

UC Irvine

ICS Technical Reports

Title

A safety-critical software design and verification technique

Permalink

<https://escholarship.org/uc/item/2cr141mp>

Author

Cha, Stephen Sungdeok

Publication Date

1991

Peer reviewed

699
C3
no. 91-62

**A SAFETY-CRITICAL SOFTWARE DESIGN AND
VERIFICATION TECHNIQUE**

Dissertation

Stephen Sungdeok Cha

Department of Information and Computer Science

University of California, Irvine

Irvine, California 92717

Technical Report No. 91-62

**Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)**

UNIVERSITY OF CALIFORNIA

Irvine

A Safety-Critical Software Design and Verification Technique

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Information and Computer Science

by

Stephen Sungdeok Cha

Committee in charge:

Professor Nancy G. Leveson, Chair

Professor K.H. Kim

Professor Debra J. Richardson

1991

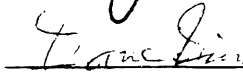
©1991

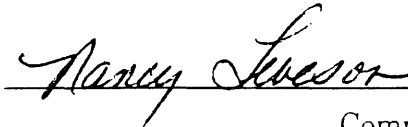
STEPHEN SUNGDEOK CHA

ALL RIGHTS RESERVED

The dissertation of Stephen Sungdeok Cha is approved,
and is acceptable in quality and form for
publication on microfilm:







Committee Chair

University of California, Irvine

1991

Dedication

To my parents, Mr. In Jun Cha and Mrs. Sung Sook Kim, and
to my wife, Yoondeok.

Contents

List of Figures	v
List of Tables	vi
Acknowledgements	vii
Abstract	ix
Chapter 1 Approaches to Software Safety	1
1.1 Introduction	1
1.2 Dissertation Overview	4
Chapter 2 Survey and Evaluation of Previous Research	7
2.1 Software Design Methodology	7
2.2 Security Design Techniques	10
2.3 Software Safety Verification Techniques	14
Chapter 3 A Safety-Oriented Design Method	21
3.1 Introduction	21
3.2 High-Level Design Analysis	24
3.3 Detailed Design Safety Verification	35
3.4 Some Safe Design Techniques	46
Chapter 4 A Safety-Oriented Management Structure	52
4.1 Software Safety Management	52
4.2 System Safety Management	57
Chapter 5 A TCAS Example	60
5.1 Introduction	60
5.2 TCAS Design Description	63
5.3 TCAS Safety Analysis	67
5.4 An Improved TCAS Design	77
Chapter 6 Conclusions and Future Work	81
6.1 Conclusions	81
6.2 Future Work	82

List of Figures

2.1	Software Safety Techniques	14
2.2	Run-Time Safety Environment	18
2.3	Roles of Safety Executive	20
3.1	Safety-Oriented Method Overview	22
3.2	A Brute Force Algorithm to Detect Safety-Critical Items	25
3.3	<i>AnalyzeANode</i> (in <i>n</i> : <i>node</i>)	26
3.4	An Enhanced Algorithm to Detect Safety-Critical Items	28
3.5	<i>AssignNodeLevels</i> (out <i>MaxLevel</i> : <i>integer</i>)	29
3.6	<i>AnalyzeANode</i> (in <i>n</i> : <i>node</i> ; out <i>MustBackUp</i> : <i>boolean</i>)	30
3.7	<i>DistributeWP</i> (in <i>n</i> : <i>node</i> ; in <i>WP</i> : <i>boolean</i> ; out <i>Successful</i> : <i>boolean</i>)	31
3.8	<i>Undo</i> (in <i>n</i> : <i>node</i> ; out <i>Resolved</i> : <i>boolean</i>)	32
3.9	Essential Firewall Requirements	34
3.10	Why Concurrency Safety Analysis is Needed	36
3.11	Impacted Concurrency States and the Rendezvous	43
4.1	Safety Management Hierarchy Recommended in the MoD-Std-0055	53
4.2	Slight Variation to Mod-StD-0055 Management Hierarchy	55
4.3	Safety-Critical System Development Management Hierarchy	57
5.1	CAS Logic Functions	61
5.2	Procedures <i>Threat-Detection</i> and <i>Setup-Parameters</i>	64
5.3	Procedures <i>Hit-Or-Miss-Test</i> and <i>Hit-Test-Init</i>	65
5.4	Procedure <i>Compute-Tau</i>	66
5.5	Procedure <i>Compute-VMD-HMD</i>	67
5.6	Procedure <i>Run-Range-Altitude-Test</i>	68
5.7	Procedure <i>Log-Threat-Info</i>	68
5.8	Structure of <i>Threat-Detection</i> Procedure	69
5.9	Control Flow of <i>Threat-Detection</i> Procedure	70
5.10	Control Flow of <i>Range-Test</i> Procedure	72
5.11	Propagation of WP from <i>Range-Test</i> Task	73
5.12	Propagation of WP from <i>Altitude-Test</i> Task	75
5.13	Firewall Installation	78
5.14	Safety-Critical Module Augmentation with Run-Time Assertions	79
5.15	Main Procedure and Safety-Independent Module	80

List of Tables

1.1	Causes of Tandem System Outages from 1985 to 1989	3
5.1	Derivation of Run-Time Assertions	76

Acknowledgements

Without the guidance, encouragement, and patience of my advisor, Professor Nancy Leveson, this dissertation could not have been completed.

I am also grateful to the committee members Professor K.H. Kim and Professor Debra Richardson. Joan Isenbarger was the proofreader of my first and the last publications as a graduate student. Her help is deeply appreciated. Reuven Greenberg suggested the safety constraints from which I developed a TCAS example. His help is appreciated.

Curriculum Vitae

February 18, 1960	Born CheongPyong, South Korea
October 30, 1980	Immigrated to the United States
1983	B.S. in Information and Computer Science, University of California, Irvine, California (Cum Laude)
1986	M.S. in Information and Computer Science, University of California, Irvine, California
1991	Ph.D. in Information and Computer Science, University of California, Irvine, California Dissertation "A Safety-Critical Software Design and Verification Technique"

Publications

N.G. Leveson, S.S. Cha, and T.J. Shimeall. Safety Verification of Ada Programs using Software Fault Trees. *IEEE Software*, July 1991.

N.G. Leveson, S.S. Cha, T.J. Shimeall, and J.C. Knight. The Use of Self-Checks and Voting in Software Error Detection: An Empirical Study. *IEEE Transaction on Software Engineering*, SE-16(4), pp 432-443, April 1990.

S.S. Cha, N.G. Leveson, and T.J. Shimeall. Safety Verification in MURPHY using Fault Tree Analysis. In *Proceedings of the 10th International Conference on Software Engineering*, pages 377-386, Raffle City, Singapore, April 1988. Reprinted in *Tutorial: Software Risk Management*, B. Boehm, editor, 1989.

S.D. Cha, J.C. Knight, N.G. Leveson, and T.J. Shimeall. An Empirical Study of Software Error Detection using Self-Checks. In *Digest of Papers: The Seventeenth International Symposium on Fault Tolerant Computing*, pages 156-161, Pittsburgh, PA, July 1987.

S.S. Cha. Dynamic Load Balancing Algorithm Complexity. Technical Report 87-24, Department of Information and Computer Science, University of California, Irvine, 1987.

S.S. Cha. A Recovery Block Model and Its Analysis. In W. Quirk, editor, *SafeComp '86*, pages 21-26. IFAC, Pergamon Press, Sarlat, France, October 1986.

Abstract of the Dissertation

A Safety-Critical Software Design and Verification Technique

by

Stephen Sungdeok Cha

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1991

Professor Nancy G. Leveson, Chair

Safe software can be developed by applying a safety-oriented design method and establishing good safety management procedures. However, safety-oriented design has not received much research attention in the past.

This dissertation proposes a software design method whose goal is to minimize the amount of safety-critical code and to produce a design whose safety can be verified. Starting from the software safety requirements, backward analysis is used to identify the safety-critical modules and derive their safety constraints. Safety constraints play an important role since they become the criteria against which the safety of detailed design is verified. This dissertation also proposes the use of information hiding principles to implement a “firewall.” The firewall protects the safety-critical modules from the safety-independent modules, thereby minimizing the amount of safety verification effort required in formally certifying the design safety. The complexity of design safety verification is further reduced by employing an incremental and selective verification. This dissertation argues that concurrency decisions on safety-critical software must be based on careful trade-off analysis and demonstrates that concurrent designs do not necessarily require exhaustive concurrency safety verification. An application of the proposed safety-oriented design method is demonstrated using a subsystem of TCAS II (Traffic Alert and Collision Avoidance System).

Management aspects of software safety are important because of the direct and significant impact management has on safety. This dissertation examines how to organize safety-critical projects and distribute safety responsibilities.

Chapter 1

Approaches to Software Safety

1.1 Introduction

Software safety became a critical concern in the 1980s because of the increased use of software to control safety-critical systems. A system is considered safety-critical if system behavior can result in death, injury, property loss, or environmental damage[27]. Twenty years ago, software failures merely annoyed users who had to perform manual recovery actions such as reissuing checks with the correct amounts. Now the failure of safety-critical software can have catastrophic consequences when effective recovery activities do not exist. For example, a software error on the Therac 25 therapy machine was responsible for the death of four patients[19, 24]. The list of computer-related accidents, reported to the ACM RISKS forum and selectively published in the ACM Software Engineering Notes[41], is long and growing. Some safety-critical computer systems have been the subject of public controversy.

In the Airbus A320[37], a commercial fly-by-wire aircraft, almost all of the direct mechanical links from the cockpit to the aircraft parts have been computerized¹; pilot commands are interpreted by the flight computers that send commands to the motors

¹The only surfaces retaining mechanical control of the hydraulics are the rudder and pitch rim.

to move the appropriate parts. The A320s have been in service for about three years now, and software has not been proven to be the direct cause of the two crashes that have occurred to date. Yet, the safety of its control software and the use of the N-version programming technique[26, 48] as a means of developing safe software continues to be debated². Some computer scientists are concerned because:

- The embedded software is designed to override any of the pilot's commands that the software determines to be dangerous. They have questioned the wisdom of such design decisions and have argued that the pilots should be given the ultimate authority especially in emergency situations.
- The results of some controlled experiments suggest that the N-version programming technique may not be very effective due to correlated failures[25, 29]. A theoretical model of N-version programming showed that a small degree of correlated failures can significantly reduce the degree of reliability improvement[7].

Other safety-critical systems include air traffic control, aircraft collision avoidance (TCAS), nuclear power plant shutdown, and patient monitoring. The safety of software controlling these systems has significant and direct impact on the lives of the general public.

Recently, Forester and Morrison[9] proposed an international ban on the use of computers in controlling safety-critical systems, expressing concern that current software engineering technology is not mature enough to put human lives at risk. Their concerns are valid, and the use of software in controlling safety-critical systems must be carefully analyzed against the feasible alternatives such as using electro-mechanical controls or depending on manual controls. However, the increasing and

²Extensive debates on the safety of the A320 aircraft and its control software have appeared in the ACM Risks Forum since May 1988.

Causes	1985	1987	1989
Software	33.5%	38.8%	62.1%
Hardware	28.8%	22.4%	6.6%
Maintenance	18.6%	12.6%	15.1%
Operation	8.8%	11.9%	15.1%
Environment	6.0%	9.5%	5.9%
Others	4.2%	4.8%	5.3%

Table 1.1: Causes of Tandem System Outages from 1985 to 1989

irreversible trend of controlling safety-critical systems via software shows that most people think the advantages outweigh the potential hazards.

Most safety-critical systems are embedded systems, where software is only a part of a larger system, which can be very large, complicated, and expensive to develop. The entire system could consist of a collection of distributed hardware, software, and human operators. Therefore, the system may fail due to permanent or transient hardware failures, software failures, hardware-induced software failures[17], operator mistakes, or environmental events.

Hardware reliability has improved impressively in the last few years; hardware failures are no longer a major cause of system failures. This trend is especially true with the availability of commercial fault-tolerant computers. Software improvements, however, have been less impressive. Based on the reported causes for Tandem system outages from 1985 to 1989 as shown in Table 1.1, Gray[11] recently argued that software has become the major bottleneck in further improving system reliability.

The development of perfectly safe software is impossible and unnecessary. A realistic goal is to develop software that is free of hazardous behavior. Development of such software, however, remains a challenging task because:

- Most of the safety-critical systems are real-time systems where the sequencing and timing of input events are determined by the real-world (environment) and not by the program. Furthermore, a real-time system must meet its deadlines. Timeliness of real-time system outputs is as important as their correctness.
- The demands on the system may occur in parallel rather than in sequence. The system must react correctly to multiple events within the limit of its specified load and capacity. For example, a traffic light controller at an intersection must sense pedestrians as well as cars approaching the intersection from all four directions. All these events may occur simultaneously or in a very short interval.
- The competitive and profit-oriented industrial environment always forces companies to develop safety-critical software with a minimal allocation of resources. The constraints are usually further compounded by the pressure of meeting deadlines.

1.2 Dissertation Overview

Safe software can be developed by applying a safety-oriented design method and establishing good safety management procedures. However, safety-oriented design has not received much research attention. The United Kingdom software safety standard[38], for example, provides little help beyond the following:

The design shall minimize the extent of safety critical software. The safety critical software shall be isolated from other equipment functions. The detailed design shall avoid common mode failures with hazardous consequences. Fault tolerance, defensive programming, graceful degradation, and fail-safe design techniques shall be used when possible. The design shall be hierarchical and modular with well-defined interfaces between modules.

In essence, the standard specifies general principles for developing safety-critical software design without providing techniques about how these principles might be accomplished.

This dissertation proposes a safety-oriented design method that consists of the following:

- An analysis procedure guiding the high-level design phase where the safety-critical modules are identified and their safety constraints, a set of conditions whose truth are necessary to ensure safety, are derived.
- A selective and incremental safety verification technique where the safety of the detailed design can be certified before coding begins.
- Various design techniques to enhance safety including reducing the probability of timing-related errors and using programming language constructs designed to deal with abnormal data or control errors at run-time.
- A safety-oriented software management hierarchy.

Chapter 2 reviews the previous research on software design techniques, security design techniques, and software safety verification techniques. Chapter 3 extends the work of Leveson[27] and proposes a safety-oriented software design method. The method consists of a design hazard analysis and design safety verification technique. Chapter 4 proposes a safety-oriented software project management hierarchy. Chapter

5 demonstrates an application of the design method and the verification technique on the official design of TCAS II (Traffic Alert and Collision Avoidance System). Chapter 6 draws conclusions and discusses directions for future research.

Chapter 2

Survey and Evaluation of Previous Research

This chapter surveys previous research on software design techniques, security design techniques, and software safety verification techniques. A thorough survey of software design techniques or security design techniques is impractical since the subjects are extremely broad. Therefore, this chapter briefly surveys and evaluates previous research from a software safety viewpoint. Finally, software safety verification techniques are reviewed.

2.1 Software Design Methodology

Following the recognition of a software crisis in the late 1960s, intensive research on software design techniques gave birth to many design methodologies. However, no software design technique classification schemes are widely accepted in the literature. Freeman and Wasserman[10], in their popular IEEE tutorial, classify software design techniques as either process-oriented or data-oriented. Yau and Tsai[56] adopt essentially the same classification although the design techniques listed for each group

are different. Pressman[46], on the other hand, further classifies the process-oriented design techniques as either data flow-oriented or data structure-oriented design techniques (e.g., Jackson system design). The process-oriented design techniques include, but are not limited to, functional decomposition, structured design, structured analysis and design technique (SADT), Jackson design methodology, Hierarchy-Input-Process-Output (HIPO). Examples of data-oriented techniques are object-oriented and conceptual database design techniques.

Experts do not always agree on such terms as design techniques (or methodologies), design principles, and design notations. For example, Yau and Tsai list modular programming as a design technique while Fairley[8] classifies modularity, used in the same context, as one of the fundamental design principles. Neither the classification of software design techniques nor the definition of sometimes blurry and abstract terms is the subject of this research. Therefore, this dissertation arbitrarily adopts the scheme used by Pressman and evaluates each approach for its applicability to safety-critical software design.

Data flow-oriented design techniques, the most general and perhaps the most widely used approaches, are certainly applicable to safety-critical software design. The resulting design, however, exhibits different characteristics depending on the selected decomposition criteria. Various module decomposition criteria have been proposed in the literature. Stepwise refinement[54] can be adapted as a decomposition criterion, and each of the major processing steps can be implemented as a module. Principles such as information hiding[42] can guide the decomposition process. The “uses hierarchy,” proposed by Parnas[44], is another decomposition criterion where software maintainability (e.g., the ease of extension or contraction) is emphasized.

While these decomposition criteria promote the properties any good software design must possess, they do not specifically address safety issues.

The applicability of data structure-oriented design techniques[18], proven effective and widely used in business (or data-processing) applications, appear to be severely limited (if applicable at all) when it comes to safety-critical software designs. Harel and Pnueli[14] used the term “transformational systems” to capture the essential characteristics of such applications where the primary function of software is almost always to process the stream of input data (e.g., transaction records) and to produce the outputs in the required format (e.g., transaction summary). Much of software complexity in such applications stems from the need to manipulate complicated input and output data structures. The algorithms to determine the values of the outputs themselves tend to be relatively straightforward. Therefore, it makes sense to base the design of such software on input and output data structures. However, safety-critical software is primarily found in embedded systems whose primary function is process control. Much of the complexity in developing process control software lies in determining how to properly control the environment. Inputs to software, usually denoting the occurrence of environmental events, often arrive from the sensors in the form of interrupts with relatively simple (if any) data structures (e.g., numeric values or escape sequences). Similarly, software outputs, often generated to the actuators, rarely require the manipulation of complicated data structures.

