12-6-95

SANDIA REPORT

SAND95–2593 • UC–905
Unlimited Release
Printed November 1995

A Software Surety Analysis Process

Sharon Trauth, Pat Tempel

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
for the United States Department of Energy
under Contract DE-AC04-94AL85000

SF2900Q(8-81)

35

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
PO Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
US Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

Distribution Category
UC-905

SAND 95-2593
Unlimited Release
Printed November 1995

A Software Surety Analysis Process

Sharon Trauth
Pat Tempel

Prepared as part of the
High Consequence System Surety
Process Development Project
Sandia National Laboratories
Albuquerque, NM 87185

Abstract

As part of the High Consequence System Surety project, this work was undertaken to explore one approach to conducting a surety theme analysis for a software-driven system. Originally, plans were to develop a theoretical approach to the analysis, and then to validate and refine this process by applying it to the software being developed for the Weight and Leak Check System (WALS), an automated nuclear weapon component handling system. As with the development of the higher level High Consequence System Surety Process, this work was not completed due to changes in funding levels. This document describes the software analysis process, discusses its application in a software environment, and outlines next steps that could be taken to further develop and apply the approach to real projects.

A Software Surety Analysis Process

Contents

Definitions 3

Introduction 4

Capturing the Hardware Fault Tree Analysis Process 4

Software Fault Tree Analysis Process 8

Future Steps 24

Summary and Conclusions 24

Acknowledgments 26

Bibliography 26

Figures

Figure 1: Hardware Fault Tree Analysis Process 5

Figure 2: Software and Hardware Fault Tree Analysis Processes
Overlayed with a General Systems Design Approach 9

Figure 3: A Portion of a Hardware Fault Tree 15

Figure 4: Example System Structure 16

Figure 5: Example Software Documentation
for Design and Implementation 20

Figure 6: Example Software Documentation for Design Implementation 22

Definitions

High Consequence	Varies with the operation and customer, but is a consequence judged to be severe, for example resulting in significant loss of investment or loss of life.
Fault Tree	An analysis documented in a diagram which indicate paths through which the fault could occur. Where multiple paths exist, two options for failure are possible. One option is that the failure can be caused by any of the paths identified singularly; this situation is represented by an OR gate to connect the failure paths. The second option is that the failure paths must occur at the same time in order for the higher level failure to actually occur; this situation is represented by an AND gate connecting the failure paths.
Surety	As defined by the High Consequence System Surety Project Team, surety includes safety, security, control, reliability, and quality.
System	For the discussion in this document, system refers to the combined hardware-software end product. Although a systems approach must also include facilities and procedures, these are not explicitly covered in this discussion. However, to the extent possible, the reader may extend applicable principles into the facilities and procedures portions of a system.
Probabilistic Risk Assessment (PRA)	A process by which all the potential outcomes of a planned activity are identified, along with the probability of their occurrence and their associated consequence(s).

Introduction

Considerable debate exists today regarding whether software can be used in safety critical applications, such as in many weapon components. The prevailing assumption is that, since software is extraordinarily more complex than mechanical hardware, it cannot be analyzed sufficiently well to verify the absence of safety-critical faults. Consequently, the approach taken today is that software not be exclusively used in such applications, but rather coupled with analyzable, characterizable mechanical devices whose behavior can be well predicted under the environments of concern and whose presence will guarantee the device failure in a safe state under a given environment. In the development of such mechanical devices for weapon systems, *e. g.*, stronglink safing devices, a fairly well-understood, iterative, but minimally documented process is used to assure that component level design approaches are sound and justified and that components and piece parts will continue to meet established safety requirements during the production phase of the product life cycle. This component-level process integrates with a comprehensive approach applied at the system level to assure overall safety is achieved and maintained.

The work described in this paper was undertaken to explore the extension and applicability of the structured methodology used in mechanical safing devices to software. The approach considered here is not intended to imply that software *can* be used in safety critical applications, but rather that when the process is successfully applied the results may offer the design engineer a better set of information from which to make design decisions regarding whether additional safety features are really needed, and why. Further, the approach is intended to ultimately offer a comprehensive documentation scheme that will identify, for current and future responsible engineers, exactly which aspects of the software are critical, why they are critical, the testing done to verify the design approach implemented in the software, and the potential ramifications if changes were to be incorporated. Without such an approach, even though extensive work may have been done to assure the absence of critical defects, once a change is introduced, there may be no way of knowing what analyses or tests need to be repeated to ensure continued absence of critical defects.

The approach described was developed by first capturing and documenting the existing approach used in the mechanical design arena, followed by detailed exploration of its extension to software, with an emphasis on walking through the extension applied to specific software examples. The discussion presented in this paper flows from that exploratory work, so that the methodology is presented and the implications explored by walking through specific, simple software examples. Since the actual project under which this work began was redirected, completion of the documentation approach did not occur. The discussion in this paper therefore focuses on the work completed and provides a discussion of what next steps should occur to fully develop the methodology and demonstrate its applicability to a real software project.

Capturing the Hardware Fault Tree Analysis Process

The process presently used during the component level design of mechanical safing devices was reviewed extensively with design engineers, both at the component level itself and at the system level to assure adequate integration. The resulting process documentation is shown in Figure 1.

The process begins at the complete system level, in the conceptual phase of the project, wherein the critical safety requirements are explored, established, and extended to the next level component subsystems (such as the firing or aft subsystems). The preliminary

