SAFETY VERIFICATION CONDITIONS FOR SOFTWARE-INTENSIVE CRITICAL SYSTEMS

by

KEN WONG

B.Sc.(Physics), The University of Saskatchewan, 1985
M.Sc.(Physics), The University of Saskatchewan, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October 1998

Abstract

This dissertation proposes an approach to generating “safety verification conditions” (SVCs) that improves upon the accuracy and thoroughness of approaches that rely primarily on engineering judgment. This approach, “Verification Tree Method” (VTM), is part of an overall system safety engineering process intended to eliminate or mitigate hazards in the development of a software-intensive critical system. VTM carried out to the level of a “black box” view of the system results in a set of system safety requirements. VTM can also be used to derive SVCs at the software component and the source code levels. The resulting SVCs can be used as input into the corresponding level of testing. VTM is based upon Fault Tree Analysis (FTA). Like FTA, VTM involves tracing a given hazard backwards through the system to cover all the ways in which a hazard can occur. VTM supplements FTA with a constrained syntax and “proof-by-contradiction” style reasoning to support the systematic derivation of key safety-related temporal relationships. The result of the analysis is an informal, rigorous safety argument that provides greater confidence that the SVCs, if satisfied, will be sufficient to mitigate the hazard. This informal argument can then be validated with a formal verification technique. VTM is illustrated in this dissertation with a (hypothetical) chemical factory information system.

Table of Contents

Abstract i

Table of Contents ii

Table of Figures iv

Acknowledgments v

1. Introduction 1

2. System Safety Engineering Process 5

2.1 System Safety 5
- 2.1.1 System Safety Terminology 5
- 2.1.2 System Safety Engineering Tasks 6
- 2.1.3 Safety Documentation 8

2.2 System Safety Engineering Process for Software-Intensive Systems 9
- 2.2.1 Hazard Identification 11
- 2.2.2 Safety System Analysis 12
- 2.2.3 Safety Requirements Analysis 12
- 2.2.4 Safety Design Analysis 13
- 2.2.5 Safety Code Analysis 13
- 2.2.6 Safety User Interface Analysis 13
- 2.2.7 Safety Verification 14
- 2.2.8 Safety Operational Analysis 14

3. Example - Chemical Factory Information System 15

3.1 System Description 15

3.2 Functional Requirement 15

3.3 Safety-Related Hazard 16

3.4 Preliminary Hazard Analysis 16

4. Safety Verification Conditions 19

4.1 Safety as a Distinct Property 19

4.2 System and Environmental Assumptions 20

4.3 Semantic Gap 21

4.4 Techniques for Generating SVCs 22
- 4.4.1 Fault Tree Analysis 22
- 4.4.2 Generating Temporal Relationships 24

5. Verification Tree Method 27

5.1 Constrained Syntax 28
- 5.1.1 Basic Notational Elements 28
- 5.1.2 Scenarios 29
- 5.1.3 Timing Constraints 30

5.2 Mathematical-Style Reasoning 30

6. Chemical Factory: Verification Tree Method 33

6.1 Initial Conjecture 33

6.2 Part 1: Corrupt Temperature Value 34

6.2.1 Step 1 36

6.2.2 Step 2 37

6.2.3 Step 3 38

6.3 Temporal Relationships 39

6.4 Part 2: Stale Temperature Value 40

6.4.1 Step 4 41

6.4.2 Step 5 42

6.4.3 Step 6 43

6.5 System level SVCs 44

7. Discussion of Results 47

7.1 Relationship to Fault Tree Analysis 47

7.1.1 Comparison to FTA 47

7.1.2 Comparison to SFTA 48

7.2 Deriving Temporal Relationships 48

7.2.1 Constrained Syntax 48

7.2.2 Mathematical Reasoning 49

7.2.3 Temporal Relationships 49

8. Formalization 53

8.1 Formal Specification 53

8.2 Formal Validation 57

9. Verifiable Code Assertions 59

9.1 Process for the Development of Verifiable Code Assertions 60

9.2 Component level SVCs 61

9.3 Source Code Level SVCs 62

9.4 Code Assertions 62

9.4.1 Pre- and Post-Conditions 62

9.4.2 SPARK Annotations 63

10. Future Work 65

10.1 Part 1: Hazard Refinement 67

10.2 Part 2: Model Extraction 68

10.3 Part 3: Verification 69

11. Conclusions 71

Bibliography 73

Appendix A. Formalized Chemical Factory SVCs 75

Table of Figures

Figure 1: The relationship of hazards to accidents. **6**

Figure 2: System lifecycle. **7**

Figure 3: Safety analyses for an iterative system development process. **11**

Figure 4. System Level Fault tree analysis of the chemical factory hazard. **23**

Figure 5: Graphical representation of the rigorous argument. **31**

Figure 6: Graphical representation of the first part of the analysis. **36**

Figure 7: Timeline of Events. **39**

Figure 8: Graphical representation of the second part of the analysis. **41**

Figure 9: Safety verification process. **66**

Figure 10: Structure of the Safety Verification Case. **67**

Acknowledgments

I would like to thank Bruce Elliott for playing the gracious host when I visited Praxis Critical Systems in Bath, UK. He also provided valuable feedback on my thesis and papers, and was a source of many keen insights into the practice of system safety engineering. I would also like to thank Raytheon Systems Canada Ltd. for the opportunity to visit their facilities and to learn from their software development practices. I was partly sold on my thesis topic by the promise that I would be able to study aspects of the Canadian Automated Air Traffic System (CAATS), a state-of-the-art object-oriented software development project which is being developed by Raytheon. During my time there, both Jim Ronback and Daniel Chang were supportive of my efforts to learn more about the company's system safety engineering practices. My decision to do research in an area of software engineering was partly inspired by the work I did with Carolyn Wick (a fellow UBC graduate student) on a number of projects. Being carried along by her enthusiasm, vision, and demanding standards, created a lasting interest in software engineering, an awareness of my own capabilities, and more than a few grey hairs. I was then fortunate enough to find a supervisor, Jeff Joyce, with similar qualities. He brought out of me my very best work, with encouragement, deep insights and stimulating discussions.

The work described in this dissertation is a result of a collaborative industry/university research project sponsored by the BC Advanced Systems Institute, Raytheon Systems Canada Ltd., and MacDonald Detwiller. The UBC Faculty of Graduate Studies was the source of a Graduate Student Travel Grant. I would like to thank each of these institutions for their support.

1. Introduction

This dissertation proposes an approach to generating “safety verification conditions” (SVCs) that improves upon the accuracy and thoroughness of approaches that rely primarily on engineering judgment. This approach, “Verification Tree Method” (VTM), supplements Fault Tree Analysis (FTA) with a constrained syntax and mathematical reasoning for the purpose of generating SVCs. In particular, VTM provides a mechanism for the systematic derivation of safety-related temporal relationships that are not easily derived with FTA and other traditional approaches.

This dissertation introduces the term “safety verification conditions” to represent all the necessary system safety conditions and environmental assumptions required for system safety verification. For example, a SVC for a railroad crossing gate controller might require that it detect any trains approaching the crossing, and that it lower the gate before a train reaches the crossing. Other SVCs might involve environmental assumptions about the trains and track sensors.

Though safety-related requirements will exist in the system requirements specification, it is important that an explicit set of SVCs are derived. The set of SVCs must be sufficient to eliminate or mitigate the identified hazards. In general, relying on the existing requirements specification will not be sufficient for ensuring the safety of the system.

For software-intensive critical systems, the derivation of SVCs must bridge the “semantic gap” between the definition of the hazard and the software. The hazard definition typically uses domain or end-user terminology to describe a relationship between the external effects of the system and environment. The SVCs derived from the hazard must be appropriate for the given stage in the software development. During the functional specification of the system, the SVCs are defined in terms of the environment and the system inputs and outputs. During system design, the SVCs are defined in terms of the system components. During implementation, the SVCs are defined in terms of the source code. The semantic gap is particularly acute between the high-level description of the hazard and the source code implementation of the system.

Traditionally, SVCs are constructed in an *ad hoc* fashion from the results of the hazard analysis. A more systematic approach involves the derivation of SVCs from a FTA [VGRH81] of each hazard. FTA begins with the hazard as the "top event" and then determines intermediate fault events that cause the hazard. These fault events are combined with the logical operators "AND" and "OR". The analysis continues until the basic fault events are determined. SVCs can then be constructed from the basic fault events.

However, for real-time systems, there will usually exist time-dependent hazards that involve subtle relationships between various temporal constants. The derivation of SVCs from these hazards will involve analysis of the dynamic behavior of the system and the temporal ordering of events which are not easily captured with FTA.

VTM extends traditional FTA in a number of different ways in order to generate SVCs from hazards that involve complex temporal relationships. VTM uses a constrained syntax to identify unknown temporal quantities and to expose the logical structure of the temporal relationships (e.g., A > B OR A <= B). The branches of the fault tree are then partly based on the "OR" structure of the temporal relationships. In addition to the propositional logic used in FTA, VTM includes the use of logical deduction rules based upon predicate logic. VTM also involves annotating the fault tree with SVCs to steer the direction of the analysis. The result of the analysis is an informal, rigorous safety argument that provides confidence that the SVCs are sufficient to mitigate the hazard.

The use of a constrained syntax and mathematical reasoning in VTM is influenced by formal methods. Though VTM does not require the use of formal techniques, it may be useful to formally specify the SVCs or to validate the informal safety argument by means of a formal verification technique.

The advantages of VTM over FTA are illustrated in this dissertation by means of a hypothetical chemical factory information system. This example has similarities with other safety-critical information systems such as Air Traffic Management (ATM) systems [ER96]. The application of VTM to the chemical factory example leads to the discovery of several system level SVCs, including some conditions that are not apparent from FTA.

VTM can also be used to derive both component level and source code level SVCs for the chemical factory. The source code level SVCs can be in the form of verifiable code assertions. It may then be possible to verify the assertions with a code verification tool.

VTM provides a systematic approach to the derivation of SVCs from system hazards. VTM may not appropriate for all hazards, especially those without any temporal complexity. The approach is particularly effective for hazards with non-trivial temporal relationships.

The following chapter gives an overview of system safety engineering and presents an example of a system safety process suitable for software-intensive systems. Chapter 3 introduces the chemical factory information system example. The role SVCs play in the safety engineering process is presented in Chapter 4, along with some techniques for generating SVCs. Chapter 5 describes the key elements of VTM. VTM is then used in Chapter 6 to generate SVCs for the chemical factory. A discussion of the results, and a comparison between the FTA and VTM approaches, is found in Chapter 7. Chapter 8 describes the formalization of the rigorous safety argument. The refinement of the system level SVCs into source code level SVCs is outlined in Chapter 9. Future work towards incorporating VTM into a comprehensive safety verification process is described in Chapter 10. Chapter 11 contains some conclusions.

2. System Safety Engineering Process

This dissertation focuses on the generation of SVCs as part of an overall system safety engineering process. This section introduces the concept of system safety and presents a system safety engineering process for software-intensive systems.

2.1 System Safety

System safety engineering is a matter of reducing the risk of "bad" things occurring (accidents) by eliminating or controlling states of the system (hazards) that may contribute to the "bad" things occurring. Examples of hazards include aircraft violating minimum separation standards, and gates failing to lower when trains approach a traffic crossing. Accidents and hazards are some of the key concepts in a safety engineering program. They are defined more precisely in the next section.

2.1.1 System Safety Terminology

Though terms such as accidents and hazards will have an obvious intuitive meaning for many people, it is important to have more precise definitions for safety engineering purposes. For one thing, careful understanding of these terms will help clarify what it means to mitigate hazards. However, there are no universally agreed upon meanings for these terms in the system safety field. Leveson [Lev95] provides the following definitions which are based upon a review of the traditional use of the terms in the system safety literature:

- An **accident** is an undesired and unplanned event that results in a specified level of loss. Types of loss include the loss of life, injury or damage to property.

- A **hazard** is a state or set of conditions of a system that, together with other conditions in the environment of the system, will lead inevitably to an accident.

These definitions of accidents and hazards attempt to capture the aspects of the system that are most directly relevant to safety. A clear distinction is drawn between the contributions of the system and the contributions of the environment to an accident. As well, these definitions ensures that there exists a sequence of events in the environment leading from the hazard to an accident. As a result, states of the system that may appear undesirable, but cannot lead to an accident, will not be considered hazardous. These definitions provides a clear focus for the system safety engineering activities which is the management of hazards.

