UNTANGLING SAFETY-CRITICAL SOURCE CODE

by

KEN WONG

M.Sc., University of British Columbia, 1998
M.Sc., The University of Saskatchewan, 1989
B.Sc., The University of Saskatchewan, 1985

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

in

THE FACULTY OF GRADUATE STUDIES

(Electrical And Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

April 2005

© Ken Wong, 2005
Abstract

The traditional system safety paradigm of isolating safety-critical functionality is no longer tenable in the face of the increased size and complexity of modern software systems. Software is unlike purely mechanical systems where safety concerns typically correspond to "functional slices" that can be extracted and tested in isolation. Instead, specific safety concerns often cross-cut the primary software structures and involve fragments of source code from many different software components.

The fact that safety-critical software is not always easily isolated, documented and analyzed for safety verification is an issue we have previously called the "long thin slice problem". This problem was initially identified for a class of systems that display critical information, such as Air Traffic Management (ATM) systems. The research in this dissertation is largely motivated by the author's experiences with the development of an advanced ATM system that is now deployed for operational use in Canada.

This dissertation outlines an approach intended to strengthen the “back-end” of the overall software safety engineering process, i.e., verifying the safety of the implementation. Software safety has primarily focused on “front-end” of the safety process, for good reason, as building safety into the specification and design is the most effective method of mitigating safety risk. However, the safety of the source code must ultimately still be verified to provide evidence that the risk has been truly mitigated. Various safety verification methods have been proposed, but there are few published discussions of how these techniques could be applied to anything other than relatively small, isolated subsystems.

Given the difficulty in performing safety verification of the entire system, we propose extracting a model of the part of the system that is relevant to a particular safety concern. Constructing the model involves untangling the safety concern from the rest of the system functionality. In order to create a model that can be effectively used for safety verification, we have experimented with a variety of techniques that range from the use of natural language to the use of a formal specification notation.
# Table of Contents

Abstract ii

Table of Contents iii

Table of Figures v

Acronyms vii

Acknowledgments viii

## 1. Introduction 1

1.1 Thesis Statement 3
1.2 Safety and Modern Software Architectures 4
1.3 Existing Safety Verification Techniques 6
1.4 Creating Safety Concern Models for Safety Verification 9
1.5 Case Studies 9
1.6 Contributions of this Dissertation 10
1.7 Overview of Dissertation 11

## 2. Background 13

2.1 Safety-Critical Software 13
2.2 System Safety 14
2.3 Software Safety 21
2.4 Object-Oriented Software Architectures 23
2.5 Safety Verification Case 25
2.6 Examples 29
2.7 Summary 36

## 3. Software Safety Concerns 37

3.1 Safety Concerns in the Source Code 37
3.2 Example: Chemical Factory Management System Safety Concern 41
3.3 Other Examples 48
3.4 Summary 50

## 4. Foundations of Safety Concern Models 52

4.1 Safety Concern Models 52
4.2 Safety Concern Model Properties 56
4.3 Safety Concern Model Vocabulary 57
4.4 Summary 59

## 5. Constructing Safety Concern Models 61

5.1 A Process for Constructing a Safety Concern Model 61
5.2 Supporting Techniques and Tools 75
5.3 Maintaining the Code Model 80
5.4 Validation – ATM System 81
5.5 Summary 81
Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>The relationship between hazards, hazard causes and mishaps.</td>
<td>15</td>
</tr>
<tr>
<td>2-2</td>
<td>Relationship between unsafe and incorrect behaviour</td>
<td>18</td>
</tr>
<tr>
<td>2-3</td>
<td>Basic Safety Tasks</td>
<td>19</td>
</tr>
<tr>
<td>2-4</td>
<td>Hypothetical hazard analysis for an electrical wiring hazard for a system.</td>
<td>20</td>
</tr>
<tr>
<td>2-5</td>
<td>Safety verification process.</td>
<td>27</td>
</tr>
<tr>
<td>2-6</td>
<td>Part of an air traffic controller’s “situation display”.</td>
<td>30</td>
</tr>
<tr>
<td>2-7</td>
<td>The ATM software architecture adapted from figure 5 of [KRU95]</td>
<td>32</td>
</tr>
<tr>
<td>2-8</td>
<td>Chemical Factory</td>
<td>33</td>
</tr>
<tr>
<td>2-9</td>
<td>The layers of the static architecture from the implementation view.</td>
<td>35</td>
</tr>
<tr>
<td>3-1</td>
<td>Schematic of an ABS Braking System [VW04]</td>
<td>38</td>
</tr>
<tr>
<td>3-2</td>
<td>The Hazard-Related Software as a Cross-Cutting Concern</td>
<td>43</td>
</tr>
<tr>
<td>3-3</td>
<td>Temperature data flow highlighting the key components and procedures.</td>
<td>44</td>
</tr>
<tr>
<td>3-4</td>
<td>Fragment of the “ProcessSensors” subprogram source code</td>
<td>45</td>
</tr>
<tr>
<td>3-5</td>
<td>Radar Data Processor Class Diagram from [JF04]</td>
<td>50</td>
</tr>
<tr>
<td>4-1</td>
<td>Extracting a model of the safety concern</td>
<td>55</td>
</tr>
<tr>
<td>5-1</td>
<td>Outline of Document Containing the Safety Concern Model</td>
<td>62</td>
</tr>
<tr>
<td>5-2</td>
<td>Class hierarchy in Java</td>
<td>64</td>
</tr>
<tr>
<td>5-3</td>
<td>Class C Flattened into Class C1</td>
<td>65</td>
</tr>
<tr>
<td>5-4</td>
<td>Fragment of the Server Generic</td>
<td>66</td>
</tr>
<tr>
<td>5-5</td>
<td>Flattened Source Code from the Server Generic</td>
<td>66</td>
</tr>
<tr>
<td>5-6</td>
<td>Interaction diagram for temperature updates.</td>
<td>68</td>
</tr>
<tr>
<td>5-7</td>
<td>Interaction diagram showing the update of a stale temperature value</td>
<td>68</td>
</tr>
<tr>
<td>5-8</td>
<td>Subprogram “ReadLAN”</td>
<td>70</td>
</tr>
<tr>
<td>5-9</td>
<td>Flattened and filleted subprogram “ReadLAN”</td>
<td>71</td>
</tr>
<tr>
<td>5-10</td>
<td>Pruned function “IsFound”</td>
<td>72</td>
</tr>
<tr>
<td>5-11</td>
<td>Active objects used to represent processes and threads.</td>
<td>73</td>
</tr>
<tr>
<td>6-1</td>
<td>Skeleton of the package body for the Ada model</td>
<td>92</td>
</tr>
<tr>
<td>6-2</td>
<td>Subprogram “Compute_Stereographic_Xy”</td>
<td>101</td>
</tr>
<tr>
<td>6-3</td>
<td>Subprogram “Compute_Stereographic_Xy” translated into S</td>
<td>102</td>
</tr>
<tr>
<td>7-1</td>
<td>System Level SVC</td>
<td>109</td>
</tr>
<tr>
<td>7-2</td>
<td>Design level SVC for LANToBroadcast process</td>
<td>110</td>
</tr>
<tr>
<td>7-3</td>
<td>Source code level SVC for LANToBroadcast process</td>
<td>111</td>
</tr>
<tr>
<td>7-4</td>
<td>SPARK annotation</td>
<td>113</td>
</tr>
<tr>
<td>7-5</td>
<td>Proof functions</td>
<td>114</td>
</tr>
<tr>
<td>7-6</td>
<td>Proof function “ConvertAll”</td>
<td>114</td>
</tr>
<tr>
<td>7-7</td>
<td>Formalized source code level SVC for LANToBroadcast process</td>
<td>116</td>
</tr>
</tbody>
</table>
Figure 9-1: Example functional block source code  
Figure 9-2: Example S specification for functional block behaviour
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOP</td>
<td>Aspect-Oriented Programming</td>
</tr>
<tr>
<td>ASIS</td>
<td>Ada Semantic Interface Specification</td>
</tr>
<tr>
<td>ATM</td>
<td>Air Traffic Management</td>
</tr>
<tr>
<td>CAATS</td>
<td>Canadian Automated Air Traffic System</td>
</tr>
<tr>
<td>CFM</td>
<td>Chemical Factory Management</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off the Shelf</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DVM</td>
<td>Distributed Virtual Machine</td>
</tr>
<tr>
<td>FDL</td>
<td>Functional Description Language</td>
</tr>
<tr>
<td>FMEA</td>
<td>Failure Modes and Effects Analysis</td>
</tr>
<tr>
<td>FPTN</td>
<td>Failure Propagation and Transformational Notation</td>
</tr>
<tr>
<td>FTA</td>
<td>Fault Tree Analysis</td>
</tr>
<tr>
<td>HAZOP</td>
<td>Hazard and Operability Studies</td>
</tr>
<tr>
<td>HOL</td>
<td>Higher Order Logic</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>MAATS</td>
<td>Military Automated Air Traffic System</td>
</tr>
<tr>
<td>OO</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>PPS</td>
<td>Present Position Symbol</td>
</tr>
<tr>
<td>RDP</td>
<td>Radar Data Processing</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real Time Operating Systems</td>
</tr>
<tr>
<td>SFE</td>
<td>Symbolic Functional Evaluation</td>
</tr>
<tr>
<td>SFMEA</td>
<td>Software Failure Modes and Effects Analysis</td>
</tr>
<tr>
<td>SFTA</td>
<td>Software Fault Tree Analysis</td>
</tr>
<tr>
<td>SHARD</td>
<td>Software Hazard Analysis and Resolution</td>
</tr>
<tr>
<td>SLOC</td>
<td>Source Lines of Code</td>
</tr>
<tr>
<td>SSR</td>
<td>Secondary Surveillance Radar</td>
</tr>
<tr>
<td>SVC</td>
<td>Safety Verification Condition</td>
</tr>
<tr>
<td>TCG</td>
<td>Test Case Generation</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
</tbody>
</table>
Acknowledgments