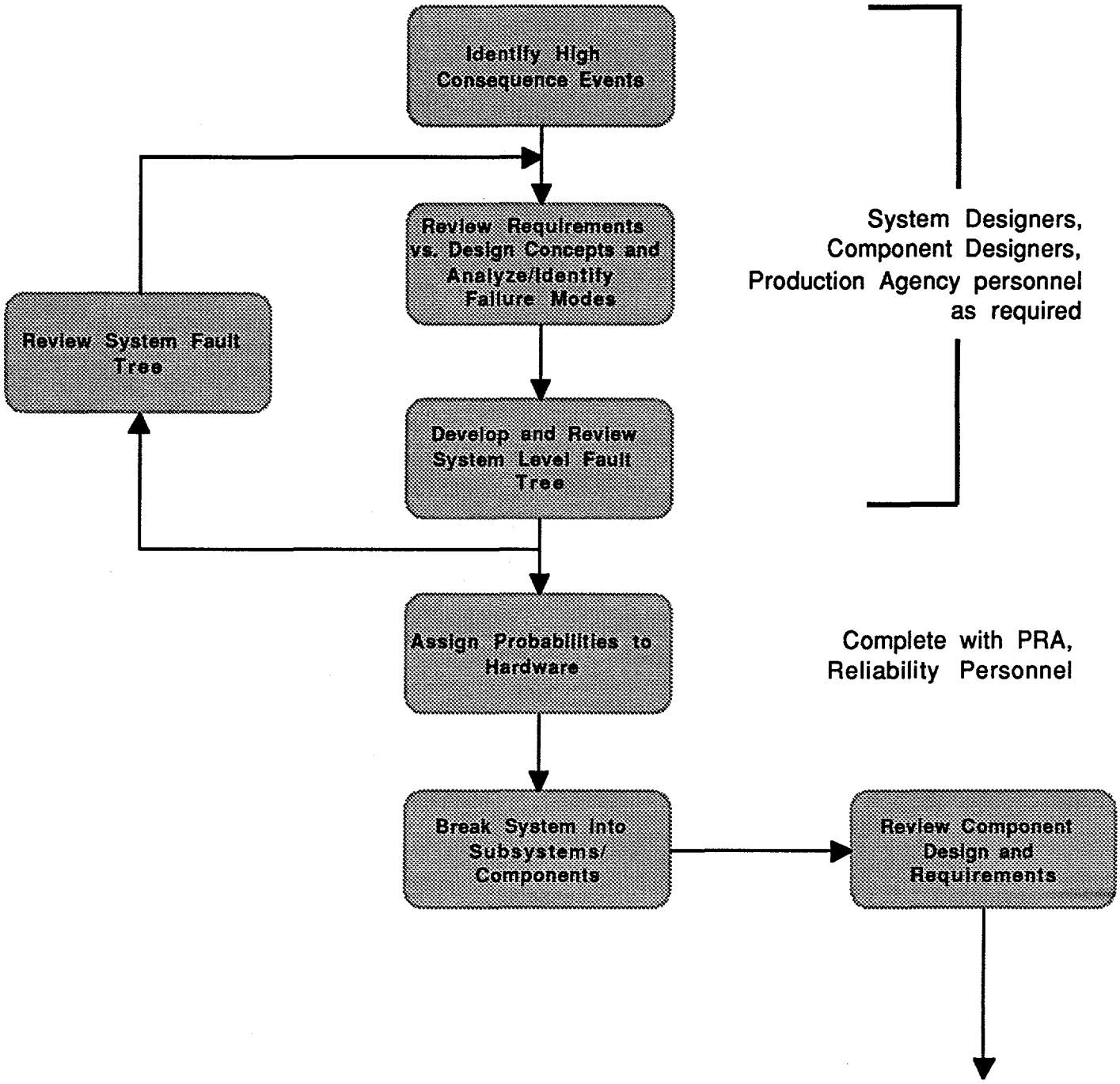

Figure 1: Hardware Fault Tree Analysis Process

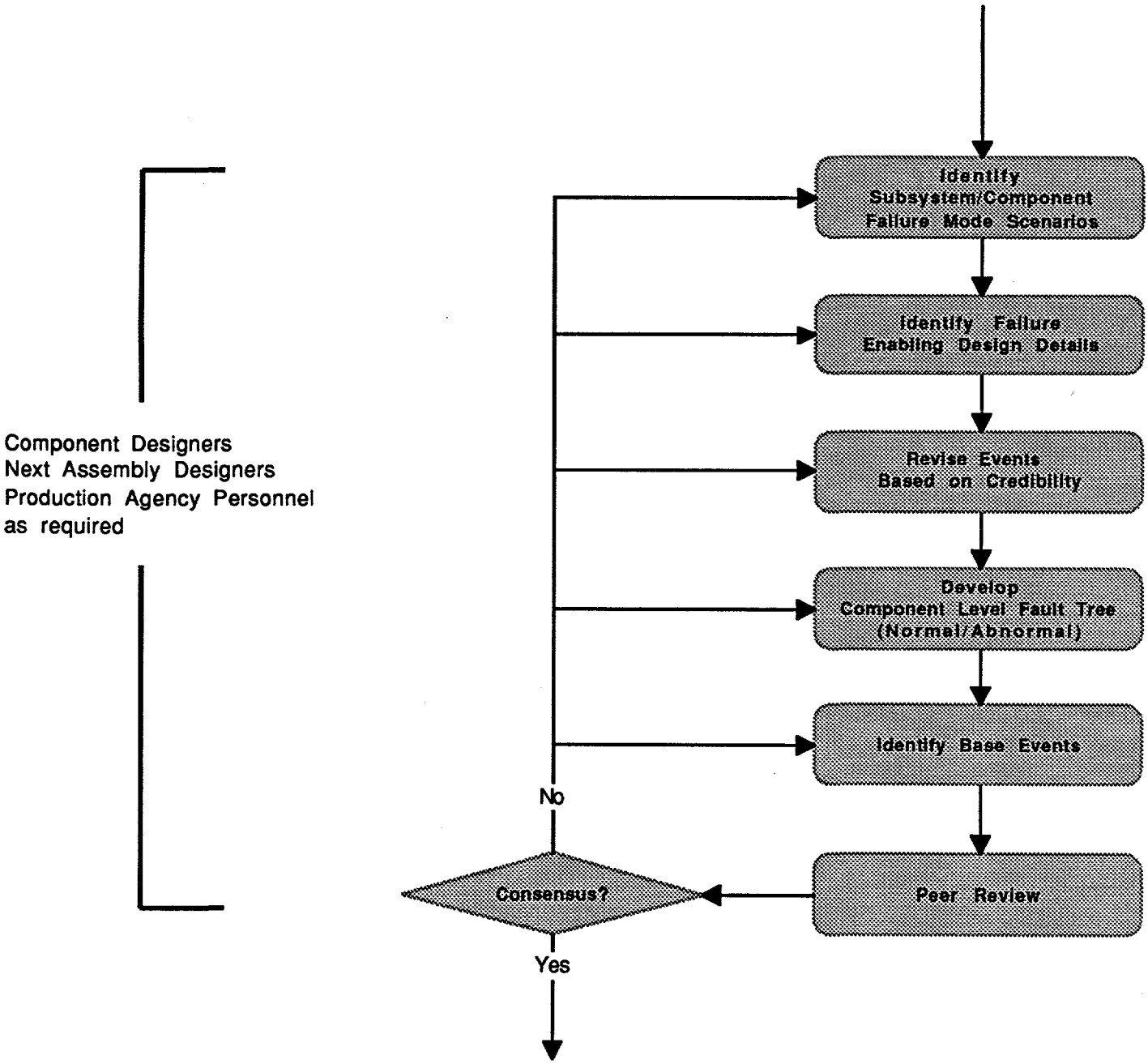

Figure 1 (continued)

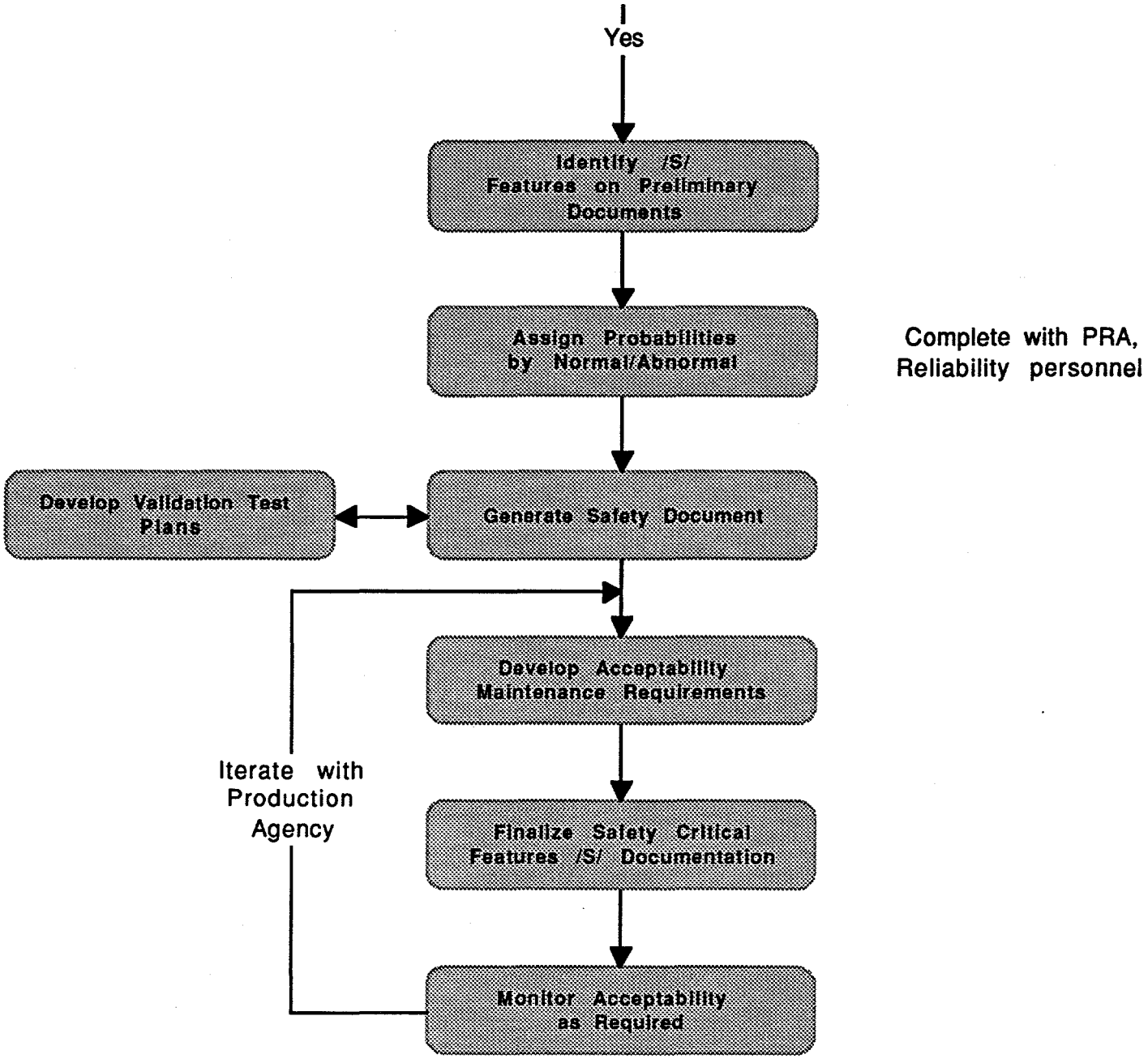

Figure 1 (continued)

conceptual approaches to assure safety against the critical scenarios are developed, reviewed, solidified, and then extended once again to the next level deeper in detail for the "significant" components.

Throughout this iterative process, systems, subsystems, and component level engineers discuss requirements, potential failure modes of the system, design alternatives, and associated implications. Ultimately, safety requirements are partitioned, extrapolated, and extended to the individual component levels for detailed design to proceed. From there, the iterative process continues, but at a more detailed level, until component design is completed, failure mode analysis is finished, and validation of the design is successfully implemented.

The process then focuses on the production and long term concerns for the device and its piece parts, homing in on the identification of specific features (contours, devices, material properties, etc.) which, if not produced correctly, could allow a fault state to occur with ultimate potential for an unsafe system condition.

Software Fault Tree Analysis Process

In exploring the extension of the process depicted in Figure 1 into the software arena, it was found that virtually all approaches, on a conceptual basis, could be applicable to software. As the software fault tree analysis process emerged, it became obvious that the hardware and software processes worked in parallel during the general product development process. Figure 2 illustrates a consolidated version of the hardware process depicted in Figure 1 (top path) coupled with the counterpart process for software (bottom path). Also shown in Figure 2 is a generalized product development process (middle path), and how the hardware and software fault tree analysis processes overlay and integrate with development. The following discussion explores each facet of the software fault tree analysis process, discusses its intent and implications, and illustrates its applicability to software through a specific, though simplified, software example. In the discussion that follows, the text refers to the title of a box within the flow of Figure 2 by using boldface type.