Object-oriented design[3] is another technique one can use to design safety-critical software. Object-oriented design proceeds by initially identifying the objects that the software must manipulate. The data structure of the objects as well as the set of operations to be performed on the objects are subsequently identified, and the operations to be performed on an object are most likely to be grouped into a module

(e.g., a package). Although this dissertation does not specifically address safety-critical software development using object-oriented design techniques, the analysis procedure presented in the dissertation can be applied to object-oriented design.

2.2 Security Design Techniques

Since society has become more dependent on computers for information processing and storage, computer security has been an active research topic. An organization must protect private and proprietary information from unauthorized access. Computer security is a critical concern not only to the financial success of a corporation but also to national security. Although there is no single strategy for achieving security, previous research has focused on cryptography and access control techniques.

Denning[5] defines cryptography as the science and study of secret writing. Privacy is preserved by transforming a plaintext M into a ciphertext C using the enciphering key E_K before storing or transmitting information. The authorized user, who has a cryptographic key D_K , performs a decipherment or decryption operation to access the plaintext. A cryptography technique must satisfy the following security requirements to be useful[5]:

- It should be computationally infeasible for a cryptanalyst to systematically determine the plaintext M from intercepted ciphertext C .
 - It should be computationally infeasible for a cryptanalyst to systematically determine the deciphering transformation from intercepted ciphertext C , even if the corresponding plaintext M is known.
-

An example of an encryption method is the Data Encryption Standard (DES), specified by the National Bureau of Standards[36] for use on unclassified U.S. Government applications. Cryptography techniques by themselves, however, have no direct relation to software safety, and further discussion is omitted.

The fundamental requirements of access control and multi-level security techniques are:

- Never allow access to or operation on information by the users without proper authorization.
- Never deny an access to or operation on information by authorized users.

Access control mechanisms can be either discretionary or mandatory depending on who controls the access policies (e.g., user versus system). Discretionary access control mechanisms proposed in the literature are passwords, capability, and access control lists.

When access control is based on a password, the user assigns an individual password to each object (i.e., file) that must be protected. The technique quickly becomes impractical in a large organization.

When access control is based on capability, each user is assigned a capability list that specifies the set of objects the user is authorized to access as well as the mode of access for each object. However, the management of capability-based access control may be inefficient. For example, when a file is deleted, the capability list of all the users must be searched and updated. Otherwise, a user may acquire an unauthorized access when another file is created with the same name.

With an access control list that specifies the authorized users and their access mode on each object, managing user access to an object is simple. However, access to the objects may be inefficient because the associated access control list must be scanned whenever an access is requested. Grouping users (e.g., by the projects), assigning suitable default access modes (e.g., allowing reads but prohibiting writes within the group), and using efficient searching technique (e.g., hashing) can allow efficient and secure access.

Multi-level security techniques were developed for the Department of Defense to protect classified military information stored in computers. All information is assigned a *classification* level, and each user is assigned a *clearance* level. A classification or clearance level consists of:

- Sensitivity Level: Unclassified, Confidential, Secret, or Top Secret.
- Category: A list of subjects or keywords on the contents of information (e.g., NATO and nuclear).

A user can access the object if and only if the user's clearance level 'dominates' the object's classification level. For example, a user with top secret clearance can read confidential documents while a user with an unclassified clearance cannot read secret documents. However, writes are prohibited to prevent the transfer of top secret information to an unclassified document. The protection of resources using access control mechanisms is an essential property of a secure system. Without such mechanisms, a system becomes useless (and even harmful) since no guarantee can be made on system security. Furthermore, secure systems must preserve security despite dynamic changes (e.g., classification and clearance levels) and authorization transfers among users.

Similarly, safety-critical software development needs to separate the safety-critical modules from the safety-independent modules. Fortunately, the access control issues for safety-critical software are simpler than the ones for secure systems. Access control in safety-critical software is a desirable (rather than a mandatory) property. The identification and proper protection of safety-critical modules simplifies the effort required for safety certification, but they do not necessarily make the system safer. Furthermore, the safety attribute of a module is static and does not change at run-time, and there is no need to deal with the transfer of safety attributes from one module to another. Therefore, adequate access control is provided as long as the safety-critical variables are never shared between the safety-critical and the safety-independent modules. This separation guarantees that the safety-critical behavior of software is completely determined by the semantic definitions of the safety-critical modules.

The security kernel, a small component of the system whose correctness is sufficient to verify the security of the entire system, is the most widely used technique for building secure systems. The kernel can encapsulate the safety-critical modules as suggested by Rushby[50]. He showed that the kernel can enforce “negative properties” (i.e., absence of commission faults) but not “positive properties” (i.e., absence of omission faults) of the system. The positive properties cannot be enforced because “no matter what ‘good’ properties a kernel may possess, there can be no guarantee that the rest of the system will use the kernel correctly.” Safety, however, can be enforced as long as the safety-critical resources are under the total control of the safety kernel.

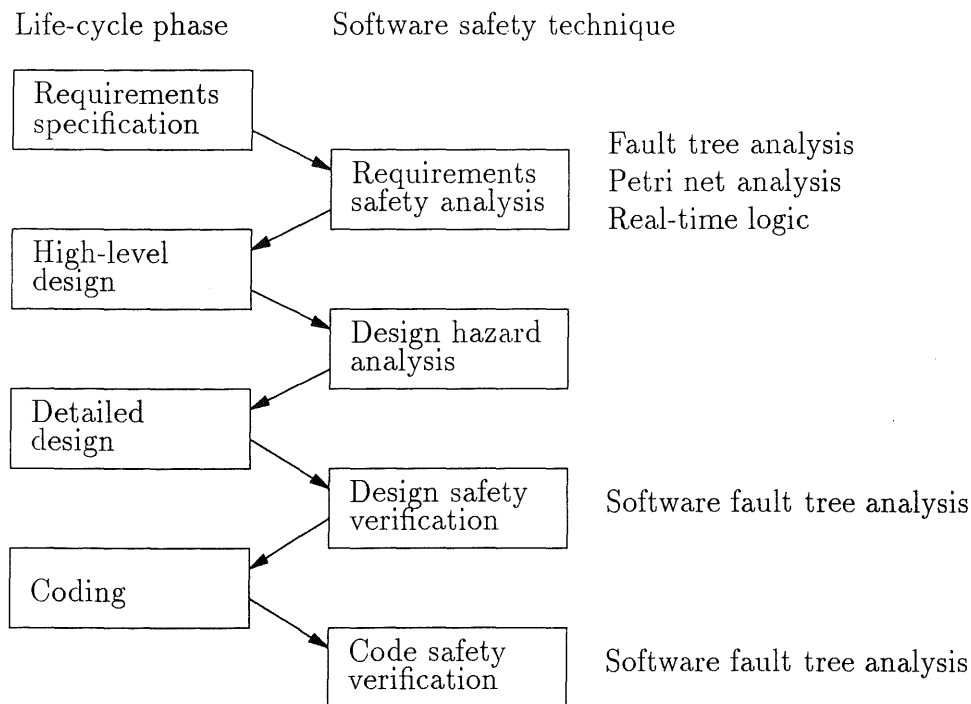


Figure 2.1: Software Safety Techniques

2.3 Software Safety Verification Techniques

Software safety has a relatively short history as a research topic. The most comprehensive and authoritative survey on software safety to date is provided by Leveson[27]. The paper discusses what software safety issues are, why software safety is an important topic, and how software safety might be achieved.

The importance of applying systematic and rigorous safety verifications at the end of each development phase has been stressed by Leveson[28], and several safety verification techniques have been proposed in the literature (Figure 2.1). While some techniques, such as software fault tree analysis, are general enough to be applicable

throughout software development, the applicability of others are limited to specific phases.

Leveson and Stolzy[34] demonstrate how software safety requirements can be derived through the modeling of the system operation using timed Petri Nets[45] with an extended notation to allow the modeling of faults and failures. The goal of software safety is to fulfill the part of system safety requirements allocated to software. In other words, software safety is essentially a system property, and any discussion of software safety outside the context of system safety would be meaningless. Reachability graph analysis derived from the Petri Net model of the system reveals the potential failure modes, and software safety requirements can be derived to prevent their occurrence, either by imposing timing constraints on the events or by requiring software to control the occurrence of event sequences. They developed the concept of “critical states” as a means of avoiding exhaustive generation of the reachability graphs without sacrificing the expressive or analytical power of the model.

Fault tree analysis (FTA) is another technique that can be used to derive software safety requirements. It was developed in the 1960s for the safety analysis of the Minuteman missile system and has become one of the most widely used system safety techniques. The fault tree handbook[53] explains the technique as follows:

Fault tree analysis can be simply described as an analytical technique, whereby an undesired state of the system is specified (usually a state that is critical from a safety standpoint), and the system is then analyzed in the context of its environment and operation to find all credible ways in which the undesired event can occur. The fault tree itself is a graphic model of the various parallel and sequential combinations of faults (or system states) that will result in the occurrence of the predefined undesired event. The faults can be events that are associated with component hardware failures, human errors, or any other pertinent events which can lead to the undesired event. A fault tree thus depicts the logical relationships of

basic events that lead to the undesired event – which is the top event of the fault tree.

It is important to understand that a fault tree is not a model of all possible failures or all possible causes for system failure. A fault tree is tailored to its top event which corresponds to some particular system failure mode, and the fault tree thus includes only those faults that contribute to this top event. Moreover, these faults are not exhaustive – they cover only the most credible faults as assessed by the analyst.

Software safety requirements can be derived by expanding the system fault trees to the software interface levels and by identifying the failure modes that the software may cause or to which it may contribute.

Software functional and safety requirements must be analyzed before the design activity begins. Jaffe and Leveson[20] developed a definition of the logical completeness of the requirements. This work was further expanded by Jaffe, Leveson, Heimdahl, and Melhart[21] to include a set of criteria that detect flaws in the black-box specification of real-time software. The criteria are specified using a general behavioral specification model called a requirements state machine and is applicable to any specification languages based on state machines (e.g., statecharts[12, 13]).

Jahanian and Mok[22] developed a procedure to perform timing-related safety analysis on software requirements using a formal logic called Real-Time Logic (RTL). The requirements are represented in the event-action model and later mechanically transformed into a set of RTL formulas and subsequently into the equivalent formulas in Presburger arithmetic with uninterpreted functions. This technique, as the authors point out, is applicable mainly to the software requirements and, in particular, analysis of the timing-related behaviors.

The most commonly used software safety technique applied to the code level is software fault tree analysis (SFTA). The technique was adapted from system fault tree

analysis by Leveson and Harvey[30] and also in parallel by Taylor[51]. Leveson and Harvey successfully applied the technique to sequential software controlling the operations of a scientific satellite. They report on the detection of an error that remained undetected despite the extensive functional testing previously performed on the software. It was a serious error that could have caused the destruction of the satellite. Subsequent research extended the SFTA technique to more complex languages with features such as concurrency and exception handling[4, 33]. The analysis is guided by the use of statement templates that describe the failure modes of each statement. This approach is the same as the one used in formal axiomatic verification[6, 16] where the weakest preconditions are derived that are necessary to satisfy the given postconditions. In fact, SFTA can be seen as a graphical application of formal axiomatic verification where the postconditions describe the hazardous conditions rather than the correctness conditions.

The informal nature of SFTA allows the results of other analysis techniques (e.g., timing analysis using Petri Nets) to be incorporated into the fault tree and allows the entire system, including hardware, software, and operators, to be systematically analyzed[39]. Unfortunately, the informal nature of the technique is also its major weakness because the success of the technique heavily depends on the ability of the analysts. SFTA is essentially a structured walk-through technique with special emphasis on safety issues rather than correctness ones. Therefore, the technique is applicable in virtually all phases of the software life cycle.

The SFTA technique has been successfully applied on several, mostly classified, military projects including F18-E and F16 control software. It was also used in verifying the safety of Canadian nuclear power plant shutdown software that consisted of about 6,000 lines of Pascal and Fortran code. The overhead was moderate in that

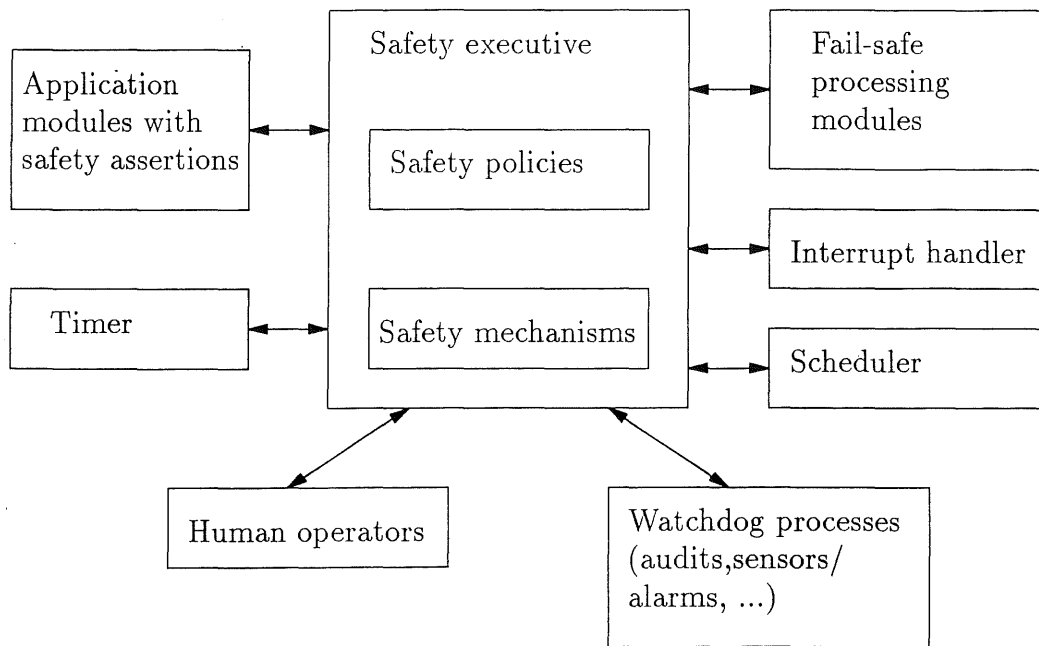


Figure 2.2: Run-Time Safety Environment

the analysis took approximately two man-months including training. Although no errors were detected using SFTA, the technique was useful in making the software more robust against errors and in developing the contents of run-time assertions to detect potentially hazardous internal software states. According to Leveson, who helped engineers to perform SFTA, the engineers found the technique to be effective and easy to use, and they are using it on other safety-critical software projects.

Leveson, Shimeall, Stolzy, and Thomas[32] proposed a general run-time structure, called a safety executive (Figure 2.2), where the hazardous run-time system states are detected by evaluating safety assertions[31] inserted in the code. The causes of hazardous run-time system states include:

- Erroneous safety requirements due to an incorrect hazard analysis. This includes erroneous assumptions made about the environment.
- Software design errors not detected by validation and verification techniques.
- Environmental failures, operator errors, or hardware failures that affect software even if software correctly implements the requirements.

Upon the detection of the occurrence of hazardous system states, the safety executive initiates appropriate recovery procedures such as a reset, shut-down, or fail-safe processing.

It is possible to design the application to include features that monitor the system states and initiate recovery routines whenever necessary, but the use of general run-time safety environments such as safety executives[32] is preferred (Figure 2.3). Although the separation of the safety responsibilities may not necessarily reduce the degree of the correlated failures between the application and the safety executives, it reduces the complexity of the application and enhances the reusability of the safety executive software.