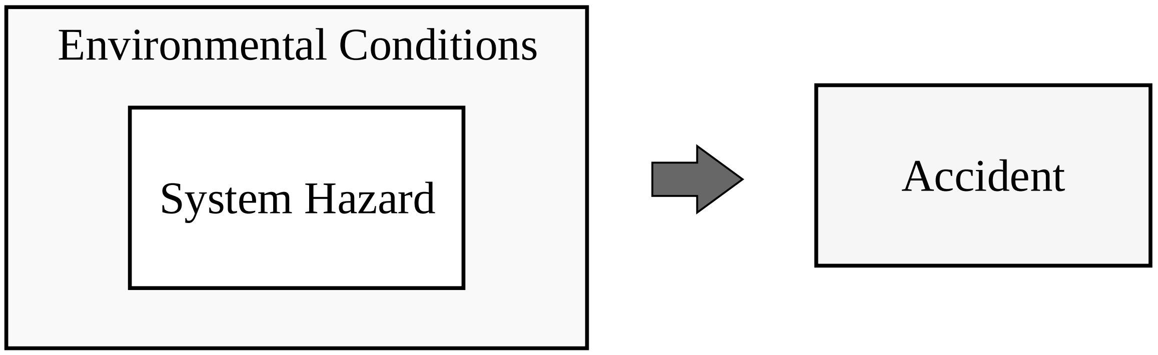


Figure 1: The relationship of hazards to accidents.

2.1.2 System Safety Engineering Tasks

The focus of safety engineering is the management of hazards. The safety engineering tasks includes identifying hazards, determining hazard causes and scenarios, performing risk assessments of the hazards and the overall system, and eliminating or controlling the hazards. The term "hazard analysis" is sometimes used to designate the various activities designed to identify, analyze, assess and resolve the hazards [IW95]. In this dissertation, hazard analysis means the identification of hazard causes, while the term "safety analysis" [IEEE94] is used to describe the various safety-related activities associated with each phase of the safety engineering process.

In general, system safety engineering tasks are conducted throughout the system's existence. There are five basic stages [Lev95] to the system lifecycle: conceptual development, design, full-scale development, production and deployment, and operation. Figure 2 depicts the system lifecycle. Hazards are identified and analyzed during initial conception of the system. Hazard identification, analysis, assessment and mitigation occurs throughout system development, deployment and operation until the system is decommissioned.

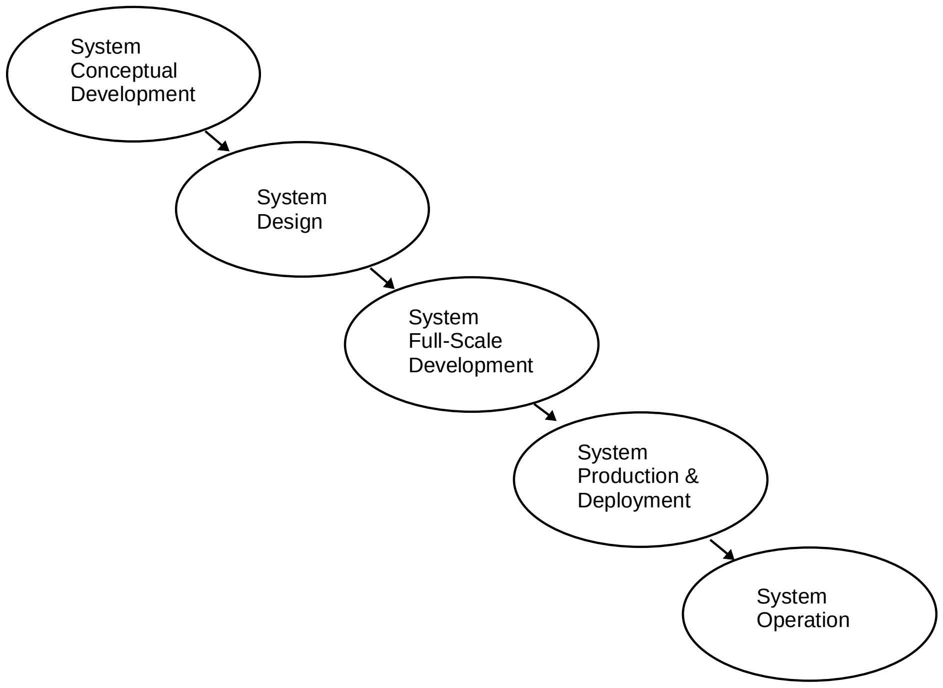

System Conceptual Development
System Design
System Full-Scale Development
System Production & Deployment
System Operation

Figure 2: System lifecycle.

There are a number of hazard analysis techniques [IW95] designed to uncover system faults that contribute to hazards. Some of the traditional system safety techniques include Hazard and Operability Studies (HAZOP), Failure Modes And Effects Analysis (FMEA) and Fault Tree

Analysis (FTA).

2.1.3 Safety Documentation

There are a number of documents that are produced by the system safety engineering process. The **Hazard List** records the hazards initially uncovered during the hazard identification stage. A set of safety requirements is maintained in a **System Safety Requirements Specification** document. The progress of the safety process is recorded in the **Hazard Log** which maintains an audit trail of hazard identification, analysis, assessment, and mitigation.

A number of safety-related standards and guidelines mandate, recommend or suggest the production of a detailed report on the safety of a software system with safety-related functionality. This system safety report is sometimes called a **Safety Case** [Bis98]. The safety case is likely to span the entire safety engineering process and includes a summary of the results of various process steps. Ideally, the safety case will provide an argument in support of system certification. Although it may be a less desirable outcome, it is possible that the safety case will provide evidence which supports an argument against system certification given certain assumptions about the environment.

An important aspect of the safety case will be an audit trail of the hazards. The hazard causes and scenarios should be traceable to the requirements, software components, and the source code. The audit trail will include the hazard mitigation efforts made at each level, such as the safety requirements, the safety design decisions and constraints, as well as safety code assertions. The safety case will also record the assessment of the effectiveness of the hazard mitigations.

2.2 System Safety Engineering Process for Software-Intensive Systems

The system safety engineering process presented in this section is appropriate for the iterative and incremental development of a software-intensive system with an object-oriented architecture. The system safety engineering process has some similarities to the process [ER96] used in the development of the Canadian Automated Air Traffic System (CAATS) by Raytheon System Canada Ltd. The chemical factory information system described in Section 3 follows such a safety engineering process. The subsequent derivation of SVCs takes place in the context of this process.

System safety engineering processes are well established for the development of critical systems such as a chemical plants, aerospace and defense systems, and nuclear power plants. However, the system safety methodologies were originally developed for systems without substantial, or any, computer control. With the increase use of computer control in safety-critical systems, software should be included in the safety engineering process.

The focus of this section is on software aspects of software-intensive systems. The word “system” will be used to mean the software system.

Software development often involves different design techniques and processes such as object-oriented methods [Boo94] and iterative lifecycle development processes [Boe88]. For example, the development of object-oriented software systems is typically produced in a number of different iterations that implement various “slices” of functionality. Each iteration may involve the incremental development of the requirements, design and implementation.

Safety engineering techniques must be adapted to the types of designs and lifecycle processes used in the software development. For example, a software iteration may span the system design and full-scale development stages. The safety engineering process should be integrated with this type of development lifecycle.

The system safety engineering phases during each stage of the system lifecycle includes the following safety analyses:

1) System Conceptual Development Stage
    - Hazard Identification
    - Safety System Analysis
2) System Design and Full-Scale Development Stages
    - Safety Requirements Analysis
    - Safety Design Analysis
    - Safety Code Analysis
    - Safety User Interface Analysis
3) System Production and Deployment Stage
    - Safety Verification
4) System Operation Stage
    - Safety Operational Analysis

As shown in Figure 3, safety analyses of the requirements, design and code are performed throughout each software iteration of system design and full-scale development. The analysis performed at each level consists of hazard identification, assessment and mitigation, as discussed in Section 2.1.2.

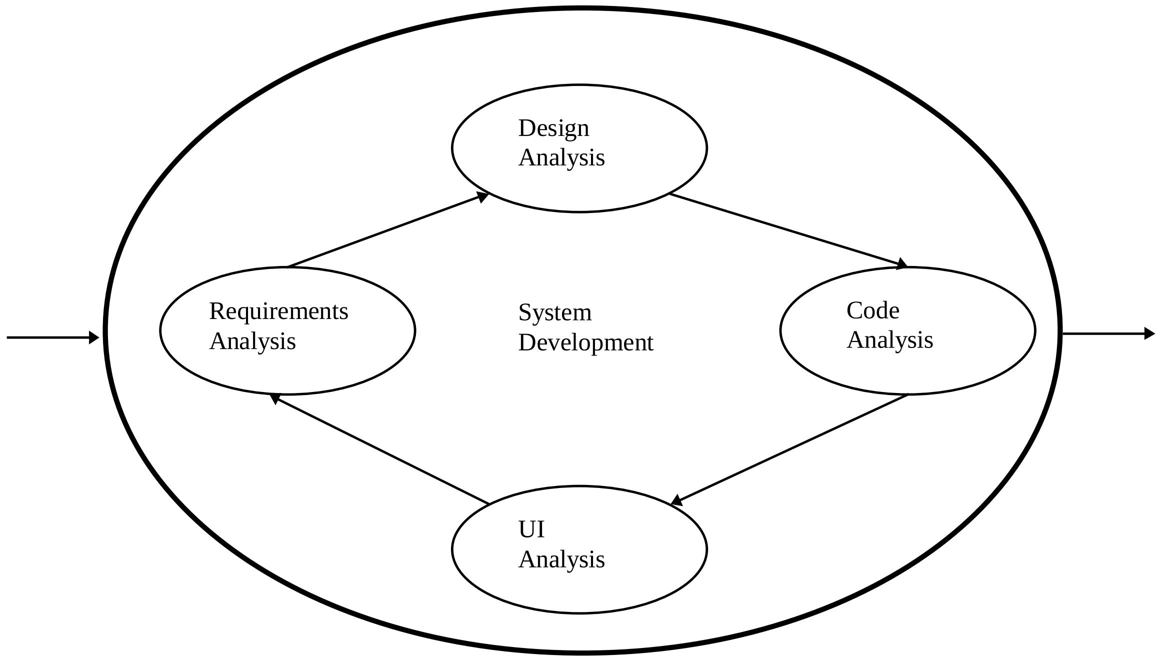


Figure 3: Safety analyses for an iterative system development process.

2.2.1 Hazard Identification

The identification of hazards begins during the initial conception of the system and continues as the system develops. The identification exercise involves brainstorming among a group of people with knowledge of the application domain. Historical data and the lessons learned from the development and operation of similar systems are important inputs into the process. The initial identification of hazards is documented in the Preliminary Hazard List [DOD93].

The definition of the hazards depends on where the system boundaries are drawn. It is important to restrict the definition of the hazard to the aspects of safety within the system design space. Accident scenarios are constructed that begin with a hazard at the system boundary and then follow a series of environmental events that lead to an accident. The scenario events can then be used to construct accident fault trees.

The identification of hazards continues throughout the system lifecycle. A "safety issues" mechanism can be employed where anyone associated with the system is able to express a safety-related concern by submitting a safety issue. The issues are controlled in some well-defined manner to ensure that they are analyzed and eventually closed. An issue may be closed in several different ways: 1) a decision that the issue does not, after all, relate to safety; 2) a determination that the issue constitutes a particular aspect of a previously identified hazard (in which case an addition may be made to the definition or analysis of the hazard) or 3) a new hazard is formally introduced based on the issue.

2.2.2 Safety System Analysis

A Preliminary Hazard Analysis (PHA) is performed to identify hazardous system states [IEEE94] The system is treated as a “black-box” and its interactions with the environment are analyzed. Hazard scenarios are constructed at the system level that consist of system and environmental events that lead to the hazard. The system events correspond to hazard causes. The results of the analysis are assessed and are used to construct system safety requirements and design criteria to eliminate or mitigate the hazard.

2.2.3 Safety Requirements Analysis

The system level hazard scenarios identified during hazard analysis are traced onto the requirements. The requirements are examined for consistency with the safety requirements. HAZOP [Ree96], FMEA [LW96], and other techniques are used to identify hazard causes. Additional hazard scenarios are constructed. The requirements can be modeled and analyzed for incompleteness and ambiguity [MLRPS97]. A state machine model of the requirements can be searched for hazard causes [Lev95]. The results of the analysis are then assessed and used to derive further safety requirements and design criteria.

2.2.4 Safety Design Analysis

The safety requirements and design criteria are used to guide the design process. The design is examined for consistency with the safety requirements and design criteria. The system level hazard scenarios are mapped into events within the system. These events occur within and between software components. If the design is object-oriented, object-scenario diagrams can be constructed to illustrate the component level hazard scenarios. FTA, HAZOP and FMEA [FMNP94] can be used to analyze the software design. The critical components are identified and their interfaces and interactions analyzed. The results of the analysis are then assessed and used to derive design constraints for the critical components.

2.2.5 Safety Code Analysis

The component level hazard scenarios are traced onto the source code. The critical procedures and the relevant code sections are identified. The source code is examined for consistency with the design constraints, by using unit testing, code reviews, formal verification and other techniques. Additional quality checks are performed for the critical code sections. Recommendations are made for design and coding changes based on the results of the analysis [IEEE94]. The results are used to construct code assertions and other run-time checks, as well as code comments.

2.2.6 Safety User Interface Analysis

The user interface is examined for possible contributions to the hazard. Standard user interface evaluation techniques can be used to investigate operators’ interactions with the system when performing critical tasks. These techniques include cognitive walkthroughs, usability testing, usability engineering, controlled experiments and usability guidelines [BGBG95].