The software fault tree analysis process is also coupled with High Consequence System Surety (HCS^2) Process (*ref:* SAND 94-3223) in the following ways. The functions performed in the boxes shown on the first page of Figure 2, up to conducting the joint requirements review, indicate more detail regarding the actions involved in the HSC^2 process through the decision step to determine if the surety theme is acceptable. The steps of Figure 2 are principally applicable to the lower level subsystems developed to meet a higher system (or end product) need. To understand these interactions, consider an aircraft being developed for delivery to the commercial airline industry. The airplane represents the higher level system depicted in the HCS^2 process, while the guidance system would represent the system (or *sub*system) level for the process in Figure 2. The remaining steps in Figure 2 represent the tasks undertaken in the Conduct Surety Theme Analysis step of the HCS^2 process, as coupled with a generalized product development process *and as applied to a subportion, or component of the higher level "system"* represented in the HCS^2 process diagram. To complete the development of the overall higher level system, the general approach indicated in Figure 2 would be applied to all subsystems, and integrated together for the overall systems approach. This aspect is indicated by the Integrate Elements step and subsequent steps identified in the HCS^2 process.

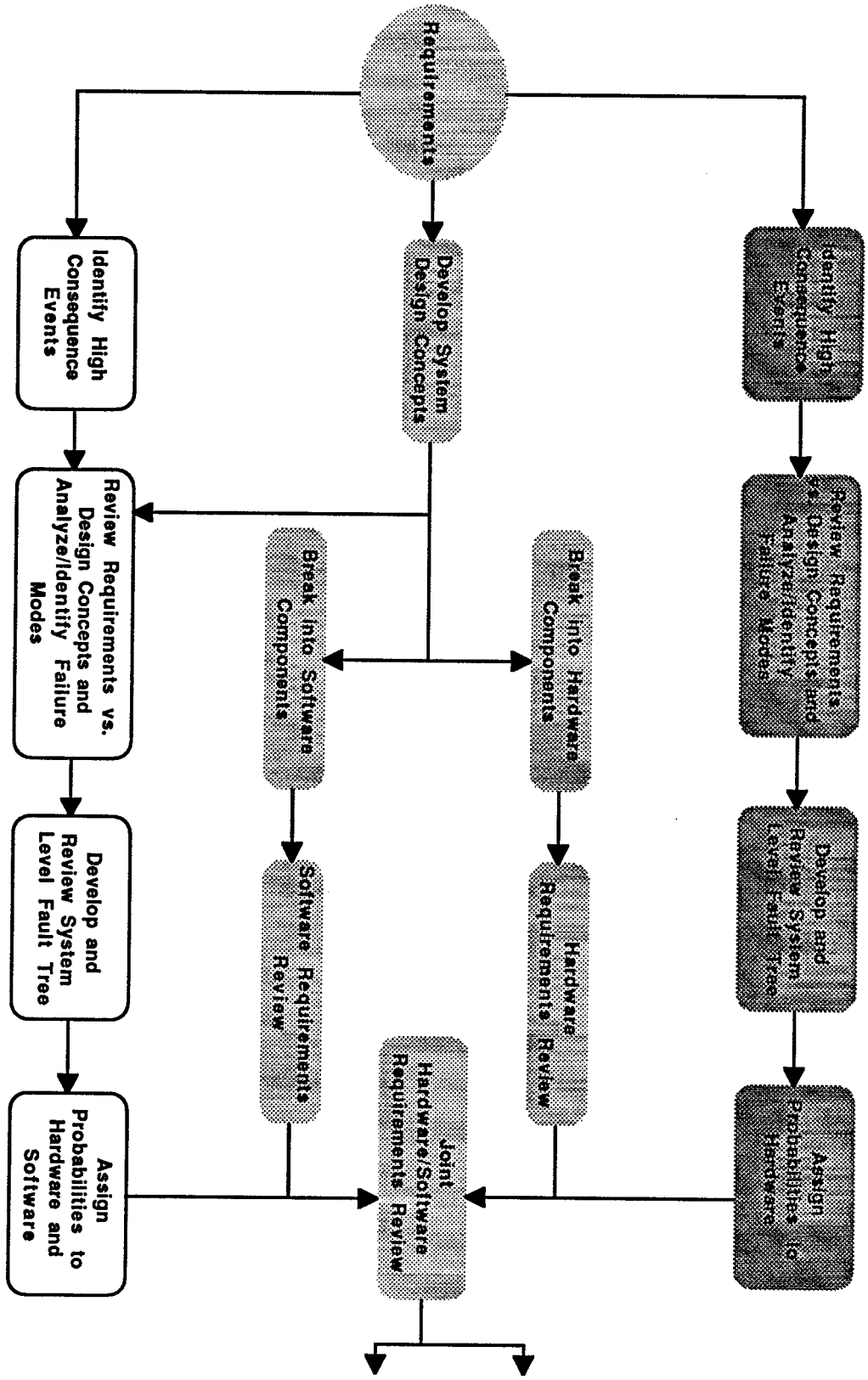

Figure 2: Software and Hardware Fault Tree Analysis Processes Overlayed with a General Systems Design Approach

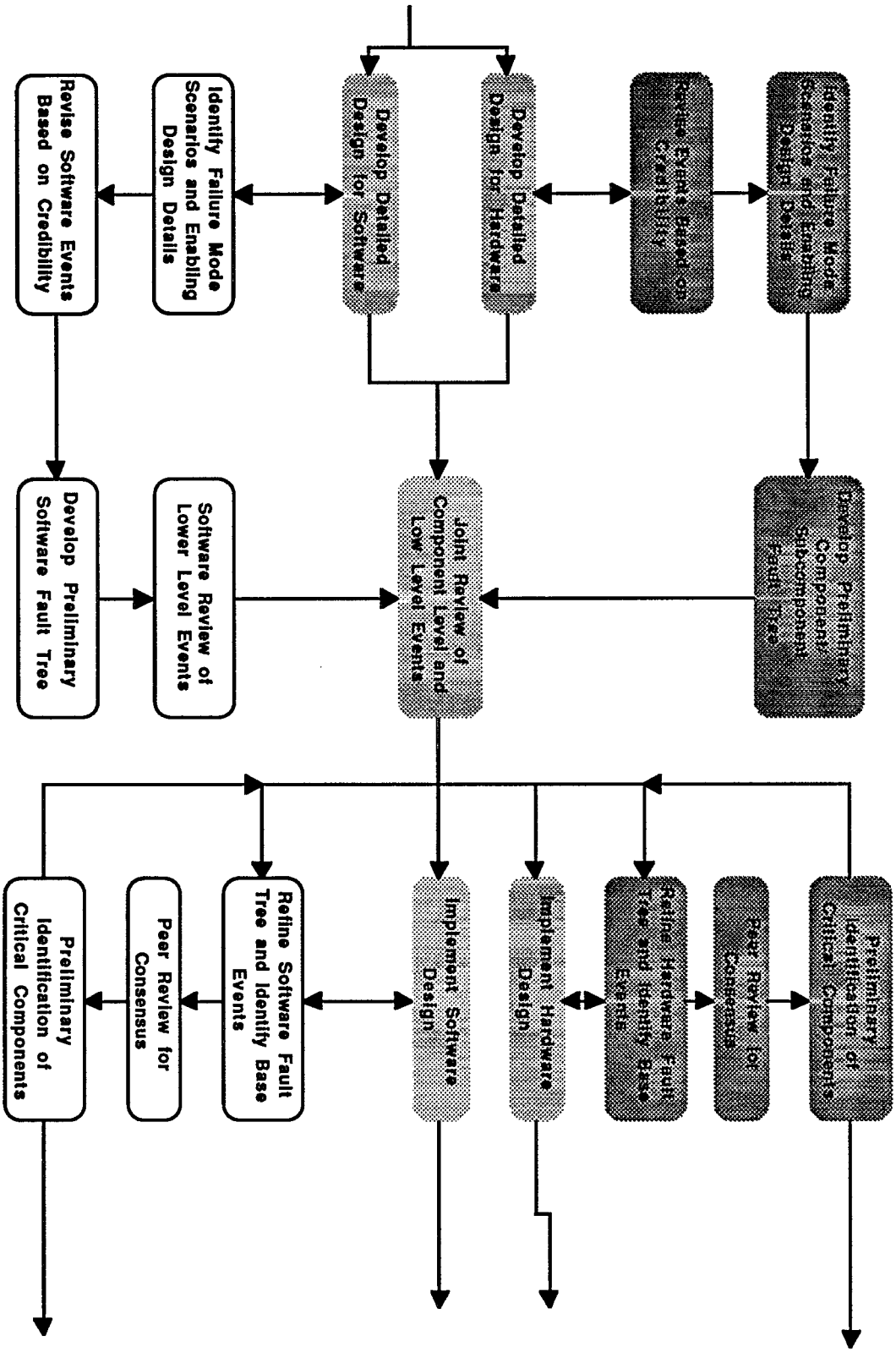

Figure 2 (continued)