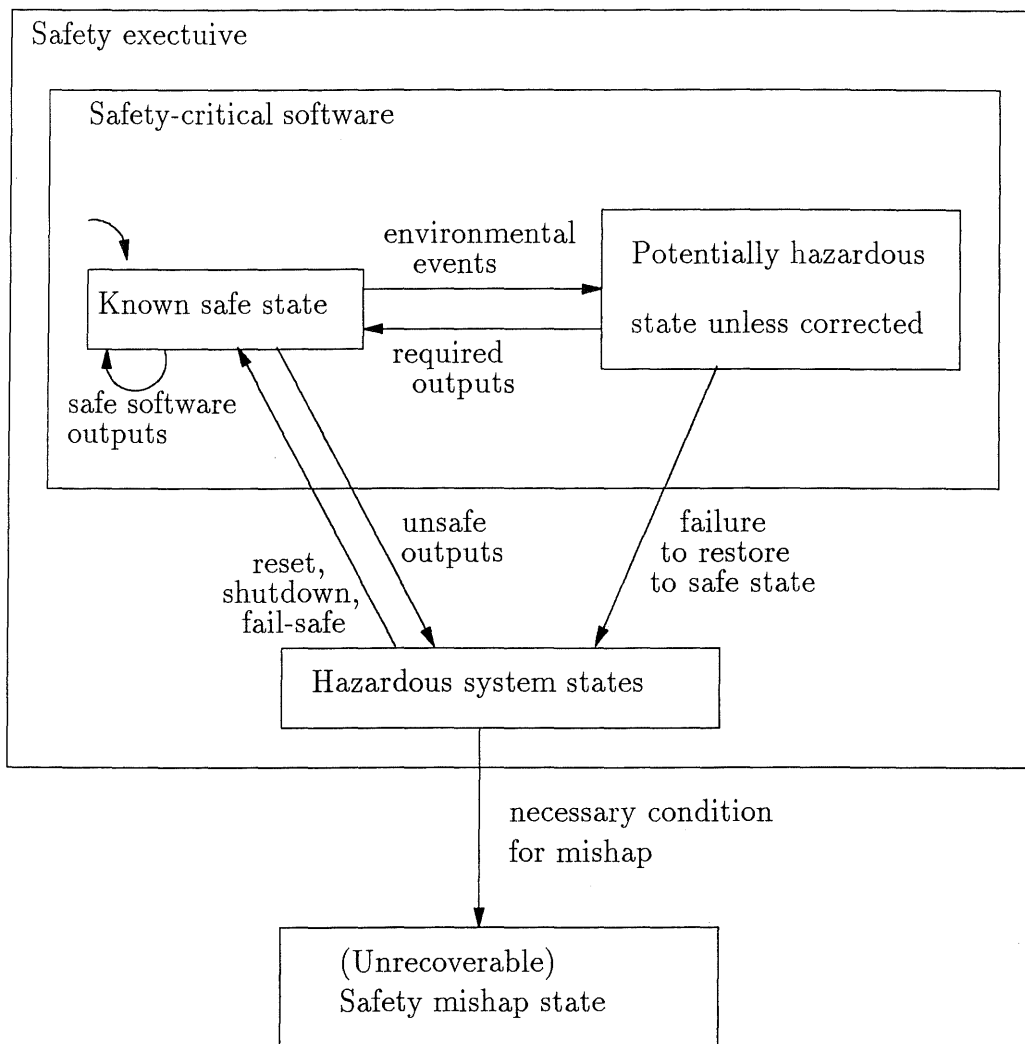


Figure 2.3: Roles of Safety Executive

Chapter 3

A Safety-Oriented Design Method

3.1 Introduction

The goal of a safety-oriented design method, such as that shown in Figure 3.1, is to minimize the amount of safety-critical code and to produce a design whose safety can be certified. Safety constraints, i.e., the set of conditions whose truths are necessary to ensure safety, play an important role in designing safety-critical software. The safety constraints of a module¹ describe the desired postconditions², and backward analysis can be applied to determine the necessary weakest preconditions. A module can be called “inherently safe” or “safety-independent” if the safety constraints are TRUE. If, on the other hand, the design contains a module whose safety constraints are FALSE, the design must be revised since the module is “inherently unsafe.” All other modules are called “safety-critical,” and the algorithmic design of the module must ensure that the safety constraints and the functional requirements are always satisfied.

¹The term ‘module’ is used to refer to the entire software or any of its component – either a procedure or a function.

²If the module consists of non-terminating cyclic routines, this refers to the end of a processing cycle.

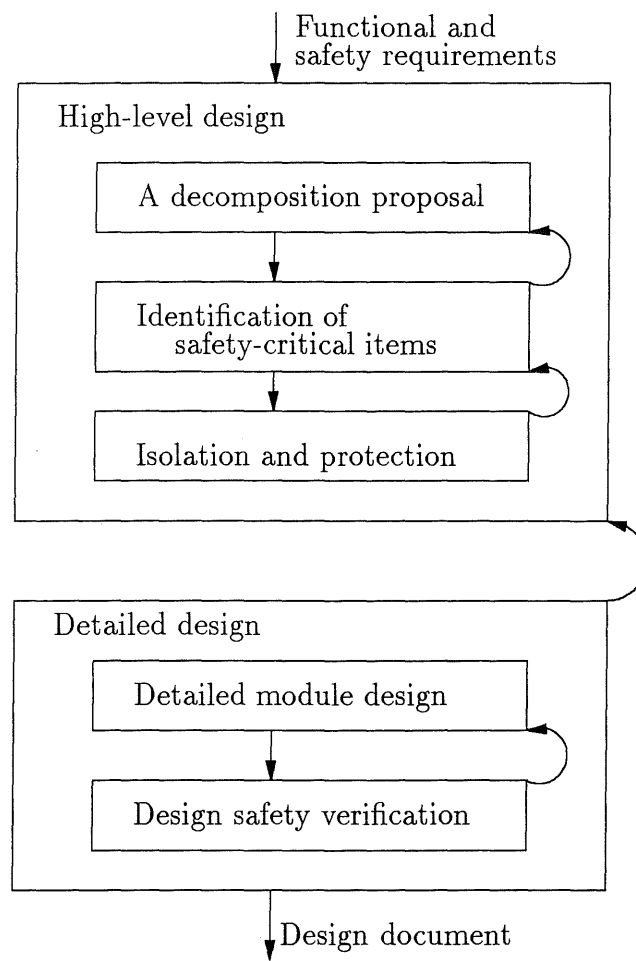


Figure 3.1: Safety-Oriented Method Overview

Software learns about the occurrence of environmental events to which it must react only via externally visible inputs. Similarly, software controls the environment only by generating externally visible outputs. Therefore, the initial safety constraints must be formulated as a predicate involving only the externally visible inputs, outputs, and time (if applicable); the initial safety constraints are equivalent to either the software safety requirements or their negation depending on how the requirements are formulated. As the modules are decomposed successively, the safety constraints for each module need to be refined. Safety constraints that are not expressed in terms of inputs, outputs, and time have been erroneously formulated.

The identification of the safety-critical modules is not enough. It is necessary to isolate and protect the safety-critical modules from the ones that are not. It is also desirable to minimize the number of safety-critical modules. When the detailed design is complete, the safety constraints become the criteria against which the design safety can be formally verified. The complexity of design safety verification can be reduced if the safety-critical modules are identified and protected from the other modules.

Section 3.2 describes a design hazard analysis technique that identifies the safety-critical modules and derives their safety constraints. It presents a brute-force analysis technique, suggests enhancements to resolve safety constraint conflicts, and discusses essential requirements of a “firewall” to protect the safety-critical modules from the rest. Section 3.3 discusses how to verify the safety of the detailed design efficiently by using an incremental and selective verification technique. Section 3.4 proposes some safe design techniques that enhance safety by reducing the possibility of timing-related errors and dealing with abnormal data and control errors.

3.2 High-Level Design Analysis

High-level design may be conceived as consisting of successive refinement into a set of modules and the interactions among them. The decomposition may also introduce a set of internal variables (i.e., data flows). The module being decomposed can be completely characterized by its functional requirements $F_M(I_M, O_M, t)$ and the safety constraints $C_M(I_M, O_M, t)$ where I_M, O_M , and t denote the module's inputs and outputs, and time (if applicable), respectively. After allocating the functional requirements, it is necessary to identify the safety-critical modules and to derive their safety constraints.

The decomposition can be formally represented as a directed graph where nodes denote the functions that the modules compute and edges denote the data-dependency among the modules. When a module M is decomposed, the process of identifying the safety-critical modules and their safety constraints starts with the safety constraints of the module C_M . This section proposes an analysis technique using the following notation:

$F(n)$: Function being computed at the node n .

$source(e), dest(e)$: Source and destination node of the edge e , respectively.

$c(e)$: Safety constraints associated with the edge e . If a node n has outgoing edges e_1, \dots, e_i , the module's safety constraint is defined as $c(e_1) \wedge \dots \wedge c(e_i)$.

3.2.1 Safety-Critical Module Identification

```

 $\forall e, c(e) := \text{TRUE};$  - - initially assumed to be safety-independent
 $\forall C_i$  loop - - for each constraint subcondition
    OutEdges := { edges with direct impact on  $C_i$  but without dest node };
     $\forall e \ni \text{OutEdges}, c(e) := c(e) \wedge C_i;$  - - initialization
    NodesToProcess := {  $\forall e \ni \text{OutEdges}, \forall n \ni n = \text{source}(e)$  };
    while (NodesToProcess  $\neq$  [ ]) loop
         $n := \{ n \ni \text{NodesToProcess} \};$  - - select a node to analyze
        NodesToProcess := NodesToProcess - [n];
        AnalyzeANode(n);
    end loop;
end loop;

```

Figure 3.2: A Brute Force Algorithm to Detect Safety-Critical Items

Figure 3.2 shows how a brute-force backward analysis can be applied for each component of the safety constraints. First, the output modules that have direct impact on the satisfiability of the safety constraints are identified (i.e., *NodesToProcess*). Then, backward analysis is applied to one node at a time until there are no more nodes to process.

The *AnalyzeANode* procedure, shown in Figure 3.3, starts with the derivation of the weakest preconditions necessary to satisfy the safety constraints. A module is safety-independent if the safety constraints (i.e., post condition) and the weakest precondition are the same. If, on the other hand, the weakest precondition is either **TRUE** or **FALSE**, the analysis need not be propagated to other nodes. Otherwise, each component of the weakest precondition must be made true either by propagating it to another module as its safety constraint or by enforcing it within the module using run-time assertions. If the variables that constitute each component of the weakest precondition in conjunctive normal form are passed from another module via data

```

 $\forall j(1 \leq j \leq i) \ni source(e_j) = n, C_n := c(e_1) \wedge \dots \wedge c(e_i);$ 
WP( $n$ ) := weakest precondition, in CNF, to satisfy  $C_n$ ;
if WP( $n$ ) = FALSE then
  HALT; - - inherently unsafe, revise design, and repeat analysis
elsif (WP( $n$ )  $\neq$   $C_n$ ) and (WP( $n$ )  $\neq$  TRUE) then
   $\forall$  WP $_i$ ( $n$ ) loop
    InEdges := { edges passing data used in WP $_i$ ( $n$ ) }
    if size(InEdges) = 1 then
       $e := \{ e \ni \text{InEdges} \};$ 
      if ( $\exists n \ni n = source(e)$ ) then - - safety-critical input assertions
        F( $n$ ) := 'if WP $_i$  then F( $n$ ) else RECOVERY; end if;
      else - - data flow from another module
         $c(e) := c(e) \wedge \text{WP}_i(n);$ 
        if  $c(e) = \text{FALSE}$  then
          HALT; - - revise design and repeat analysis
        else
          NodesToProcess := NodesToProcess + [source( $n$ )]
        end if;
      end if;
    else - - use run-time assertions to ensure safety
      F( $n$ ) := 'if WP $_i$ ( $n$ ) then F( $n$ ) else RECOVERY; end if';
    end if;
  end loop;
end if;

```

Figure 3.3: *AnalyzeANode (in n : node)*

flow, the condition can be best enforced by the module supplying the data. Otherwise, the functional definition of the module needs to be augmented using assertions so that the safety-critical computations take place only when it is safe to do so. It should be emphasized that the designer should always specify the proper recovery activities to be invoked when the assertions do not hold.

The brute-force approach presented above has some drawbacks. If safety conflicts are detected (e.g., FALSE weakest precondition), the analysis must be applied in entirety on a revised design. Due to the ad hoc order of applying the analysis on each component of the safety constraints, such conflicts may not be detected quickly.

The enhanced algorithm, shown in Figure 3.4 through 3.8, improves the brute force algorithm in the following two ways:

- The data-flow dependency (i.e., *AssignNodeLevels* procedure) determines the order of module analysis.
- If there is a module whose current design cannot guarantee the satisfaction of the safety constraints, an attempt is made to substitute a functionally equivalent but algorithmically different design to see if the conflicts can be resolved. If not, the *AnalyzeANode* procedure returns with a flag (e.g., `MustBackUp = TRUE`) indicating the need to attempt recovery at other nodes (e.g., *Undo* procedure).

The *AssignNodeLevels* procedure (Figure 3.5), invoked as a part of initialization, uses data-flow dependency and assigns a level to each node so that the levels of source nodes are always greater than that of the destination nodes for all the data flows. Backward analysis can then be applied in the ascending level order.

The *AnalyzeANode* procedure (Figure 3.6) is basically the same as the one previously presented except that the *DistributeWP* procedure (Figure 3.7) sets the

```

 $\forall e, c(e) := \text{TRUE};$  - - assume safety-independence
 $\forall C_i$  loop - - for each constraint condition
    OutEdges := { edges with direct impact on  $C_i$  but without dest node };
     $\forall e \ni \text{OutEdges}, c(e) := c(e) \wedge C_i;$ 
end loop;
CurLevel := 1; AssignNodeLevels (MaxLevel); - - analyze data-dependency
OLoop: while (CurLevel < MaxLevel) loop - - in ascending level order
    NodesToProcess := { $\forall n \ni \text{level}(n) = \text{CurLevel}$ }
    while (NodesToProcess  $\neq$  [ ]) loop
         $n := \{ n \ni \text{NodesToProcess} \};$ 
        NodesToProcess := NodesToProcess - [n];
        AnalyzeANode(n, MustBackUp);
        if (MustBackUp) then - - unsafe module detected
            Undo (n, Resolved);
            exit OLoop when ( $\neg$  Resolved); - - parent substitution failed
        end if;
    end loop;
    CurLevel := CurLevel + 1;
end loop;
if (not Resolved) then
    HALT; - - revise design and repeat analysis
end if;

```

Figure 3.4: An Enhanced Algorithm to Detect Safety-Critical Items

```

 $\forall e \ni \neg \exists \text{ dest}(e), \text{ level}(\text{source}(e)) := 1;$ 
CurLevel := 1; MaxLevel := 1;
loop
  Edges := {  $\forall e \ni \text{ level}(\text{dest}(e)) = \text{CurLevel}$  };
  exit when Edges = [ ];
   $\forall e \ni \text{ Edges}$  loop
     $\text{ level}(\text{source}(e)) := \text{ level}(\text{dest}(e)) + 1;$ 
    if  $\text{ level}(\text{source}(e)) > \text{ MaxLevel}$  then
       $\text{ MaxLevel} := \text{ level}(\text{source}(e));$ 
    end if;
  end loop;
  CurLevel := CurLevel + 1;
end loop;

```

Figure 3.5: *AssignNodeLevels* (out *MaxLevel* : integer)

parameter *Successful* to FALSE upon the detection of safety constraint conflicts. In such cases, attempts are made if the substitution of semantically equivalent but algorithmically different functions can resolve the conflicts. If not, the node returns the boolean flag *MustBackUp* to TRUE so that the analysis can be backtracked to the nodes that are the destination of the data flows originating from the node n .

The *Undo* procedure (Figure 3.8) determines if the safety conflicts that could not be resolved locally can be resolved by modifying the functional definition at one of the parent nodes (e.g., the destination node of data flows). Safety constraint conflicts are considered to be resolved if and only if backward analysis applied at the parent node (i.e., PNode) and the child node (i.e., n) do not require any further backups. The modification of the functions at the parent node requires that backward analysis

```

MustBackup := FALSE;
∀j(1 ≤ j ≤ i) ∃ source(ej) = n, Cn := c(e1) ∧ ... ∧ c(ei);
loop
  WP(n) := weakest precondition, in CNF, to satisfy Cn;
  exit when (WP(n) = Cn) ∨ (WP(n) = TRUE); - - ignore
  ∀e ∃ n=source(e), save c(e);
  DistributeWP (Successful);
  exit when Successful;
  if ∃ G(n) then - - local substitution failed
    F(n) := G(n);
  else
    MustBackUp := TRUE;
    exit;
  end if;
end loop;

```

Figure 3.6: *AnalyzeANode* (in n : node; out $MustBackUp$: boolean)

```

Successful := WP( $n$ )  $\neq$  FALSE;
if Successful then
   $\forall$  WP $i$  loop
    Edges := { edge supplying value used in WP $i$ };
    if size(Edges) > 1 then - - data-flow from multiple modules
      F( $n$ ) := 'if WP $i$  then F( $n$ ) else RECOVERY; end if';
    else - single data-flow source
      if ( $\nexists n \ni n = source(e)$ ) then - - input assertion
        F( $n$ ) := 'if WP $i$  then F( $n$ ) else RECOVERY; end if';
      else - - another module
        c( $e$ ) := c( $e$ )  $\wedge$  WP $i$ ;
        if c( $e$ ) = FALSE then
           $\forall e \ni n = source(e)$ , restore c( $e$ );
          Successful := FALSE;
          exit;
        end if;
      end if;
    end if;
  end loop;
end if;

```

Figure 3.7: *DistributeWP* (in n : node; in WP : boolean; out Successful : boolean)

```

Resolved := FALSE;
ParentNodes := {  $\forall m \ni \exists e$  (source( $e$ )= $n$ )  $\wedge$  (dest( $e$ ) =  $m$ )};
OLoop: while (not Resolved) and (ParentNodes  $\neq$  [ ]) loop
    PNode := {  $m \ni$  ParentNodes };
    ParentNodes := ParentNodes - [PNode];
     $\forall e \ni$  PNode=dest( $e$ ), save  $c(e)$ ;
AParent: loop
    exit AParent when  $\nexists$  G(PNode); - - parent substitution failed
    F(Pnode) := G(PNode);
    ParentWP := Weakest precondition for PNode;
    AnalyzeANode(PNode, MustBackup);
    if ( $\neg$  MustBackup) then
        AnalyzeANode( $n$ , MustBackup);
        if ( $\neg$  MustBackup) then
            Resolved := TRUE;
            ChildNodes := {  $\forall m \ni \exists e$  (source( $e$ ) =  $m$ )  $\wedge$  (dest( $e$ ) = PNode)};
            NodesToProcess := NodesToProcess + ChildNodes;
            exit OLoop;
        else
             $\forall e \ni$  PNode=dest( $e$ ), restore  $c(e)$ ;
        end if;
    end if;
end loop; - - for each parent node
end loop; - - not resolved

```

Figure 3.8: Undo (in n : node; out Resolved : boolean)

be applied again on some child nodes because different safety constraints may be propagated.

