quad x[i] \ approx \ y[i]$

CoilCurrentValues

$idle : CoilCurrent$

$IdleCurrent : current$

$\forall c : current \mid c \in \text{ran} \ idle \bullet$
$\quad c = IdleCurrent$

CoilCurrents

$$
\begin{array}{l}
CoilCurrentValues \\
CoilCurrents : time \rightarrow CoilCurrent \\
\hline
CoilCurrents(t) = c \Leftrightarrow \\
\quad \exists_1 \, i : \mathbb{N} \mid MoveCommand[i].time \leq t \wedge \\
\qquad\qquad MoveCommand[i+1].time > t \bullet \\
\qquad MoveCommand[i] \neq StartMove \wedge c = idle \\
\qquad \vee \; MoveCommand[i] = StartMove \wedge \\
\qquad\quad start = MoveCommand[i].stamp \wedge \\
\qquad\quad im = fluoro[start] \wedge \\
\qquad\quad move(im.left, im.right, MoveCommand[i].target) = \\
\qquad\qquad\qquad (currents, duration) \wedge \\
\qquad\quad stop = min(MoveCommand[i+1].stamp, \\
\qquad\qquad\qquad start + duration) \wedge \\
\qquad\quad (t \geq stop \wedge c = idle \vee \\
\qquad\quad t < stop \wedge c = currents)
\end{array}
$$

ShutdownQueue

$$
\begin{array}{l}
shutdown : \mathrm{seq}[time] \\
CoilMonitor : time \rightarrow CoilCurrent \\
XrayMonitor : time \rightarrow XrayValues \\
HeaterMonitor : time \rightarrow HeatValues \\
\hline
\forall \, t : time \bullet \\
\quad \neg \, (CoilCurrents(t) \; approx \; CoilMonitor(t)) \Rightarrow \\
\qquad insert(shutdown, t) \\
\forall \, t : time \bullet \\
\quad \neg \, (XrayIntensity(t) \; approx \; XrayMonitor(t)) \Rightarrow \\
\qquad insert(shutdown, t) \\
\forall \, t : time \bullet \\
\quad \neg \, (HeaterIntensity(t) \; approx \; HeaterMonitor(t)) \Rightarrow \\
\qquad insert(shutdown, t) \\
\forall \, t : time \mid t \in \mathrm{ran} \, shutdown \bullet \\
\quad insert(MoveCommands, t, shutdown) \wedge \\
\quad insert(HeatCommands, t, shutdown) \wedge \\
\quad insert(XrayCommands, t, shutdown)
\end{array}
$$

CoilCommand

$stamp : time$
$value : CoilCurrent$

CoilCommands

$CoilCommands : \text{seq}[CoilCommand]$

$\#CoilCommands = \#MoveCommands$
$\forall i : \mathbb{N} \mid i \in 1 \,..\, \#CoilCommands \bullet$
$\quad MoveCommands[i].stamp = CoilCommands[i].stamp \land$
$\quad MoveCommands[i].type = StopMove \Rightarrow$
$\quad\quad CoilCommands[i].value = idle \land$
$\quad MoveCommands[i].type \in \{Reset, Shutdown\} \Rightarrow$
$\quad\quad CoilCommands[i].value = CoilsOff \land$
$\quad MoveCommands[i].type = StartMove \Rightarrow$
$\quad\quad CoilCommands[i].value =$
$\quad\quad\quad CoilCurrents(MoveCommands[i].stamp)$

XrayCommand

$type : XrayCommandTypes$
$stamp : time$
$duration : interval$
$intensity : XrayValues$

XrayCommands

$XrayCommands : \text{seq}[XrayCommand]$

$\forall c : XrayCommand \mid c \in \text{ran}\, XrayCommands \bullet$
$\quad c.type = Shutdown \Rightarrow$
$\quad\quad c.stamp \in \text{ran}\, shutdown \land$
$\quad c.type = TimedXray \Leftrightarrow$
$\quad\quad c.stamp \in \text{ran}\, images \land$
$\quad\quad c.duration = FixedXrayDuration \land$
$\quad\quad c.intensity = FixedXrayIntensity$

window

$origin : point$
$xsize : \mathbb{N}$
$ysize : \mathbb{N}$
$status : \{show, hide\}$
$image : table[color]$

$\text{rows } image = ysize$
$\text{cols } image = xsize$

ScreenPoint

$ScreenPoint : point$

$ScreenPoint.x \leq ScreenWidth \land$
$ScreenPoint.y \leq ScreenHeight$

Screen

$ScreenWidth : \mathbb{N}$
$ScreenHeight : \mathbb{N}$
$default : color$
$windows : \text{seq}[window]$
$image : table[color]$