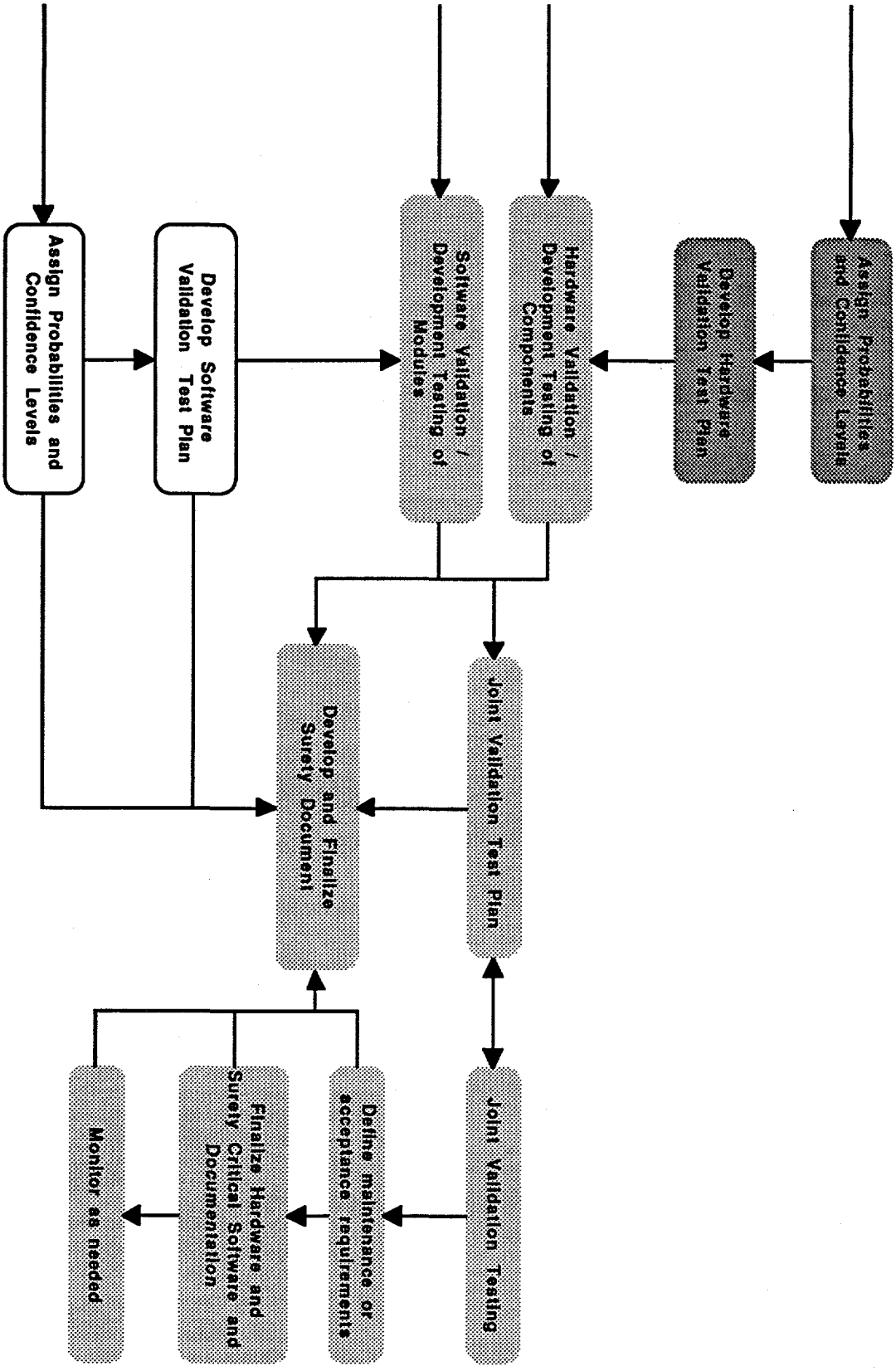

Figure 2 (continued)

The analysis process begins at the integrated hardware/software (*system*) level during the early conceptual stages of the product. At this point in time, the system requirements may be available either in draft or final form. When requirements are still draft, the process will be less formal and will likely result in considerable iteration through the initial stages of the analysis at the system level.

An important first step in the fault tree process is the formulation of a basic **identification of high consequence event(s).** A high consequence event is one whose *consequence is judged to be severe enough to cause significant loss of investment or loss of life.* It is important to establish the boundaries of the software being studied and to examine failures within those boundaries that might lead to a high consequence event. This means that when a specific software function is defined, it needs to be stated in terms of its boundaries - what input conditions are valid, who has authorization, when it should not be performed, and so on. Situations that could lead to possible critical failures include: (1) a system performing a function that it should not perform; (2) a system failing to perform a function it is supposed to perform; or (3) a system successfully performing a function under the specific conditions when the function should not be performed. A software system which operates a valve it is not supposed to operate (1), which fails to maintain a required door lock (2), or which provides access to an unauthorized user on February 29 (3), are examples of what could be critical, high consequence events.

In identifying the event(s), the requirements and possible use environments for the system form the basis for future analysis. Typically, representatives from a broad experience base would engage in dialog to identify the event(s). Human error is considered an enabler for a high consequence event. When multiple events are identified, as with a possible safety critical event and one which has only reputational consequences, they may be prioritized based on the potential significance of their undesirable outcomes. Multiple events could be identified for any of the surety concerns, such as quality, reliability, safety, security, and control. Analysis proceeds for one single event at a time.

The analysis proceeds with a **review of the system requirements vs. design concepts** for their expected ability to meet these requirements, and an **analysis to identify failure modes.** This portion of the process is intended to identify possible ways failures could occur which would bring about the occurrence of the undesired high consequence event. Often these activities are conducted concurrently, perhaps in a single meeting. The activities may be conducted more independently as the complexity and formality of the project warrants. The participants in these reviews are typically the systems engineers, as well as the component engineers who only *may* be known based upon the degree of specificity in the preliminary conceptual design. For example, in the first of these meetings, the review and requirements examination may be conducted by the systems engineers in conjunction with the next level subsystem designers. Subsequently, as the details of the design unfold, additional meetings may be needed involving deeper level component engineers with next assembly and systems engineers. In a project of this layered type of complexity, it is likely that several meetings would be needed, iteratively, throughout the initial development stages as more detail becomes known about both specific requirements and design concepts.

When reviewing the system requirements, the participants try to gain a deep understanding of the surety requirements, evaluate their achievability, and prioritize their potential consequences. Surety requirements need to include what the system will *not* do, and the associated performance specifications. While it is often easy to develop and verify software which will successfully do what you want it to do, it is often not easy to verify that it will not do the desired function when it should not be permitted. Thus,

circumstances need to be well defined and understood for which the undesired function is *not* to be performed. To illustrate this point, a requirement might state that the software is to permit access when the user enters the correct numerical sequence. That the software successfully achieves this requirement is measurable. But without a specification regarding what is to happen when the incorrect sequence is entered, it is even possible that the access could be provided, say, when the user enters an *alphabetic* sequence. At this stage in the development process, there is likely to be variability in the amount of information both known and specified about a given requirement. In addition, some aspects will be better specified at a deeper level, *i.e.*, at the software equivalent of a component level. Thus, the participants will need to resolve such variations and agree on the necessary specificity to properly define the system.

In any case, the hardware portion of the review will focus on a clear understanding of the physical boundaries of the system and the system interfaces. Participants develop an understanding of the functions, environment and the operations of the system, specifying both normal (environments the system is expected to operate in) and abnormal (environments that are out of the systems operating range) environments. Similarly, for software, the participants reach an understanding of what constitutes normal use conditions (correct expected input data, task or event sequence), and what will be considered as abnormal input (incorrect data, inappropriate task or event sequence). Special attention will need to be placed on defining the software performance in the event that the abnormal condition occurs.

During this review the preliminary design concepts will be examined along with details regarding operation and maintenance. This portion of the review focuses on establishing engineering confidence that the system design can be expected to meet the surety requirements that exist and their associated understanding. Participants will be trying to establish the potential for specific design concepts and approaches to reduce the occurrence of system failures.

As the requirements and design concepts solidify, the team can begin to analyze failure modes. Portions of the design whose failure would have a potentially significant, or high consequence, are examined in as much detail as possible. This examination is usually a coordinated brainstorming session attended by system experts to develop a list of undesired events, including possible hardware-initiated failure modes. For each possible failure, the team identifies the potential consequences of the malfunction, tracing through the event failure, coupled with local interactions, and extended to next level functions. For example, it might be noted that permitting access to an individual who enters the incorrect password is the undesirable event. The team would analyze the possible ways the system might permit this to happen. As an example, the team might determine that one possible way the fault could occur is for the software to correctly determine the password was invalid, but give the user access anyway. Another way might be that the software correctly determines the incorrect password and denies access by not sending the enable signal to the mechanical lock, but that access is provided anyway if the mechanical lock was left in the enabling position after completion of the previous correct user functions. Thus, the focus at this stage of the analysis is on the combined system level performance. However, when specific functions within the design concepts have been allocated to software or to hardware, the analysis can focus on the associated specifics to the extent possible. Since this portion of the analysis involves component-level designers (both hardware and software) and occurs before any software implementation, it may be possible to avoid the selection of software implementation approaches which could lead to the unwanted problems. After all credible events and potential failures are listed, the events would be prioritized starting with the most critical event.

As a result of the previous interrelated and sometimes concurrent activities, the team determines the system failure events. For each area of major concern, such as security, reliability, safety, or control, the top level event is determined. Examples are: incorrect X-ray level causes patient death; equipment malfunction causes fabrication shutdown and substantial profit loss; or improper signal causes inadvertent detonation. In order to avoid any possible confusion, analyses should proceed separately for each of the identified events, since a failure mode causing an explosion may be caused by a completely different aspect of the design than might one which leads to inadvertent access. The top event(s) is (are) then identified in a fault tree diagram. This step is typically done by a team including system designers, some component or subsystem designers, and software developers, and often includes representatives from the production team or agency and possibly software maintenance staff.