It is possible, in principle, to extend the *Undo* procedure so that the backtracking continues until either the safety constraint conflicts are resolved or there are no more modules to which one can backtrack. However, the idea of “global” (or extensive) backtracking seems impractical because:

- For a complete backtracking analysis, one must examine the various combinations of functional definitions at all the modules involved.
- Since the analysis is applied recursively *at each level of decomposition*, the number of modules one must analyze at a time is not expected to be large.

The *Undo* procedure, therefore, halts the analysis if the substitution of various functional definitions at the node and at the parent node fail to resolve the conflict.

3.2.2 Safety-Critical Module Protection

Once the safety-critical modules have been identified, it is important to protect the safety-critical items by a “firewall.” The firewall allows the safety-critical modules to be clearly identified and reduces the effort required for safety verification. The basic idea behind a firewall is to restrict the interactions between the safety-critical modules and the safety-independent ones so that the behavior of the safety-critical modules can be completely determined given only their definitions.

Security techniques can be used to implement firewalls because their goal is to control access to objects. Capability and access lists are general concepts used to

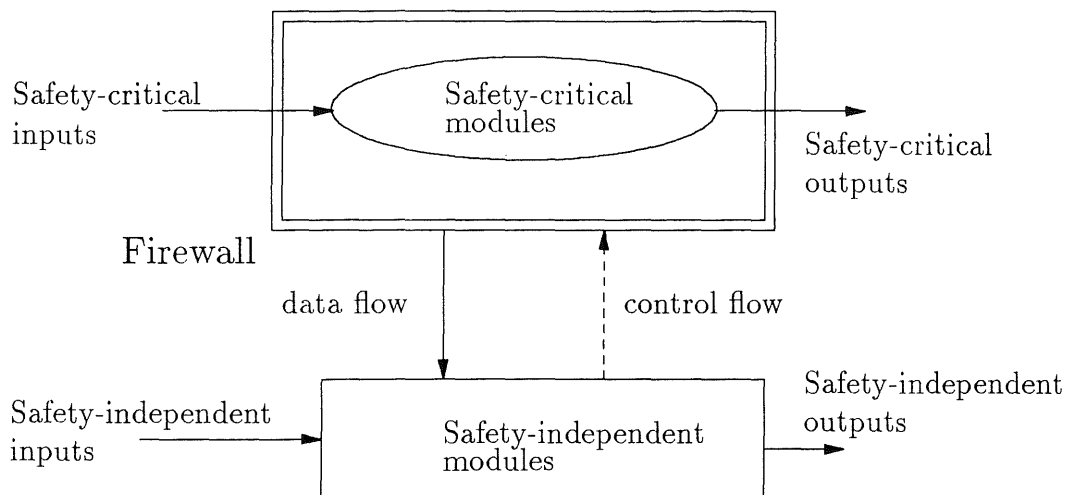


Figure 3.9: Essential Firewall Requirements

enforce access limitation in security. Rushby[50] suggested the use of kernel encapsulation as a means of protecting safety-critical items. While security techniques such as capabilities or kernelization provide adequate means of protecting the safety-critical items, this dissertation proposes the use of information hiding principles[43] instead (Figure 3.9) because:

- The implementation of a capability or kernel within the application is a non-trivial task, and the implementation is subject to errors. Information hiding, on the other hand, does not require any implementation overhead.
- The formal verification of the correctness of a capability or kernel is a complex and difficult task, but verification of information hiding is simple and can be provided easily by the compiler.

To implement a firewall, data flow from the safety-independent modules to the safety-critical modules is prohibited. Similarly, calls from the safety-critical modules

to the safety-independent modules are prohibited. However, the opposite unidirectional data and control flows need to be allowed. When an input triggers multiple outputs, some may be safety-critical while others may not. In such cases, the inputs should be considered safety-critical and must be processed by the safety-critical modules. A unidirectional data flow allows the safety-independent modules to access such inputs. When a package construct is used to group the safety-critical modules, the package specification must declare only the following items to be visible:

- The variables whose values are to be passed unidirectionally to the safety-independent modules. The safety-critical modules may assign values to these variables, but the safety-critical outputs must not depend on the values of these variables.
- The subprograms that safety-independent modules may call. The parameter passing mode must be strictly limited to the **out** mode that is semantically equivalent to the unidirectional data flow.

The strict enforcement of such restrictions, despite the potential inconvenience to the developers, simplifies the safety verification process and should enhance safety.

3.3 Detailed Design Safety Verification

After the detailed design is complete, design safety verification should be applied to detect errors before the coding begins. Intermediate verification requires additional

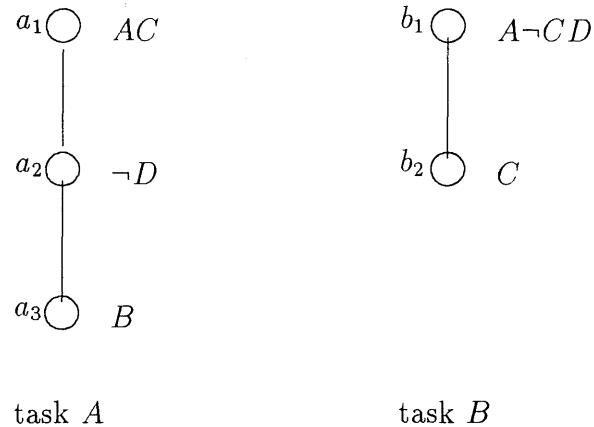


Figure 3.10: Why Concurrency Safety Analysis is Needed

resources, but should simplify the safety verification of the final code. If the safety-oriented design method presented in the previous section is employed during the high-level design phase, the safety verification of the detailed design can be accomplished by proving that:

- The safety-critical modules are protected from the others. This allows one to verify the safety of the entire software by analyzing the safety of the safety-critical modules only.
- Algorithmic definitions of the safety-critical modules satisfy their safety constraints. This dissertation proposes an incremental approach to safety verification: sequential safety verification followed by an optional concurrency safety verification.

If the safety-critical software is implemented in a sequential language, the application of the second phase is unnecessary. Otherwise, concurrency safety verification is necessary, even if all the safety-critical modules are proven to be safe, because the concurrent execution of the individually safe modules might be unsafe. For example,

suppose that the safety constraints are given as $(A \vee B) \wedge (C \vee D)$ and that the two safety-critical modules, whose control-flow graphs are shown in Figure 3.10, are executed concurrently. Although each module satisfies the safety constraints individually, the concurrency state (a_2, b_1) might fail to satisfy the safety constraints. The safety of the concurrency state depends on the pace of task execution, which may not be under the programmer's control.

3.3.1 Firewall Adequacy Verification

Verifying the firewall's adequacy shows that the interactions among the safety-critical and safety-independent modules have no effect on the semantic behavior of the safety-critical modules. The firewall adequacy verification for a block structured language such as Ada can be accomplished by determining satisfaction of the following static criteria:

- The safety-critical and the safety-independent modules have no variables that are visible to both of them except those specifically declared for the purpose of unidirectional data flows. Furthermore, the values of the variables used for data flow should never be used in determining the values of the safety-critical outputs.
- The safety-critical modules may not call any safety-independent modules. When the safety-independent modules call the safety-critical modules, the actual parameters should not violate the unidirectional data flow rules. The prohibition of **in** -type or **in out** -type parameters in such calls guarantees the absence of side-effects.

3.3.2 Design Safety Verification: Sequential Phase

Design hazard analysis, as described above, identifies the safety constraints C_M and the desired weakest preconditions $wp(R_M, C_M)$ for each safety-critical module based on its functional requirements R_M . Sequential safety verification on the detailed design attempts to prove that the detailed algorithmic definitions of module F_M satisfy these constraints.

The verification can be performed in either a forward or backward manner. Forward analysis attempts to prove that all the states reachable from the known initial state are safe. Using the desired weakest precondition, $wp(R_M, C_M)$, as the initial condition, the postcondition R can be derived (e.g., $\{wp(R_M, C_M)\} F_M \{R\}$). The module is safe in a sequential execution environment if and only if R is logically equivalent to or a “stronger” condition than the safety constraints (e.g., $R \Rightarrow C_M$). Forward analysis, however, becomes impractical if there are a large number of states to consider.

Backward analysis, on the other hand, starts with the safety constraints as the initial condition. The weakest precondition derived based on the detailed module design, $wp(F_M, C_M)$, can then be compared against the desired weakest precondition $wp(R_M, C_M)$. The module is safe, when executed sequentially, if and only if the predicate

$$wp(F_M, C_M) \Rightarrow wp(R_M, C_M)$$

evaluates to TRUE.

When backward sequential module safety verification is performed, not all the computational steps within the safety-critical modules need analysis. Nor do their

results (e.g., intermediate conditions leading to the module's weakest precondition) need to be saved for use in subsequent concurrency verification. Only the computations that cause the intermediate weakest preconditions to be changed need to be saved. Additionally, the statements that deal with concurrency, such as entry calls or rendezvous points, need to be saved so that correct concurrency safety verification may be performed.

3.3.3 Design Safety Verification: Concurrency Phase

While past software designs were predominantly sequential, the use of concurrent designs is increasing. Concurrency is advantageous when a natural and logically concise solution can be developed. However, concurrency decisions on safety-critical software must be based on careful trade-off analysis because:

- The run-time overhead of creating and managing tasks is significant. Such run-time overhead, in some hard real-time systems, may cause failures due to missed deadlines or even safety hazards.
- The complexity may increase due to the various ways tasks may communicate. Some rendezvous may result in hazardous states, but exhaustive verification is usually impractical.
- The application of concurrent software verification techniques is likely to be more expensive and prone to errors than the sequential counterparts.

Where the use of concurrency in safety-critical software design can be justified, a safety-oriented design method must provide a technique to verify the safety of concurrent design efficiently. This dissertation proposes that the use of concurrency

in safety-critical software design be limited only to the cases where software must simultaneously control logically distinct subsystems. Therefore, the decision to use concurrency must be made early in the design phase, and the subsequent refinements must proceed sequentially.

Concurrency safety verification proves that the concurrent execution of the modules that are proven to be safe in the sequential execution environment is also safe. Since concurrency safety verification is a form of static analysis, the following assumptions are often needed to enable or to simplify the analysis[35]:

- Arrays of tasks are not allowed, and at most a fixed number of tasks are active simultaneously.
- Tasks do not share variables (e.g., no race conditions).

While the former is an inherent limitation of any static analysis technique, the latter simplifies the analysis by forcing the tasks to communicate only by explicit rendezvous and eliminates the possibility of side-effects.

While it is possible, as noted by Long and Clarke[35], to extend the analysis technique to handle side-effects, the techniques become more complicated and less efficient due to the introduction of nondeterminism. Development of software tools to (fully or partly) automate the analysis becomes more costly. More importantly, analysis results themselves could be subject to more errors. The possibility of erroneous analysis cannot be ignored for safety-critical software design. Therefore, one must carefully evaluate the trade-offs between the restricted design activities that require relatively simple verification procedures and the unrestricted and flexible design activities that require more sophisticated (and potentially more erroneous) verification procedures. It is almost always wiser to choose the former unless the restrictions

are too severe to prevent designers from developing understandable and maintainable designs. This is especially true for the design of safety-critical software.

A straightforward approach to concurrency safety verification generates all the feasible combinations of the critical states and proves that the safety constraints are satisfied in all the concurrency states. There are two functionally equivalent methods of determining the feasible concurrency states. The first method starts from the known initial concurrency state and identifies all the concurrency states that are reachable (e.g., concurrency graph[52], Petri Net reachability graph[45], or task interaction graph[35]). The second method first generates all possible combinations of component states regardless of their reachability (e.g., the Kleene star operator in constrained expression formalism[2]). The pruning process is then followed to eliminate the infeasible concurrency states. The rendezvous points, which must have been introduced to fulfill the functional requirements, serve as a pruning tool. The pruning allows one to achieve the effect of applying the shuffle operator in the constrained expression formalism. The satisfiability of the safety constraints at each concurrency state can then be determined by serializing the concurrent execution into the list of feasible sequential executions. However, this brute-force analysis is impractical due to the enormous number of serial executions that must be analyzed.

An efficient concurrency safety verification technique has been developed based on the following argument:

- Concurrency safety verification can be achieved by proving that all the reachable concurrency states satisfy the safety constraints (e.g., the brute-force technique described above). Or, one can accomplish the same objective by proving that

there are no concurrency states that lead to the violation of the safety constraints.

- Unlike the sequential safety analysis where *all* the safety-critical modules must be analyzed, only the safety-critical modules that could be executed concurrently with other safety-critical modules (henceforth called “concurrent safety-critical modules”) need to be analyzed in concurrency safety verification.
- Even some of the statements setting the safety constraint conditions to false within the concurrent safety-critical modules can be ignored because not all the safety constraints are subject to violation during concurrent execution. Therefore, only the safety constraints that are subject to violation in the concurrent execution environment need to be analyzed, and only the statements setting such safety constraint conditions to false need to be examined.

Suppose, for example, that the safety constraint the concurrent modules must satisfy is given as $(A \vee B) \wedge (C \vee D)$ and that the modules have the control-flow graph shown in Figure 3.11. The initial and final states of the tasks are represented as a_1 and b_1 and as a_n and b_n , respectively. The states a_i , a_j , and b_k change the value of the safety constraint C as indicated. The *potential* rendezvous points are represented as r_{a1} , r_{a2} , and r_{a3} for task A and as r_{b1} and r_{b2} for task B . The rendezvous points (1) and (2) represent the synchronization of task A at states r_{a1} and r_{a3} , respectively, with task B at state r_{b1} . Rendezvous points (3) and (4) represent the rendezvous at the states r_{a2} and r_{b2} , respectively. The condition C remains false between the states a_i through a_j for the task A . The concurrency states that include any of states a_i through a_j as their members could *possibly* fail to satisfy the safety constraints, and these states are collectively referred to as “impacted concurrency states.”

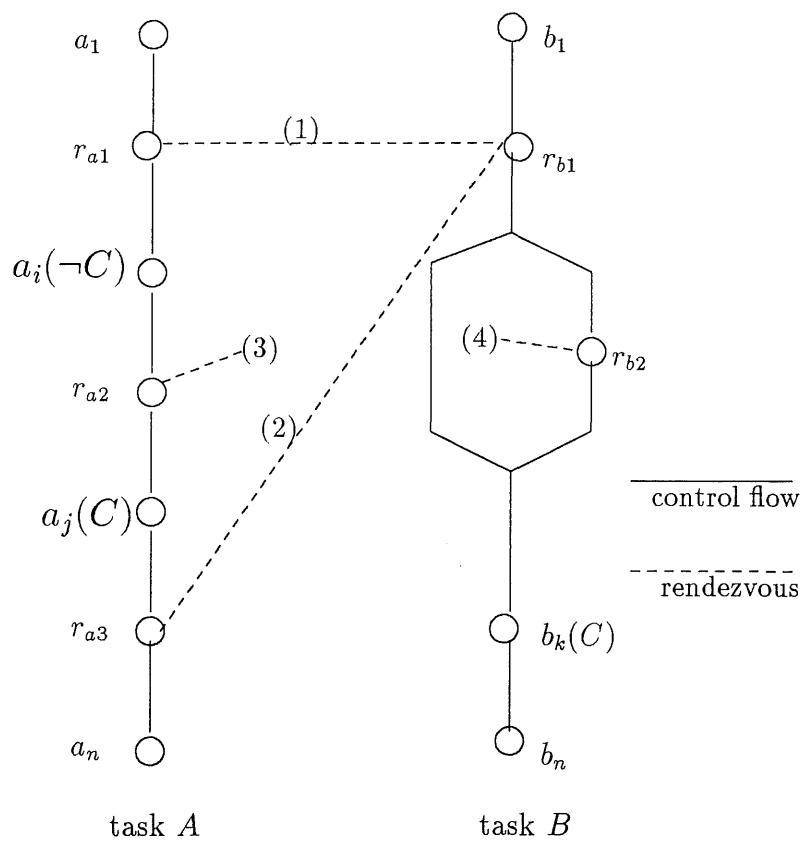


Figure 3.11: Impacted Concurrency States and the Rendezvous

The scope of the impacted concurrency states depends on the formulation of the safety constraints themselves and the placement of the rendezvous points among the concurrent modules. The term “complementary conditions” is used to refer to the conditions that can still satisfy the safety constraints regardless of the current values of some other conditions. If, for example, the safety constraints are given as $(A \vee B) \wedge (C \vee D)$, the conditions A and B are complementary conditions in that the safety constraints can be satisfied by either (or both) of the conditions being TRUE. Similarly, the conditions C and D are the complementary conditions. Representing the safety constraints in the conjunctive normal form clearly reveals the complementary conditions. If, on the other hand, the safety constraints were $(A \vee B) \wedge C$, there is no complementary condition to C , and setting C to false would always cause the safety constraints to be violated.