I would like to thank Bruce Elliott, Nancy Day and Drasko Sotirovski for providing feedback on an earlier draft of my dissertation. Each had a unique perspective that proved invaluable. I am also grateful for the time and effort contributed by my committee: Robert Rohling, Lee Iverson, and Philippe Kruchten. In particular, insightful suggestions provided during both the qualifying and departmental exams helped shape the final dissertation.

Much of what I know about system safety and software development was learned during my time at Raytheon Canada Limited, first as a M.Sc. student and later as a system engineer. I was privileged to work with many bright engineers, to have first-hand experience with an innovative software safety program, and to have access to a state-of-the-art software architecture.

I am grateful for the support and encouragement of my friends and family, which includes my parents’ unwavering conviction that the completion of a Ph.D. degree is a good thing. I am particularly thankful for all the help and support provided by my significant other, Sally, especially during the final, difficult stretch in completing the thesis.

Of course, my greatest debt is to my supervisor, Jeff Joyce. He introduced me to the field of software safety and the central problem addressed in the thesis. Always bristling with ideas, enthusiasm and encouragement, he provided the inspiration for much of the thesis.
1. Introduction

It is has become a well-known maxim that complexity is an essential property of software [Bro87]. Furthermore, software-implemented functionality has increased dramatically in almost all safety-critical sectors [Sto96]. The increased size and complexity of software has provided great challenges to the practice of “system safety”, which focuses on the management of system hazards in order to mitigate the risks of mishaps occurring. Hazards are states of the system that potentially contribute to a mishap occurring. “Software safety” [Lev95] has emerged as sub-discipline of system safety to help address these challenges specifically for software-intensive systems. Software safety has primarily focused on “front-end” of the safety process such as safety requirements specification and hazard analysis, as opposed to the “back end” of the process where we verify the safety of the software implementation. There is good reason for this, as building safety into the specification and design is the most effective method of mitigating safety risk. However, the safety of the source code must still be verified to provide evidence that the risk has been truly mitigated.

For a non-software system, system safety experts commonly expect that “safety concerns” can be separated from the rest of the system for safety analysis. By safety concern, we mean a hypothesis about a specific combination of internal or external events that might lead to an occurrence of a hazard. In particular, the safety concern often corresponds to a “functional slice” of the system that can be extracted and tested in isolation, like the braking subsystem of an automobile. Similarly, a traditional design paradigm of software safety is the isolation of the safety-critical software from the rest of the system.

The notion of separating and encapsulating software functionality is a cornerstone of the Object-Oriented (OO) development approaches (as exemplified in the Unified Modeling Language (UML) [BRJ99] and Rational Unified Process (RUP) [Kru98]). OO techniques are especially important for a class of systems that display critical information [Joy03], such as Air Traffic Management (ATM) systems. In particular, the research in this dissertation is largely motivated by the author’s experiences with the development of the Canadian Automated Air Traffic System (CAATS), an advanced ATM system [PKT93]. CAATS possesses an object-oriented, distributed software architecture primarily implemented in the Ada 83 programming language [KT94].

The OO approach, with a layered and service-oriented architecture, is perhaps the only effective way known for building complex software systems such as CAATS. For these types of systems, there exist other essential system goals such as performance, maintainability and robustness that compete with safety as influences on the decisions made by software developers about the software architecture. As
well, core functionality is safety-critical. As a result, a primary claim of this dissertation is that the safety-critical source code is not easily isolated, documented and analyzed for safety verification, an issue we have previously called the “long thin slice problem” [Won98b]. Though the focus of the dissertation is on object-oriented, critical decision information systems, we believe this problem will be evident in any software system with a modular software architecture that has core display or control functionality that is safety-critical.

In particular, we will show that safety is typically a “cross-cutting” concern that involves a number of different software components. As well, there is a preponderance of non-executable code statements in support of class hierarchies that obscure the critical execution code path. Though a compiler might easily keep track of this necessary detail, the result is a high cognitive overhead for the stakeholder attempting to understand the source code for safety purposes.

The fact that the safety-critical software might not be simple or isolated provides challenges to proposed methods of verifying source code with respect to identified hazards [Lev95]. Safety verification techniques tend to be standard software verification methods [Sto96] that do not focus on safety. In the case of approaches that are specifically designed for safety verification, there are few published discussions of how these techniques could be applied to anything other than relatively small, isolated subsystems.