$\text{rows } image = ScreenHeight$
$\text{cols } image = ScreenWidth$
$\exists\, background : window \bullet$
$\quad windows[\#windows] = background \land$
$\quad \text{rows } background.image = ScreenHeight \land$
$\quad \text{cols } background.image = ScreenWidth \land$
$\quad \forall\, pt : ScreenPoint \bullet background.image[pt] = default$
$\forall\, pt : ScreenPoint \bullet$
$\quad image[pt] = w.image[pt - w.origin] \land$
$\quad w = highest(windows, pt)$

highest

$$
\begin{array}{l}
highest : ScreenPoint \times \mathrm{seq}[window] \rightarrow window \\
\_contains\_ : window \leftrightarrow ScreenPoint \\
\hline
\forall\, stack : \mathrm{seq}[window];\ pt : point;\ w : window \bullet \\
highest(stack, pt) = w \Leftrightarrow \\
\quad \exists\, z : \mathbb{N} \mid z \in \mathrm{dom}\, stack \bullet \\
\qquad stack[z] = x \wedge \\
\qquad stack[z].status = show \wedge \\
\qquad stack[z] containspt \wedge \\
\qquad \forall\, y : \mathbb{N} \mid y \in \mathrm{dom}\, stack \bullet \\
\qquad\quad (stack[y].status = show \wedge \\
\qquad\qquad stack[y]\ contains\ pt) \Rightarrow y \leq z \\
\forall\, w : window;\ pt : point \bullet w\ contains\ pt \Leftrightarrow \\
\quad \exists\, r : point \mid r = pt - w.origin \bullet \\
\qquad r.x > 0 \wedge r.x \leq w.xsize \\
\qquad \wedge\ r.y > 0 \wedge r.y \leq w.ysize
\end{array}
$$

UCS

$$
\begin{array}{l}
GetSeedUCSPosition : time \rightarrow point \\
\hline
GetSeedUCSPosition(t) = seed \Leftrightarrow \\
\quad \exists\, T : OrthogonalTransformation \bullet \\
\qquad apply(T, RelativeMarkers)\ PermEq \\
\qquad\qquad IdentifiedMarkers \\
\qquad \wedge\ T(SeedPosition(t)) = seed
\end{array}
$$

feedback

$$
\begin{array}{l}
HeatActual : \mathbb{R} \\
HeatProject : \mathbb{R} \\
Target : vector
\end{array}
$$

GetFeedback

$$
\begin{array}{l}
HeatFeedback \\
MoveFeedback \\
GetFeedback : time \rightarrow feedback \\
\hline
\forall\, t : time;\ fb : feedback \bullet \\
GetFeedback(t) = fb \Leftrightarrow \\
\quad fb.HeatActual = HeatFeedbackActual(t) \land \\
\quad fb.HeatProject = HeatFeedbackProject(t) \land \\
\quad fb.Target = TargetFeedback(t)
\end{array}
$$

flatten

$$
\begin{array}{l}
FlattenFunction : ImageWindowView \rightarrow (point \rightarrow point) \\
\hline
FlattenFunction(view)(p) = q \Leftrightarrow \\
\quad q.z = 0 \land \quad ((view = Lateral \land \\
\qquad q.x = -p.y \land q.y = p.z) \\
\quad \lor (view = Frontal \land \\
\qquad q.x = p.x \land q.y = p.z) \\
\quad \lor (view = Coronal \land \\
\qquad q.x = p.x \land q.y = p.y))
\end{array}
$$