Similarly, the rendezvous might reduce the size of the impacted concurrency states. These rendezvous points might have been introduced to fulfill the functional requirements of the module or as a safe design technique during the sequential module safety verification. For the control-flow graph shown in Figure 3.11, the impacted concurrency states are $(a_i..a_j, b_1..b_n)$ in the absence of any rendezvous. Rendezvous point (1), however, prevents states b_1 through r_{b_1} (excluding the boundary state r_{b_1}) from being affected by the event of setting C to FALSE at a_i . Accordingly, the impacted concurrency states are reduced to $(a_i..a_j, r_{b_1}..b_n)$. The rendezvous point (2) similarly limits the impacted concurrency states to $(a_i..a_j, b_1..r_{b_1})$.

However, not all rendezvous points are effective in reducing the size of impacted concurrency states. Rendezvous points (3) and (4) are such examples. They are ineffective for either of the following reasons:

- The rendezvous occurs at the state where the safety invariant conditions remain FALSE.
- The rendezvous occurs only on one path or a limited number of paths in the concurrent modules. Therefore, it is possible for the rendezvous not to occur.³

If the analysis reveals that there are some impacted concurrency states that fail to satisfy the safety constraints, assertions or rendezvous can be introduced as appropriate. Assertions are used to provide further constraints on executing the safety-critical statements so that they are executed only when their safety can be guaranteed not only in the sequential but also in the concurrent execution environment. Rendezvous points are used as a means of controlling the pace of task executions so that the concurrency states that violate the safety constraints can never occur in the absence of abnormal control-flow errors. If the assertions or rendezvous points are unable to guarantee the satisfaction of the safety constraints at some impacted concurrency states, the design must be revised.

The traditional concurrency analysis techniques would have required the generation and analysis of a large number of concurrency states that either have nothing to do with safety or that can never possibly cause the safety constraints to be violated. However, incremental safety analysis allows the analysis to focus only on statements that have direct impact on safety in the concurrent execution environment.

³The possibility of task deadlocks due to the absence of the expected rendezvous is a different issue. While it is possible for hazardous states to occur consequently, the standard deadlock detection techniques would reveal such possibilities.

3.4 Some Safe Design Techniques

While the rigorous application of the safety-oriented design method presented in the previous section should prevent hazardous states from occurring, the hazardous states still may occur due to:

- Timing errors.
- Generation of software outputs in an incorrect order.
- Abnormal data and control flow errors.

This section proposes some safe design techniques to reduce or control these problems. Section 3.4.1 explains how the possibility of timing errors can be reduced. Section 3.4.2 recommends the use of software interlocks to ensure the correctness of outputs. Section 3.4.3 shows a simple redundancy technique using programming language constructs that can detect or tolerate data and control flow errors.

3.4.1 Timing Errors

Timing analysis on software requirements has received some attention in the past few years. For example, timing analysis techniques on the requirements have been developed by Jahanian and Mok[22, 23]. Since the design may introduce timing errors, it is desirable to verify the timing aspect of the design, too. However, timing analysis on the software design is more difficult than the analysis on the requirements or the code.

Timing analysis at the requirements level is feasible because the requirements address the “what” aspect of software rather than “how.” That is, each activity

specified in the requirements can be modeled as either an atomic or composite activity. Timing information on atomic activity is *assumed known or can be specified*. Timing information on composite activities can be derived from that of the atomic activities and their relationships. Therefore, for timing analysis on requirements, one need not worry if an atomic action can be completed within the deadline; in fact, the analysis starts with the assumption that it will be completed within its deadline.

Similarly, timing analysis on the source code is feasible because the source code represents a concrete representation of the system being developed. Furthermore, the source code can be compiled easily into lower level languages whose upper bound on the execution time is known[15]. Therefore, maximum response time can be determined by analyzing the module through all the feasible paths. If computation involves loops with an indefinite number of iterations, no definite upper bound on execution time can be derived. Watchdog timers, however, can be added to detect timing errors at runtime.

On the other hand, no matter how detailed the design, it is an *abstract* statement of “how” software fulfills “what” is required. If the same design documents are distributed to different programmers (or programming teams), it is reasonable to expect different implementations to have different characteristics. For example, the same operation (e.g., sorting) could be implemented using different algorithms. Even if the algorithm is specified in the design documents (e.g., quick sort), different implementations almost always exhibit different timing behavior. Therefore, it is possible that some implementations satisfy the timing requirements while others do not. These factors are essential in performing meaningful timing analysis but are still unknown at the design phase, making timing analysis very difficult (if possible at all).

However, there are measures designers can take to reduce the possibility of hazards due to timing errors. Hazardous states may occur if the required response occurs either too soon or too late. This includes the possibility of required events never occurring. If an event is hazardous when it occurs too soon, the designer can tighten the trigger conditions (e.g., require additional confirmation). If, however, the event is hazardous when it occurs too late, the designer can relax the trigger conditions (e.g., provide an alternative source for the trigger conditions) so that timing errors occur only when there are multiple errors in all the trigger conditions. Suppose, for example, that the gate movement at a train crossing has been computerized. Hazardous states occur with either of the following:

- The gate is not lowered before the train arrives at the crossing.
- The gate is raised before the train leaves the crossing.

The possibility of the former can be reduced by installing multiple sensors to detect train movement. While multiple sensors increase the possibility of erroneously lowering the gate when the sensors fail, these failures do not compromise safety. Equivalently, a separate routine to issue the gate movement command could ensure that the gate movement is delayed no more than the predefined period of time once a train approaching the crossing is detected. The possibility of the latter can be reduced by raising the gate only when separate sensors confirm that the train has left the crossing.

3.4.2 Unsafe Output Sequences

Software mishaps can occur not only from inherently unsafe software outputs but from an unsafe sequence of outputs. Suppose, for example, the generation of the

successive outputs O_A immediately followed by O_B is unsafe and that the generation of the intermediate output O_I is needed to ensure safety. Software interlocks provide an effective means of enforcing the desired output sequences, and they can be implemented easily by modifying the output trigger conditions. If the trigger condition functionally required for the output O_B is denoted c_B , one can “strengthen” the trigger condition to be $I \wedge c_B$ where I is the interlock condition. The output sequencing can be accomplished if the interlock condition I , initially set to TRUE, is set to FALSE after the generation of the output O_A . Setting the interlock condition to TRUE only after the generation of the required intermediate output O_I guarantees that the output O_A is never immediately followed by O_B without the required output O_I in between. The interlock conditions, however, do not block the output O_B from being generated if the preceding output is not O_A .

3.4.3 Abnormal Data and Control Errors

Another issue that must be addressed in developing high-quality safety-critical software is that of the programming language constructs. Programming languages impact productivity as well as safety. If the implementation language does not support the concepts used in the design, the program becomes longer, more expensive to develop, and less reliable. This section proposes some programming language constructs that are useful in preventing and detecting abnormal data and control errors.

Data errors occur when incorrect values are assigned to the variables. Data errors, almost always the result of design (logic) errors, can be detected using the safety verification technique proposed earlier. However, data errors may occur abnormally; environmental stress or hardware failures may cause data errors[17]. One such type

of stress occurs when a bit of memory holding the variable is toggled unexpectedly when a gamma ray hits the memory cells hard enough to cause a change. Any unexpected modification of these variables can be detected (or tolerated if desired) by using redundancy (e.g., allocate the variable at two or three distinct memory locations). Data redundancy is hardly a new idea. For example, the VAXft 3000, the first fault-tolerant VAX computer, comes with a layered product called a volume shadowing server⁴. The users can create a shadow set that consist of up to three disks, and the shadow server software guarantees the data integrity among the redundant data through multiple updates and voting upon data retrieval.

Selective data redundancy capability can also be provided by the programming language if users are allowed to declare the safety attribute of variables along with their data types.

```
procedure monitor_patient is
    age : integer; - - default (safety independent)
    blood_pressure : critical integer;
```

This relieves the programmer of the burden of designing software that can cope with its own failures due to abnormal data errors. However, the designer need not manually identify all the safety-critical variables because the compiler can identify all the safety-critical variables through static data dependency analysis based on the declaration of the safety-critical software outputs. While the implementation of data redundancy requires additional resources, it is a simple and effective solution to potentially serious software failures that are difficult to handle in software.

⁴Volume shadowing is DEC's terminology for data redundancy.

Control errors refer to the cases where the program (or module) execution occurs due to the execution of an incorrect path. This often occurs due to logic errors (e.g., an incorrect branch condition), and the safety verification technique discussed earlier should detect such errors. Abnormal data errors may also cause control errors. For example, a data error in the program counter register or the variables used in branch condition evaluation results in control errors. These types of errors can be either detected or tolerated using the safety attributes proposed above.

An alternative is the use of batons[55]. The basic idea is to determine the desired control flow of the module and to assign a unique identifier to each place in the graph. The baton is an internal variable that gets passed from place to place and modified. Each place compares the baton values against its list of potential values and detects the occurrence of abnormal control flow errors. The overhead of the baton, due to the runtime comparison and modification of the baton variable, can be kept reasonable if the compiler allows the technique to be limited to the safety-critical modules only.

Chapter 4

A Safety-Oriented Management Structure

Software project management techniques are very important due to their direct and significant impact on software quality. This is also true of their impact on safety. As Leveson points out in her survey paper[27], “the degree of safety achieved in a system depends directly on management emphasis,” and the development of high-quality safety-critical software is unlikely without management’s recognition of the seriousness of the software safety problems and its commitment of adequate resources. Management needs to be aware of the effectiveness and the costs of potential approaches to software safety so that a wise decision can be made about how the resources are to be spent.

Software project management is a very broad subject that covers topics such as planning, organizing, staffing, directing, and controlling[47]. This chapter addresses the project management issues that are unique to software and system safety.

4.1 Software Safety Management

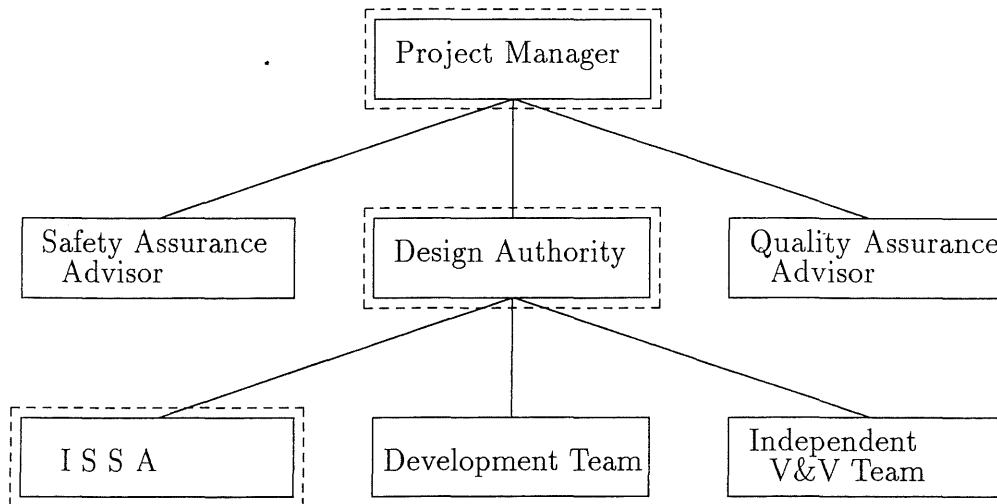


Figure 4.1: Safety Management Hierarchy Recommended in the MoD-Std-0055

Figure 4.1 shows the organizational structure recommended by the UK software safety standard. It consists of a project manager, design authority, independent software safety assessor (ISSA), an independent verification and validation (IV&V) team, and a number of advisors assisting the project manager. The parties whose approval is required in issuing the software safety certification are shown in dashed boxes. The standard specifies the safety responsibilities as follows:

- The project manager, acting as the safety authority, bears the ultimate responsibility for software development and its safety.
- The design authority, appointed by the project manager, appoints the ISSA and the IV&V team. The primary responsibility of the design authority is safety management which includes the following:
 - Safety plan preparation. The safety plan is prepared at the beginning of each phase and requires the project manager’s approval. The plan provides

a detailed description of safety goals and the means of accomplishing such goals.

- Safety log maintenance. The safety log, main depository of information related to safety, contains documents on review results, formal correctness and safety proofs, hazard analysis results, etc.
- Software design and verification. The design authority carries out high-level design analysis by identifying the safety-critical modules and by isolating them. The design authority also conducts formal correctness and safety verification on the detailed design and the code.
- The ISSA, independent from the design authority, audits and reviews all activities and documents involved in safety-critical software development.
- The IV&V team, independent of the design team, checks the correctness of the design and proofs. The results are recorded in the safety log.
- A number of advisors and authorities assist the project manager and the design authority on various subjects. For example, the safety assurance advisor “advises the project manager on all safety matters.” This includes the assessment of the work by the design authority and the ISSA.

Figure 4.2 proposes a slightly different hierarchy where the ISSA is appointed and supervised directly by the project manager – not by the design authority. The advantages of this include:

- The elimination of a redundant position.

The role of safety assurance advisor, assessing the work of the design authority and the ISSA, may seem reasonable as providing yet another

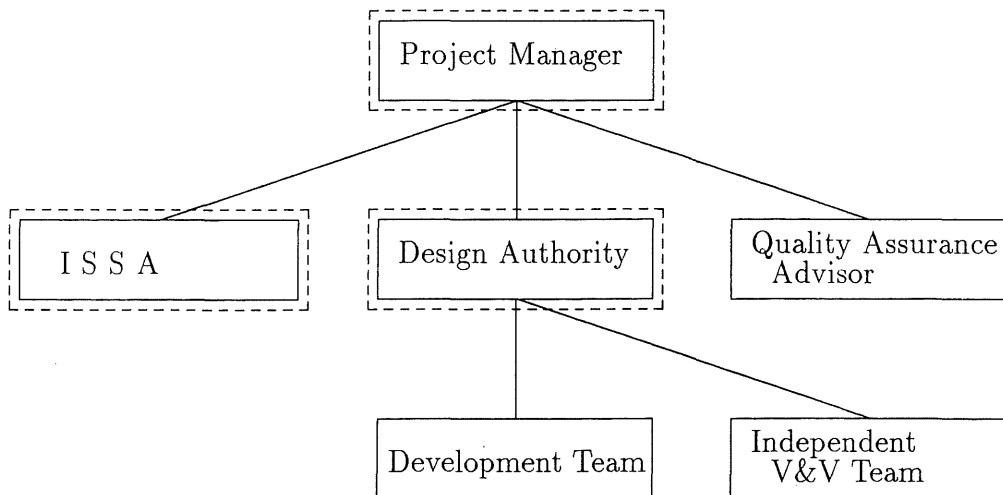


Figure 4.2: Slight Variation to Mod-StD-0055 Management Hierarchy

level of “defensive line” against software mishaps. However, one of the most fundamental management principles is that excessively redundant positions reduce productivity due to increased communication overhead.

The revised structure eliminates the position of the safety assurance advisor by having the ISSA and the IV&V team advise the project manager and the design authority, respectively, on safety matters.

- Managerial independence.

The standard recognizes the importance of technical as well as managerial independence of the ISSA from the design authority. Having the ISSA appointed and supervised directly by the project manager ensures (or at least enhances) managerial independence from the design authority.

- Reduced load on design authority.

The standard explicitly requires the design authority to be responsible for safety management. Consequently, the design authority is responsible for an overwhelmingly large number of tasks as detailed in Annex B of the standard. Some tasks (e.g., safety log maintenance) require administrative skills while others require technical skills (e.g., software hazard analysis, safety-oriented software design, etc). While this heavy concentration of responsibilities allows the design authority to control software development, it may result in a negative impact on software safety. Furthermore, finding people who possess both administrative and technical talents can be difficult.

Therefore, it is best to separate the technical and managerial aspects of software safety responsibility. The technical aspects can be performed by the design authority whose role is similar to that of the chief programmer[40], and the administrative aspects can be performed by the project manager.

- Lower turnover rates.

The standard notes that the person acting as the design authority may vary from phase to phase (e.g., due to staff turnovers or individual specialties). It is naive to expect no staff turnovers during development, but it is reasonable to expect lower staff turnover rates among the positions with greater responsibility. Software safety is more likely to receive the continuous attention it deserves if the responsibility of the actual safety management is shifted to a higher-ranking authority.