This dissertation proposes an approach intended to strengthen the “back-end” of the overall safety engineering process for software-intensive safety-critical system. The approach involves extracting a model of the part of the system that is relevant to a particular safety concern. A safety concern model captures the “footprints” of the safety concern in the source code. The models are then used in support of safety verification and the overall system safety case.

There have been many proposed approaches to modeling the software specification and design for safety analysis. Typically they do not contain sufficient detail for safety verification of the software implementation. For example, UML diagrams alone are not sufficient, unless the semantics of the source code are also captured in the model. As noted by Brooks, “descriptions of a software entity that abstract

---

1 The terms “cross-cutting”, “tangled” and “scattered” are borrowed from the aspect-oriented design community [KL+97]. However, the terms are not used in the same precise, technical sense in this dissertation, but rather, they are meant to provide intuition on some different possible perspectives of the “long thin slice problem”.

2
away its complexity often abstract away its essence" [Bro87]. The source code details are crucial when attempting to analyze the software implementation for hazards.

1.1 Thesis Statement

The thesis of this dissertation is as follows:

The traditional system safety paradigm of isolating safety-critical functionality is no longer tenable in the face of the increased size and complexity of modern software systems. In a purely mechanical system, for instance, safety concerns typically form functional slices that can be conveniently partitioned from the rest of the system through design. Similarly, when software is involved, the conventional wisdom has been to isolate the software safety concern. However, isolating safety-critical software is not always possible in many modern systems, especially large, software-intensive systems with a modular software architecture where the safety-critical functionality is interleaved with core system functionality. There are a variety of legitimate reasons why a safety concern is not easily separated at the source code level from the rest of the system. As a result, instead of attempting to impose a strict partitioning of safety from non-safety software, the safety engineer should use techniques to extract models of the safety concerns for safety verification.

For non-software systems, the scope of a safety concern is often closely aligned with the partitioning of the system into functional subsystems. For example, the scope of a safety concern regarding the possibility of brake failure in an automobile is closely aligned with the braking system of the automobile. In particular, the braking system can be separated from components of the air conditioning system. Any connections that might exist between the two subsystems (e.g., electrical components) are likely to be fairly straightforward and easily understood based upon well-known principles of physics and engineering.

Given this alignment of safety concerns with functional subsystems, we can analyze the safety of the vehicle's braking system without needing to consider the possibility that the air conditioning system may be activated while the car is braking. It may seem absurd to suggest that the braking system of an automobile cannot be understood without also understanding the design of the air conditioning system. However, this kind of linkage is commonplace and not always explicitly evident in complex software systems, especially object-oriented systems that have been designed with an emphasis on software re-use. If automobiles were designed in the same manner with an emphasis on re-use and sharing of components, we might well be driving automobiles with brakes that could fail because of an air conditioning problem.
The possibility that safety concerns cannot always be isolated in modern software systems is a fundamental break from traditional system safety practices and undermines one of the basic principles of software safety design. The cross-cutting nature of safety concerns is a paradigm shift for how system safety engineers are to view and verify software-intensive systems. Failure to recognize this paradigm shift and to discover appropriate techniques to address the issue will potentially result in a failure to adequately verify the safety of modern software systems.

1.2 Safety and Modern Software Architectures

A key system safety task is hazard analysis, which involves analyzing hazards at the system and subsystem level. For example, a hazard for an automobile is the condition that the "brakes fail to engage when needed". The hazard could then be traced to the braking system and then to the braking system components. In this case, the hazard is traced to a subsystem, the braking system, which forms a functional slice that can be analyzed and tested independently of the rest of the automobile. In general, this isolation of the hazard to a functional slice is typical for non-software systems and is very useful for hazard analysis, mitigation and verification.

Similarly, a traditional design goal of software safety is to keep the safety-critical software as simple and isolated as possible. Parnas et al. have argued in favour of "keeping safety-critical software as small and simple as possible by moving any functions that are not safety critical to other computers" [PVP90]. This notion makes particular sense in the context of protection systems, such as a nuclear shutdown system, which monitors another system for hazardous conditions and takes corrective action to ensure the hazard does not occur or does not lead to an accident. In this case, the safety function could conceivably be isolated in a separate component or even computer.

However, as noted previously, the amount of software has increased dramatically in almost all safety-critical sectors despite the safety design goal of keeping software simple. The increase in software size is true for command and control systems, where ATM systems are being built with over a million lines of production source code. Modern information systems often involve sophisticated, object-oriented software architectures. The architecture might support real-time, distributed and concurrent processing, as well as a complex graphical user interface. Parts of the architecture could be implemented with Commercial-Off-The-Shelf (COTS) products. The increase in software size typically corresponds in an increase of complexity. For example, in the case of ATM protection functions, such as monitoring the conformance of a flight to the expected flight profile, what is being monitored is itself software, which is performed recursively for many layers of software.
As well, software is increasingly being used in safety-critical system for purposes besides the implementation of a protection function. Command and control systems [KS92] are an important class of software-intensive systems that include other types of safety-critical control functions. One such example is an ATM system that includes display and control functions with safety implications. In this case, the safety-critical functionality is interleaved with core system functionality or might even be core system functionality. For instance, the incorrect display of aircraft position is an ATM hazard, but the display of radar information is also core ATM functionality, i.e., it is not a separate safety function. Similarly, core control and display functions are being implemented with software in a wide variety of safety-critical systems, including “fly-by-wire” aircraft [Sto96], engine management systems [BMW04], and medical devices [VSM04]. In some cases, the added software functionality is for safety purposes, such as conflict prediction functions in ATM systems. However, there is debate as to whether increased software automation of user functions and the replacement of hardware safety functions with software equivalents truly add to the safety of the system [Lev95]. In any case, increasing use of software in safety-critical system functionality is a reality that must be addressed by developers and safety engineers.

Given that large safety-critical software systems are being built, the OO approaches to software development likely contain the current industry “best practices” for successfully building such complex systems. The OO approach is based upon the principles of abstraction, encapsulation, modularity and hierarchy [Boo94]. One key notion is decomposing software into components, where each component encapsulates a particular “concern”. In this fashion implementation detail is abstracted away (or hidden) by providing a public interface. Another key notion is providing an ordering to the abstractions and components through hierarchy, such as class inheritance.

The OO approach supports system goals such as fault tolerance, maintainability and software reuse, though an object always has limitations and should not be reused outside the scope of its applicability. The problem is that it is extremely difficult to define concepts such as 'scope of applicability' for objects. As a result, object-oriented approaches potentially complicate the task of software testing. As noted by Jorgensen [Jor02]:

“One of the original hopes for object-oriented software was that objects could be reused without modification or additional testing. This was based on the assumption that well-conceived objects encapsulate functions and data ‘that belong together’ and once such objects are developed and tested, they become reusable components. The recent interest in aspect-oriented programming (Kiczales, 1997) is one response to some of the limitations of the object-oriented paradigm. The
new consensus is that there is little reason for this optimism – object-oriented software has potentially more severe testing problems than those for traditional software.”