2.2.7 Safety Verification

The source code is verified with respect to the hazards. This step is necessary even after hazards have been identified and controlled through design, in order to determine if any mistakes were made in the safety engineering process.

Safety verification includes both static and dynamic analysis, such as system testing and code inspections [SL94], as well as software fault tree analysis [LCS91] and formal verification techniques [Bar97]. Process checks are conducted to ensure that the hazard mitigations identified in the earlier stages of development have been implemented. Final system risk assessment is performed to ensure that the risks of the accident occurring have been adequately mitigated.

2.2.8 Safety Operational Analysis

The system will require periodic safety audits to ensure that the operational level of safety is maintained [Lev95]. In particular, it will be necessary to evaluate the impact on safety of any changes that may have occurred in the system, operations or the environment.

3. Example - Chemical Factory Information System

For illustrative purposes, this dissertation focuses on an information system for a chemical factory. This hypothetical example is similar to existing real-time information systems, like Air Traffic Management systems, in that environmental data is received, processed and displayed to operators. The operators then make safety-critical decisions based on the information displayed by the system.

The development of the chemical factory information system is assumed to follow the safety engineering process described in Section 2.2. This includes the production of a set of documents such as the Hazard List, System Safety Requirements Specification, Hazard Log and Safety Case. These safety documents are in addition to other development artifacts such as the System Requirements Specification. Though these documents do not in fact exist, they would be produced if the chemical factory information system was built according to the described safety engineering process.

3.1 System Description

The factory consists of a set of reactor vessels equipped with sensors that record data such as temperature and pressure. The sensors are connected through a LAN to the chemical factory information system, which runs on a central server and a set of workstations. The information system maintains and processes the vessel information it receives over the LAN and displays it on the workstation monitors.

3.2 Functional Requirement

The system is responsible for maintaining and displaying the temperatures of the reactor vessels, along with other vessel information. The following is a functional requirement from the System

Requirements Specification concerning the display of vessel temperature[1]:

ROID 356.

Upon receipt of a sensor update from the external monitoring system containing the temperature of a vessel, the system shall update the displayed temperature of the vessel in no more than 200 milliseconds.

3.3 Safety-Related Hazard

One of the hazards identified in the Hazard List is the display of an "invalid" value for the temperature of a vessel:

Hazard 3.

An invalid vessel temperature is displayed.

The identification of this hazard resulted from an analysis which shows that the display of an invalid value for the temperature of a vessel, in combination with other conditions, could lead to an accident such as a fire or explosion.

3.4 Preliminary Hazard Analysis

An “invalid” vessel temperature may be caused by a number of different factors. A temperature value may not be delivered in a timely fashion, a displayed valid temperature value may become

[1] The acronym ROID stands for Requirements Object Identifier, a method of identifying individual requirements borrowed from the CAATS development methodology.

stale, or a temperature value may become corrupted before being displayed.

The system’s failure to display a valid vessel temperature was also considered as a possible hazard cause during the preliminary hazard analysis. However, it was determined that this will not cause the hazard. When the system is unable to display a valid temperature for a particular vessel, the system is required to mark the temperature field for this vessel as "unavailable". Even though the appearance of "unavailable" on the operator display in the temperature field for some particular vessel may be a result of a system fault, it has been determined that its appearance is not unsafe. This determination was partially based on the assumption that the human operator should not be misled by the "unavailable" indication. Hence, the term "invalid" is used in the definition of this particular hazard to refer to a temperature that is invalid but appears to be a valid temperature. In particular, the appearance of "unavailable" on the operator display in the temperature field for some particular vessel is excluded from the definition of this hazard.

There are two basic hazard scenarios. In one scenario, the vessel temperature display is updated with a stale or corrupted temperature value. In the other scenario, a displayed valid temperature value becomes stale.

4. Safety Verification Conditions

Safety verification is typically defined in terms of verification of the safety critical source code with respect to the safety requirements. For example, the international safety standard IEC 61508 [IEC95] defines a software safety validation phase for demonstrating that the safety-related software satisfies the software safety requirements specification.

For this definition of safety verification to be valid, it is important that the safety requirements are sufficient to mitigate the hazards. The safety verification of the software requires a complete set of "safety verification conditions" (SVCs) that, if satisfied, ensures that the hazard is eliminated or mitigated. This includes any implicit assumptions about the system and the environment in the safety requirements specification. This dissertation introduces the term "safety verification conditions" to emphasize the importance of having a complete set of conditions for safety verification.

4.1 Safety as a Distinct Property

It might seem to someone developing a safety-critical system that the derivation of an explicit set of SVCs would be superfluous if the system is "correctly" built, satisfying the system functional requirements. For example, the functional requirement ROID 356 (given in Section 3.2) is designed to ensure the timely and correct delivery of temperature values. It may appear at first glance to a system developer that ensuring the timely and correct delivery should be sufficient to mitigate the hazard of displaying an "invalid" temperature.

However, requirement ROID 356 is not sufficient to mitigate the hazard. The requirement would not rule out the possibility of the system displaying a stale temperature value. For example, a valid display could become invalid if temperature updates cease and the system fails to set the display to "unavailable". The requirement would also not rule out unintended functionality, such as the possibility of the displayed temperature being updated to D for vessel V, as well as for a different vessel V' which is not at temperature D.

It may be argued that the functional requirement ROID 356 does not have anything at all to do with safety since the hazard does not result if the system fails to display the temperature. During hazard identification, it was determined that the environmental conditions were such that the operator had alternative methods of obtaining a vessel temperature if the system failed to display one. Therefore, a system may "correctly" implement this requirement without any significant safety benefit.

4.2 System and Environmental Assumptions

SVCs that place constraints on system functionality typically involve assumptions about the system and the environment. For example, an environmental assumption for the chemical factory might be the timely and correct delivery of the temperature value from the external vessel sensors to the system. Another assumption might involve the ways in which the temperature display may be updated. These assumptions are often fairly obvious and implicit in the statement of the SVCs that involve system functionality.

However, overlooking implicit or buried assumptions made during the safety analysis can contribute to accidents. The ARIANE 5 failure [AIB96] is an example of software reuse where the original environmental assumptions were not re-evaluated for the new system. The catastrophic failure of this satellite resulted in direct costs of approximately $370 million. The Therac-25 accidents [Lev95] resulted when new software functions replaced hardware interlocks, without careful consideration of the impact on safety. Problems with the source code contributed to six lethal overdoses.

It is important that the underlying implicit assumptions about the system and environment are made explicit, especially if they are necessary for the mitigation of the hazard. The hazard-related environmental and system assumptions should be included as additional SVCs.

4.3 Semantic Gap

Hazards lead to a set of safety conditions which are essentially the negation of each hazard. For the chemical factory, the corresponding safety condition is that only “valid” vessel temperatures are displayed. This can be refined by the introduction of SVCs which assert that displayed temperature values have been delivered in a timely and correct fashion, and that displayed temperatures will be updated to “unavailable” before they become stale.

However, hazards are typically defined at a relatively high level of abstraction. For good reason, the definition of a hazard will be based upon the terminology of the end-user rather than the terminology of the software developer implementing application-level functionality on top of lower layers of software infrastructure and primitives. To perform safety verification of the software, the definition of the hazard must be mapped to a relationship between elements of the software implementation. This mapping must bridge the "semantic gap" between the terminology used to define the hazard and the terminology of the software developer.

The semantic gap can be bridged in a series of refinement steps by constructing SVCs at the system, component and source code levels. System level SVCs correspond to safety requirements that constrain system functionality and performance, and include environmental and system assumptions. Component level SVCs include constraints on the system components, as well as assumptions on underlying services and mechanisms. Source code level SVCs can be expressed as verifiable code assertions. These can then be used in the safety verification of the software implementation.

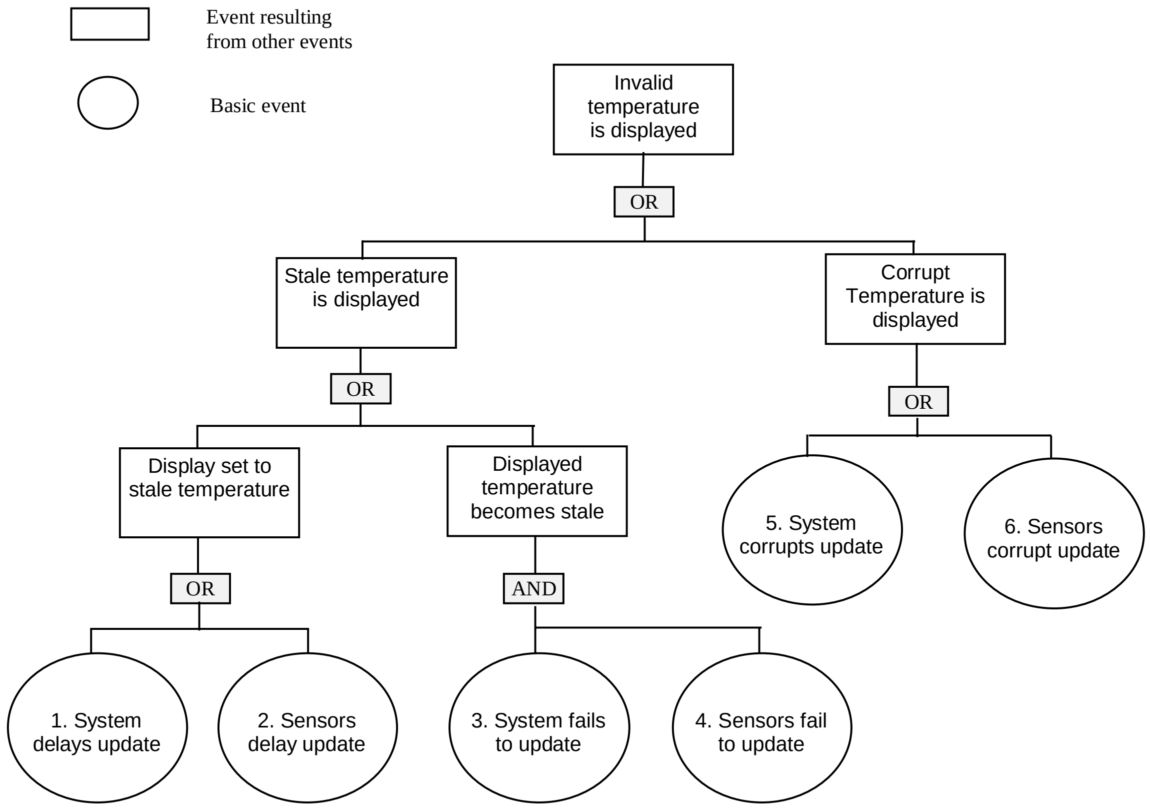
At the system level, functionality is usually expressed in terms of system inputs and outputs. The hazard, however, typically relates the external effects of the system to the environment. For the chemical factory, the hazard is ultimately tied to the vessel's actual temperature, which is reported by the external monitoring system. In order to verify the safety of the software, it would be useful to have a set of system level SVCs expressed in terms of the system input and outputs, as well as system level constants and terms.

4.4 Techniques for Generating SVCs

SVCs are typically constructed in an *ad hoc* manner, relying primarily on engineering judgment. Some SVCs are obtained by identifying the safety-related requirements already present in the system requirements specification. Other SVCs are based upon the lessons learned from the development of similar existing systems. In general, the results of the hazard identification and analysis phases of the safety process provide the basis for the SVCs.

4.4.1 Fault Tree Analysis

Aside from following a purely *ad hoc* approach, the derivation of SVCs is likely to involve a form of Fault Tree Analysis (FTA) or a related hazard analysis technique. FTA is often used during hazard analysis to reveal some possible hazard causes.



Event resulting from other events
Basic event
Invalid temperature is displayed
OR
Stale temperature is displayed
Corrupt Temperature is displayed
OR
OR
Display set to stale temperature
Displayed temperature becomes stale
5. System corrupts update
6. Sensors corrupt update
OR
AND
1. System delays update
2. Sensors delay update
3. System fails to update
4. Sensors fail to update

Figure 4. System level fault tree analysis of the chemical factory hazard.

FTA begins with the hazard as the "top event" and then determines the intermediate events that cause the hazard, which are combined using logical operations such as "AND" and "OR". The analysis is performed in a top-down fashion beginning with events at the system level. The analysis continues with events at the component level, down to events at the source code level. The system level fault tree for the chemical factory hazard is shown in Figure 4.

FTA is intended as a means of organizing hazard causes in a top-down fashion. The resulting fault tree can then be used as the basis for constructing SVCs. For example, SVCs can be derived from the basic fault events by taking the negation of each fault. For example, the chemical factory system level SVCs would include "System does not delay temperature update", "Sensors do not delay updates", "System updates vessel temperature display as 'unavailable' before the displayed value becomes stale" and "System does not corrupt temperature updates".