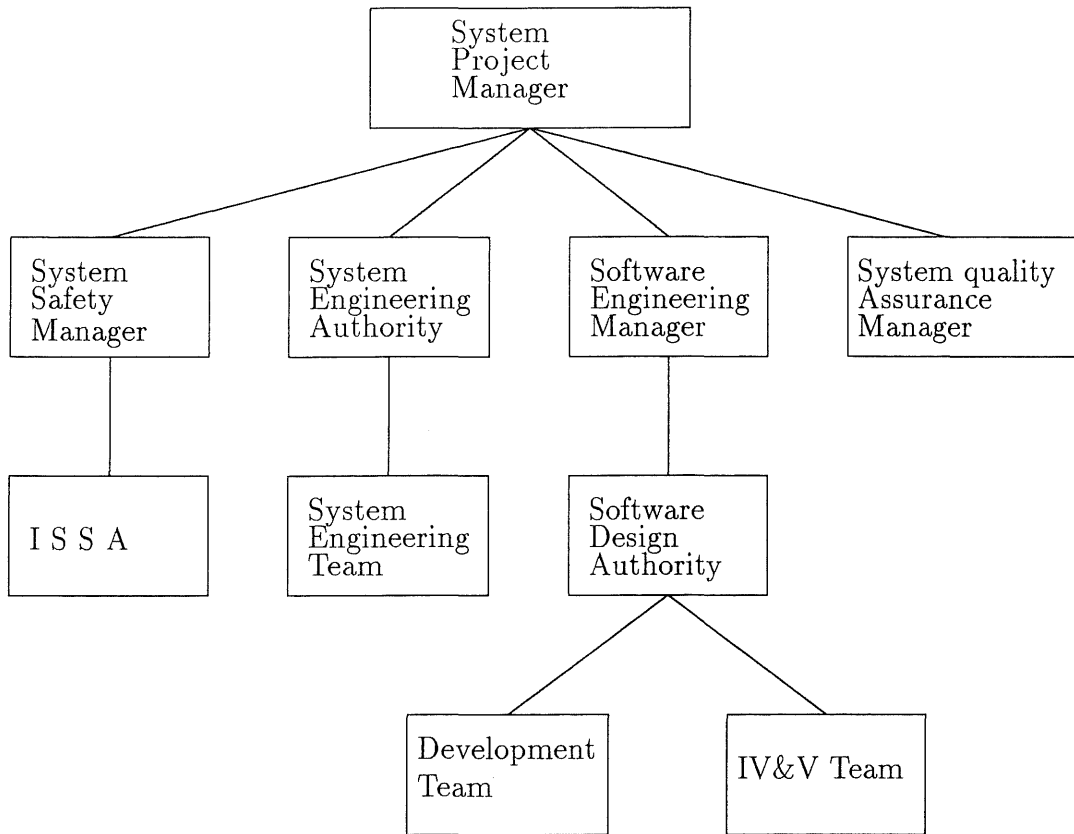


Figure 4.3: Safety-Critical System Development Management Hierarchy

4.2 System Safety Management

A generalization of the safety-oriented software management hierarchy to include system safety is proposed in Figure 4.3. It recommends the appointment of a system safety manager who is responsible for the verification of both system and software safety. Since software safety is a system property, the ISSA can be appointed and supervised by the system safety manager.

The system engineering authority and the system safety manager should be responsible for the early phases of system development (i.e., the system requirements specification and system design). The system engineering authority develops the system requirements, performs the preliminary hazard analysis (PHA), and allocates the system requirements to subsystems. Upon the submission of the system design for approval, the system safety manager should review the work of the system engineering authority and make a recommendation on its safety to the system project manager.

Software safety responsibilities should be similarly allocated. The software design authority is responsible for the validation of the logical completeness and safety of the requirements specification. The IV&V team should independently validate the work of the software design authority. The software design phase, however, only begins when the software engineering manager gives a formal approval to the independent safety verification by the ISSA. The software design authority performs the design hazard analysis and verifies the detailed design safety as presented in this dissertation. The ISSA provides an independent recommendation on software design safety to the software engineering manager who formally approves the completion of the design phase. The same process is repeated in the coding phase.

The software engineering manager aids the software design authority by assuming responsibilities on document and configuration controls. This allows the software design authority to devote his or her efforts to the technical aspects of the project. However, the software design authority must be kept informed about the documentation and configuration changes. The project manager should maintain the safety log as well. A software librarian could be hired if necessary to keep the load of the software engineering manager at a reasonable level.

Upon the completion of software development, the system quality assurance manager performs system integration and makes a recommendation to the system project manager on its acceptance. The system safety manager must perform the safety verification of the integrated system and certify the system safety. The system project manager, based upon these recommendations, accepts the system and certifies its safety.

Chapter 5

A TCAS Example

5.1 Introduction

This chapter describes an application of the safety-oriented design method described in Chapter 3. It is demonstrated using the threat detection subsystem of the TCAS II (Traffic Alert and Collision Avoidance System) software design. A Federal Aviation Administration (FAA) publication[1] provides the best and most concise description of the goals and the basic designs for the TCAS II. The complete description of the detailed CAS logic is published by the Radio Technical Commission for Aeronautics (RTCA)[49].

TCAS, a family of airborne devices that function independently of the ground-based air traffic control system, provide collision avoidance protection for a broad spectrum of aircraft types. TCAS is based on the concept of range/range rate (τ) which defines time-to-go, rather than distance-to-go, to the closest point of approach. TCAS II provides traffic advisories and resolution advisories (i.e., recommended escape maneuvers) in a vertical direction to avoid conflicting traffic. Effective CAS logic operation requires a trade-off between necessary protection and unnecessary

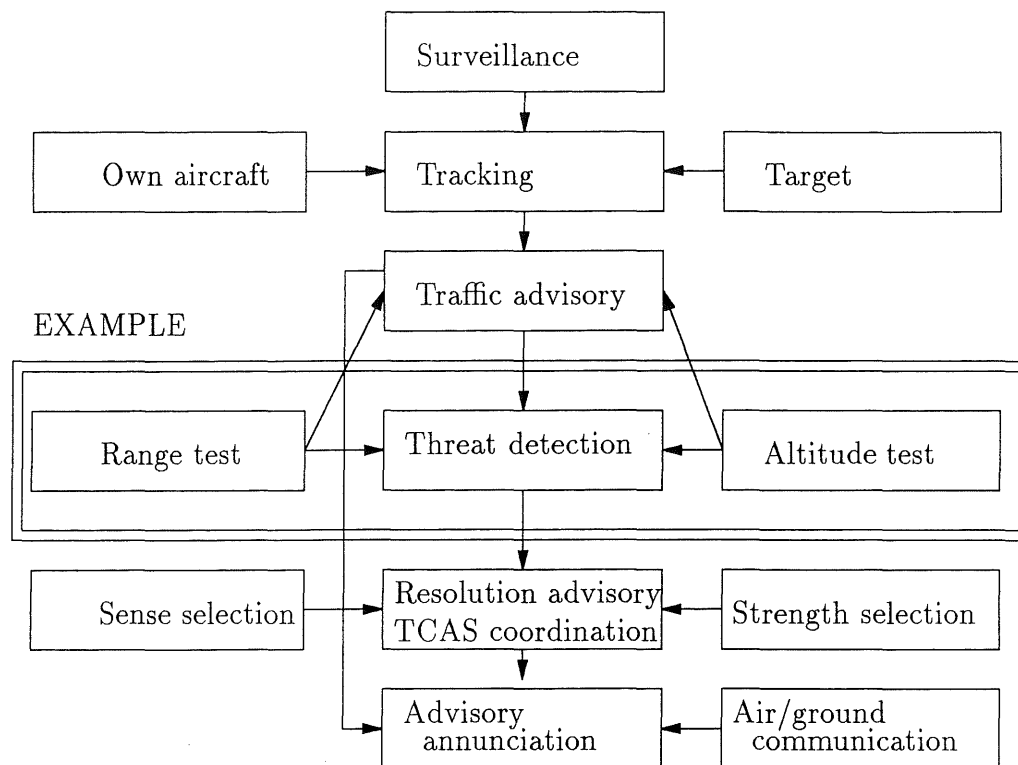


Figure 5.1: CAS Logic Functions

advisories. Controlling the sensitivity level (SL) changes the dimensions of the protected air space around each TCAS-equipped aircraft. A higher SL provides better protection but also increases the probability of unnecessary alerts.

The logic functions used by TCAS II to perform the collision avoidance task are shown in Figure 5.1. The surveillance data on the ‘own’¹ and intruder aircraft are processed by the “Tracking” subsystem which creates the intruder track file (ITF). The “Threat Detection” subsystem searches the list of intruders and identifies the threats about which the traffic or resolution advisories are issued. The “Resolution

¹The aircraft the TCAS must protect is referred to as ‘own’ while any other aircraft is called the ‘intruder’.

Advisory” subsystem coordinates the recommended maneuvers to avoid collision with other TCAS-equipped aircraft and displays the resolution advisories to the pilots.

Because of the complexity of the complete TCAS II design, the example in this dissertation is limited only to the *Threat-Detection* subsystem. The threats are identified by performing range and altitude tests on every altitude-reporting intruder. The intruders are declared a threat when both the range and the altitude tests are passed. However, the threat declaration may be delayed, even if the intruder passes both tests, depending on the geometry of the encounter and the quality and length of the vertical track data.

The PDL descriptions used in this chapter are slightly different from the one published by the RTCA in the TCAS Minimum Operational Performance Standards (MOPS). The differences are:

- *Track-Firmness-Test*. This procedure examines the quality and length of the vertical track data to determine if the threat declaration should be delayed even if the intruder passes both the range and the altitude criteria. While the RTCA design invokes the *Track-Firmness-Test* only when the intruder passes the range and the altitude tests, the modified PDL first invokes the *Track-Firmness-Test* to see if the application of the the range and altitude tests are needed².
- *Concurrency*. The RTCA design is strictly sequential. To illustrate the application of the method on concurrent designs, the modified PDL performs the range and altitude tests as tasks.

²The modification is intended to simplify the complexity of backward analysis. However, the modified design is more efficient than the RTCA design when a large number of intruders pass both the range and altitude test but fail the *Track-Firmness-Test* which makes them non-threatening.

- **Notation.** The RTCA design uses record structures extensively. For example, information related to an intruder is stored in an *ITF* record while the system parameters and the global variables are stored in the *P* and *G* records, respectively. Because unique field names are in the design, the modified PDL omits the record names.
- **Algorithmic Simplification.** Computational definitions of the various tests (e.g., altitude tests, tau computations, etc.) have been slightly simplified to make the example self-contained. This also enhances the understandability of the example by non-TCAS experts.
- **Intruder Logging.** The display of the intruder information on the pilot console is safety-critical because incorrect information may make a pilot issue a potentially hazardous command. Since all the modules within the *Threat-Detection* subsystem of the RTCA design are safety-critical, a safety-independent module logging the intruder information for post-flight analysis is introduced.

5.2 TCAS Design Description

The procedures *Threat-Detection* and *Setup-Parameters* are shown in Figure 5.2. An intruder is declared a hit when the results of the tests (i.e., *firm*, *zhit*, and *rhit*) are TRUE. The *Setup-Parameters* procedure initializes the following parameters:

tvpcmd: Max tau for vertical miss distance (VMD) calculation.

h1: Range—range rate hyperbola threshold.

trthr: Range tau threshold.

tvthr: Time-to-coaltitude threshold.

```
procedure Threat-Detection;
begin
  Setup-Parameters;
  hitflg := FALSE;
  Track-Firmness-Test (firm);
  if (firm) then
    Hit-or-Miss-Test (zhit, rhit);
    hitflg := zhit  $\wedge$  rhit;
    if hitflg then
      Log-Threat-Info;
    end if;
  end if;
end Threat-Detection;

procedure Setup-Parameters;
  tvpcmd := tvpctbl (lev); - - lev = max (index, plint)
  h1 := h1tbl (lev);
  if (itf.eqp = $TCAS) then
    trthr := trtetbl(lev); tvthr := tvtetbl(lev);
  else
    trthr := trtutbl(lev); tvthr := tvtutbl (lev);
  end if;
end Setup-Parameters;
```

Figure 5.2: Procedures *Threat-Detection* and *Setup-Parameters*

```

procedure Hit-or-Miss-Test (out zhit, rhit : boolean);
begin
    Hit-Test-Init;
    Compute-Tau (taur, trtru);
    Compute-VMD-HDM (vmd, hmd);
    Run-Altitude-Range-Test (zhit, rhit);
end Hit-or-Miss-Test;

procedure Hit-Test-Init;
    rz := zown - zint;
    rzd := zdown - zdint;
    a := |rz|;
    adot := rzd * sign (zdint);
    rdtemp := rd;
    if (rd ≥ - rdthr) then
        rdtemp := - rdthr;
    end Hit-Test-Init;

```

Figure 5.3: Procedures *Hit-Or-Miss-Test* and *Hit-Test-Init*

The values of these parameters are based on the higher of the sensitivity levels between the own and intruder aircraft. Once the parameters are decided, the *Track-Firmness-Test* examines the length and quality of the vertical tracking data. Upon passing the *Track-Firmness-Test*, the *Hit-or-Miss-Test* evaluates the range and the altitude tests in parallel.

The *Hit-or-Miss-Test* (Figure 5.3) performs its own initialization, calculates the tau values, calculates the vertical and horizontal miss distances, and invokes the range and the altitude test. The *Hit-Test-Init* procedure initializes the following:

rz: Relative altitude (i.e., own altitude minus intruder altitude).

```

procedure Compute-Tau is
begin
  tauv :=  $-\frac{a}{adot}$ ;
  trtru := max (mintau,  $-\frac{r}{rdtemp}$ ); - - true tau
  if (r > 0) then
    taur :=  $-\frac{r - \frac{dmod^2}{r}}{rdtemp}$ ;
  else
    taur := mintau;
  end if;
  taur := max (mintau, taur); - - modified tau
end Compute-Tau;

```

Figure 5.4: Procedure *Compute-Tau*

rzd: Relative altitude rate.

a: Absolute value of the relative altitude.

adot: Signed value of relative altitude rate.

rdtemp: Temporary variable for tau calculation (i.e., either tracked range rate *rd* or the negation of the system threshold on range rate *rdthr*).

The *Compute-Tau* procedure (Figure 5.4) determines the values of the true and modified tau, *trtru* and *taur*, respectively. TCAS uses the modified tau *taur* because simulations have shown that an intruder with slow horizontal or vertical closure rate can become dangerously close without crossing the true tau boundary[1]. The modified tau declares such intruders as threats earlier than the true tau can.

```

procedure Compute-VMD-HMD is
begin
  hmd := r + rd * tauv;
  vmd1 := rz + rzd * min (tvpcmd, trtru);
  vmd2 := rz + rzd * min (tvpcmd, taur);
  if (vmd1 * vmd2 ≤ 0) then
    vmd := 0;
  elsif (vmd1 > 0) then
    vmd := min (vmd1, vmd2);
  else
    vmd := max (vmd1, vmd2);
  end if;
end Compute-VMD-HMD;

```

Figure 5.5: Procedure *Compute-VMD-HMD*

The *Compute-VMD-HMD* procedure (Figure 5.5) determines the vertical and horizontal miss distances using the tau values supplied by the *Compute-Tau* procedure. The algorithmic definitions of the *Range-Test* and *Altitude-Test* tasks involve the examination of the encounter geometries as shown in Figure 5.6. The *Log-Threat-Info* procedure (Figure 5.7) is provide to assist data analysis after the flight.

5.3 TCAS Safety Analysis

Figure 5.8 shows the structure of the *Threat-Detection* procedure and data flows. Suppose that the safety requirements specify that the intruder located inside the protected airspace be declared a threat. The initial safety constraints can be

```

procedure Run-Altitude-Range-Test (out zhit, rhit : boolean);
task body Range-Test is
begin
  if (rd > rdthr) ∨ (taur ≥ trthr) ∨ (r > rmax) then
    rhit := (r ≤ dmod) ∧ (|r * rd| ≤ h1);
  else
    rhit := TRUE
  end if;
end Range-Test;
task body Altitude-Test is
begin
  zhit := FALSE;
  if (a < zthr) then
    zhit := |vmd| < zdthr;
  elsif (adot < zdthr) then
    zhit := (tauv < tvthr) ∧ ( |vmd| < zthr ∨ (|hmd| < dmod ∧ tauv < trtru) );
  end if;
end Altitude-Test;
begin
end Run-Range-Altitude-Test;

```

Figure 5.6: Procedure *Run-Range-Altitude-Test*

```

procedure Log-Threat-Info is
begin
  - - append the following threat info to the end of threat-log-file
  seek (threat-log-file, eof);
  save !TF.ID, taur, trtru, vmd, hmd, time-of-day
end Log-Threat-Info;