Similarly, it may have been hoped that object orientation would simplify the task of safety analysis by the encapsulation of safety concern in objects or closely related groups of objects [KL+97]. In general, it is only possible to encapsulate concerns if they are orthogonal, so that some concerns will cross-cut another. The existence of cross-cutting concerns has given rise to software development approaches such as Aspect-Oriented Programming (AOP) [KL+97]. Given that there are a number of non-orthogonal concerns including safety that must be addressed when developing a safety-critical system, it will not always be possible or desirable to encapsulate the safety functions.

1.3 Existing Safety Verification Techniques

This section provides a brief overview of existing techniques that can be used for software safety verification and representing safety concerns. A more comprehensive and in-depth survey is provided in Chapter 8.

1.3.1 Software Safety Verification

For engineering professions, there is a strong expectation by the public and regulatory authorities that a conclusion about the safety of a system will be based upon an examination of the physical system. In particular, safety-critical components are inspected and reviewed to ensure their implementation is consistent with the safety requirements [PVP90]. Engineers are expected to inspect the actual system – for instance, to walk around the site of a chemical factory to examine all of the safety-critical subsystems. For software systems, too little attention is typically paid to the inspection of the implementation.

The focus of software safety analysis methods has been on the software requirements and design, i.e., the “front end” of the process. For example, Leveson and her colleagues have proposed techniques for analyzing a state machine representation of the requirements [ML+97], while McDermid and his colleagues [FM+94], have proposed approaches for analyzing the software design. There is good reason for focusing on the front end of the process as safety risk is most effectively mitigated during the requirements stage and when safety is designed into the system. However, there has not been as much attention paid to the safety verification of the implementation. For instance, examination of the authoritative text on software safety by Leveson [Lev95] reveals a relatively short chapter on safety verification and with no discussion of the impact of software techniques such as object-oriented approaches on safety verification.
In general, existing methods for verifying source code with respect to identified hazards rely on standard software verification techniques. For example, a review paper by Scott and Lawrence [SL94], as well as Chapter 12 in Storey’s text on safety-critical computer systems [Sto96] describes software verification techniques that might be applied to safety verification. However, software verification approaches focus on software correctness and not safety. Code inspections for safety-critical software typically focus on code quality checks and adherence to coding standards [AC+03], as opposed to the hazards. Though these types of checks are important for ensuring overall code quality and reliability, it is essential that safety verification also provide evidence that the hazards have been sufficiently mitigated.

The most common safety verification technique is likely safety testing. Often, safety verification is seen as simply a matter of verifying the safety requirements. We have previously argued that a requirements-based approach to safety verification is insufficient for demonstrating that the safety risk has been adequately mitigated [JW03]. Even if more exhaustive safety testing is performed, it is a well-known limitation of testing that due to the large number of possible software inputs and paths, it is not feasible to exhaustively test them all [PVP90]. Testing can only provide a limited degree of assurance that the hazards have been mitigated.

There have been a few proposed static analysis techniques that focus on safety verification, such as Software Fault Tree Analysis (SFTA) [LCS91]. However, these techniques have primarily been applied to relatively small, isolated subsystems. Ultimately, in addition to testing, safety verification will require a static analysis of the safety-critical software to provide assurance that the hazard risk has been adequately addressed.

1.3.2 Representing Safety Concerns

Analysis of the safety-critical software is complicated by the cross-cutting nature of the safety concern. As discussed in the previous section, existing static analysis techniques are better suited for small, self-contained software systems. To address this limitation, we propose creating a model of the safety concern for safety verification.

The problem of cross-cutting concerns has received a lot of attention, as exemplified in the aspect-oriented [KL+97] and hyperspace [OT01] approaches to software development. However, these approaches focus on incorporating cross-cutting concerns into the software design and implementation, and not on modeling existing concerns in the source code. As well, preliminary investigations by Joyce and Feng indicate that the types of safety concerns examined in this dissertation do not appear to correspond naturally to the sort of aspects that are defined in aspect-oriented design [JF04].
There are many approaches to modeling the software at the specification and design level. These methods typically deliberately abstract away details of the software implementation that are not relevant during the software design phase. However, these details are important when verifying the safety of the source code.

There are existing techniques that could be adapted to represent safety concerns in the source code. These methods include:

- textual documentation of delocalized “plans” (developer’s intent or design) [SP+88],
- use of concern graphs to represent scattered concerns [Rob03],
- executable program slices for debugging [Wei84], and
- software visualization tools (subset of reverse-engineering tools) as an aid to program comprehension [SFM97].

In general, these techniques focus on the issue of documenting concerns at the implementation level for software maintenance purposes, as opposed to creating a model for safety analysis. In the case of program slices, which are potentially executable and testable, the slicing criterion typically does not readily correspond to a safety concern.

There are a variety of existing software tools and techniques that could be used to help construct a model of the source code. For example, there are tools for searching and analyzing source code, which could be used to help identify source code related to a safety concern. Code searching tools range from basic lexical searching tools like the unix utility “grep” to the code cross-referencing capabilities in many software development environments. There are also code analysis tools that use the semantic and syntactic information available in a compiler environment [Coo97]. In the case of Ada environments, the Ada Semantic Interface Specification (ASIS) [ACM97] provides access to compiler-generated information through an interface that is at the same semantic level as the Ada language [Ehr94]. Alternatively, there exist tools such as JANUS for parsing source code to build an in-memory knowledge-based abstract syntax tree (KBAST) model of the entire system [ND01]. Though these tools can support the creation of code models, they alone are not sufficient for identifying and specifying the details relevant to the safety concern.
1.4 Creating Safety Concern Models for Safety Verification

Our proposed approach to safety verification involves creation of models of the source code related to the safety concern that can be analyzed during safety verification and documented in the system safety case. To address the issues related to the cross-cutting nature of the safety concern, we propose a process that involves four basic steps:

1. **Flatten** - modify classes by making explicit all the attributes, methods, conditions, and invariants the class inherits from other classes. Expand parameterized classes (generics or templates) by replacing parameters with the actual parameters, changing the parameters names as necessary to avoid namespace conflicts.

2. **Fillet** – identify and extract the critical code paths. Prune the less relevant code branches and, if necessary, replace with something simpler or with an assumption of the expected behaviour.

3. **Partition** – identify the portions of the critical code path that execute in separate processes.

4. **Translate** – represent the source code in a notation that allows the irrelevant details to be omitted.

The existing software tools and techniques mentioned in Section 1.3 (and described in greater detail in Chapter 5) can be used to help carry out the different steps in the approach.

The output of the approach should be an approximately conservative, sufficiently complete and tractable model of a safety concern. The safety concern model captures the highest risk elements of the software with respect to a system hazard. Use of the model during safety verification should result in a more efficient and effective safety analysis. For example, it much more likely that safety analysis techniques designed for small, isolated safety-critical software systems can be applied to a safety concern model, than to a safety concern scattered across a large software system.

The analysis of the safety concern model is intended supplement existing safety verification efforts. In particular, it essential that the software still be tested on the hardware that will be used operationally.

1.5 Case Studies

Raytheon engineers informally applied the code model construction process to several hazards during the safety analysis of the ATM system. As described in Chapter 2, the ATM System is a large real-time, command and control system with an object-oriented software architecture. The problems uncovered
during the creation of the code models were not identified by the other software analysis techniques employed, such as code quality reviews, requirements analysis and system testing. The experiences of the Raytheon engineers provide evidence of the effectiveness of using code models to supplement existing safety verification techniques.