Software Fault Tree Analysis (SFTA) is performed at the source code level. SFTA begins by assuming a hazard arises from the output of a given line of source code. The hazard causes are then traced backwards through the code with the help of language templates. The templates are based on the semantics of the programming language and determine the various ways a code statement can contribute to the hazard or to an intermediate event. The analysis continues until a contradiction is reached or a hazard code path is uncovered. Though it may be possible to introduce SVCs at the source code level with SFTA, its primary use is to verify the source code with respect to a hazard.

4.4.2 Generating Temporal Relationships

For a real-time system, some hazards will depend on the temporal ordering of events. These time-dependent hazards typically lead to SVCs which place real-time constraints on the system. For example, in Section 4.4.1, the following SVC was introduced: "System updates vessel temperature display as 'unavailable' before the displayed value becomes stale". However, the phrase "before the displayed value becomes stale" is potentially vague. Updating the vessel temperature display will depend on events such as the sensor's acquisition of data values, and the reception and display of temperature values by the system. The processing of incoming temperature values will be concurrent with the monitoring of current temperature values for staleness. In order to have a set a verifiable conditions, the specification of the SVC will require the precise relationships between these environmental events, and the system input and output events.

FTA is a hierarchical, top-down modeling technique and is not particularly suited for representing the temporal aspects of the hazard, such as the chronology of events leading to the hazard [Lev95]. In particular, the basic “OR” and “AND” operators do not specify the time ordering of events or time delay. There are other operators that could be used to allow for a partial treatment of time. However, the use of additional operators may compromise the simplicity and readability of fault trees. Leveson suggests that if chronology is important, it may be more appropriate to use a different analysis technique [Lev95].

There are a number of different temporal and real-time logics [Gup92] that can be used for the specification and analysis of the temporal aspects of a real-time system. For example, a formal logic known as Real Time Logic is used, along with a model of system events and actions, to analyze a system for timing constraints [JM86]. Besides temporal and real-time logics, Petri Nets [LS87], statecharts and statechart variants such as RSML [LHHR44] can be applied to the safety analysis of real-time systems. Though these approaches can be used to specify and verify timing constraints, they do not provide a systematic method for deriving the necessary temporal relationships.

A temporal or real-time logic could be used to introduce a notion of time ordering to fault trees. There have been a number of attempts to provide fault trees with a formal semantics [Han96]. For example, one approach uses a real-time interval logic known as duration calculus [KRS98]. Safety requirements can then be derived from the formalized fault tree.

However, simply expressing the fault tree events in a formal notation does not result in the necessary temporal relationships. In general, it will be necessary to perform an additional timing analysis of the fault tree events to arrive at some timing constraints [GW96]. Even then it is not clear if the resulting timing constraints are sufficient to eliminate the hazard.

5. Verification Tree Method

This dissertation proposes a approach to deriving SVCs known as “Verification Tree Method” (VTM). VTM supplements FTA with a constrained syntax and mathematical reasoning. The analysis begins by first assuming that the hazard exists. The analysis then proceeds in a stepwise manner by working backwards from the hazard occurring at a particular instant of time, to discover other events which must have occurred at some earlier instant of time. The analysis may branch as a result of reasoning by cases. When the analysis branches into one or more cases, each branch is "closed" by showing that each branch leads to a logical contradiction. The closure of these branches may require steps that are purely mathematical, not involving event occurrences. In the course of generating contradictions, SVCs are introduced. Each SVC is intended to be the minimum condition required to close a particular branch of the analysis.

SVCs are derived at the system level by treating the system as a “black box”. This involves defining the system boundaries and the relevant system inputs and outputs. For the chemical factory information system, reports from the vessel monitoring system are received as inputs and vessel temperature values are displayed as outputs. The system level SVCs are expressed in terms of these system inputs and outputs.

VTM uses the results of the hazard analysis as input. This includes hazard causes and hazard scenarios. For the chemical factory, one hazard scenario involves the arrival of a temperature update which results in the display of a corrupt or stale temperature value. In another scenario, the cessation of updates leads to a displayed valid temperature becoming stale. The system and environmental events that make up the hazard scenario are used to guide the introduction of SVCs.

5.1 Constrained Syntax

VTM uses a simple constrained syntax to describe: 1) scenarios involving the occurrence of specific events at specific instants in time; 2) constraints on the behaviour of the system.

5.1.1 Basic Notational Elements

Unknown or variable quantities of time (i.e., durations) are represented by logical constants. These constants are symbols which may be given descriptive names such as MAX_DISP_TEMP_STALE. The time units are not usually expressed explicitly. Notationally, both quantities of time and instants in time are treated as arithmetic values, typically natural numbers. The time unit (e.g., milliseconds, seconds, minutes, hours or days) will vary depending on the application.

VTM involves the analysis of events at particular instants in time. These instants in time are explicitly named by logical constants. These constants are symbols which, by convention, have names such as T1 and T2.

Simple arithmetic operators, + (addition) and **-** (subtraction), are used to refer to unnamed instants in time in terms of their relationship to other instants in time. For example, T1 + D1 is the instant of time, D1 time units after T1. These expressions serve as references to specific instants in time, or for short, "time references".

Time references are nouns within phrases (e.g., “temperature D is displayed at time __”) which describe the occurrence of particular events at particular instants in time. These phrases are called "predicates". By filling in the parameters (represented by "___" in our example) of a predicate with a time reference, we obtain an assertion such as "temperature D is displayed at time T". A predicate is typically parameterized by just one time reference, but there may be situations in which a predicate is parameterized by multiple time references as well as other values.

The description of a scenario often involves specifying constraints on the location of time references on the timeline relevant to the location of other time references on the timeline. Several different notational approaches (e.g., temporal logic expressions) could be used to express these constraints. In our approach, we use familiar mathematical operators such as "=", "<", "<=", ">" and ">=" for comparison of arithmetic values. The comparison operators may be used to represent temporal relationship as inequalities such as (T1 + D1) < T2. These operators are used because they are more concise than the natural language phrases that are often used to express temporal relationships, such as "at the same time as", "before", and "not after". Moreover, the use of these operators avoids the potential ambiguity that may arise from using these various phrases of natural language.

VTM follows the convention where upper case letters instances (e.g., V, D, T, T1, T2, T3, T4) are used to denote specific. Lower case variables (e.g., v, d, t, t', t'') are used for variables that are universally quantified by a quantifier such as a "for all" or existentially quantified by an "exists" operator.

5.1.2 Scenarios

A scenario is usually described by two kinds of statements. One kind of statement, as described in the previous section, is an assertion that results from the application of a predicate to a time reference to express the occurrence of a particular event at a particular instant of a time, e.g.,

Temperature D is displayed at time T.

The other kind of statement is an inequality which constrains the position of a time reference on the timeline relative to other time references, e.g.,

At some time T’, T’ > T and T’ - T <= S1, E occurs.

It is also possible to combine these two basic types of assertions with logical connectives to create a new assertion, e.g.,

If temperature D is displayed at time T,

then at some time T’, T’ > T and T’ - T <= S1, E occurs.

5.1.3 Timing Constraints

In addition to representing scenarios with a logical notation, we also want to represent constraints on the behaviour of the system. Like the description of scenarios, the representation of these constraints will be formulated in terms of temporal relationships. However, a constraint is generally more than a relationship between a particular fixed set of time references. Instead, a constraint is usually a relationship that holds "universally" at all points along the timeline.

For this purpose, our approach uses bound variables and quantification operators. Bound variables are used to denote instants in time. These are bound by a quantification operator such as "for all" and "exists". In addition, it is often necessary to use logical operators such as NOT, AND, OR and IMPLIES to express constraints. A typical pattern for the expression of a constraint is:

For all t, if E1 occurs at t,

then there exists a time T, such that t < T and E2 occurs at T.

5.2 Mathematical-Style Reasoning

VTM uses a style of mathematical reasoning called “proof-by-contradiction”. To prove an assertion “X”, this style of reasoning begins with the introduction of a conjecture that X is not true, i.e., “not X”. The argument proceeds by showing that “not X” inevitably leads to a contradiction. If this can be demonstrated, then we may conclude that “not X” is false – that is, X is true.

The task of showing that “not X” inevitably leads to a contradiction typically includes steps where the argument is split into multiple branches. Each branch of a split in the argument represents one of the cases in a case analysis. When the structure of the argument is viewed graphically, the splitting of some steps of the argument by case analysis has the effect of giving the graphical representation a tree-like appearance, as illustrated in Figure 5.

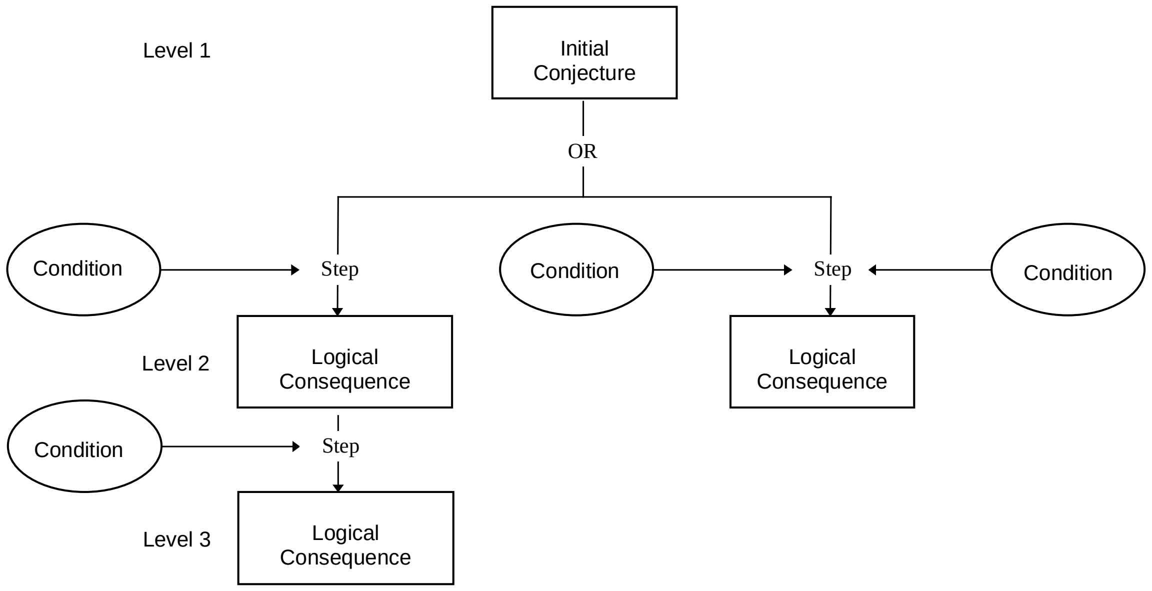


Figure 5: Graphical representation of the rigorous argument.

The cases are based on tautologies exposed by expressing the argument in the constrained syntax of VTM. For example, a scenario may involve a relationship like S1 + S2 <= S3. This relationship implies two different cases 1) S1 + S2 < S3 or 2) S1 + S2 = S3. The analysis may branch at this point where case 1) is assumed in one branch and case 2) is assumed in the other branch.

In the course of developing a proof by contradiction, assumptions are introduced. The assumptions close off particular branches of the argument and “steer” the direction of the proof. The validity of the assertion proved in this manner with the aid of these assumptions will depend on the validity of the assumptions. These assumptions are the SVCs.

Figure 5 provides a graphical representation of the structure of the argument. The box at the top of the figure represents the initial conjecture (IC). The remaining boxes represent logical consequences (LC) of this initial conjecture. The ovals represent logical conditions which are introduced as assumptions at various steps in the analysis. Different assumptions may lead to different cases. The cases are represented as different branches combined by an “AND/OR” gate. The assumptions are used along with the IC or LC of the “current” level to generate a LC for the next level of the analysis. Each level of the analysis is linked to the next lower level by an arrow. The arrow may be read as “implies”. For example, the initial conjecture (IC), in combination with a condition, implies a logical consequence.

6. Chemical Factory: Verification Tree Method

In this section, VTM is used to derive SVCs for the chemical factory information system. The analysis begins with the conjecture that the chemical factory hazard has occurred at some particular instant in time. The subsequent analysis is then guided by the hazard scenarios identified during the Preliminary Hazard Analysis. There are two parts to the analysis.

The first part of the analysis introduces SVCs which eliminate the display of a corrupt temperature value as a possible hazard cause. This part of the analysis corresponds to the hazard scenario where a temperature update results in the display of a corrupt temperature value. This hazard scenario can be expressed in terms of system and environmental events, where the system is treated as a "black box". The first part of the analysis then involves tracing each of these events backwards to their cause, beginning with the hazard as the top-level event.

The second part of the analysis introduces additional SVCs that eliminate the possibility of a stale temperature value being displayed. This part of the analysis corresponds to the scenario where a displayed valid temperature value becomes stale. These steps are purely a matter of logical reasoning. The analysis makes use of logical laws (i.e., tautologies) as well as arithmetic laws. Although these laws may be cited in the written record of the analysis, they are not shown in the graphical representation. These steps lead to a logical contradiction that completes the proof-by-contradiction argument.