```

Figure 5.7: Procedure *Log-Threat-Info*

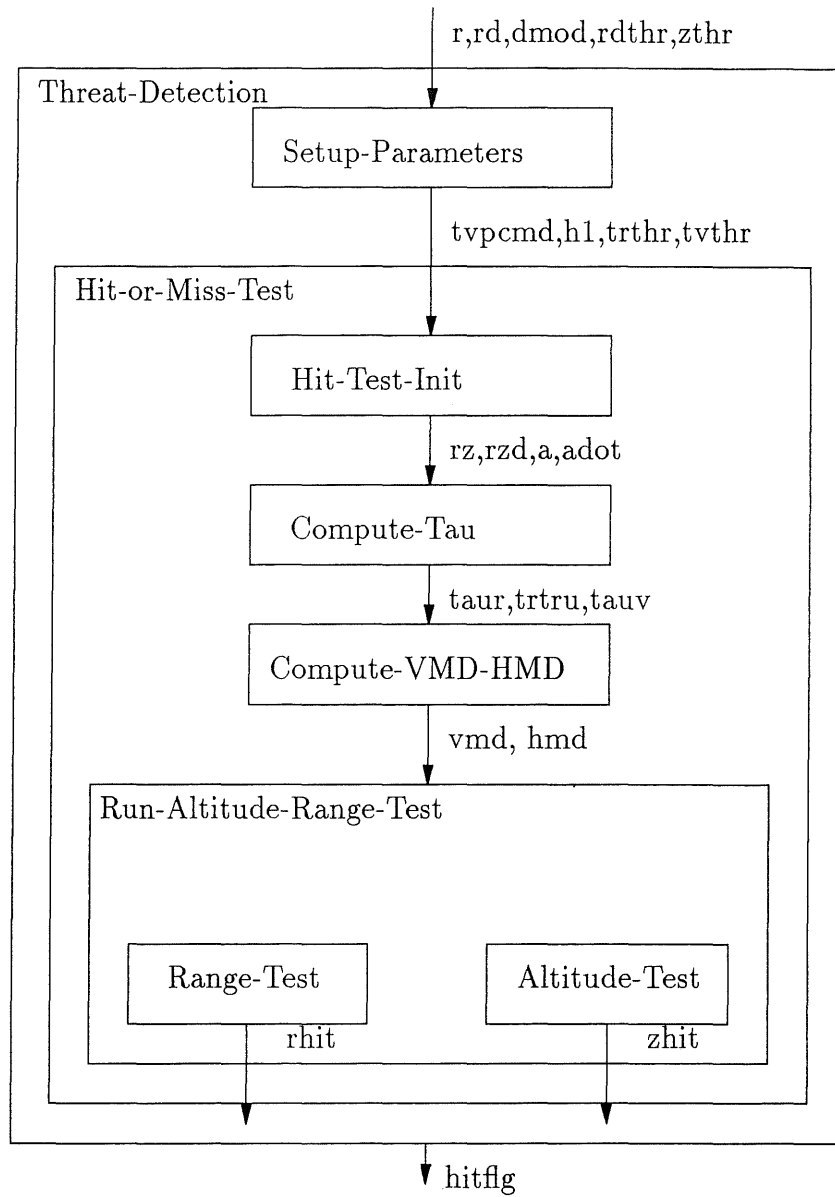
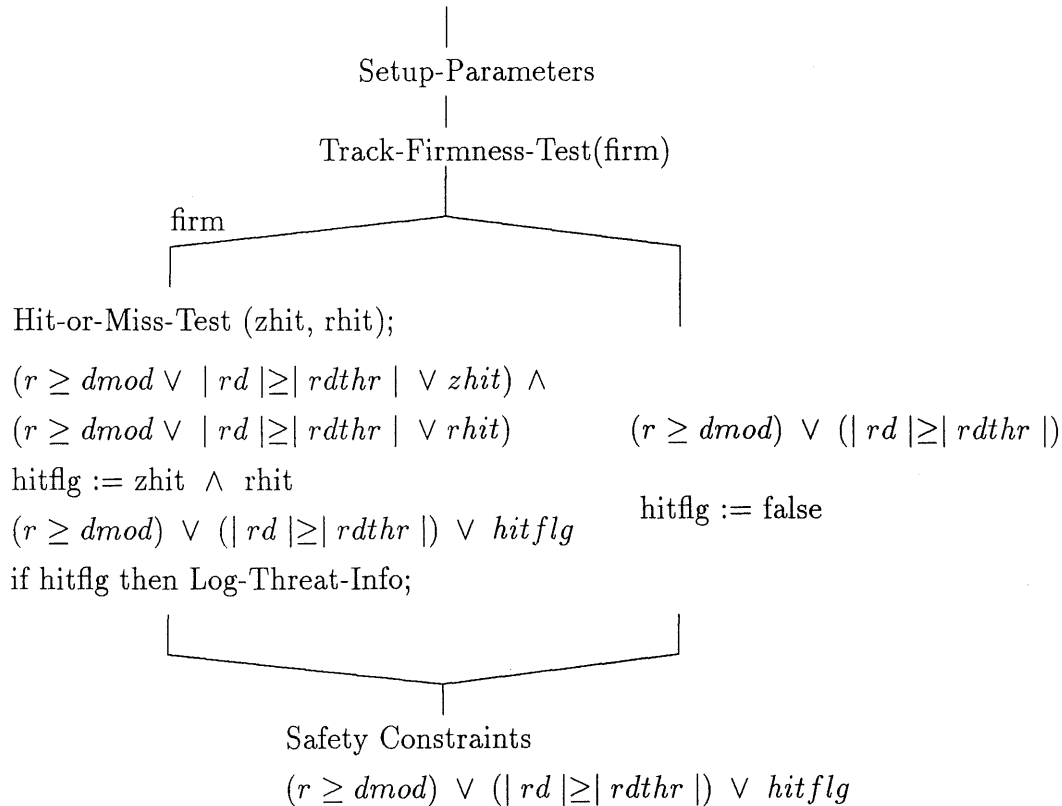


Figure 5.8: Structure of *Threat-Detection* Procedure

Figure 5.9: Control Flow of *Threat-Detection* Procedure

formally specified as:

$$\begin{aligned}
 & (r < dmod) \wedge (|rd| < |rdthr|) \rightarrow hitflg \\
 = & (r \geq dmod) \vee (|rd| \geq |rdthr|) \vee hitflg
 \end{aligned}$$

where `hitflg` is the output (i.e. TRUE means a threat) and the others are the inputs.³

Figure 5.9 shows how backward analysis is applied on the *Threat-Detection* procedure where the initial safety constraints and the intermediate weakest preconditions

³`Dmod` is actually a local variable within the *Threat-Detection* procedure in the RTCA design. Its value is `Dmodtbl(lev)` where `lev` is the maximum of the sensitivity level of the own and intruder aircraft.

are shown in italics. Backward analysis reveals that the *Log-Threat-Info* procedure is safety-independent because the weakest precondition is the same as the safety constraints. The procedure requires access to variables such as the intruder ID that we assume are available, the values of the true and the modified tau, and the vertical and horizontal miss distances. If any of these variables turn out to be safety-critical, they must be protected by a firewall so that the *Log-Threat-Info* procedure may not modify their values accidentally.

The safety constraints of the *Run-Range-Altitude-Test* procedure are:

$$(r \geq dmod \vee |rd| \geq |rdthr| \vee rhit) \wedge (r \geq dmod \vee |rd| \geq |rdthr| \vee zhit)$$

Therefore, the safety constraints of the *Range-Test* and the *Altitude-Test* become $(r \geq dmod \vee |rd| \geq |rdthr| \vee rhit)$ and $(r \geq dmod \vee |rd| \geq |rdthr| \vee zhit)$, respectively.

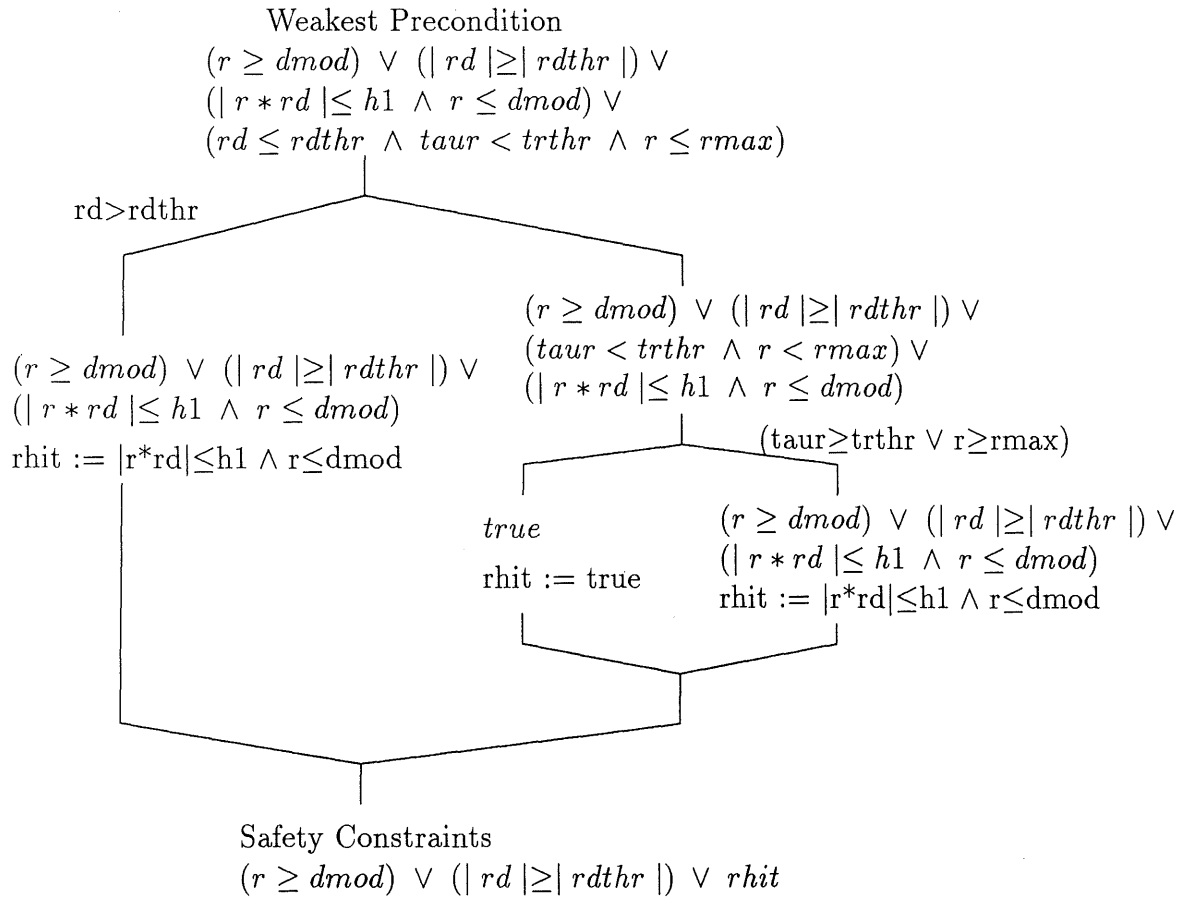
Figure 5.10 shows the continued application of backward analysis to the *Range-Test* task that results in the following weakest precondition:

$$\begin{aligned} &(r \geq dmod) \vee (|rd| \geq |rdthr|) \vee \\ &(|r * rd| \leq h1) \wedge (r \leq dmod) \vee \\ &(rd \leq rdthr \wedge taur < trthr \wedge r \leq rmax) \end{aligned}$$

The weakest precondition, given in disjunctive normal form, can be regarded as having several lines of defense toward the satisfaction of the safety constraints. For example, the truth of the predicate c_1

$$(r \geq dmod) \vee (|rd| \geq |rdthr|)$$

is based only on the input values. If the condition c_1 is TRUE, the *Range-Test* task is safe in a sequential execution environment in the absence of abnormal data or control

Figure 5.10: Control Flow of *Range-Test* Procedure

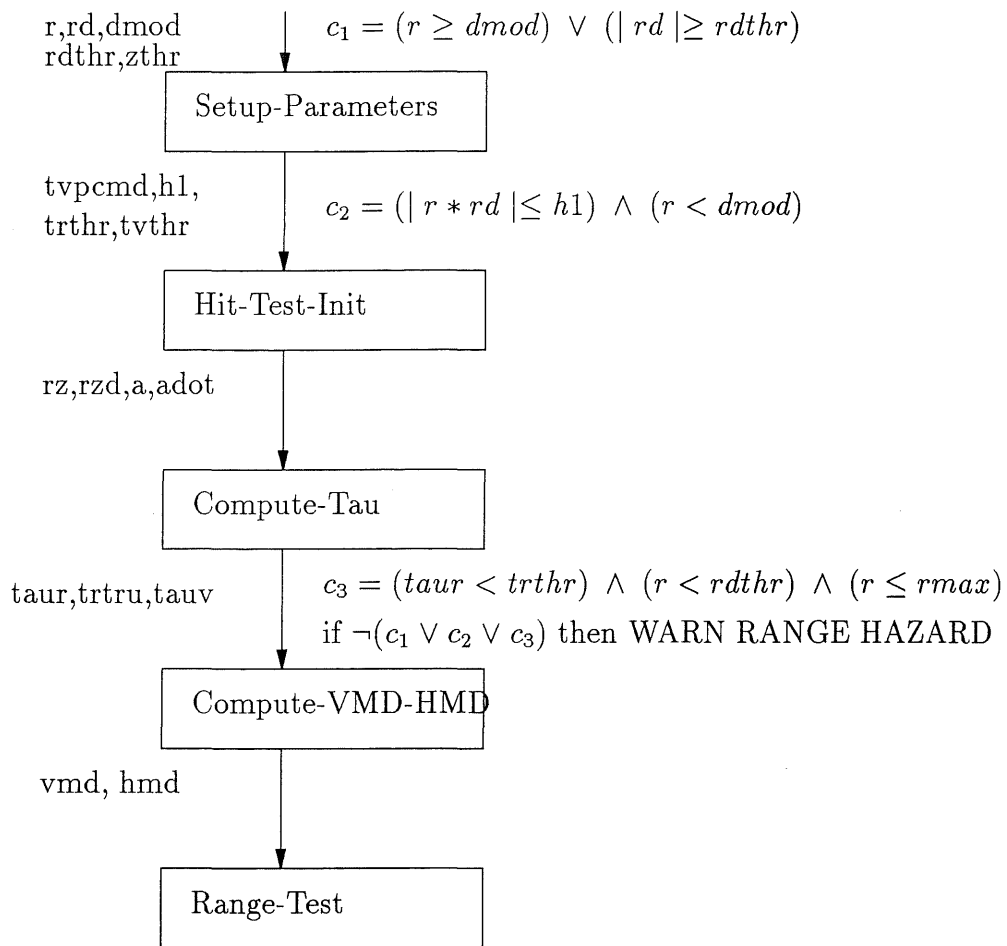


Figure 5.11: Propagation of WP from *Range-Test* Task

errors. If, on the other hand, the condition c_1 is FALSE, the condition c_2

$$(| r * rd | \leq h1) \vee (r \leq dmod)$$

is the second line of defense. Its predicate is expressed using variables internal or external to the routine. In such cases, the development of run-time assertions requires that either the values of the variables (i.e., r , $dmod$) or the predicate (i.e., $r < dmod$) be passed from one module to another. Similarly, the predicate c_3

$$(taur < trthr) \wedge (r < rdthr) \wedge (r \leq rmax)$$

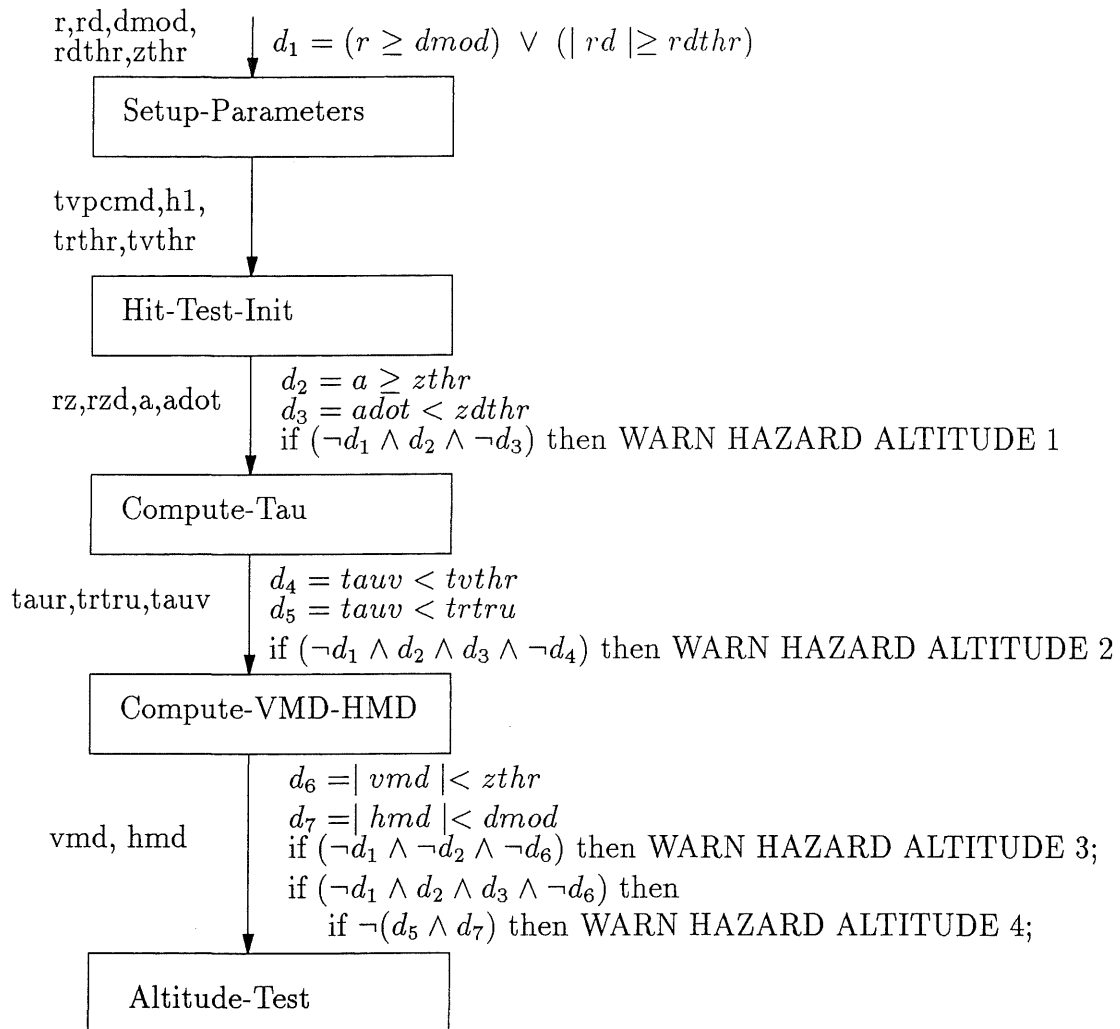
serves as the last line of defense toward the satisfaction of the *Range-Test* safety constraints. Figure 5.11 shows the assertions inserted to detect the occurrences of hazardous states.