The code model construction process is further evaluated in this dissertation with two substantial case studies involving a Chemical Factory Management System and the ATM system. The Chemical Factory Management System is a "toy model" of a command and control system with a software architecture similar to the ATM system.

Safety concern models are created using different representational notations including informal English, a programming language, and a formal mathematical notation. Examples of the resulting models can be found in the appendices.

In order to evaluate the effectiveness of the safety concern models, we address the following questions when applying the approach in the case studies:

- How tractable is the resulting model?
- How analyzable is resulting model?
- How faithful is the resulting model?
- How complete is the resulting model?
- How usable is the approach and resulting model?

The overriding question is whether the resulting safety concern model is something coherent that can be understood by the relevant stakeholders and effectively analyzed during safety verification.

1.6 Contributions of this Dissertation

The following summarizes the contributions of this dissertation:

1. A reassessment of the traditional system safety design principle that safety-critical functionality should be isolated.
2. An in-depth analysis of how object-oriented software can create difficulties for safety verification.
3. Developing the foundation for modeling a safety concern.
4. An approach for creating a safety concern model involving four basic steps.
1.7 Overview of Dissertation

Chapter 2 presents background material that provides context for the problem of cross-cutting safety concerns. The fundamentals of system safety are presented, followed by the impact of software and, in particular, object-oriented software on system safety, especially in the area of safety verification. The notion of a “safety verification case” that documents the safety verification, including the safety concern models, is described. As well, the two main examples used in this dissertation are introduced, an ATM system and a Chemical Factory Management System.

Chapter 3 demonstrates the difficulties in identifying and representing the software safety concern for safety verification. In particular, the fact that safety is a cross-cutting concern in the source code will be further described and then illustrated with a Chemical Factory Management System hazard. Additional examples of cross-cutting safety concerns in other systems are also presented.

Chapter 4 elaborates on the concept of a safety concern model, including the desired model properties and allowed vocabulary for describing the model.

Chapter 5 describes the approach for creating a model of the safety concern source code. The steps for constructing the safety concern model are then presented in greater detail, as well as how the model might be maintained in the event of changes to the software. This is followed by a discussion of an application of the process by engineers involved with the safety analysis of the ATM system.

Chapter 6 describes the creation of safety concern models with the approach using different notations including informal English, a programming language, and a formal mathematical notation. The usefulness of the approach with respect to the different notations is evaluated, with the main emphasis on translation step and the effectiveness of the notation used.

Chapter 7 describes how the safety concern models can be extended by the formal construction of program pre- and post-conditions.

Chapter 8 describes related work including an overview of existing safety verification techniques. The overview is followed by a review of approaches that explicitly focused on identifying and documenting concerns in source code, as well as the various techniques and tools that can be used in support of modeling safety concern source code.
Chapter 9 provides a summary, a set of recommendations, an overview of potential future research, and some concluding thoughts. One avenue of future research is overcoming the limitations in existing concern documentation techniques when applied to safety concerns. Another is applying existing analysis techniques and tools to the safety concern model.
2. Background

The discipline of system safety must adapt to the complexity of modern software systems. In particular, the traditional principles and practices of system safety should take into account the unique characteristics of software. As noted in the introduction, the safety-critical functionality cannot always be isolated for some modern software systems.

This chapter presents background material that provides additional context for the problem of cross-cutting safety concerns. The fundamentals of system safety are presented, followed by the impact of software and, in particular, object-oriented software on system safety. The notion of a safety verification case that documents the safety verification is then described. Finally, the two main examples used in this dissertation are introduced, along with a description of typical system hazards and their object-oriented software architectures. Although the work of other researchers has served as background material for this research, we defer a discussion of related work until Chapter 8 of this dissertation.

2.1 Safety-Critical Software

Parnas et al. note that it is becoming common to find safety-critical software functionality where it previously did not exist, such as civilian and military aircraft, medical devices, and nuclear devices [PVP90]. As well, safety-critical software is becoming increasingly complex and integrated.

Levenson states that the growing presence of safety-critical software is partly due to the increase in automation of formerly manual operations [Lev95]. For example, microprocessors currently play a role in most control functions in plants and systems such as petrochemical plants.

The growing complexity of safety-critical software is partly attributable to the corresponding growth in size. Storey notes that the size of the software in words of executable code, doubled every two years over a thirty-year period for a single aircraft manufacturer [Sto96]. Modern safety-critical software systems potentially contain over a million lines of source code. In contrast, the Therac-25 radiation therapy machine was made available for commercial use in 1982 and involves assembly code that runs on a 32K PDP-11/23 [Lev95]. The Darlington nuclear generator station went operational in 1990 and consists of two shutdown systems, one with about 7000 lines of FORTRAN code, while the other has about 13,000 lines of Pascal code [Sto96]. Though the increase in size is not necessarily proportional to an increase in functionality, it is clear that amount of software functionality has dramatically increased.
The increase in software size also reflects a trend in the re-use of previously developed code and the use of "Commercial Off the Shelf" (COTS) products that adds to software complexity. The use or re-use of previously developed software often includes a large amount of "dead code" into the implementation, i.e., functionality that is not used by the system. As well, integrating previously developed code into a new system sometimes results in extra layers of source code (i.e., software "wrappers") that could make the implementation even more convoluted and difficult to analyze. The ARIANE 5 failure is an example of the safety risks involved with software reuse [AIB96].

Software is also "increasingly integrated" in the sense that the amount of external inputs to a safety-related function is increasingly large or the function relies on an increasing amount of "state" information. ATM is a rich example of this problem. Older ATM systems display radar targets directly as "plots", which are relatively simple because the inputs are not much more than the target azimuth and slant range, radar source location on the display, as well as display orientation and scale. More modern ATM systems will typically maintain and display radar "tracks", which are derived from the radar targets and a history of the radar targets, as well as additional inputs such as discrete Secondary Surveillance Radar (SSR) codes. A fully modern ATM system might maintain and display an extrapolated track based upon the aircraft profile that is derived from a wide range of input data, including flight plan information, flight clearances, weather, aircraft information, radar data and so forth. The modern ATM system is significantly more complicated and "integrated" than older, simpler ATM systems.

2.2 System Safety

This section provides an overview of System Safety.

2.2.1 Definitions

The following definitions are primarily based upon the system safety standard MIL-STD-882C [DOD93]:

- **Mishap** (or Accident): An unplanned event or series of events resulting in death, injury, occupational illness, or damage to or loss of equipment or property, or damage to the environment.

- **System Hazard** (or Hazard): A state of the system that, possibly in combination with environmental conditions, leads to a mishap.

- **Hazard Cause**: An internal (i.e., system) or external (i.e., environmental) condition that, possibly in combination with other internal and environmental conditions, leads to a hazard.