6.1 Initial Conjecture

The analysis begins with a conjecture that an instance of the hazard has occurred:

An invalid temperature D is displayed for vessel V at time T.

Expressing the chemical factory hazard with the constrained syntax of VTM results in a more precise definition of how a displayed temperature value may be "invalid". The “initial conjecture” (IC) can be re-written as:

IC: invalid temperature D is displayed at time T.

The temperature D displayed for vessel V at time T has not been within MAX_DISP_TEMP_DIFF degrees of the actual temperature of the vessel at any time within MAX_DISP_TEMP_STALE milliseconds before time T.

MAX_DISP_TEMP_DIFF and MAX_DISP_TEMP_STALE are “requirements-level” system constants defined as follows:

- MAX_DISP_TEMP_DIFF is the maximum difference allowed between the actual temperature of the vessel and the value displayed to the operator;

- MAX_DISP_TEMP_STALE is the maximum amount of time that a value may be displayed before it is considered "stale".

The system constant MAX_DISP_TEMP_STALE should play a key role in the temporal relationships derived from the hazard.

6.2 Part 1: Corrupt Temperature Value

The first part of the analysis introduces SVCs which eliminate the display of a corrupt temperature value as a possible cause of the hazard.

To bring to the surface the implicit temporal relationships, SVCs are expressed in terms of events occurring at particular instants in time. These include environmental events, as well as events corresponding to system inputs and outputs. These events will be causally related, where an event occurrence will the result of a different, earlier event occurrence. The constrained syntax of VTM is then used to represent the temporal relationships between these events.

Defining the meaning of "invalid" temperature in terms of system and environmental events leads to SVCs expressed in a "backwards" fashion. According to the Preliminary Hazard Analysis, the system displaying a vessel temperature as "unavailable" in place of a valid temperature value will not cause the hazard. As a result, the SVC need only cover the case when a temperature value is actually displayed. This leads to "backwards" SVCs that specifies the expected system input (e.g., temperature update), given a system output (e.g., displayed temperature value). This is in contrast to a typical functional requirement that specifies the expected system output, given a system input.

The first part of the analysis is shown in Figure 6. The analysis steps establish a backward chain of causal relationships. Step 1 traces the occurrence of the hazard at time T (represented by the IC in the first level of analysis) backwards to an event at time T1 (represented by LC-1 in the second level of the analysis). Step 2 traces this event at time T1 to an earlier event at time T2. In turn, Step 3 traces this event at time T2 to an earlier event at time T3.

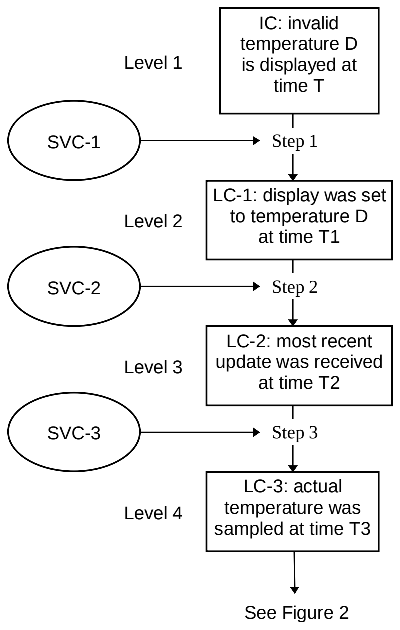

Level 1
IC: invalid temperature D is displayed at time T
SVC-1
Step 1
Level 2
LC-1: display was set to temperature D at time T1
SVC-2
Step 2
Level 3
LC-2: most recent update was received at time T2
SVC-3
Step 3
Level 4
LC-3: actual temperature was sampled at time T3
See Figure 2

Figure 6: Graphical representation of the first part of the analysis.

6.2.1 Step 1

The display part of the chemical factory information system is implemented by Commercial-Off-The-Shelf (COTS) hardware and software. The following analysis will focus on the application-specific, custom software which drives the COTS-based display subsystem. To this end, an SVC is introduced that is a high level assumption about the display subsystem which allows the cause of the hazard to be traced directly back to the application-specific, custom software.

SVC-1.

For all temperatures d, times t, and vessels v, if d is displayed at time t as the temperature of vessel v, then there is some time t’, t’ <= t, when the temperature of vessel v was set to d and this was the most recent change made to the displayed temperature for vessel v.

Given this SVC, LC-1 is derived as a logical consequence of the initial conjecture.

LC-1: display was set to temperature D at time T1.

At time T1, T1 <= T, the temperature of vessel V was set to D and this was the most recent change made to the displayed temperature for vessel V.

6.2.2 Step 2

The displayed temperature value is the result of a system output which can be traced back to a system input. A second SVC is introduced to ensure that the displayed vessel temperature is the result of a temperature update from the external sensor monitoring system. Furthermore, the SVC ensures that the temperature update has been delivered correctly and within the time constraint S1:

SVC-2.

For all vessels v, displayed temperatures d, and times t, if the displayed temperature of vessel v is set to d at time t, then at some time no earlier than S1 milliseconds before t the system received a report from the external sensor monitoring system that the temperature of vessel v is d.

Given SVC-2, the following is derived as a logical consequence of LC-1:

LC-2a.

At some time T2, T2 < T1 and T1-T2 <= S1, the system received a report from the external sensor monitoring system that the temperature of vessel V is D.

Without loss of generality, this logical consequence may be refined into:

LC-2: most recent update received at time T2.

T2 is the most recent time before T1 when the system received a report from the external sensor monitoring system that the temperature of vessel V is D.

The derivation of LC-2 from LC-2a is an example where VTM uses more than propositional reasoning. In this case, the step involves making an informal deduction that is based on a formal inference rule of predicate logic that is often called existential instantiation.

6.2.3 Step 3

The temperature update received by the system can be traced back to the vessel sensors. A third SVC is then introduced to ensure that the temperature update is correct, within a given tolerance, and has been delivered to the system within the time constraint S2:

SVC-3.

For all vessels v, displayed temperatures d, and times t, if the system receives a report at time t from the external sensor monitoring system that the temperature of vessel v is d, then at some time no earlier than S2 milliseconds before t the actual temperature of vessel v was within MAX_DISP_TEMP_DIFF degrees of d.

This SVC is used to derive the following logical consequence from LC-2:

LC-3: actual temperature was sampled at time T3.

At some time T3, T3 < T2 and T2-T3 <= S2, the actual temperature of vessel V was within MAX_DISP_TEMP_DIFF degrees of D.

6.3 Temporal Relationships

A backward chain of events has been constructed with the first part of the analysis using symbolic names, T, T1, T2 and T3, to represent the times of these events. The order of these times is represented by the timeline in Figure 7.

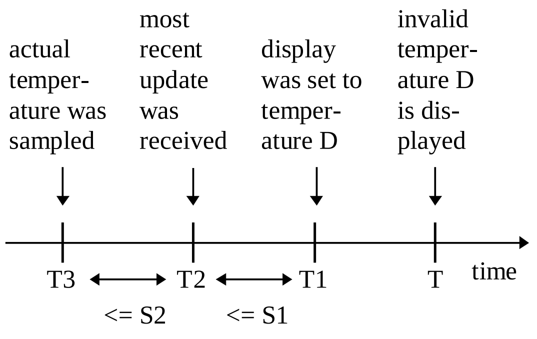

actual temper-ature was sampled
most recent update was received
display was set to temper-ature D
invalid temper-ature D is dis-played
T3
T2
T1
T
time
<= S2
<= S1

Figure 7: Timeline of Events.

Two "system level" time constants, S1 and S2, have been identified which have direct relevance to the hazard. A third system level time constant, S3, will be required that is associated with the system monitoring of temperature value staleness. These system level time constants are defined as follows:

- S1 - the maximum time required for a temperature value to be propagated through the chemical factory information system;

- S2 - the maximum time required for a temperature value to be propagated from the temperature sensors to the chemical factory information system via the external monitoring system;

- S3 - the maximum amount of time that the system will display the value in the absence of an external update before the system will set the displayed temperature to "unavailable".

S1 and S3 are system level constants which denote upper bounds on the performance of the software system. S2 is an upper bound on the performance of an external system.

Intuition suggests that there should be dependencies between these system constants. Moreover, intuition also suggests that there should be relationships between these constants and the "requirements-level" system constant MAX_DISP_TEMP_STALE used in the definition of the hazard. These temporal relationships will be determined in the second part of the analysis.

6.4 Part 2: Stale Temperature Value

The second part of the analysis introduces SVCs to eliminate the display of a stale temperature as a hazard cause. However, unlike the introduction of SVCs 1-3, introducing SVCs to eliminate this hazard cause is not a simple matter of tracing an event back to its cause. Some additional reasoning is required to determine the necessary SVCs. This part of the analysis demonstrates some of the distinctions between FTA and VTM. A graphical representation of the second part of the analysis is shown in Figure 8.

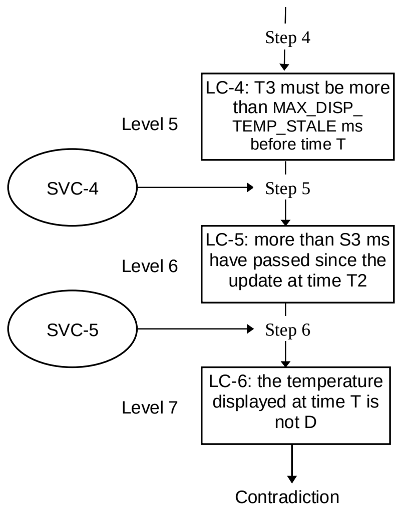

Step 4
LC-4: T3 must be more than MAX_DISP_ TEMP_STALE ms before time T
Level 5
SVC-4
Step 5
LC-5: more than S3 ms have passed since the update at time T2
Level 6
SVC-5
Step 6
LC-6: the temperature displayed at time T is not D
Level 7
Contradiction

Figure 8: Graphical representation of the second part of the analysis.

6.4.1 Step 4

In the first part of the analysis, the cause of the hazard occurrence has been narrowed to the situation where a displayed temperature has become stale. This can be seen more clearly if LC-1 and LC-2 are used to derive the inequality T2 < T, and this inequality is, in turn, used to derive the following logical consequence from LC-3:

LC-4a.

At some time T3, T3 < T, the actual temperature of vessel V was within MAX_DISP_TEMP_DIFF degrees of D.

The initial conjecture states that the actual temperature of vessel V was not within MAX_DISP_TEMP_DIFF degrees of D at any time within MAX_DISP_TEMP_STALE milli-seconds prior to T. So, in light of LC-4a, T3 must be more than MAX_DISP_TEMP_STALE milliseconds prior to T. This reasoning yields LC-4:

LC-4: T3 must be more than MAX_DISP_TEMP_STALE ms before T.

At some time T3, T - T3 > MAX_DISP_TEMP_STALE, the actual temperature of vessel V was within MAX_DISP_TEMP_DIFF degrees of D, and this was the most recent time before T that the displayed temperature was within MAX_DISP_TEMP_DIFF degrees of D.

6.4.2 Step 5

A new SVC is introduced to constrain the value of S3. MAX_DISP_TEMP_STALE is the maximum amount of time that a value may be displayed before the value is considered to be stale. Similarly, S3 is the maximum amount of time the system may display a temperature value before the value is considered to be stale, but as measured from the time the value is first received by the system. The value of S3 must be less than MAX_DISP_TEMP_STALE to take into account the time required for the temperature value to reach the system from the external monitoring system. Since S2 is the maximum time allowed for a temperature value to reach the chemical factory information system, the following SVC is proposed:

SVC-4.

MAX_DISP_TEMP_STALE > S2 + S3

This SVC is used to derive (by transitivity of “>”) the following logical consequence from LC-4:

LC-5a.

T - T3 > S2 + S3

The system will set the vessel temperature display to “unavailable” depending on the time elapsed since the most recent receipt of a temperature update for that vessel. This can be determined by first obtaining the relationship T2 - T3 <= S2 from LC-3. This relation is used, along with various rules of arithmetic, to derive the following logical consequence from LC-5a:

LC-5: more than S3 ms has passed since the update at time T2.

T - T2 > S3

6.4.3 Step 6

The assumption is made at this point that the receipt of an update at time T2 initiates a process that will cause the displayed temperature to be changed to “unavailable” if a subsequent update is not received before the displayed temperature becomes stale. This should occur before S3 ms has elapsed, but only after S1 ms has elapsed to allow time for a subsequent update to arrive. If a subsequent update is received in time, the display is updated to the new value. This leads to the fifth SVC:

SVC-5.

For all vessels v, and times t and t’, if time t is the most recent time that an update for vessel v at temperature d was received prior to time t’, and t’-t > S3, then at some time, t”, such that t + S1 < t” <t’, the temperature value displayed for vessel v shall have been set to "unavailable" or shall exhibit some value other than d.

Considerable care went into the formulation of SVC-5 to ensure that it is both sufficiently general, so that it places minimal constraints on the actual implementation, and practically feasible, in that there is likely to be a practical implementation of this SVC. This new SVC is used to derive the following logical consequence of LC-5:

LC-6a.