Often the identification of the top level event(s) is done in conjunction with completing the fault tree analysis at a high level. The fault tree is a graphical model of event combinations that can lead to the occurrence of a specific hazard or event. The **system fault tree events are developed** by successively breaking down events into lower level events. The analysis may take the team down to failure events for a subsystem and several specific components before the component design actually proceeds and a detailed component level fault tree is generated. The information developed up to this point is then entered into a system fault tree diagram using one of several available software packages. Event numbers are assigned during the process of generating the fault tree. Each event consists of a unique designation that is denoted with a unique code consisting of a letter and a series of digits. The "higher" level event is the output of the gate and the "lower" event is the input to the gate. Each event number is unique to a given fault tree. Figure 3 illustrates this documentation convention for an actual hardware safing device.

After the preliminary fault tree is developed as just discussed, a **review of the system fault tree** is conducted with necessary personnel to verify the events and make any necessary changes. Once again, this is generally conducted jointly at this early stage in the development process. Again, representatives from the systems, component (both hardware and software), and production or maintenance organizations are generally included in the review.

At this point, enough should be understood about the system to begin a decomposition of the conceptual design into the various hardware components which will be needed and which will require further design and development. For example, a system might require a motor, a cooling subsystem, a pumping subsystem, and a monitoring subsystem.

It is also possible to **break the system into software components** as well. It may be less obvious to the reader how this software decomposition might proceed, so we will consider what might happen by exploring an example which begins with a software conceptual design.

Suppose we were to design a system which opens a valve when an authorized operator requests the valve to be opened. For such a system, we would expect the *hardware* components to consist of the valve and the computer system which authenticates the operator and opens the valve. Conceptually, the software would perform the authentication and initiate the opening of the valve. This type of preliminary concept begins a high level functional allocation to the hardware and software portions of the system.

Proceeding further, the software might include a module which captures input from the operator, one which authenticates the operator, and one which causes the valve to open. Were we to proceed further we might look in more depth at the authentication component

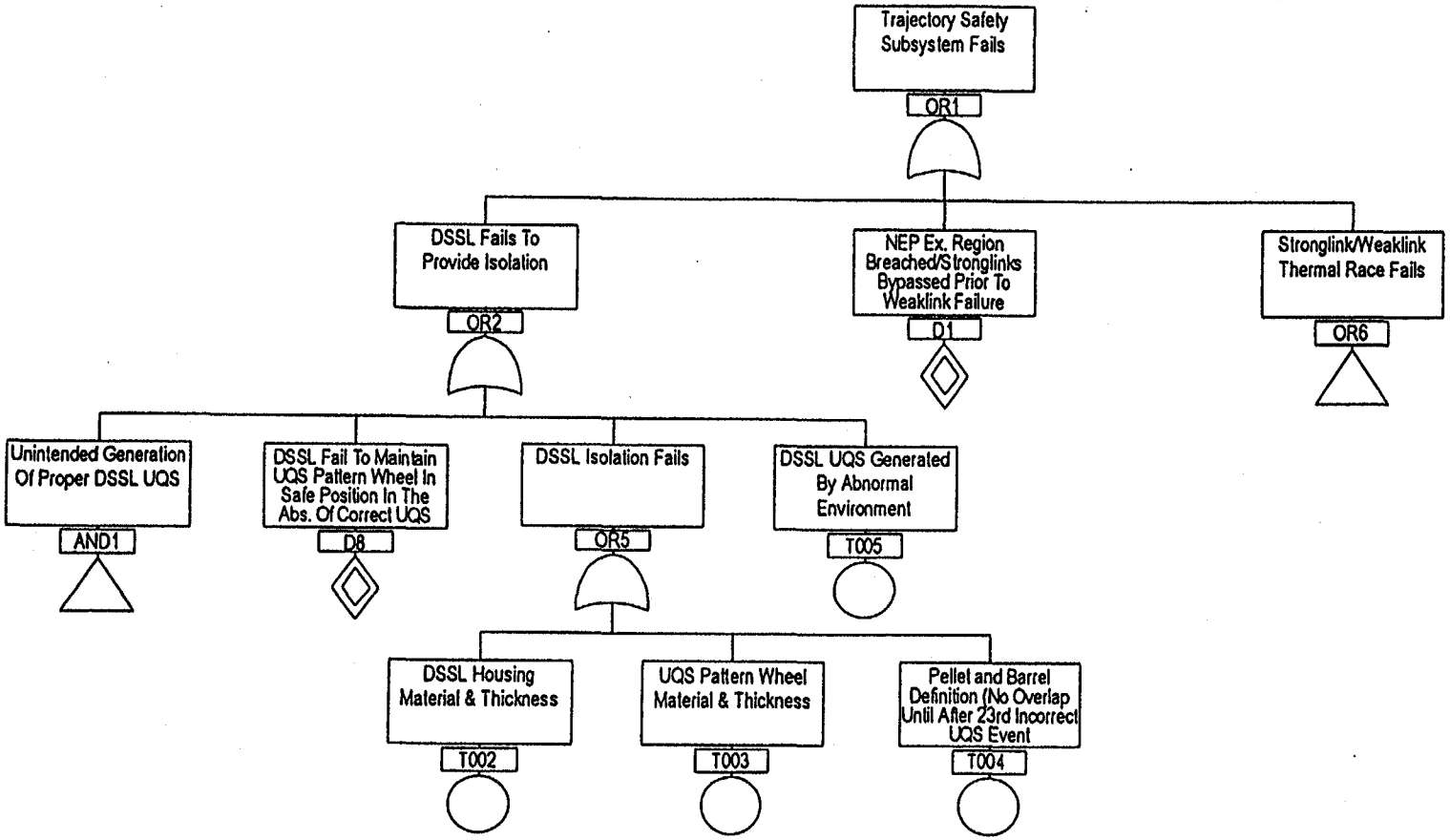

Figure 3: A Portion of a Hardware Fault Tree

1TRAJ_2
DLG:5355
7/24/92

(or module) of the software and determine that it would need a component that retrieves the correct user identification data, one which retrieves the user input from the module which captured user data, one which compares the two sets of data for a match, and one which returns the message regarding user authentication or an error message. Figure 4 illustrates this example. Although this example is rather simplistic, it will be referred to later to illustrate some further concepts in this methodology.

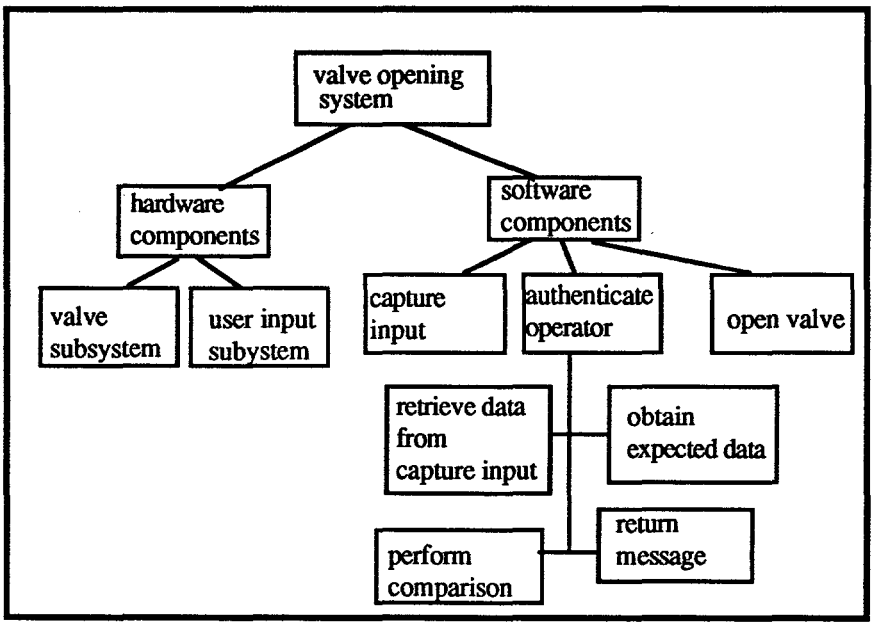

Figure 4: Example System Structure

Even though at this point the software design is mostly high level and conceptual, it is important to begin adopting a mindset focused on identifying inherent weaknesses, or faults, in the approach. At this point, a brainstorming session could be employed to identify what could go wrong at the subsystem or component level and to refine the design hierarchy.

As development of the software design progresses, there comes a point where it is believed that all major module/component decomposition has been identified. A **review of the design against the software requirements** to assure that there are no glaring oversights or deficiencies would be conducted using the completed hierarchy. This review should include appropriate software development staff and is intended to identify any problems requiring resolution. For example, from Figure 4, focusing on the authenticate operator component, the review session could reveal that an additional component is needed to "reset" the system, so that the next operator could not open the valve using the previous operator's authorization. This need may be uncovered during discussion of the possible ways that the software could allow an unauthorized individual to open the valve.

The participants in this review should all have a clear understanding of the software design approach, modules, functional allocation, and requirements. Participants should focus on verifying that the proposed design concepts are expected to meet the requirements, and that there are no obvious flaws in the design logic and algorithms which could lead to unwanted high consequence events.

Assigning probabilities starts at the system level but often the numeric allocation is made at the subsystem or component level. Probabilistic Risk Assessment (PRA) and Reliability personnel have thus far generally assigned the probabilities. For hardware safety systems, the probabilities may be assigned based on the definition of normal and abnormal environments. For hardware reliability concerns, a system reliability figure is established and failure mechanisms established. Often, a reliability model is compiled from a similar system or collection of similar components and is used to assign a specific reliability number to a particular subsystem or component.