ImageWindow

$$ImageWindow : ImageWindowView \times time \rightarrow Window$$

$$\begin{aligned}
&ImageWindow(view, t) = w \Leftrightarrow \wedge \\
&\quad \exists\, flatten : point \rightarrow point \mid \\
&\qquad flatten = FlattenFunction(view) \bullet \\
&\quad \exists\, scale : \mathbb{R};\ fb : Feedback;\ l1, l2, l3, l4 : image \bullet \\
&\quad \exists\, seed3D : point \bullet \\
&\qquad scale = w.xsize / ImageWidth \wedge \\
&\qquad scale = w.ysize / ImageHeight \wedge \\
&\qquad seed3D = GetSeedUCSPosition(time) \wedge \\
&\qquad seed = flatten(seed3D) \wedge \\
&\qquad l1 = GetImage(w.view, seed3D) \wedge \\
&\qquad l2 = SeedOverlay(seed, scale, l1) \wedge \\
&\qquad fb = GetFeedback(t) \wedge \\
&\qquad ((fb.HeatProject = 0 \wedge l3 = l2) \vee \\
&\qquad (fb.HeatProject > 0 \wedge \\
&\qquad\quad l3 = ProjectHeatOverlay(seed, \\
&\qquad\qquad fb.HeatProject, scale, l2)) \\
&\qquad \wedge ((fb.HeatActual = 0 \wedge l4 = l3) \vee \\
&\qquad (fb.HeatAtual > 0 \wedge \\
&\qquad\quad w = ActualHeatOverlay(seed, fb.HeatActual, \\
&\qquad\qquad scale, layer3)) \\
&\qquad \wedge ((fb.Target = seed \wedge w.image = l4) \vee \\
&\qquad (fb.Target \neq seed \wedge \\
&\qquad\quad w = MovementOverlay(seed, \\
&\qquad\qquad flatten(fb.target), scale, layer4))
\end{aligned}$$

SeedOverlay

$$SeedOverlay : point \times \mathbb{R} \times Window \rightarrow Window$$

$$\begin{aligned}
&SeedOverlay(seed, scale, before) = after \Leftrightarrow \\
&\quad after.xsize = before.xsize \land \\
&\quad after.ysize = before.ysize \land \\
&\quad after.view = before.view \land \\
&\quad \forall x, y : \mathbb{N}_1 \mid x \leq before.xsize \land y \leq before.ysize \bullet \\
&\qquad (distance(seed * scale, pt) \leq SeedRadius * scale \\
&\qquad\quad \land after.image[x, y] = SeedColor) \\
&\qquad \lor (distance(seed * scale, pt) > SeedRadius * scale \\
&\qquad\quad \land after.image[x, y] = before.image[x, y]
\end{aligned}$$

ProjectHeatOverlay

$$ProjectHeatOverlay : point \times \mathbb{R} \times \mathbb{R} \times Window \rightarrow Window$$

$$\begin{aligned}
&ProjectHeatOverlay(seed, radius, scale, before) = after \Leftrightarrow \\
&\quad after.xsize = before.xsize \land \\
&\quad after.ysize = before.ysize \land \\
&\quad after.view = before.view \land \\
&\quad \forall x, y : \mathbb{N}_1 \mid x \leq before.xsize \land y \leq before.ysize \bullet \\
&\qquad (distance(seed * scale, pt)\ approx\ radius * scale \\
&\qquad\quad \land after.image[x, y] = HeatProjectColor) \\
&\qquad \lor \neg ((distance(seed * scale, pt)\ approx\ radius * scale) \\
&\qquad\quad \land after.image[x, y] = before.image[x, y]
\end{aligned}$$

ActualHeatOverlay

$$ActualHeatOverlay : point \times \mathbb{R} \times \mathbb{R} \times Window \rightarrow Window$$

$$\begin{aligned}
&ActualHeatOverlay(seed, radius, scale, before) = after \Leftrightarrow \\
&\quad after.xsize = before.xsize \wedge \\
&\quad after.ysize = before.ysize \wedge \\
&\quad after.view = before.view \wedge \\
&\quad \forall x, y : \mathbb{N}_1 \mid x \leq before.xsize \wedge y \leq before.ysize \bullet \\
&\qquad (distance(seed * scale, pt) \leq radius * scale \\
&\qquad\quad \wedge after.image[x, y] = HeatActualColor) \\
&\qquad \vee (distance(seed * scale, pt) > radius * scale \\
&\qquad\quad \wedge after.image[x, y] = before.image[x, y]
\end{aligned}$$

ProjectMoveOverlay

$$ProjectMoveOverlay : point \times point \times \mathbb{R} \times Window \rightarrow Window$$

$$\begin{aligned}
&ProjectMoveOverlay(seed, target, scale, before) = after \Leftrightarrow \\
&\quad after.xsize = before.xsize \wedge \\
&\quad after.ysize = before.ysize \wedge \\
&\quad after.view = before.view \wedge \\
&\quad \forall x, y : \mathbb{N}_1 \mid x \leq before.xsize \wedge y \leq before.ysize \bullet \\
&\qquad (pt\ between(seed * scale, target * scale) \\
&\qquad\quad \wedge after.image[x, y] = ProjectMovementColor) \\
&\qquad \vee \neg ((pt\ between\ (seed * scale, target * scale) \\
&\qquad\quad \wedge after.image[x, y] = before.image[x, y]
\end{aligned}$$