At some time T4, T2 + S1 < T4 < T, the temperature value displayed for vessel V shall have been set to “unavailable” or shall exhibit some value other than D.

From LC-2a, it is known that T1-T2 <= S1 and, from LC-6a that T2 + S1 < T4. Using rules of arithmetic reasoning for inequalities, the inequality T1-T2 <= S1 can be used as justification for replacing S1 by T1-T2 in T2 + S1 < T4 to obtain T2+(T1-T2) < T4. In its simplified form, T1 < T4, this result clearly shows that T4 must be some time after T1. Therefore, the following logical consequence of LC-6a can be derived:

LC-6: the temperature displayed at time T is not D.

At some time T4, T1 < T4 < T, the temperature value displayed for vessel V shall have been set to “unavailable” or to some value other than D.

LC-6 contradicts LC-1, which states that the most recent change to the displayed temperature value occurred when it was set to D at time T1.

QED

6.5 System level SVCs

The application of VTM to the chemical factory hazard results in five distinct safety verification conditions. The SVCs include:

- system and environmental assumptions (SVC-1, SVC-3);
- system functional correctness conditions (SVC-2, SVC-5);
- constraints on variable system parameters or constants (SVC-4).

As discussed in Section 6.2, SVC-2 is an example of a “backward” condition which describes the expected system input (a temperature update), given a system output (a displayed temperature value). This backwards condition is in contrast to the “forward” functionality requirement ROID 356 (given in Section 3.2). This requirement expresses the expected system output, given a system input. Verifying SVC-2 involves analyzing all system functionality that may create the given system output, including the functionality expressed by ROID 356.

SVC-5 ensures that a temperature display is updated to “unavailable” before it becomes stale. The software implementation could satisfy SVC-5 by setting a timer when an update is received. If the timer is not reset by a subsequent update, the system would then eventually initiate an action to cause the displayed temperature to be changed to “unavailable”. A crude (but possibly more robust) implementation would involve a process that blanks out all displayed temperature values at regular intervals independently of when they were last updated.

7. Discussion of Results

There are three key elements of the VTM approach for generating SVCs: (1) a constrained syntax for explicitly representing temporal aspects of the analysis, (2) proof-by-contradiction style reasoning, and (3) the introduction of SVCs during the reasoning process to "steer" the proof-by-contradiction towards the closure of each branch of the proof. These basic elements can be used in the application of the method to other hazards and systems.

The approach presented in this dissertation is not a purely "mechanical" method for the generation of SVCs. Some inspiration is required to formulate each of the SVCs. However, rather than depending on a few "big" inspirations as with an *ad hoc* approach, a number of "small" inspirations are required with an approach based on FTA.

7.1 Relationship to Fault Tree Analysis

VTM is based on FTA. The relationship of VTM with FTA is described further in this section.

7.1.1 Comparison to FTA

Like FTA, VTM begins with an assumption that the hazard has occurred and then works "backwards" to systematically cover all of the possible ways in which this condition might have arisen. However, unlike FTA, not all of the steps in the analysis will necessarily involve a relationship between an event and its cause. For example, the mitigation of a stale temperature value display described in Section 6.4 involved steps 4, 5 and 6 that were a matter of pure logical reasoning.

In general, FTA is limited to a form of propositional reasoning where each branch in the tree corresponds to a disjunction or a conjunction. VTM also uses Boolean logic, but the cases are based on the "AND/OR" structure exposed by the temporal relationships when expressed in the constrained syntax of VTM. In addition to Boolean logic, VTM makes informal use of quantification and rules of reasoning based on predicate logic.

7.1.2 Comparison to SFTA

SFTA also employs "proof-by-contradiction" style reasoning to show that each disjunctive branch of the argument leads to a logical contradiction. However, for VTM, the reasoning is not tied specifically to templates based on the syntax and semantics of a programming language. Whereas SFTA is intended to be used as means of verifying the source code with respect to a defined hazard, the approach presented in this dissertation is meant to support the derivation of system level SVCs.

7.2 Deriving Temporal Relationships

VTM enhances FTA with a constrained syntax and mathematical reasoning for deriving safety-related temporal relationships.

7.2.1 Constrained Syntax

Expressing the SVCs with the constrained syntax of VTM helps identify key unknown quantities. For example, the chemical factory hazard depends upon the allowed difference between the actual and displayed temperature, MAX_DISP_TEMP_DIFF, and the length of time a temperature value can be displayed without being considered stale, MAX_DISP_TEMP_STALE. These unknowns may be hidden or obscured by more conventional natural language. It is also easier to reason about the relationships between these unknowns when they are represented with symbolic names.

Though not demonstrated with the chemical factory example, casting a temporal relationships into a notational form exposes the "AND/OR" structure of the temporal relationship. Once this has been done, it is then possible to use familiar FTA-style of reasoning based on the "AND/OR" structure.

7.2.2 Mathematical Reasoning

VTM uses informal but rigorous mathematical reasoning for analysis of the safety-related temporal relationships. Most of the reasoning required by the chemical factory example was a matter of arithmetic reasoning about inequalities. The reader may try his or her hand at the derivation of LC-5 from LC-5a in Section 6.4.2 to get a sense that this kind of reasoning, though error-prone, is nothing more than “high school level mathematics”. A bit of predicate logic reasoning was implicitly used in several steps, such as the refinement of LC-2a into LC-2. To ensure greater confidence in the analysis and the resulting SVCs, it may be useful to perform the analysis with the aid of a formal verification tool.

7.2.3 Temporal Relationships

Application of VTM to the chemical factory system results in temporal relationships such as SVC-4, a relationship between system constants S2 and S3. This constraint is necessary to ensure that the process monitoring the staleness of a temperature value sets the temperature display to “unavailable” before it goes stale. It is possible that this relationship, among others, may have been postulated based on engineering judgment alone. However, the fact that SVC-4 is required to mitigate the hazard, and not others, was not obvious to the author before applying VTM to the hazard.

The use of VTM results in some subtle temporal details in the SVCs. For example, SVC-5 states that the system shall set the temperature display to “unavailable” for vessel within S3 milliseconds of the most recent update for vessel V. However, SVC-5 also states that the update should occur only after S1 milliseconds had elapsed since the most recent update. The choice of S1 as the lower bound is necessary if SVC-5 is to be used to derive the contradiction, LC-6, from LC-5. If a value smaller than S1 is chosen, LC-6 would then allow the temperature of vessel V to be set to "unavailable", or to some other value other than D, at some time before time T1. As a result, there would be no contradiction. This corresponds to the following hazard scenario:

1. The system receives a final update.
2. Before the update is displayed, the process that monitors the staleness of the update determines that the update is stale and sets the temperature display to "unavailable".
3. The final update is subsequently displayed and eventually becomes stale.

The inclusion of the lower bound closes off that particular hazard scenario. The necessity of a lower bound is not particularly obvious – and its specification is easily overlooked.

Another subtle feature about SVC-5 is the requirement that the temperature display exhibit some other temperature value if it has not been set to "unavailable". This eliminates the possibility of a displayed temperature value becoming stale due to temperature updates being received but not displayed. This corresponds to the following hazard scenario:

1. The system displays a valid temperature value.
2. Further updates are received but not displayed by the system.
3. The system does not set the vessel temperature display to "unavailable".
4. The displayed temperature value eventually becomes stale.

In general, it would be difficult to determine the precise details of these temporal relationships without using a method such as VTM.

8. Formalization

VTM involves the use of informal techniques such as a constrained syntax and mathematical reasoning to generate SVCs. Though these techniques were found to be effective, there are alternative formal techniques that could be used. The constrained syntax could be replaced with a formal specification notation. For example, a temporal or real-time logic notation could be used to capture the temporal aspects of the SVCs. As well, the informal, rigorous argument constructed with VTM could be formally validated.

It may be worthwhile to formally validate the rigorous argument with a formal verification technique. For example, an earlier version of the derivation of the chemical factory SVCs contained errors that invalidated the final rigorous argument. These errors included a version of SVC-5 that contained a "loophole" that would have allowed a stale temperature value to be displayed if the system received but failed to propagate a temperature value. This corresponds to the second hazard scenario described in Section 7.2.3. These errors were sufficiently subtle to escape detection by three external reviewers in an earlier version of the analysis.

The rest of this section illustrates how aspects of the VTM approach can be formalized. In particular, the hazards and SVCs can be expressed in a formal specification notation. As well, the rigorous safety argument can be represented as a logical conjecture in predicate logic and then validated with a formal verification tool.

8.1 Formal Specification

A notation based on typed predicate logic is used to formally specify the SVCs. This notation, called S, was developed at the University of British Columbia to serve as a foundation for a variety of different approaches to formal specification [JDD94]. Statements expressed in the constrained syntax of VTM can be systematically translated into S.

An S specification is a sequence of paragraphs. Each paragraph is a fragment of ASCII text terminated by a semi-colon which serves one of the following purposes:

- declares or defines a new type;
- introduces an abbreviation for a type expression;
- declares or defines a new constant;
- declares or defines a new function;
- declares or defines a new predicate;
- expresses an assertion.

Quantities of time and instants in time are treated as numbers. For instance, an S type abbreviation,

```
:time==num;
```

is used to introduce the name, `time`, to formally represent time as the pre-defined S type, `num`. Instants in time (e.g., T1) and durations (e.g., S1) can then be represented as constants of type `time`:

```
T1:time;
S1:time;
```

Events are represented as an uninterpreted type, as defined by the following S type declaration:

```
:event;
```

Events typically involve vessels and their temperatures. Vessels are defined as uninterpreted types, and temperatures as numbers, as given by the type declarations:

```
:vessel;
:temperature==num;
```

Events can then expressed as uninterpreted, infix functions of vessels and temperatures. For example, the event involving the display of a vessel temperature is given by the infix predicate:

```
(_ displayed_for _) : temperature -> vessel -> event;
```

As presented here, this predicate is left "uninterpreted" on the basis of a decision that the relationship expressed by this predicate does not need to be formalized. Alternatively, this predicate could be defined, rather than declared, as a means of formalizing this relationship.

As discussed in Section 5.1, the SVCs involve time references that are predicates which describe the occurrence of particular events at particular instants in time. The time reference can be expressed as the uninterpreted, infix predicate:

```
(_ at_time _): event -> time -> bool;
```

Predicates can be used in combination to make assertion. For instance, the infix predicate "__ displayed_for __" can be used in combination with the infix predicate "__ at_time __" to formalize the phrase "temperature D is displayed for vessel V at time T":

```
(D displayed_for V) at_time T
```

The precedence rules for infix predicates allow the parentheses in this assertion to be dropped, yielding:

```
D displayed_for V at_time T
```

The constrained syntax of VTM uses bound variables and quantification operators. Bound variables are used to denote instants in time. These are bound by a quantification operator such as "for all" or an existential operator such as "exists". The logical operators such as NOT, AND, OR and IMPLIES are then used to express constraints. These constraints are similarly expressed in S. For example, SVC-1:

SVC-1.

For all temperatures, d, times, t, and vessels, v, if d is displayed at time t as the temperature of vessel v, then there is some time t', t' <= t, when the temperature of vessel v was set to d and this was the most recent change made to the displayed temperature for vessel v.

can be formalized in S as follows:

SVC-1 (formalized in S).

```
forall (d:temperature) (t:time) (v:vessel).
if ((d displayed_for v) at_time t)
then (exists (t':time).
    (let temperature_set := (d set_for v) in
    (t'<=t
    AND (temperature_set at_time t)
    AND (Most_recent temperature_set t' t))));
```

The above formalized version of SVC-1 employs a number of auxiliary predicates, including set_for and Most_recent.

set_for is an uninterpreted, infix function:

```
(_ set_for _) : temperature->vessel->event;
```

Most_recent is a predicate defined by:

```
Most_recent (E:event) (t1:time) (t2:time) :=
(E at_time t1) AND
```

```
(forall t.
if (t1 < t AND t <= t2)
then NOT(E at_time t));
```

The formalized hazard and SVCs for the chemical factory can be found in Appendix A.

8.2 Formal Validation

Though significant benefit may be achieved from simply representing the hazards and SVCs in a formal notation, it may also be useful to formally validate the rigorous safety argument constructed with VTM.

The safety argument can be expressed as a logical conjecture in predicate logic:

```
SVCs AND Definitions -> NOT(Hazard)
```

Validating the safety argument is then a matter of demonstrating that this conjecture is a theorem.

Formal verification systems, such as PVS [OSRH95] and HOL [GM93], can be used to automate aspects of the proof. For example, the safety argument could be systematically translated into a proof problem in HOL. In fact, this translation can be performed algorithmically. A prototype tool [JDD94] exists for automatically translating S statements into a format acceptable as input into HOL. HOL can then be used to help demonstrate that the above safety conjecture is a theorem, though this has not yet been done.

The result of the translation is that a safety problem is converted into a HOL proof problem. Though it may turn out that some domain and safety knowledge is also needed, carrying out the proof should require mainly HOL expertise.