To date, software has been deemed sufficiently complex and without failure mode models which are themselves reliable to a high confidence level. Thus, software is typically assigned a failure probability of one and the analysis proceeds for hardware exclusively. Methods for allocation of probabilities to software, other than 1, need further research. In fact, later discussion illustrates one approach which may prove useful in identifying failure probabilities (and conversely reliabilities) for software. In order to proceed with the analysis for software, the process has to temporarily assume a failure probability of 1 for the hardware, so that the fault paths that arise in the software portion of the system may be examined freely and thoroughly. The analysis also proceeds with the traditional failure probability of 1 assigned to the software so that the hardware fault paths may be thoroughly examined and identified.

Up until this point the focus has been on assuring that the design concepts to be employed are likely to perform as expected and *not lead to the occurrence of high consequence events*. It is this perspective of the *requirements* which set these activities and reviews apart from those traditionally viewed as "quality" reviews. Returning again to our example, it may be determined that the best option for the system is to include a mechanical "lock" on the valve, and require the opening task to have both a software "authorization" and a hardware "enable." It should be clear that without dialog between the hardware and software development teams, the needs which may emerge from either point of view may not be apparent to the other side. For instance, the hardware team could notice that the software would have to obtain data from a permanent memory chip, and merely assume that the software development team will build this into their design. Conversely, the software team could identify the need for a mechanical device to couple with the software, but have no natural mechanism to discuss this possibility. Consequently, it is essential to have a **joint hardware-software requirements review** with respect to higher level fault tree events. Both the hardware and software teams have been working on their respective fault paths, identifying their own mitigation options, and will have considerable information to share. This joint review, however, must not resort to merely summary information, for detailed discussion is necessary to be sure significant faults are not overlooked. The review of higher level events for hardware and software will determine if the proper top level event is being analyzed and also to determine if all credible events have been identified.

Each development team now proceeds with developing further detail on their respective components. The results of the work thus far in this process will be the identification of those portions of the software (or hardware) which have a perceived "risk" or likelihood of failing and causing a high consequence event. Also identified are those portions of the software which are of less concern from a failure perspective -- possibly a prioritization of concerns will be obvious.

For our example, if it were possible to assure that no matter how the component which captures input data were to fail, the component which compares and returns the authorization message would always return the correct answer, then the capture input

component would be "less critical" than the perform comparison and return message components.

The software design begins to take on the character where the high consequence events are localized to specific software components. It will now be possible to concentrate efforts to mitigate the consequences only where the major concerns really are. We would thus expect to find a majority of our efforts regarding verification, review, validation, and detailed documentation to be applied to those modules which are identified in the high consequence failure paths. Clearly, though, if resources and schedule permit, the development team may want to apply a similar level of effort to lesser consequence components to assure the absence of errors which could adversely impact customer satisfaction.

The development team participants now face analyzing the software design in detail, before the code has been written, to identify potential failure paths and to assure that proper decisions are made regarding these paths. The analysis done to **identify software subsystem failure mode scenarios** proceeds iteratively during this detailed phase in the development. At this time, the attention turns to the specific algorithms, execution sequences, memory utilization, and error handling. To illustrate the mindset essential to this portion of the analysis, we will examine several situations using our example in Figure 4.

It is likely that software providing user authorization will employ some form of data encryption algorithms. Before the software code is actually written, the analysis team would closely critique the elected approach, looking for all the possible ways the software could fail and lead to the undesired event -- say permitting someone to open the valve who is not an authorized individual. A close examination might reveal several scenarios under which this might happen. First, the software might capture the user information incorrectly such that on comparison with the "real" stored information, it would be a match. Second, the software might capture the user information correctly, correctly compare with the stored valid information, find the mismatch, but return authorization anyway. Third, the software might capture the user information correctly, but compare with the wrong data from memory, such that there is a match and authorization is verified. As many of these types of scenarios are identified as the development team can generate. This information alerts the developers to areas where special efforts to mitigate these scenarios from occurring will be necessary.

Another example can be seen in the data encryption algorithm and reprocessing sequence. Two approaches come to mind for the processing sequence. First, the software could capture the user information, encrypt it, retrieve encrypted data from memory, compare, and do a checksum calculation. Alternatively, the software could capture the user information, retrieve an encrypted value from a memory location, decrypt it, compare with the user input and compute a checksum. In this latter approach, a possible high consequence security fault could occur. In yet another scenario, the software could implement the selected encrypt/decrypt algorithm incorrectly, yielding incorrect checksums and also leading to a high consequence fault.

As such scenarios are being identified, the fault tree begins to emerge. The next level of detail is determined as the team **identifies software failure enabling design details**. The development team, with the assistance of system designers, hardware designers, and production personnel as required, look for all possible ways for a failure to occur within the critical software component or module. Again returning to our example, we explore some ways in which the software could compare the information as entered by the operator with the wrong stored information and return authorization inappropriately.

Consider that the system has been used once and correctly gave access to an authorized individual to open the valve. With the task successfully completed, the computer system is left on. The next user (unauthorized) now interacts with the system to open the valve. The individual correctly enters his unauthorized information. Now, since the software design did not include "reset," the contents of the temporary storage location from the previous user access is used, and the comparison is made with the contents of memory. Since the first user's information was valid, the software will now again authorize access. In this way, no matter what subsequent users enter, access could always be granted until system restart. In this case, the enabling design feature would be the storage of the user information in the temporary location.

A second possibility exists for this fault to occur if the software were to employ a lookup table to identify the location of the correct information for the comparison. A single point error in the table entries could cause the software to look in the wrong spot for the information, and subsequently compare to find a match. The enabling design detail in this case would be the lookup table. Yet a third mechanism for failure might be that the software uses multiple memory locations for the storage of the correct information. The software might incorrectly retrieve only a subset of the necessary locations, such that upon comparison with the correctly entered, encrypted information from the second user, a match might result. The enabling design detail here would be the retrieval sequence in the software.

Typically in concurrent discussions, the identified enabling design details are reviewed for their credibility based on the design and use scenarios as presently understood. For example, if the software were to employ an automatic restart, including memory initialization for every user request to the system, the scenario wherein the information from the previous operator is used would be impossible. Thus, methodically, one by one, each scenario is examined and the team will **revise the software events based on credibility**. Events not needed in the fault tree analysis are eliminated.

It is important to note that for the activities just discussed the team participants first assume that the undesirable event will occur and then begin to look for the possible, credible ways by which it *could* occur. As in hardware fault tree analysis, these events are compiled together in a tree, using logical AND and OR gates. In this manner, **the preliminary software fault tree** (and similarly the **preliminary hardware fault tree**) are generated. Figure 5 shows how this approach could be applied to the example just presented.

By completing this process in detail for those portions of the software which are identified as critical in their ability to cause an undesirable high consequence event, it is possible to identify in advance of software implementation those approaches which will lead to the occurrence of the fault. The software team will want to review **lower level events against design options** to identify what has to be done to assure that the incorrect implementation is not coded into the software and what tests will be needed to demonstrate the acceptability of the implementation.

After the initial independent development of the hardware and software fault trees and their respective reviews, a **joint hardware/software component-level review of lower level events** is done to determine if enough detail has been given to the fault tree analysis, and to assure similar interpretation of the events and joint consensus regarding their importance. In addition, this interaction provides a further opportunity to be sure software features and requirements which impact hardware are properly communicated to the hardware folks and vice versa. For example, consider the case where the software

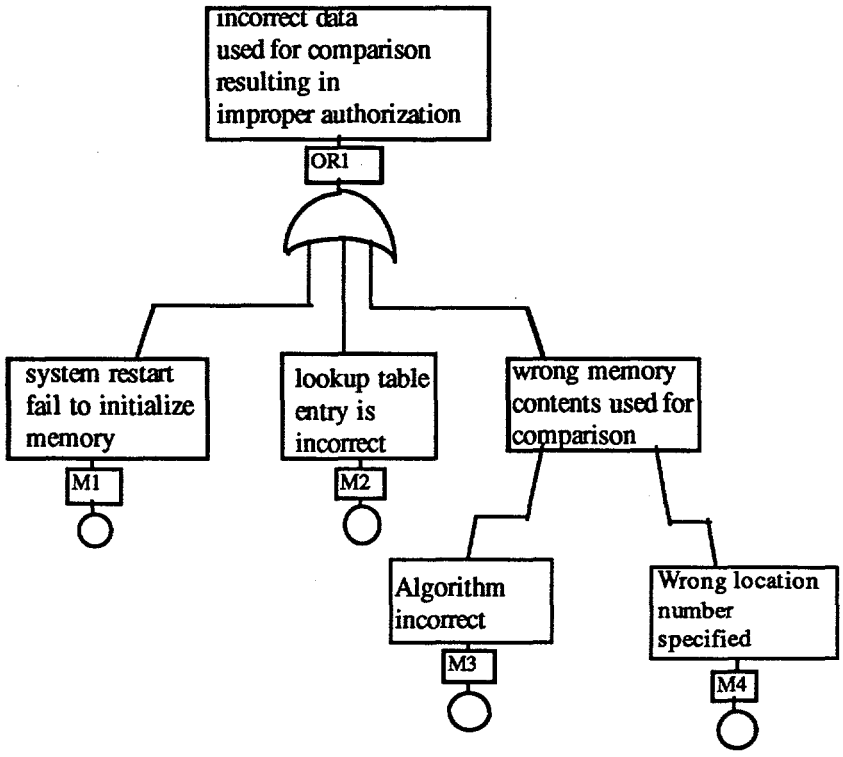

Figure 5: Partial Fault Tree Example

analysis has determined that a single software failure could result in inadvertent opening of a valve. The potential consequences of this failure may be significant enough that the software team recommends inclusion of a hardware backup enabling device. This recommendation must be communicated early in the project, and this joint review provides a forum for the recommendation to be explored in detail to the consensus of the hardware and software design teams.