between

$$\_between\_ : point \leftrightarrow (point \times point)$$

$$\begin{aligned}
&a\ between\ (b, c) \Leftrightarrow \\
&\quad ((a.x \leq b.x \land a.x \geq c.x \land \\
&\qquad a.y \leq b.y \land a.y \geq c.y \land \\
&\qquad a.z \leq b.z \land a.z \geq c.z) \\
&\quad \lor (a.x \geq b.x \land a.x \leq c.x \land \\
&\qquad a.y \geq b.y \land a.y \leq c.y \land \\
&\qquad a.z \geq b.z \land a.z \leq c.z)) \\
&\quad \land (a.x - b.x)/(a.y - b.y) = (a.x - c.x)/(a.y - c.y) \\
&\quad \land (a.x - b.x)/(a.z - b.z) = (a.x - c.x)/(a.z - c.z)
\end{aligned}$$

$ImageWindowView \triangleq \{Coronal, Lateral, Frontal\}$

ImageFile

$$\begin{aligned}
&xsize : \mathbb{N} \\
&ysize : \mathbb{N} \\
&view : ImageWindowView \\
&origin : point \\
&patient : name \\
&checksum : \mathbb{N} \\
&image : table[color]
\end{aligned}$$

$$\begin{aligned}
&\text{rows } image = ysize \\
&\text{cols } image = xsize
\end{aligned}$$

MRIStack

$$\begin{aligned}
&MRIStack : \text{seq } ImageFile \\
&\#MRIStack = slices \\
&\forall i, j : \mathbb{N} \mid i, j \in \text{dom } MRIStack \bullet \\
&\quad MRIStack[i].origin > MRIStack[j].origin \Leftrightarrow i > j \\
&\exists v : plane \bullet \\
&\quad \forall i : ImageFile \mid i \in MRIStack \bullet i.view = v \\
&\forall i : ImageFile \mid i \in \text{ran } MRIStack \bullet \\
&\quad i.xsize = ImageWidth \land i.ysize = ImageHeight
\end{aligned}$$

GetImage

$$GetImage : ImageWindowView \times point \rightarrow Window$$

$$
\begin{array}{l}
\forall\, view : ImageWindowView;\ seed : point;\ w : Window \bullet \\
GetImages(view, seed) = w \Leftrightarrow \\
view = Lateral \land \\
\quad \exists\, i : \mathbb{N};\ I : ImageFile \mid (i \mapsto I) \in LateralStack \bullet \\
\quad w.image = I.image \\
\qquad seed \leq I.origin \land i = 1 \lor \\
\qquad seed \geq I.origin \land i = slices \lor \\
\qquad (b = Distance(seed, I.origin) \land \\
\qquad\quad Distance(seed, LateralStack[i-1]) \geq b \land \\
\qquad\quad Distance(seed, LateralStack[i+1]) \geq b) \\
\lor\ view = Coronal \land \\
\quad \exists\, i : \mathbb{N};\ I : ImageFile \mid (i \mapsto I) \in CoronalStack \bullet \\
\quad w.image = I.image \\
\qquad seed \leq I.origin \land i = 1 \lor \\
\qquad seed \geq I.origin \land i = slices \lor \\
\qquad (b = Distance(seed, I.origin) \land \\
\qquad\quad Distance(seed, CoronalStack[i-1]) \geq b \land \\
\qquad\quad Distance(seed, CoronalStack[i+1]) \geq b) \\
\lor\ view = Frontal \land \\
\quad \exists\, i : \mathbb{N};\ I : ImageFile \mid (i \mapsto I) \in FrontalStack \bullet \\
\quad w.image = I.image \\
\qquad seed \leq I.origin \land i = 1 \lor \\
\qquad seed \geq I.origin \land i = slices \lor \\
\qquad (b = Distance(seed, I.origin) \land \\
\qquad\quad Distance(seed, FrontalStack[i-1]) \geq b \land \\
\qquad\quad Distance(seed, FrontalStack[i+1]) \geq b)
\end{array}
$$

$[T]$

$$\_ \oplus \_ : table[T] \times table[T] \rightarrow table[T]$$

$$\begin{aligned}
&\forall a, b, c : table[T] \bullet \\
&\quad a \oplus b = c \Leftrightarrow \\
&\qquad a.sizex = b.sizex \land b.sizex = c.sizex \land \\
&\qquad a.sizey = b.sizey \land b.sizey = c.sizey \land \\
&\qquad \forall x, y : \mathbb{N}_1 \mid x \leq a.sizex \land y \leq a.sizey \bullet \\
&\qquad\quad b[x, y] \neq default \land z[x, y] = b[x, y] \land \\
&\qquad\quad b[x, y] = default \land z[x, y] = a[x, y]
\end{aligned}$$