9. Verifiable Code Assertions

The safety verification of software-intensive systems will ultimately require verifying the source code with respect to the hazards. Toward that end, it would be useful to be able to refine the hazard into a set of source code level SVCs. As discussed in Section 4.3, one difficulty with performing the refinement is the large "semantic gap" between the source code and the abstract "system level" concepts and language used to define hazards. To address this problem, VTM can be extended beyond the system level to systematically refine a safety-related hazard into a set of verifiable code assertions.

The derivation of safety code assertions was originally motivated by our interest in the possibility of using code verification tools such as SPARK Examiner [Bar97] as part of the safety verification of a large software system. Chapter 18 of Leveson's seminal textbook on software safety [Lev95] hints at the possibility of using such tools, but expresses concern about their practical feasibility. Clearly, it would be naive to expect that the safety verification task could be automated by simply feeding the source code for an entire system into a verification tool along with a representation of a safety-related hazard. If code verification tools are to be used in the safety verification of the system, it will be necessary to process the hazard and the source code into a form that the tools can accept as input.

One problem is that the safety-related hazard is likely to be expressed at a much higher level of abstraction, and in a different form, than the assertions expected as input by the code verification tool. VTM can be used as part of a hazard-refinement process to address this particular problem. There are other problems with using code verification tools for the safety verification of a large software system. The discussion of these problems will be the subject of future work. The hope is that code verifications tools such as SPARK Examiner may prove to be useful in the safety verification of a large software system. Even if such tools are not used, the refinement of a safety-related hazard into a set of verifiable assertions would support other methods of static analysis such as manual inspection.

9.1 Process for the Development of Verifiable Code Assertions

The process for generating source code level SVCs bridges the semantic gap between the hazard and source code though a series of refinement steps. This includes the development of SVCs at the system, component and source code level.

The inputs of the process are:

- Hazards;
- Software architecture;
- Hazard-related source code.

The safety code assertions are created for the hazard-related source code. In this dissertation, it is assumed that a representation for the hazard-related source code has been created in the SPARK Ada subset [Bar97].

The output of the process is:

- Safety code assertions.

The safety code assertions are pre- and post-conditions written as SPARK annotations.

The steps of the refinement process are:

1. **Generate system level SVCs**. VTM is used to produce the system level SVCs.
2. **Generate component level SVCs**. The system level SVCs are refined into component level SVCs by extending the rigorous argument to the design level. The hazard-related software is partitioned into “functional blocks” of code, and the system level SVCs are mapped into conditions on the functional block’s input and output parameters.
3. **Generate source code level SVCs.** The component level SVCs are refined into source code level SVCs through analysis of the source code. The functional block parameters are identified in the source code, and the functional block SVCs are re-written in terms of the source code parameters.
4. **Generate Code Assertions.** The “backwards” source code level SVCs are reformulated in a “forward” fashion and translated into SPARK annotations.

A demonstration of the derivation of system level SVCs with VTM was given in Section 6. The rest of Section 9 will outline steps involved in the derivation of component level and source code level SVCs.

9.2 Component level SVCs

The system level SVCs are refined into component level SVCs by extending the system level rigorous safety argument to the relevant software components. For example, the hazard-related software can be partitioned into blocks which are invoked asynchronously. This could be the result of code running in a process on a different computer, or which executes in a different “thread” on the same processor. The resulting “functional blocks” of code may be viewed as procedures with input and output parameters. Component level SVCs can then be derived for these functional blocks.

The refinement is performed on the system level SVCs that place constraints on system functionality. This involves identifying the functional blocks that carry out the "black box" behavior described by these SVCs. VTM can then be applied where the initial conjecture is that the system level SVC does not hold. During the construction of the proof-by-contradiction argument, SVCs are introduced for the functional blocks.

The system level SVCs are then refined into separate functional and temporal conditions, with the system events mapped onto the functional block invocations and output. Furthermore, the functional block becomes the causal agent in that it takes input and creates output. This replaces the temporal sequence of events. Functional block pre- and post-conditions can then be derived from the time-independent functional block SVCs.

9.3 Source Code Level SVCs

The component level SVCs are refined into source code level SVCs by identifying the functional block input and output parameters, along with any other relevant source code element. The input and output parameters of the functional block are determined from examination of the subprograms that make up the block. The component level SVC can then be re-written in terms of these source code elements.

9.4 Code Assertions

To derive the code assertions, the source code level SVCs must first be expressed in pre- and post-condition style. This involves reformulating "backward" SVCs into a "forward" conditions . The "forward" conditions are then translated into SPARK annotations.

9.4.1 Pre- and Post-Conditions

Pre- and post-conditions are such that, given the pre-conditions on the input parameters, the execution of the program should result in the output variables satisfying the post-conditions:

If pre-conditions on input

then post-conditions on output

SVCs are sometimes expressed in a “backward” fashion:

If conditions on output

then conditions on inputs

As discussed in Section 6.2, some of the system level SVCs for the chemical factory information system are examples of this "backwards" orientation.

It will be necessary to write the SVCs in a “forward” fashion, in order to arrive at a pre- and post-condition style specification. Often it is possible to obtain a “forward” version of the SVCs from its “backward” formulation by taking its contrapositive:

If NOT(conditions on input)

then NOT(conditions on output)

This should place the input conditions into the antecedent of the SVC.

9.4.2 SPARK Annotations

At this point, the source code level SVCs can be used as input into the code verification process. One possibility is conventional software testing. However, in this dissertation, we consider the possibility of using a tool-based method based on static verification, in particular, use of SPARK Examiner.

SPARK Examiner may be used to reduce the problem of verifying a "slice" of the code with respect to a source code SVC into the purely mathematical task of verifying a logical expression called a "verification condition". To complete the overall process, the verification conditions must then be verified either by manual efforts or by use of tools such as the SPARK Simplifier and Proof Checker [Bar97].

SPARK annotations makes use of the SPARK Ada subset and appears in the code as Ada comments. The annotations are used to express code assertions such as pre- and post-conditions. However, they do not support quantification, so it may not be possible to directly express the source code level SVC in the SPARK annotation language. If quantification is required, a “proof function” is defined which has the same syntax as an Ada function and the post-condition is then expressed in terms of this proof function. The proof function can then be defined as a proof rule expressed in the FDL language. FDL supports quantification and is the required input for the SPARK Proof Checker.

Transforming the hazard into a set of machine-readable code assertions is the first step towards the use of a code verification tool such as SPARK Examiner in safety verification. A key benefit of a code verification tool is that it relieves the human analyst of the task of tracing through the code slice statement-by-statement.

10. Future Work

VTM can be incorporated into an overall method for performing system safety verification. While methods for performing “front-end” steps such as hazard analysis are relatively well understood and widely used, methods for performing safety verification are less well established. Several methods have been proposed as a means of verifying source code with respect to identified hazards, but there are few published discussions of how these techniques could be applied to anything other than relatively small isolated subsystems.

The goal of this future work would be to strengthen the “back-end” of the overall safety engineering process for large software-intensive information systems. The approach would center upon the production of a document that records the safety verification of the source code The term “safety verification case” is introduced to refer to this document. This use of this term is intended to emphasize the relationship of this document to the overall system safety case. The safety verification case is intended to ensure the inspectability, maintainability and repeatability of the source code safety verification.

The proposed process for developing the safety verification case accepts as inputs:

- a large amount of **source code** (e.g., several hundred thousand SLOCs) which serves as a concrete representation of the implementation of the system;
- a set of identified **hazards** expressed at a relatively high level of abstraction.

The output of the process is:

- a **safety verification case** with detailed evidence that provides a rigorous argument that the hazard cannot occur given specific, explicitly stated assumptions about the hardware and software.

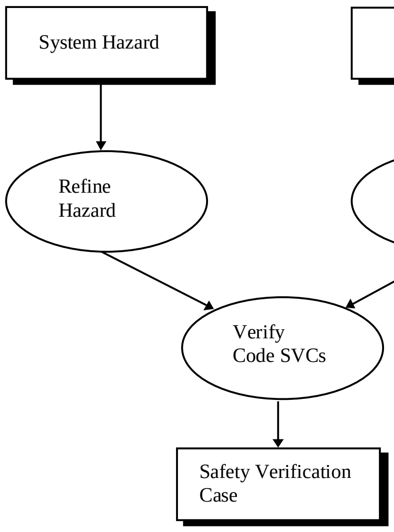


Figure 9: Safety verification process.

The process is based on the strategy of systematically refining the safety verification process into a series of simpler steps. These steps isolate the relevant details of the source code, and bridge the "semantic gap" between the source code and the abstract system level concepts used to define hazards. It may be possible to partially automate the process steps with tool-based support.

The steps of the process include:

1. step-wise refinement of the hazard into one or more source code level SVCs;
2. step-wise extraction of models of one or more critical code paths;
3. step-wise verification that each of the source code level SVCs (obtained from refinement documented in Part 1) is satisfied by the model of the source code implementation (obtained as a result of the extraction documented in Part 2).

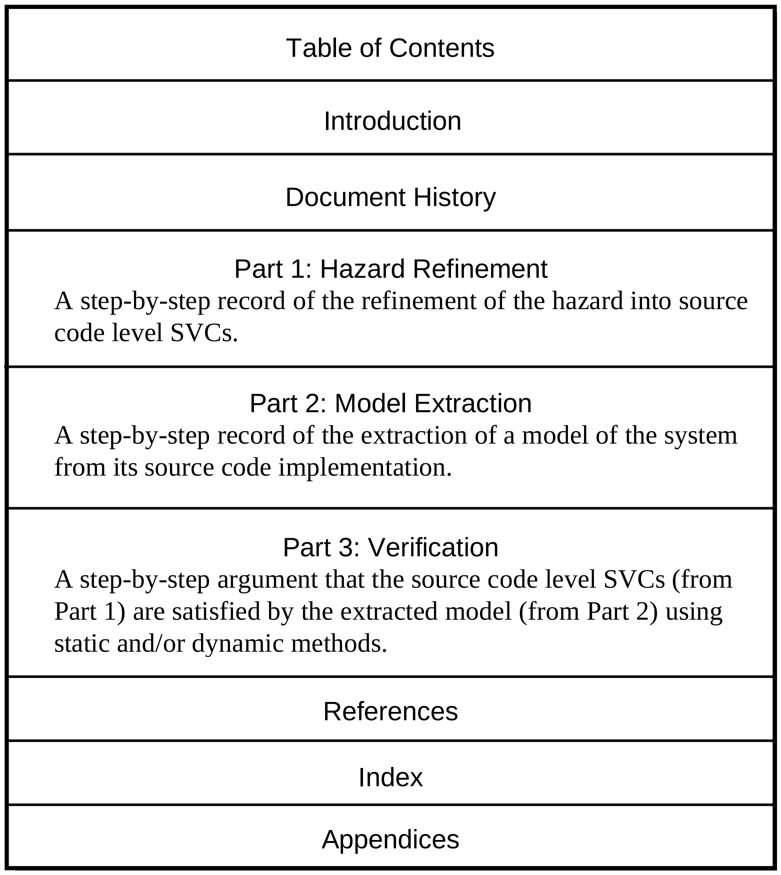


Table of Contents

Introduction

Document History

Part 1: Hazard Refinement
A step-by-step record of the refinement of the hazard into source code level SVCs.

Part 2: Model Extraction
A step-by-step record of the extraction of a model of the system from its source code implementation.

Part 3: Verification
A step-by-step argument that the source code level SVCs (from Part 1) are satisfied by the extracted model (from Part 2) using static and/or dynamic methods.

References

Index

Appendices

Figure 10: Structure of the Safety Verification Case.

Figure 9 shows a process flow diagram that represents the development of the safety verification case. The left-hand side of this diagram represents the derivation of source code level SVCs from the definition of a hazard. The right-hand side represents the extraction of a model for one or more critical code paths for the hazard. At the bottom of the process flow diagram, the outputs of the hazard refinement and of the model extraction are used as inputs to the verification process. The output of the process is the safety verification case. The structure of the safety verification case is shown in Figure 10.

10.1 Part 1: Hazard Refinement

The "Hazard Refinement" part of the safety verification case contains a set of source code level SVCs along with a record of the execution of the process used to obtain these SVCs as a refinement of the hazard definition. This refinement may be performed in an *ad hoc* manner or it may employ a more systematic approach such as FTA, or a related technique such as VTM. In either case, both the execution of the process and its results must be recorded in a form which would allow the process to be repeated.

The source code level SVCs may be expressed as informal English statements. Alternatively, some of the SVCs may be expressed in a “codified form” using a notation that can be parsed by a software tool. The use of a codified form may be effective as a means of ensuring the precision of the SVCs as well as avoiding ambiguity and other specification related problems associated with the exclusive use of informal English. The specification of SVCs in codified form such as the SPARK annotation notation [Bar97] may also allow enable verification tools such as SPARK Examiner to be used.

In addition to the specification of the results of the hazard refinement process, this part of the safety verification case must also contain a record of the refinement. If the refinement was performed in an *ad hoc* manner, this record may only be a narrative account of the engineering judgment used to obtain the SVCs. On the other hand, the use of FTA or a related technique as a means of deriving the SVCs can be recorded by a graphical representation supplemented by textual annotation.