- **Hazard Scenario**: A chronology of events that identifies one possible combination of internal and external hazard causes that results in the occurrence of a hazard.
- **Hazard Analysis**: The identification of hazards and hazard causes.
- **Hazard Probability**: The aggregate probability of an occurrence of the individual events that create a specific hazard.
- **Hazard Severity**: An assessment of the consequences of the worst possible mishap that could be caused by a specific hazard.
- **Risk**: An expression of the probability/impact of a mishap in terms of hazard severity and hazard probability.
- **Safety**: Freedom from those conditions that can cause mishaps.

The distinction between mishap, hazards and hazard causes is not always clear from the definitions. Figure 2-1 provides an overview of the relationship between them. For a blood pressure monitoring system, an example of a mishap is the improper treatment of hypertension/hypotension. A hazard would be the display of an incorrect blood pressure reading. An internal hazard cause, at the system level, would be a miscalculation of the blood pressure reading. The hazard cause could be further traced to a software cause, such as the software corruption of a sensor reading. An external hazard cause would be an improper adjustment of the pressure cuff around the patient’s arm.

The resulting hazard scenario might then be:

1. A blood pressure reading is received.
2. The reading is corrupted by the system.
3. An incorrect blood pressure is displayed.

During hazard analysis, the hazard scenario can then be refined into a sequence of events within or between a set of subsystems and subsystem components.
2.2.2 Theories of Accidents

Researchers from a variety of engineering and other disciplines (psychology, business administration, sociology) have developed theories about the causes of accidents. These theories seek to improve the effectiveness of accident investigations towards the goal of preventing future accidents. An increasingly common theme of this research points to the need to look beyond the immediate causes of an accident. For example, James Reason [Rea97] writes:

"Indeed, it is only within the last 20 years or so that the identification of proximal active failures would not have closed the book on the investigation of a major accident. Limiting responsibility to erring front-line individuals suited both the investigators and the organizations concerned - to say nothing of the lawyers who continue to have problems with establishing causal links between top-level decisions and specific events."

The truth of this new thinking about accident investigation has been highlighted in catastrophes such as the E coli contamination of the Walkerton, Ontario, water supply, in 2000, which killed at least seven people [Oco02]. The investigation of such accidents shows the cause of an accident cannot always be isolated to one component of a system.

We might well define the task of safety verification as a form of accident investigation before the accident has occurred. Applying these new insights about accident investigation to the discipline of system safety encourages us to consider more broadly how a system may behave in an unsafe manner.

2.2.3 Safety as a Distinct Quality

Safety is a matter of ensuring that “bad” things do not happen, i.e., accidents that are unplanned events resulting in damage to property, injury, or death. System hazards are characterized by their external effects, and are expressed at a relatively high level of abstraction.

For example, a typical ATM system hazard involves the incorrect display of the aircraft position. Although it may be very clear to an experienced air traffic controller how the display of an aircraft position may be "incorrect", this abstract term needs to be given a more precise, concrete interpretation. It may not be enough to simply rely upon the functional requirements of the system for a clarification of the term “incorrect” since the meaning of this term with respect to the hazard may not be exactly the same as its meaning with respect to the functional requirements. For instance, there is a variety of ways in which the display of an aircraft position may fail to satisfy the functional requirements. But only some of these potential failures may constitute a hazard. Moreover, there may be sequences of events involving
the display of an aircraft position that have been found to be hazardous, but are not explicitly excluded by the functional requirements.

The difference between safety and correctness is apparent when "safety requirements" are explicitly contrasted with functional requirements. By "safety requirements" we mean the set of constraints or conditions that the system must satisfy to mitigate or prevent the hazard from occurring. Typical software functional requirements describe a "forward" (stimulus-response) view of system functionality, while safety requirements often involve a "backward" look at the system. This use of the words "forward" and "backward" refers to the direction of the casual links from causes to events.

The forward orientation of a typical functional requirement is illustrated by the following example that may appear in the specification of a computer-based system for monitoring conditions in a reactor vessel of a chemical factory.

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.

The focus of this requirement is the system functionality involved in the processing of sensor updates. It is a "forward" look at the system functionality, from a system input to an output.

In contrast, a safety requirement focuses on system hazards, such as "An invalid vessel temperature is displayed". The basic safety requirement derived from the hazard might be stated as:

If temperature D is displayed for Vessel V at time T, then at some time not more than S2 seconds before T, the actual temperature of Vessel V was within C degrees of D.

The safety requirement is stated in a "backwards" fashion, focusing on the desired, safe output and defining the necessary input. Such a requirement is not easily tested. It would involve determining all possible inputs that may violate this requirement.

The safety requirement is not simply a reformulation of the functional requirement. In fact, it is possible to satisfy the functional requirement, while failing to satisfy the safety requirement. One trivial case that illustrates the difference is when the system never displays a temperature value. Although this is clearly a
case where the functional requirements are not satisfied, this behaviour does not conflict with the safety requirement.

Figure 2-2: Relationship between unsafe and incorrect behaviour

In general, a system may satisfy the contractual requirements, be reliable, and yet be unsafe. The requirements themselves may not exclude hazardous behaviour. It is even possible that strict conformance to the specific requirements of the system could result in hazardous software behaviour. Even if the requirements are demonstrated to be “safe”, there may be unintended functionality in the system, not specified by the requirements that could be hazardous. In general, though there obviously is an overlap between unsafe and incorrect system behaviour, as illustrated in Figure 2-2, they are not the same. As a result, safety verification must focus on analyzing the system with respect to system hazards.

Though properties such as reliability and correctness are important aspects of building a critical system, and may provide circumstantial evidence of system safety, they alone will not ensure safety. As Leveson observes:

> Arguments have been advanced that safety is a subset of reliability or a subset of security or a part of human engineering - usually by people in these fields. Although there are some commonalities and interactions, safety is a distinct quality and should be treated separately from other qualities, or the tradeoffs, which are often required, will be hidden from view.

A simple example of such a trade-off is the realization that the safest aircraft are those that never leave the ground. A less extreme, more realistic example is the use of run-time method binding and

---

2 Page 182 of [Lev95].
polymorphism to support software reuse and maintainability. The benefits of run-time method binding must be balanced against the difficulties it creates for a safety analysis of the source code.

2.2.4 System Safety Overview

System safety involves controlling safety-related risks through management of potential hazards during development. The goal is to eliminate or mitigate states of the system that might contribute to a mishap.

2.2.4.1 Safety Tasks

The basic safety tasks include identification of hazards, analysis of hazard causes, assessment and mitigation of the associated risk, and verification of the risk mitigations. The sequence of safety tasks is displayed in Figure 2-3.

Figure 2-3: Basic Safety Tasks

More generally, systems like command and control systems will carry out some control functions that will have safety implications. In that case, it is necessary to identify and assess the safety-related functionality. For example, an automobile will have core functionality like the steering or braking that is safety-related. An obvious system hazard would be the automobile failing to brake when needed to avoid a collision.
2.2.4.2 Types of Hazards

The safety of the system can be divided into primary and functional safety aspects [Sto96]. The primary safety of the system involves hazards that arise directly from the system hardware itself, such as an electrical shock or fire. Functional safety involves the functioning of the software and hardware.

Figure 2-4 summarizes the results of the basic safety tasks for a primary safety hazard involving the electrical wiring for a system. The safety tasks can be performed without a detailed examination of the internal system. In particular, software is not a risk factor.