The outcome of this joint review process may necessitate changes in either fault tree, so that both design teams may need to independently **refine hardware and software fault trees** as the designs are being implemented. This may simply require additional details or may require fault path modifications and event changes. As part of this process, the software development team continues its analysis of increasingly more detail until the events are determined to require no further decompositional information. This implies that a decision needs to be made as to what level of detail the fault tree will be developed. Once the detail level is established, this will signify that the appropriate limit of resolution has been reached for the fault tree. The lowest level faults indicated on the resulting tree are considered base events and the developers **identify base events**, as indicated by M1, M2, M3, and M4 in Figure 5, on the fault tree.

Once the changes are implemented and the fault tree refined, the software team conducts a **peer review** to assure complete coverage of the concerns and consensus regarding the results of the analysis. The review would typically involve the software development and also maintenance team(s) and associated surety experts, and may involve the next assembly and component designers, and possibly Production Agency staff. If consensus is not reached at this time, the software development team would then go back and review subsystem failure mode scenarios, design details, software fault tree events, base events, and the software level fault tree, revising as needed and iterating with peer review until consensus is reached.

Given that the fault tree is now complete and that most certainly software development has progressed to some degree in parallel with the analyses, software code whose functions or algorithms were identified by the analysis as critical must now be analyzed and reviewed to assure adequacy and to **identify them on preliminary documents.** Many options could be used for identifying these critical software design features on associated software documentation. The option chosen for this work is to use /c/ (pentagon c, for "critical"). The /c/ designation parallels the /s/ used in safety critical hardware. The /c/ designation would be used in each type of documentation of the particular feature, along with the critical event number. Types of documentation to receive the annotation include scripts, flow charts, information models, hierarchy charts, data flow diagrams, code listings, algorithm definitions and mathematical proofs, test plans, test results, user screens, relational tables, variable definitions, memory allocation schemes, or any other form of documentation used to define the software design for designers, evaluators, users, or maintainers. The /c/ designator indicates that the design feature or approach was judged to be critical and that failure would contribute to the occurrence of a high consequence event. This identification alerts the necessary individuals that changes could have adverse affects, and that additional evaluations may be needed to validate and justify changes.

Returning to the example of Figure 5 and considering events M3 and M4, we notice that it is crucial to be certain that the correct memory locations are used for the comparison. Thus, any portion of the software which implements a location counting algorithm (M3) or which simply designates a specific memory location (M4) are associated with the undesired failure. In the software design (prior to coding) developers may create a mathematical counting algorithm for subsequent implementation during repetitive functions. Alternatively, the design may also employ specific address identification in the retrieve and compare modules. *Both* these approaches would have to be identified as critical features and identified with a /c/ on associated documentation. Thus, any text-based algorithm development documents and resulting specifications would be marked with a /c/ and a note that the algorithm is associated with the failure event. Subsequent code implementation would then place some header information indicating that the module contains critical implementations associated with critical failure, as well as comment information at the actual source code implementation lines. Figure 6 indicates how this might be applied in the design and implementation.

In conducting this sort of in-depth code analysis, the code can be broken into templates according to the semantics of the programming language. Analyzers and team members can then review the logic structure of the software to detect software logic errors, even before the onset of formal testing. Since the entire software package is not generally deemed critical, the in-depth analysis can focus on those portions which can have adverse impact. As time and resources permit, other less critical sections can be subjected to analysis as well. This natural allocation of project efforts to the critical areas can be coupled with other analysis methods, such as Pareto analysis, to identify areas of principal concern, and to assure that resources and efforts are focused on items of high potential payback.

The next portion of the process focuses on the **assignment of probabilities or desired confidence levels** in the analysis breakdown. As mentioned earlier, software is typically considered today to fail with Probability 1, since complexities generally have prohibited more detailed assessments. Consider again for a moment the example illustrated in Figure 5. Figure 5 identifies four possible failure mechanisms which could

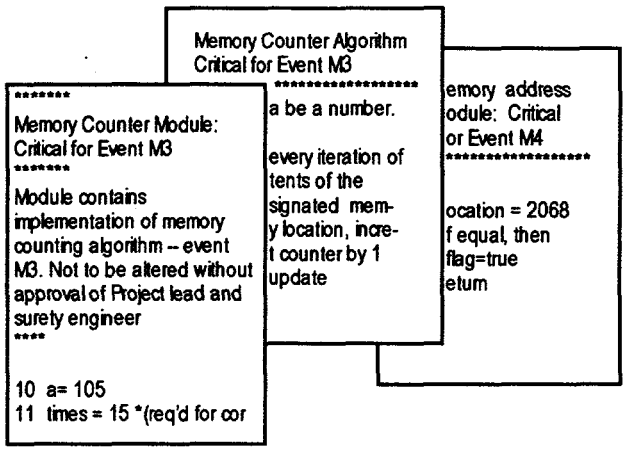

Figure 6: Example Software Documentation for Design and Implementation

cause the undesired failure. Since initial data may not exist to set limits otherwise, let us assume that each possible mechanism could occur with equal likelihood. Then each of the four events would have a 25% probability of occurring, if the undesired failure is considered to occur with probability 1. As data are collected regarding occurrences of particular failure types, these probability estimates will become more realistic, and this is an area ripe for future research and study. For this simplistic example, the design team may decide they will not implement any specific location calls for the memory content comparison. The inherent "reliability" of the software against this particular failure type would thus increase, since the likelihood of failure from three of the four possible mechanisms is reduced to 75%. Further, as mitigating actions are taken in the software design and implementation, it may be possible to demonstrably reduce the likelihood of the occurrence of base events M1, M2, and M3. By demonstrably, we mean through either testing, determining that the path is physically impossible, or other means. As this concept is extended to a failure type that has perhaps 100 ways of being implemented (all with equal likelihood) then a design which employed only one of them would have at most only a 10% chance of failure and 90% chance of success. Thus, as this kind of approach is adopted and refined, assignment of more understandable and meaningful approaches to software reliability and associated confidence may be possible.

At this point in the development of the system, the results of the analysis for both the hardware and software portions are merged and combined into the **surety document** for the overall component, subsystem, or system. The surety document describes the safety requirements and the environments in which those requirements must be met. The document also lists the event identifiers, the title of the event as listed on the fault tree, the parent event, and failure order number. The failure order number indicates whether events must occur singly or in combination (AND gates) with other events in order for the undesired failure to occur. In addition, this document is revised and expanded until it ultimately contains information regarding the necessary control requirements, rationale and background behind the selected design implementation, reasons for its being identified as a failure enabling design feature, actions taken to mitigate adverse consequences of failure, analysis and test reports generated during the validation of the design (next section), references to product (hardware and software) drawings, and any ongoing acceptance criteria. The design itself may be verified by doing any number of boundary and tolerance studies, material analyses, test plans and results, or other studies or simulations. This

document is iterated as needed to ultimately provide complete documentation regarding the analyses and verifications conducted.

As the analyses conducted indicate those areas of principle concern in assuring that a particular failure mode does not occur, the team's attention turns to **software level validation development testing** in order to assure that all identified critical features have been appropriately validated. This can be done through formal inspections and walkthroughs (these may be conducted and phased in along with any of the analyses). A detailed test plan is then developed, focusing on both verification and validation of the software. As discussed in the Sandia Software Guidelines, validation activities are geared towards assuring the requirements are met, while verification activities are aimed at showing that the design logic and structure are adequate and cohesive and that the design has been properly and correctly implemented. Team members focus on how much testing for completeness should be done based on the potential consequences of the undesired events and those techniques employed to mitigate the identified failure methods. The testing plans are implemented and the results documented; both are generally included in the surety document.

Wherever possible and sensible, tests are conducted at the lowest level so that software gets examined for errors independently of the hardware. Many tests, however, will need to be combined hardware/software tests. These joint hardware and software tests are developed and documented in **joint hardware/software (HW/SW) validation plans and testing**. Once again, test plans are refined as necessary and the results documented and generally included in the surety document. As documentation grows, the team may also decide to compile test results (or other documentation) into a separate document. As with the documentation of the software design, each test of the critical software elements are identified with event number and /c/ annotation.

Although software may be thought of as finished after the completion of testing, many software products may require mass production, such as is the case when the product is supplied in diskette form, or when it is placed in Read-Only Memory (ROM) devices. The team will need to explore what acceptance criteria will be applied to these devices to assure that the correct code has been provided. They may even wish to consider periodic refresher loads of the code into the manufacturing equipment, or periodic inspections of integrated circuit masks. Such **maintenance acceptability requirements for mass production** are compiled and documented. Once again, /c/ notations are used to identify activities associated with the event numbers they are intended to address.

As these previous tasks are undertaken and tests conducted, it may become necessary to revise either the event tree or the software design and implementation. In addition, it could be determined that costs for conducting particular tests are too high and that the tradeoff in reliability and surety improvement do not justify the costs, so that a different design approach would be needed. In such cases, changes are incorporated into the appropriate form of documentation. Critical software features are reviewed throughout the documentation to assure the inclusion of the /c/ notation. In this way, the /c/ **documentation is finalized** and the comprehensive documentation package compiled.