10.2 Part 2: Model Extraction

The "Model Extraction" part of the safety verification case contains a conservative, complete and tractable model of a critical code path for the hazard along with a record of the process used to extract this model from the source code level implementation of the system.

The critical code paths correspond to the hazard scenarios revealed by the results of the hazard refinement contained in Part 1 of the safety verification case. As described in Section 2.2.5, hazard scenarios are mapped onto the source code. The source code identified by a source code level hazard scenario is what we refer to as the “critical code path” for the scenario.

A variety of different formats may be used to represent the model of a critical code path extracted from the source code. One possibility is the representation of the model in an executable format, such as the same programming language used for the actual implementation or possibly in a closely related programming language. For example, the model of a critical code path for a system implemented in Ada might be represented in the SPARK programming language. An executable model would allow the use of software tools to partially automate the verification task by means of dynamic methods (e.g., testing) or static methods (e.g., code verification).

10.3 Part 3: Verification

The third part of the safety verification case documents the rigorous argument that demonstrates that each SVC generated by the refinement documented in Part 1 is satisfied by the representation of the code developed in Part 2. The basic elements of the rigorous argument are claims, assumptions, facts and inference steps.

11. Conclusions

The safety verification of critical systems relies upon the derivation of SVCs which, if satisfied, provides reasonable evidence that the hazards have been eliminated or mitigated. At the system level, SVCs may be regarded as safety requirements that provide the definition of safety during system development. In addition to constraints on the system functionality, system level SVCs include system and environmental assumptions, as well as relationships between system constants. SVCs can also be constructed at the software component and the source code levels. This closes the “semantic gap” between the definition of the hazard and the source code. The source code level SVCs can be translated into machine-readable code assertions that may be verified with the help of a code verification tool.

VTM provides a systematic approach to the derivation of SVCs. At the system level, VTM is based upon the results of the system hazard analyses, where the system is treated as a “black box”. FTA is performed to determine the system and environmental events that lead to a hazard. A constrained syntax is used to capture the temporal relationships between these events. SVCs are introduced to close off branches of the fault tree and to steer the argument toward a contradiction. Mathematical reasoning is employed to close off the remaining branches of the fault tree.

VTM uses the constrained syntax and mathematical reasoning to analyze complex temporal relationships in real-time hazards. The ability to systematically derive key temporal constraints and relationships from hazard definitions is what distinguishes VTM most clearly from other existing methods.

The result of applying VTM to a system hazard is a complete set of SVCs that support a rigorous argument that may be used to increase confidence in the safety of the system. This informal argument can be validated with the help of a formal verification system. Of course, the validity of this argument ultimately depends on showing that the implementation of the system satisfies the SVCs.

Bibliography

[AIB96]	ARIANE 5 Inquiry Board, “ARIANE 5 Flight 501 Failure Report by the Inquiry Board”, Paris, July 1996.
[BGBG95]	Ronald M. Baecker, Jonathan Grudin, William A. S. Buxton and Saul Greenberg, “Readings in Human-Computer Interaction: Toward the Year 2000” (Second Edition), Morgan Kaufmann Publishers Inc., San Francisco, California, 1995.
[Bar97]	John Barnes, “High Integrity Ada The SPARK Examiner Approach”, Addison Wesley Longman Ltd., 1997.
[Bis98]	Peter G. Bishop and Robin E. Bloomfield, “A Methodology for Safety Case Development”, in Safety-Critical Systems Symposium, Birmingham, UK, February 1998.
[Boe88]	Barry Boehm, “A Spiral Model of Software Development and Enhancement”, *IEEE Computer*, May 1988.
[Boo94]	Grady Booch, “Object-Oriented Analysis and Design with Applications (Second Edition)”, Benjamin/Cummings Pub. Co., Redwood City, California, 1994.
[DOD93]	Department of Defense, “Military Standard 882C: System Safety Program Requirements”, 1993.
[ER96]	Bruce Elliott and Jim Ronback, “A System Engineering Process For Software-Intensive Real-Time Information Systems, in *Proceedings of the 14th International System Safety Conference*, Albuquerque, New Mexico, August 1996.
[For98]	http://www.cs.ubc.ca/formalWARE/
[FMNP94]	P. Fenelon, J.A. McDermid, M. Nicholson and D. J. Pumfrey., “Towards Integrated Safety Analysis and Design”, *ACM Computing Reviews*, 2(1), p. 21-32, 1994.
[GM93]	Mike J. Gordon and Tom F. Melham, “Introduction to HOL: A Theorem Proving Environment for Higher Order Logic”, Cambridge University Press, Cambridge, UK, 1993.
[GW96]	J. Gorski, A. Wardzinski. "Deriving Real-Time Requirements from Safety Analysis", Proceedings of the EUROMICRO Workshop on Real Time Systems. IEEE Computer Society Press. L'Aquila, Italy. June 12-14 1996. pp. 9-14.
[Gup92]	Aarti Gupta, “Formal Hardware Verification Methods: A Survey”, *Formal Methods in System Design*, October 1992.
[Han96]	Kirsten Mark Hansen, “Linking Safety Analysis to Safety Requirements - Exemplified by Railway Interlocking Systems”, Ph.D. Thesis, Technical University of Denmark, Department of Information Technology, August 1996.
[KRS98]	Kirsten M. Hansen, Anders P. Ravn and Victoria Stavridou, “From Safety Analysis to Software Requirements”, *IEEE Transactions on Software Engineering*, vol. 22, no. 7, July 1998.
[IEEE94]	The Institute of Electrical and Electronic Engineers, Inc., “IEEE Standard for Software Safety Plans”, IEEE Std 1228-1994, NY, 1994.
[IEC95]	International Electrotechnical Commission, “Draft International Standard IEC 1508: Functional Safety: Safety Related Systems”, Geneva, 1995.

[IW95] Laura M. Ippolito and Dolores Wallace, "A Study on Hazard Analysis in High Integrity Software Standards and Guidelines", NISTIR 5589, National Institute of Standards and Technology, January 1995.

[JM86] F. Jahanian and A. K. Mok, "Safety Analysis of Timing Properties in Real-Time Systems", *IEEE Transactions on Software Engineering*, vol. 17, no. 3, March 1991.

[JDD94] Jeff J. Joyce, Nancy Day, and Mike Donat, "S: A Machine Readable Specification Notation Based on Higher Order Logic", in *7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pp. 285-299, 1994.

[Lev95] Nancy G. Leveson, "Safeware: System Safety and Computers", Addison-Wesley, 1995.

[Lev91] Nancy G. Leveson, "Software Safety in Embedded Systems", *Communications of the ACM*, 34(2), pp. 34-46, February 1991.

[LCS91] Nancy G. Leveson, Steven S. Cha, and Timothy J. Shimall, "Safety Verification of Ada Programs using software fault trees", *IEEE Software*, 8(7), pp. 48-59, July 1991.

[HHR94] Nancy G. Leveson, Mats P. E. Heimdahl, Holly Hildreth and Jon D. Reese, "Requirements Specifications For Process-Control Systems", *IEEE Transactions on Software Engineering*, vol. 20, no. 9, March 1994.

[LS87] Nancy G. Leveson and Janice L. Stolzy, "Safety Analysis Using Petri Nets", *IEEE Transactions on Software Engineering*, vol. 13, no. 3, March 1987.

[LW96] Robyn R. Lutz and Robert M. Woodhouse, "Experience Report: Contributions of SFMEA to Requirements Analysis", in *Proceedings of ICRE'96*, 1996.

[MLRPS97] Francesmary Modugno, Nancy G. Leveson, Jon D. Reese, Kurt Partridge, and Sean D. Sandys, "Integrated Safety Analysis of Requirements Specifications", in *Proceedings of the 3rd International Symposium on Requirements Engineering*, Annapolis, Maryland, January 1997.

[OSRH95] S. Owre, N. Shankar, and J.M. Rushby, and F. von Henke, "Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS*," IEEE Transactions on Software Engineering*, vol. 21, no. 2, February 1995.

[RHH93] Anders P. Ravn, Hans Rischel and Kirsten Mark Hansen, "Specifying and Verifying Requirements of Real-Time Systems", *IEEE Transactions on Software Engineering*, vol. 19, no. 1, January 1993.

[Ree96] Jon D. Reese, "Software Deviation Analysis", Ph.D. Thesis, University of California, Irvine, 1996.

[SL94] John A. Scott and J. Dennis Lawrence, "Testing Existing Software For Safety-Related Applications", Fission Energy and Systems Safety Program, Lawrence Livermore National Laboratory, UCLR-ID-117224, Revision 7.1, December 1994.

[Sto96] Neil Storey, "Safety-Critical Computer Systems", Addison-Wesley, 1996.

[VGRH81] W. E. Vesley, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. "Fault Tree Handbook". NUREG-0942, U.S. Nuclear Regulatory Commission, 1981.

Appendix A. Formalized Chemical Factory SVCs

```
%include startup.s

% Type Definitions

:event;
:time==num;

:temperature==num;
:vessel;

% Functions
(_ at_time _): event -> time -> bool;

(_ displayed_for _) : temperature -> vessel -> event;
(_ set_for _) : temperature -> vessel -> event;
(_ received_for _) : temperature -> vessel -> event;
(_ actual_for _) : temperature -> vessel -> event;

ABS : num -> num;

Most_recent (E:event) (t1:time) (t2:time) :=
(E at_time t1) AND
(forall t.
    if (t1 < t AND t <= t2)
    then NOT(E at_time t));


% Constants
Unavailable : temperature;

MAX_DISP_TEMP_DIFF:temperature;
MAX_DISP_TEMP_STALE:time;

S1:time;
S2:time;
S3:time;

% Hazard - Initial Conjecture
%IC. The temperature, D, displayed for vessel V at time t has not been within
MAX_DISP_TEMP_DIFF degrees of the actual temperature of the vessel at any
time within MAX_DISP_TEMP_STALE milliseconds before time t.


Hazard :=
exists (D:temperature) (V:vessel) (t:time).
((D displayed_for V) at_time t)
```

```
AND NOT(exists (D1:temperature) (t1:time).
     (t1 <= t) AND (t - t1 <= MAX_DISP_TEMP_STALE)
     AND (ABS(D - D1) <= MAX_DISP_TEMP_DIFF)
     AND ((D1 actual_for V) at_time t1));
```

% Safety Verification Conditions

% SVC-1.For all temperatures d, times t, and vessels v, if d is displayed at time t as the temperature of vessel v, then there is some time t', t' <= t, when the temperature of vessel v was set to d and this was the most recent change made to the displayed temperature for vessel v.

```
SVC1 :=
forall (d:temperature) (t:time) (v:vessel).
if ((d displayed_for v) at_time t)
then (exists (t':time).
     (let temperature_set := (d set_for v) in
     (t'<=t
     AND (temperature_set at_time t)
     AND (Most_recent temperature_set t' t))));
```

% SVC-2. For all vessels v, displayed temperatures d, and times t, if the displayed temperature of vessel v is set to d at time t then at some time no earlier than S1 milliseconds before t the system received a report from the external sensor monitoring system that the temperature of vessel v is d.

```
SVC2 :=
forall (d:temperature) (t:time) (v:vessel).
if ((d set_for v) at_time t)
then (exists (t':time).
     t'<=t AND t - t' <= S1
     AND ((d received_for v) at_time t'));
```

% SVC-3. For all vessels v, displayed temperatures d, and times t, if the system receives a report at time t from the external sensor monitoring system that the temperature of vessel v is d, then at some time no earlier than S2 milliseconds before t the actual temperature of vessel v was within MAX_DISP_TEMP_DIFF degrees of d.

```
SVC3 :=
forall (d:temperature) (t:time) (v:vessel).
if ((d received_for v) at_time t)
then (exists (t':time) (d':temperature).
     t'<=t AND t - t' <= S2
     AND ABS(d - d') <= MAX_DISP_TEMP_DIFF
     AND ((d' actual_for v) at_time t'));
```

% SVC-4. MAX_DISP_TEMP_STALE > S2 + S3

```
SVC4 :=
MAX_DISP_TEMP_STALE > S2 + S3;
```

% SVC-5. For all vessels v, and times t and t’, if time t is the most recent time that an update for vessel v at temperature d was received prior to time t’, and t’-t > S3, then at some time, t’’, such that t+S1 < t’’ <t’, the temperature value displayed for vessel v shall have been set to "unavailable" or shall exhibit some value other than d.

```
SVC5 :=
forall (d:temperature) (v:vessel) (t:time) (t’:time).
if ( (Most_recent (d received_for v) t t’) AND (t’ - t > S3) )
then (exists (t’’:time).
     t+S1 <= t’’ AND t’’ < t’
     AND ( ((Unavailable displayed_for v) at_time t’’)
                 AND (Unavailable != d))
         OR (exists (d’:temperature).
         ( ((d’ displayed_for v) at_time t’’) AND (d’ != d))));
```