HeatIntensity

$$HeaterIntensity : time \rightarrow HeatValues$$

$$\begin{aligned}
&HeaterIntensity(t) = h \Leftrightarrow \\
&\quad \exists_1 : i : \mathbb{N} \mid HeatCommand[i].stamp \leq t \\
&\qquad \land HeatCommand[i+1].stamp > t \bullet \\
&\quad HeatCommand[i] \in \{Reset, StopHeat\} \land h = 0 \\
&\quad \lor HeatCommand[i] = StartHeat \\
&\qquad \land h = HeatCommand[i].intensity \\
&\quad \lor HeatCommand[i] = TimedHeat \\
&\qquad \land HeatCommand[i].stamp + \\
&\qquad\qquad HeatCommand[i].duration \geq t \\
&\qquad \land h = HeatCommand[i].intensity \\
&\quad \lor HeatCommand[i] = TimedHeat \\
&\qquad \land HeatCommand[i].stamp + \\
&\qquad\qquad HeatCommand[i].duration < t \\
&\qquad \land h = 0
\end{aligned}$$

HeatFeedback

$$
\begin{array}{l}
HeatFeedbackActual : time \rightarrow \mathbb{R} \\
HeatFeedbackProject : time \rightarrow \mathbb{R}
\end{array}
$$

$$
\begin{array}{l}
HeatFeedbackActual(t) = r \Leftrightarrow \\
\quad \exists_1 : i : \mathbb{N} \mid HeatCommand[i].stamp \leq t \wedge \\
\qquad\qquad HeatCommand[i+1].stamp > t \bullet \\
\qquad HeatCommand[i] \in \{Reset, StopHeat\} \wedge h = 0 \\
\qquad \vee (HeatCommand[i] = StartHeat \vee \\
\qquad\quad (HeatCommand[i] = TimedHeat \wedge \\
\qquad\quad HeatCommand[i].stamp+ \\
\qquad\qquad\qquad HeatCommand[i].duration \geq t)) \wedge \\
\qquad\quad h = heating(HeatCommand[i].intensity, \\
\qquad\qquad\qquad HeatCommand[i].stamp - t) \\
\qquad \vee HeatCommand[i] = TimedHeat \wedge \\
\qquad\quad HeatCommand[i].stamp+ \\
\qquad\qquad\qquad HeatCommand[i].duration < t \wedge \\
\qquad\quad h = 0 \\
HeatFeedbackProject(t) = r \Leftrightarrow \\
\quad \exists_1 : i : \mathbb{N} \mid HeatCommand[i].stamp \leq t \wedge \\
\qquad\qquad HeatCommand[i+1].stamp > t \bullet \\
\qquad HeatCommand[i] \neq TimedHeat \wedge h = 0 \\
\qquad \vee (HeatCommand[i] = TimedHeat \wedge \\
\qquad\quad HeatCommand[i].stamp+ \\
\qquad\qquad\qquad HeatCommand[i].duration \geq t)) \wedge \\
\qquad\quad h = heating(HeatCommand[i].intensity, \\
\qquad\qquad\qquad HeatCommand[i].duration) \\
\qquad \vee HeatCommand[i] = TimedHeat \wedge \\
\qquad\quad HeatCommand[i].stamp+ \\
\qquad\qquad\qquad HeatCommand[i].duration < t \wedge \\
\qquad\quad h = 0
\end{array}
$$

MoveFeedback

$$TargetFeedback : time \rightarrow point$$

$$
\begin{aligned}
&target(t) = p \Leftrightarrow \\
&\quad \exists_1 i : \mathbb{N} \mid MoveCommand[i].stamp \leq t \wedge \\
&\qquad\qquad MoveCommand[i+1].stamp > t \bullet \\
&\qquad MoveCommand[i] \neq StartMove \wedge \\
&\qquad\qquad c = SeedPosition(t) \\
&\qquad \vee\ MoveCommand[i] = StartMove \wedge \\
&\qquad\qquad start = MoveCommand[i].stamp \wedge \\
&\qquad\qquad im = fluoro[start] \wedge \\
&\qquad\qquad move(im.left, im.right) = (currents, duration) \wedge \\
&\qquad\qquad stop = min(MoveCommand[i+1].stamp, \\
&\qquad\qquad\qquad\qquad start + duration) \wedge \\
&\qquad\qquad (t \geq stop \wedge c = SeedPosition(t) \vee \\
&\qquad\qquad t < stop \wedge c = MoveCommand[i].target)
\end{aligned}
$$