As identified earlier in the determination of the verification and acceptability requirements, ongoing **monitoring of acceptability of the integrated hardware/software component or system may be needed**. Tests could be conducted which contribute new information from new test paths to the statistical basis for any allocated software failure probabilities. Such evaluations are conducted and documented as needed throughout the life of the product.

As the details regarding the fault analysis, the verification activities undertaken and implemented, the options pursued during development, the possible concerns if critical software is changed, and testing results unfold during the development phase, these results would be incorporated into the surety document for the particular system or subsystem. The surety document will then provide both evaluators and system maintenance staff with the necessary information to assure ongoing confidence in the product. To be of maximum value, this document, along with the other forms of software and system documentation, will require updating as changes warrant.

Future Steps

As previously noted, the original project intent was to apply this methodology to the Weight and Leak Check System (WALS), an automated nuclear weapon component handling system, and look for refinements to the process as needed. Redirection of the High Consequence System Surety project precluded this from happening at this time. Such an application will need to be identified and explored to determine what full potential benefits could be derived from the use of this process. The application selected should be one in which the process can be applied from the conceptual stages on, not one in which the software development is in progress. One of the distinguishing aspects of this methodology is that it is applied *before* software code has been developed so that potential pitfalls in development may be avoided. The trial project will need to be committed to avoid natural tendencies and *delay* actual code implementation so that this methodology and any benefits can be fully realized.

It seems apparent from this work and from our engineering discussions that this process can be readily applied to a simple software example. However, its utility and manageability in a more complex application needs to be investigated. It may be that the most significant benefit will be derived from the early application of the process in the conceptual stages, and that detailed application to the software design and code implementation slows project pace and counteracts additional benefits. Only a study of its trial application will reveal lessons such as these for further use of the process.

Complete development of the associated /c/ documentation scheme will also need attention as the approach is applied to a trial project. Details such as standard notes and comments will need to be worked out, as will examples of the various necessary forms for software documentation. A special focus is needed to be sure that all forms of software documentation are intuitively linked to each other through the /c/ approach.

As also mentioned, the potential for failure/reliability allocations to software other than the traditionally used "1" was revealed. This potential needs considerable study to determine its long term possibilities and impacts. This, and other as yet indeterminate approaches, could provide new opportunities in considering whether software is inherently characterizable. Once again, however, the utility of the approach will need to balance the potential benefits. Even with the advent of today's high power information systems, it should nonetheless be possible to begin collecting data for typical fault mechanisms so that characterization could become more determinate.

Summary and Conclusions

The work undertaken in this effort explored the extension of using the existing safety analysis and development methodology for the design and development of stronglink safing devices into the software world. Initial discussions and the resulting approach clearly demonstrate that the principles involved in the hardware process are directly

applicable to a software project. A simplistic software example was explored showing those thoughts and concepts necessary for successful utilization of the new approach.

As the methodology unfolded it also revealed the need for coordinating joint hardware/software discussions to ensure that "unwritten assumptions" about the functionality and design implementation approaches are not carried by either the hardware or software teams. Natural points in the process were identified for this coordination of communication, although more interactive teaming should definitely be encouraged throughout the development process.

Further application of the analysis approach points to new possibilities regarding the quantification of probabilistic determinations for software. Routine application of such analyses to identified critical portions of the software is more plausible than exhaustive application of rigorous analysis and assessments throughout the software. Further exploration can reveal the long range utility and potential for characterization.

The approach explored in this paper illustrates how a software product could be reviewed and analyzed *before the creation of any software code* to identify those portions which are critical in assuring the absence of an undesired high consequence event. In fact, it is this *a priori* approach to analysis that would facilitate *error prevention* rather than detection by methods used after code implementation. This method for analysis can couple easily with other structured approaches to yield software products whose critical functions are isolated in smaller, more manageable software entities or modules. This compartmentalization can result in software modules of sufficiently small complexity that complete exhaustive testing can be possible when necessary. For example, if we were to isolate the portion of a software product which actually sends the radiation dosage to the patient into a small module with only four executable paths and two boundary conditions, then we might expect to be able to test for correctness 100%. Thus, the method provides a basis for making decisions to allocate resources and focus testing and analysis efforts first onto the portions of the software which have the highest judged potential for creating the most adverse conditions. As project schedule and budget permit, more in-depth testing could be applied to the remaining, non-high consequence modules.

The documentation scheme suggested is intended to provide a comprehensive snapshot of both the software product, the design approach implemented, and the verification and validation activities undertaken to demonstrate that the implementation is indeed error free. This documentation provides future project maintenance staff (after the original team members have left) with the basis for the implementation decisions made, a definition of the potential implications of changing the approach, and a clearer understanding of what testing may be required to demonstrate the acceptability of a contemplated change. Instead of a new team member wondering why such a "strange coding approach" had been chosen and "improving" it with a "correction" which results in severe consequences, the new individual can review the rationale and implications to perhaps avoid making a change without full knowledge and verification that surety requirements are still being met.

While preliminary indications are that this approach can provide significant gains in mitigating the occurrence of high consequence events in software, they are only preliminary. Further investigation is needed to determine both the long range utility and potential benefits of the approach. It should be further noted that even with this approach, software suitability of any high-consequence purpose is not guaranteed. However, by having more detailed, up-front information, engineers and designers will have better supporting documentation for the approach selected.

Acknowledgments

The authors wish to acknowledge the special contributions to this work from Scott Nicolaysen, Bill Greenwood, Carl Vanecek, and Doug Gehmlich. These individuals provided valuable feedback regarding the mechanical process applied at the component level and the systems level approach to the safety theme. Additional contributions were received from Ed Fronczak regarding his work in conducting fault tree analysis of software code in security-critical applications. The efforts of Howard Kimberly, Mike Eckley, and Larry Dalton were also invaluable in reviewing and refining the software process as it evolved. Special thanks to Louis Hernandez for his help in creating the process flow diagrams and to Gary Randall in editing this report.

Bibliography

N.G. Leveson, S. S. Cha, T. J. Shimeall, "*Safety Verification of ADA programs using Software Fault Trees*," IEEE Software, Vol. 8, No. 4, pp. 48-59, July 1991.

Sandia Software Guidelines, Vol 1, "Software Quality Planning," SAND85-2344, Sandia National Laboratories, Albuquerque, NM, September 1992.

Sandia Software Guidelines, Vol 3, "Standards, Practices, and Conventions," SAND85-2346, Sandia National Laboratories, Albuquerque, NM, July 1986.

Sandia Software Guidelines, Vol 4, "Configuration Management," SAND85-2347, Sandia National Laboratories, Albuquerque, NM, June 1992.

Sandia Software Guidelines, Vol 5, "Tools, Techniques, and Methodologies" SAND85-2348, Sandia National Laboratories, Albuquerque, NM, September 1992.

"High Consequence System Surety Process Description," SAND-94-3223, Sandia National Laboratories, Albuquerque, NM, September 1995

Ed Fronczak, "A Top-Down Approach for Analyzing Microprocessor Systems", presentation to the Information Surety Affinity Group, Sandia National Laboratories, Albuquerque, NM, November 1994.

Distribution

1	MS0319	Ray Leuenberger, 2645
1	MS0319	Scott Nicolaysen, 2645
1	MS0319	Bill Greenwood, 2645
1	MS0319	Carl Vanecek, 2645
5	MS0319	Gary Randall, 2645
1	MS0329	Ruben Urenda, 2643
1	MS0329	Ken Varga, 2643
1	MS0405	Todd Jones, 12333
1	MS0431	Sam Varnado, 9400
1	MS0451	Sharon Fletcher, 9411
1	MS0458	Bill McCulloch, 12333
1	MS0458	Laura Gilliom,5133
1	MS0484	Roxie Jansma, 9415
1	MS0484	Judy Moore, 9415
1	MS0486	Stan Kawka, 2122
1	MS0487	John Franklin, 2122
1	MS0492	Mark Ekman, 12324
1	MS0507	Kathleen McCaughey, 9700
1	MS0535	Larry Dalton, 2615
1	MS0503	Steve Giles, 2335
1	MS0535	Mike Eckley, 2615
1	MS0535	Laney Kidd, 2615
1	MS0560	Paul Longmire, 2106
1	MS0627	Ed Fronczak, 12334
1	MS0637	Joe Chiu, 12336
1	MS0856	Stu Rogers, 14308
1	MS0638	Mike Blackledge, 12326
1	MS0660	Margaret Olson, 9622
5	MS0535	Pat Tempel, 2615
1	MS0661	Sue Bodily, 4816
1	MS0661	Louis Hernandez, 4816
1	MS0746	Jim Campbell, 6613
1	MS0746	Maria Armendariz, 6613
1	MS0747	Heather Schriner, 6412
1	MS0759	Bill Paulus, 5845
1	MS0759	Mark Snell, 5845
1	MS0762	Sabina Jordan, 5861
1	MS0769	Dennis Miyoshi, 5800
1	MS0801	Melissa Murphy, 4900
5	MS0812	Sharon Trauth, 4923
1	MS0830	Elmer Collins, 12335
1	MS0830	Tom Kerschen, 12335
1	MS0833	Johny Biffle, 9103
1	MS0977	Dave Darsey, 9416
1	MS1006	Bill Drotning, 96711
1	MS1007	Howard Kimberly, 9672
1	MS9036	Doug Gehmlich, 2254
1	MS9214	Len Napolitano, 8117
2	MS0100	Document Processing, 7613-2 For DOE/OSTI
1	MS0619	Print Media, 12615
5	MS0899	Technical Library. 4414
1	MS9018	Central Technical Files, 8523-2