In the case of functional safety of the system, hazards might lead to a mishap directly (e.g., braking system fails to operate) or through misleading information (e.g., invalid blood pressure reading is displayed). In general, a deeper analysis of the internal system behaviour will typically be required for functional safety. The analysis should include system, subsystems, and subsystem interactions. In particular, if software functionality is involved, the software subsystem must be analyzed. The analysis will be performed at the requirements, design and implementation levels.

<table>
<thead>
<tr>
<th>Hazard</th>
<th>Hazard Cause</th>
<th>Risk Class</th>
<th>Risk Mitigation</th>
<th>Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exposed electrical wires</td>
<td>No or inadequate insulation</td>
<td>1-High</td>
<td>All exposed wiring shall be insulated.</td>
<td>Wiring inspected during qualification testing</td>
</tr>
</tbody>
</table>

Figure 2-4: Hypothetical hazard analysis for an electrical wiring hazard for a system.

For the hazards involving the display of critical information, there can be some subtlety in specifying the hazard [Joy03]. An accident occurrence will depend on the human operators response to the information. For example, a blood pressure reading will be "invalid" depending on how the user applies the information. As a result, the existence and criticality of the hazard will be especially dependent on environmental conditions and other information displayed by the system. It is likely that the hazard definition will be refined as the development of the system progresses.

2.2.4.3 Hazard Analysis

Hazard analysis is performed throughout the development of the system. The results are used to determine risk mitigations, such as design choices that eliminate, mitigate or control the hazards.

A Preliminary Hazard Analysis is performed to determine hazard causes arising from the environment and system operation. For instance, the system is treated as a "black-box" and its interactions with the
environment (i.e., inputs and outputs) are analyzed. Various operational scenarios are examined, including the interactions between the system and human operators, as well as between system components. Hazard scenarios are constructed that consists of system and environmental events that lead to the hazard. These events can be used to determine hazard causes.

As the system is developed, the hazard analysis continues with the system requirements, design and architecture. Hazard analysis techniques such as Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA) can be performed on subsystems to determine their contributions to the hazard. The hazard scenarios are traced onto the subsystems, refined into a sequence of events within or between a set of subsystems and subsystem components.

The braking hazard, for example, is traced to the braking system and then to the braking system components. In this instance, the hazard is traced to a subsystem, the braking system that can be analyzed and tested independently of the rest of the automobile. In general, this decomposition of the system into independent subsystems is very useful for hazard analysis and mitigation.

In the case of subsystems like the braking system, the overall design is fairly well understood. In particular, the functionality of each component is relatively simple and isolated, along with how they collaborate with other components in the subsystem. As a result, the hazard analysis can focus on the potential of individual component failure contributing to the hazard. Traditional hazard analysis techniques such as FTA and FMEA are especially effective for analyzing hazards in terms of component failure modes.

2.3 Software Safety
Software safety is understood within the context of system safety. In particular, software can be considered another component in the system and the hazard analysis extended to software. The software safety analysis involves refining hazard scenarios into a sequence of events within or between a set of software components. The fault tree can be extended to include software hazard causes. Hazard analysis can be performed on the relevant software components.

However, there are a number of fundamental differences between software systems and non-software systems. For example, Parnas et al note the following [PVP90]:

- **Complexity**: Brooks suggests that the complexity of software is an “essential property” [Bro87]. Booch attributes software complexity to a number of factors including complexity of the problem
domain, the sheer volume of the system requirements, and software flexibility that encourages software changes and non-standard software components [Boo94].

- **Error Sensitivity**: Software is very sensitive to small errors. For non-software systems, the conventional engineering practice involves building to a specified tolerance to control the size of the error. The use of specified tolerances to control errors does not work with software as even a small error can lead to large consequences.

- **Hard To Test**: This is due to the discrete nature of software. Testing of analogue devices are based upon interpolation and the assumption that a device functioning well at two close points will also function well in between the two points. Due to the discrete nature of software, this assumption does not hold.

- **Correlated Failures**: As software does not “wear out”, the dominant errors will be specification and design errors. For non-software systems, the primary concern is with manufacturing errors and wear-out phenomena, so the assumption can be made that the failures are not strongly correlated and simultaneous failure is unlikely. This assumption cannot be made with software, as there is the distinct possibility that failures in one component can be correlated to a failure in another.

The system safety process also needs to be integrated with the software development process, which is intended to help manage software complexity during development.

### 2.3.1 Traditional Software Architectures and Safety

A traditional design goal of software safety is keeping the safety-critical software as simple and isolated as possible. In particular, keeping a safety-critical function encapsulated in a single subsystem or a small number of components is often considered to be standard software engineering practice [Som01]. The U.S. National Science and Technology Council identified “architectural approaches to isolate safety-critical system components from non-safety-critical components” as a research objective for “high confidence systems” [NTSC99].

Parnas et al. have argued in favour of “keeping safety-critical software as small and simple as possible by moving any functions that are not safety critical to other computers” [PVP90]. For example, the Darlington nuclear generator station consists of four reactors, each with a control system and two different shutdown systems, SDS1 and SDS2. The control and shutdown systems are completely separate and independent computer systems [Sto96].
Software products such as Real Time Operating Systems (RTOS) are specifically designed to support the achievement of safety objectives through isolation of safety-related functionality. A RTOS may include mechanisms such as virtual memory to implement spatial isolation of critical software processes.

The isolation of the safety critical software functions is usually possible in the case of a protection subsystem designed to detect failure conditions and to take action to mitigate the failure. In that case, the entire subsystem encapsulates a safety concern. In fact, some safety standards like IEC 61508 focus on the notion of isolated “safety functions” that places more emphasis on the provision of protection functions than on hazard avoidance and mitigation [IEC95]. However, for systems like command and control systems, the safety-related functionality is part of core functionality and is not an independent safety function.

2.3.2 Software Components

There is a significant difference between non-software and software components. As noted in the previous section, for subsystems like the braking system with a well-understood design, the non-software component functionality is relatively simple and well understood, along with how they collaborate with other components in the braking system. As a result, a large part of the hazard analysis involves determining the potential of component failure contributing to the hazard.

Software components tend to be complex, including their interfaces and interactions with other components. Even if modular software development practices are used and the component encapsulates a single concern, it will be at a much higher level and greater complexity than a non-software component. For example, depending on the granularity, the component could be a graphical user subsystem, a windowing package, or a widget in the window. The component might serve a single purpose, display of information, but the display functionality could be greatly varied in type and method.

Ultimately, it is inadequate to simply focusing on component failures for software safety purposes. In particular, the system safety process must take into account large and complex software components and component interactions.

2.4 Object-Oriented Software Architectures

Modern software architectures pose additional challenges for software safety verification. In particular, object-oriented architectures are based upon fundamental software engineering principles such as modularity and hierarchy. However, these principles also help give rise to the possibility of cross-cutting safety concerns.
2.4.1 4+1 View Model

A "4+1 View Model" [Kru95] can be used to describe software architecture, consisting of five different perspectives, or views, that include the design, process, physical and implementation views.