Similarly, Figure 5.12 shows how the following weakest precondition of the *Altitude-Test* task is distributed as the safety constraints of other modules:

$$\begin{aligned} & (r \geq dmod \vee |rd| \geq |rdthr|) \vee \\ & \left(\begin{array}{l} (a \geq zthr \vee |vmd| < zdthr) \wedge \\ (a < zthr \vee adot < zdthr) \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} (tauv < tvthr) \wedge \\ (|vmd| < zthr \vee |hmd| < dmod) \wedge \\ (|vmd| < zthr \vee tauv < trtru) \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right) \\ = & d_1 \vee ((d_2 \vee d_6) \wedge (\neg d_2 \vee d_3) \wedge (\neg d_2 \vee \neg d_3 \vee (d_4 \wedge (d_5 \vee d_6) \wedge (d_6 \vee d_7)))) \end{aligned}$$

The truth of condition d_1 ensures the satisfaction of the safety constraints. Since the *Hit-Test-Init* may control the truth of only predicates d_2 and d_3 , one can consider the following cases:

In the first case, the assertions are introduced within the *Compute-VMD-HMD* procedure that are triggered when the condition $(\neg d_1 \wedge \neg d_2 \wedge \neg d_6)$ holds. Similarly,

Figure 5.12: Propagation of WP from *Altitude-Test* Task

$d_2 = \text{FALSE}$	$WP = d_6$
$d_2 = \text{TRUE}, d_3 = \text{FALSE}$	$WP = \text{FALSE}$
$d_2 = \text{TRUE}, d_3 = \text{TRUE}$	$WP = d_4 \wedge d_6 \wedge (d_5 \vee d_7)$

Table 5.1: Derivation of Run-Time Assertions

warnings of expected violations of the safety constraints of the *Altitude-Test* task are issued if the condition $(\neg d_1 \wedge d_2 \wedge \neg d_3)$ holds. Table 5.1 shows how to develop the assertions that detect the potentially hazardous software states for the last case.

The identification of the safety-critical modules and the derivation of their safety constraints are similarly applied to other procedures. For example, the safety constraints of the *Compute-VMD-HMD* procedure are:

$$(|vmd| < zthr) \wedge (|hmd| < dmod)$$

while that of the *Hit-Test-Init* procedure is:

$$(a < zthr) \vee (adot < zdthr)$$

The continued application of backward analysis reveals the safety-critical modules, their safety constraints, and the safety-critical variables.

The backward analysis technique is used during the software design phase for the following reasons:

- To identify the safety-critical modules and to derive their safety constraints during the high-level design phase.

- To verify the safety of the safety-critical modules in the detailed design phase and to derive the contents of the run-time assertions that make the software more robust against potentially hazardous internal states.

Despite the use of concurrency in the *Threat-Detection* procedure, concurrency safety verification is unnecessary since there are no safety-critical variables that are subject to race conditions. Not all concurrent executions of safety-critical modules require concurrency safety verification.

The analysis reveals that all the modules except the *Log-Threat-Info* are safety-critical and that the safety-critical variables `taur`, `trtru`, `vmd`, `hmd` are shared between the safety-critical modules and the safety-independent module.

5.4 An Improved TCAS Design

Upon the completion of the software design phase, a detailed design document whose safety can be certified must be produced. The safety-critical modules must be clearly identified as such and protected from the safety-independent modules to ensure the absence of hazardous side-effects. The following PDL descriptions show how the basic design of the *Threat-Detection* TCAS subsystem can be augmented with assertions to detect the occurrences – regardless of their causes – of the potentially hazardous software states.⁴ The augmented design also introduces several variables (e.g., `log-vmd`, `log-hmd`, etc) serving as a medium for unidirectional data flow from the safety-critical modules to the safety-independent module.

⁴The lines that are missing in or different from the less robust TCAS design are indicated by asterisks.

```
* package Detect-Threat-Critical is
*   procedure Threat-Detect; - - no longer main procedure
*   - - types definitions are assumed
*   distance log-vmd, log-hmd; - - for unidirectional data-flow
*   second log-taur, log-trtru;
*   boolean log-hitflg;
* end Detect-Threat-Critical;

* package body Detect-Threat-Critical is
*   procedure Threat-Detect;
*     begin
*       Setup-Parameters;
*       hitflg := FALSE;
*       Track-Firmness-Test (firm);
*       if (firm) then
*         Hit-or-Miss-Test (zhit, rhit);
*         hitflg := zhit  $\wedge$  rhit;
*       else
*         hitflg := FALSE;
*       end if;
*       log-hitflg := hitflg;
*     end Threat-Detect;
*     - - other procedures invisible to outside package go here
*     - - Compute-Tau is shown below as an example
* end Detect-Threat-Critical;
```

Figure 5.13: Firewall Installation

```

procedure Compute-Tau is
begin
  tauv :=  $-\frac{a}{\text{adot}}$ ;
  trtru := max (mintau,  $-\frac{r}{\text{rdtemp}}$ ); - - true tau
  if (r > 0) then
    taur :=  $-\frac{r - \frac{\text{dmod}^2}{r}}{\text{rdtemp}}$ ;
  else
    taur := mintau;
  end if;
  taur := max (mintau, taur); - - modified tau
  *   c3 := taur < trthr  $\wedge$  r < rdthr  $\wedge$  r  $\leq$  rmax;
  *   d4 := tauv < tvthr;
  *   d5 := tauv < trtru;
  *   if  $\neg$  (c1  $\vee$  c2  $\vee$  c3) then
  *     WARN RANGE HAZARD; - - place recovery routine if desired
  *   end if;
  *   if ( $\neg$  d1  $\wedge$  d2  $\wedge$  d3  $\wedge$   $\neg$ d4) then
  *     WARN ALTITUDE HAZARD 2
  *   end if;
  *   log-taur := taur; - - assign values for unidirectional data-flow
  *   log-trtru := trtru;
end Compute-Tau;

```

Figure 5.14: Safety-Critical Module Augmentation with Run-Time Assertions

```
* procedure Main is
* begin
*   Threat-Detect;
*   if (log-hitflg) then
*     Log-Threat-Info;
*   end if
* end Main;

procedure Log-Threat-Info is
begin
  - - append the threat info to the end of threat-log-file
  seek (threat-log-file, eof);
*   save ITF.ID, log-aur, log-trtru, log-vmd, log-hmd, time-of-day
end Log-Threat-Info;
```

Figure 5.15: Main Procedure and Safety-Independent Module

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This dissertation proposes a safety-oriented design method whose goal is to minimize the amount of safety-critical code and to produce a design whose safety can be certified. Backward analysis, which was used to verify safety of source code, is used to guide the software design process. Design hazard analysis allows the analyst to augment the design being developed with run-time assertions or rendezvous, as appropriate, to prevent the occurrence of hazardous software states. It is also shown that a limited backtracking can be applied to resolve safety constraint conflicts upon the detection of unsafe modules.

Information hiding is recommended as a means of protecting the safety-critical modules from the others so that the safety verification needs to be applied only on the former. Safety verification is a labor-intensive and costly process. While firewalls alone do not necessarily make software safer, they minimize the cost of safety verification during the development and maintenance phases.

This dissertation also proposes an incremental and selective verification technique that further reduces the complexity of design safety verification. It argues that

concurrency decisions on safety-critical software must be based on careful trade-off analysis. The dissertation proposes a criterion where the use of concurrency in the safety-critical software development can be justified. It also argues that a concurrent design does not necessarily require exhaustive concurrency safety verifications.

The management aspect of software safety is as important as the technical one. This dissertation examines how to organize safety-critical projects and to distribute safety responsibilities.

While the design of safety-critical software remains a challenging task, this dissertation provides useful guidelines on how the design activity can be organized around the goal of enhancing safety.

6.2 Future Work

Safety-critical software has direct and significant impact on the lives of the general public. The following research is in order:

- **Industrial application and evaluation.** The safety-oriented method proposed in this dissertation is practical and seems applicable on large industrial projects as demonstrated on a subsystem of the TCAS II design. The application of the method to various industrial projects would reveal its applicability and scalability. The method is currently being applied on an air traffic control system being developed.
- **Software Safety Requirements Derivation and Analysis.** The application of the safety-oriented design method proposed in this dissertation can

produce a design whose safety can be certified. However, a design is safe only when the software safety requirements are correctly derived.

Software safety requirements are derived by performing system hazard analysis such as fault tree analysis and Petri Net analysis[34]. The success of fault tree analysis heavily depends on the capability of analysts due to its informal nature. While Petri Net analysis is useful in determining how a system might fail, its applicability on large industrial systems is not yet proven. It is also unclear how fail-soft behavior of a system can be modeled in Petri Nets.

- **Safe design derivation.** This dissertation demonstrates how a decomposition proposed by the designer can be evaluated from safety viewpoints. With increasing use of formal specification languages, it may be possible to mechanically derive a safe design from the requirements specification.
- **Programming language constructs.** Further works are necessary in guiding safety-critical software design activities and in enhancing safety through programming language constructs.

Bibliography

- [1] Federal Aviation Administration. Introduction to TCAS II. U.S. Department of Transportation, 1990.
- [2] G.S. Avrunin, L.K. Dillon, J.C. Wilenden, and W.E. Riddle. Constrained Expressions: Adding Analysis Capabilities to Design Methods for Concurrent Software Systems. *IEEE Transaction on Software Engineering*, SE-12(2):278–292, February 1986.
- [3] G. Booch. *Software Engineering with Ada*. Benjamin/Cummings, 1986.
- [4] S.S. Cha, N.G. Leveson, and T.J. Shimeall. Safety Verification in Murphy using Fault Tree Analysis. In *Proceedings of the 10th Interational Conference on Software Engineering*, pages 377–386, Raffle City, Singapore, April 1988.
- [5] D. Denning. *Cryptography and Data Security*. Addison Wesley, 1982.
- [6] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [7] D.E. Eckhardt Jr. and L.D. Lee. A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors. *IEEE Transaction on Software Engineering*, SE-11(12):1511–1517, December 1985.
- [8] R.E. Fairley. *Software Engineering Concepts*. McGraw-Hill Book Company, 1985.
- [9] T. Forester and P. Morrison. Computer Unreliability and Social Vulnerability. *Futures*, pages 462–474, June 1990.

- [10] P. Freeman and A.I. Wasserman, editors. *Tutorial on Software Design Techniques*. IEEE Computer Society, fourth edition, 1983.
- [11] J. Gray. A Census of Tandem System Availability between 1985 and 1990. *IEEE Transaction on Reliability*, 39(4):409–418, October 1990.
- [12] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.
- [13] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shuti-Trauring. Statemate: A Working Environment for the Development of Complex Reactive Systems. In *Proceedings 10th International Conference on Software Engineering*, pages 396–406, Raffle City, Singapore, April 1988.
- [14] D. Harel and A. Pnueli. On the Development of Reactive Systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, 1985.
- [15] V.H. Hasse. Real-Time Behavior of Programs. *IEEE Transaction on Software Engineering*, SE-7(9):494–501, September 1981.
- [16] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, October 1969.
- [17] R.K. Iyer and P. Velardi. Hardware-related Software Errors: Measurement and Analysis. *IEEE Transaction on Software Engineering*, SE-11(2):223–231, February 1985.
- [18] M. Jackson. *Principles of Program Design*. Academic Press, 1975.
- [19] J. Jacky. Programmed for Disaster: Software Errors that Imperil Lives. *The Sciences*, 29(5):22–27, September/October 1989.

- [20] M.S. Jaffe and N.G. Leveson. Completeness, Robustness, and Safety in Real-Time Software Requirements Specification. In *11th International Conference on Software Engineering*, pages 302–311, Pittsburgh, Pennsylvania, May 1989. IEEE Computer Society Press.
- [21] M.S. Jaffe, N.G. Leveson, M. Heimdahl, and B.E. Melhart. Software Requirements Analysis for Real-Time Process-Control Systems. *IEEE Transaction on Software Engineering*, 17(3):241–258, March 1991.
- [22] F. Jahanian and A.K. Mok. Safety Analysis of Timing Properties in Real-Time Systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.
- [23] F. Jahanian and A.K. Mok. A Graph-Theoretic Approach for Timing Analysis and Its Implementation. *IEEE Transactions on Computers*, C-36:961–975, August 1987.
- [24] E. Joyce. Software Bugs: A Matter of Life and Liability. *Datamation*, pages 88–92, 15 May 1987.
- [25] J.C. Knight and N.G. Leveson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transaction on Software Engineering*, SE-12:96–109, January 1986.
- [26] J.C. Knight and N.G. Leveson. A Reply to the Criticisms of the Knight & Leveson Experiment. *Software Engineering Notes*, 15(1):24–35, January 1990.
- [27] N.G. Leveson. Software Safety: What, Why, and How. *ACM Computing Surveys*, 18(2):125–164, June 1986.
- [28] N.G. Leveson. Software Safety in Embedded Systems. *Communications of the ACM*, February 1991.

- [29] N.G. Leveson, S.S. Cha, T.J. Shimeall, and J.C. Knight. The Use of Self-Checks and Voting in Software Error Detection: An Empirical Study. *IEEE Transaction on Software Engineering*, SE-16(4), April 1990.
- [30] N.G. Leveson and P.R. Harvey. Analyzing Software Safety. *IEEE Transactions on Software Engineering*, SE-9(5):569–579, September 1983.
- [31] N.G. Leveson and T.J. Shimeall. Safety Assertions for Process Control Systems. In *Proc. 13th International Symposium on Fault Tolerant Computing*, Milan, Italy, 1983.
- [32] N.G. Leveson, T.J. Shimeall, J.L. Stolzy, and J.C. Thomas. Design for Safe Software. In *AIAA '83*, 1983.
- [33] N.G. Leveson and J.L. Stolzy. Safety Analysis of Ada Programs using Fault Trees. *IEEE Transactions on Reliability*, R-32(5):479–484, December 1983.
- [34] N.G. Leveson and J.L. Stolzy. Safety Analysis Using Petri Nets. *IEEE Transaction on Software Engineering*, SE-13(3):386–397, March 1987.
- [35] D.L. Long and L.A. Clarke. Task Interaction Graph for Concurrency Analysis. In *11th International Conference on Software Engineering*, pages 44–52, Pittsburgh, Pennsylvania, May 1989. IEEE Computer Society Press.
- [36] Data Encryption Standard. National Bureau of Standards, 1977.
- [37] Airbus A320, The New Generation Aircraft. *Aviation Weekly & Space Technology*, pages 45–66, February 2, 1987.
- [38] *Interim Defense Standard 00-55 (Draft): Requirements for the Procurement of Safety Critical Software in Defense Equipment*. United Kingdom Ministry of Defense, May 1989.

- [39] J.W. McIntee Jr. Fault Tree Techniques as Applied to Software (Soft Tree). Technical report, USAF, March 1983.
- [40] H. Mills. Chief Programmer Teams: Principles and Procedures. IBM Federal Systems Division, 1971.
- [41] P.G. Neumann. Some Computer-Related Disasters and Other Egregious Horrors. *ACM Software Engineering Notes*, 10(1):6-7, January 1985.
- [42] D.L. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Communications of the ACM*, pages 1053-1058, December 1972.
- [43] D.L. Parnas. A Technique for Software Module Specification with Examples. *Communications of the ACM*, 15(5):330-336, May 1972.
- [44] D.L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transaction on Software Engineering*, SE-5(2):128-138, March 1979.
- [45] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.
- [46] R.S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Book Company, second edition, 1987.
- [47] D. Reifer, editor. *Tutorial: Software Management*. IEEE Computer Society, third edition, 1986.
- [48] J.C. Rouquet and P.J. Traverse. Safe and Reliable Computing on Board the Airbus Aircraft. In W. Quirk, editor, *SafeComp'86*, pages 93-97. IFAC, Pergamon Press, October 1986.
- [49] RTCA/DO-185. Minimum Operational Performance Standards for TCAS Airborne Equipment. Technical report, Radio Technical Commission for Aeronautics, 1989.

- [50] J. Rushby. Kernels for Safety. In *Proc. CSR Workshop on Safety and Security*, Glasgow, Scotland, October 1986.
- [51] J.R. Taylor. Fault Tree and Cause Consequence Analysis for Control Software Validation. Technical Report RISO-M-2326, RISO National Laboratory, January 1981.
- [52] R.N. Taylor. A General-Purpose Algorithm for Analyzing Concurrent Programs. *Communications of the ACM*, 26(5):362–376, May 1983.
- [53] W.E. Vesely, F.F. Goldberg, N.H. Roberts, and D.F. Haasl. *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, Washington, D.C., January 1981. NUREG-0492.
- [54] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, April 1971.
- [55] S.S. Yau and R.C. Cheung. Design of Self-Checking Software. In *Int. Conf. on Reliable Software*, pages 450–457, April 1975.
- [56] S.S. Yau and J.J.P. Tsai. A Survey of Software Design Techniques. *IEEE Transaction on Software Engineering*, SE-12(6):713–721, June 1986.