1. The design view of the software shows the logical decomposition of the system. For an object-oriented architecture, this encompasses the classes, interfaces and collaborations that form the basis of the logical design of the system.
2. The process view describes the concurrency and synchronization aspects of the design. This view encompasses the processes and threads (light-weight processes executing within a process) that correspond to the different flows of control in the system.
3. The physical view describes the mapping of the software onto the hardware and the distributed aspects of the software.
4. The implementation view describes the organization of the software in the development environment.

The fifth view is a set of scenarios that incorporate elements of the other four views.

2.4.2 Object Model

The object-oriented approach to software development is intended to address the problem of software complexity during design and development. OO uses an object model that is based upon the principles of abstraction, encapsulation, modularity and hierarchy [Boo94].

In the design view, the software is decomposed into a set of abstractions known as classes or objects (a particular instance of a class).

In the implementation view, the source code is decomposed into components. The components implement the class abstractions, though there is not necessarily a one-to-one correspondence between classes and components. A key design goal is for the components to be cohesive and loosely coupled as possible.

Encapsulation involves information hiding, where the details of the implementation are hidden and the public behaviour is accessible through the interface.
Finally, the various abstractions can be clustered into a given hierarchy, providing a ranking or ordering of the abstractions. In particular, inheritance is an example of classes sharing structure or behaviour of another class.

### 2.4.3 Object-Oriented Software Architectures and Safety

Based upon the object model, two key characteristics emerge concerning object-oriented software architectures that impact safety concerns. One is the modular nature of the architecture. The other is the notion of class hierarchy.

In general, it is a sound and well-accepted software engineering practice to build software architectures in a modular fashion. Modularity is also a key objective of the object-oriented approach to software development. The class abstraction and software components provide a decomposition of the system in both the logical and implementation view of the software. Encapsulation ensures that only the details of interest to those outside the component and class are revealed.

In addition, the object-oriented approach provides a rich set of relationships between the components through the class hierarchy. As result, the components must include “scaffolding” for implementing the class hierarchies, such as the class concept in Java or C++. In the case of Ada 83, there is no explicit class mechanism. Ada packages and generics (similar in use to C++ class templates) can be used to implement classes and class relationships.

The intent of modularity and hierarchy is to help master software complexity, which should also help make the software more robust and maintainable. However, as will be discussed further in Chapter 3, the result will likely be cross-cutting software safety concerns, scattered among many components, with a large amount of non-executable source code.

### 2.5 Safety Verification Case

The issue of cross-cutting safety concerns poses particular challenges to safety verification and the documentation of the safety verification results. The results of the system safety process are typically documented in a final safety report (often referred to as the “safety case”) intended to provide evidence that the system is safe to deploy, along with any remaining residual risk. We have previously introduced the term “safety verification case” to refer to a document that records the safety verification of the source code [WJR98]. Our use of the term safety verification case is intended to emphasis the contributory 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 safety verification of the source code.
Ideally, the safety verification case will provide a rigorous argument in support of system certification. The argument will almost certainly depend upon assumptions made about the non-software components of the overall system, including reasonable assumptions about operators and the likelihood that they will use the system in accordance with specified operating procedures. These assumptions should be presented explicitly in the safety verification case. Although it may be a less desirable outcome, it is possible that the safety verification case will provide evidence that supports an argument against system certification given certain assumptions about the non-software components.

The safety verification case should also be sufficiently detailed to support post-development safety reviews conducted during the operational use of the system. These safety reviews will be necessary, for example, if there are any modifications of the system or changes in the environment. If the system is the first in a family of systems, the safety verification case should be documented in such a way that the development team can re-generate the document for the next generation of the system. In particular, the argument should be “repeatable”, so that system maintainers and developers different from the original development team can use the process and the document as the basis of their safety assessments.

The structure for the source-code safety verification case is most appropriate for hazards that are caused by the "normal" execution of the software, as opposed to hazards that may arise in the wake of a general, systematic failure of the system.

The safety verification case development process 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 the system level.

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.

The process for developing a safety verification case is based on the strategy of systematically refining the safety verification process into a series of simpler steps, as shown in Figure 2-5. The topic of this
The dissertation is the right hand branch of the process, i.e., the problem of cross-cutting safety concerns and the construction of code models to address this problem.

In earlier work, the author addressed the problem represented by the left-hand side of Figure 2-5, i.e., the problem of refining system hazards into small granularity “Safety Verification Conditions” (SVCs) [Won98a]. The incorporation of source code level SVCs into the concern model is discussed further in Chapter 7.

![Figure 2-5: Safety verification process.](image)

The remainder of Section 2.5 provides additional details on safety verification. In addition, Chapter 8 reviews existing safety verification techniques, while Chapter 9 suggests some potential approaches for verifying the safety concern model with respect to the source code level SVCs.

### 2.5.1 Safety Verification

An important step in the overall safety engineering process is the verification of the source code with respect to identified hazards. This step is necessary even after system hazards have been identified and controlled through design. The safety verification process must determine if any mistakes were made in the safety engineering process. The safety verification case documents the efforts made to carry out this step of the process.

Safety verification is mandated in several software safety standards, such as the international standard, IEC 61508 [IEC95], and the US military standard, MIL-STD-882C [DOD93]. For example, the IEC
61508 defines a software safety validation phase, which is designed to ensure that the safety-related software is compliant with the software safety requirements specification.

As well, the record of the safety verification is an integral part of the safety case. 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 report is often called a "safety case". The safety case is likely to span the entire safety engineering process including a summary of the results of various process steps.

The discussion of safety verification within this dissertation is limited to the analysis of the functionality of the system with respect to specific hazards. A thorough approach to safety verification at this level would also require two additional kinds of verification efforts:

- procedural checks, e.g., checking that hazard mitigations identified at earlier stages of development have been implemented;
- code quality checks, e.g., checking that variables are set before they are used.

Both of the above additional kinds of verification efforts are necessary to ensure thoroughness. However, these two additional kinds of verification efforts should not be considered as alternatives to the kind of verification discussed in this dissertation.

2.5.2 Safety Analysis of the Source Code

Common-sense would lead us to expect that "safety verification" is an integral part of the safety engineering process in other engineering domains such as chemical, mechanical, nuclear, electrical and transportation systems.

For instance, we would expect that the safety engineering process for the construction of nuclear generating station involve more than the analysis of the "blueprints". We would expect that a qualified team of safety specialists would directly inspect the "implementation" of the blueprints by walking through the power station to physically examine all of the mechanical and electrical systems that are directly related to an identified hazard. If a serious accident occurred sometime later, and it was discovered that the safety specialists had never visited the generating station for an inspection, then there is good reason to believe that the safety specialists would found liable for having failed to meet the expected standard of care. By analogy, we suggest that safety specialists for a software-intensive system
must also do more than consider the "blueprints". They must additionally seek verification evidence derived from the actual implementation of the system.

Thus, we assert that the safety engineering process for a software-intensive system must include close scrutiny of source code specifically with respect to identified hazards. It is not enough to ensure that the functional requirements and design level documentation have undergone safety analysis. As remarked in an U.S. Air Force system safety handbook,

_Unlike the fairy tale Rumplestiltskin, do not think that by having named the devil that you have destroyed him. Positive verification of his demise is required._

With appropriate assumptions about the compiler and the underlying computing platform, the source code is the most concrete, tangible representation of the behaviour of a software system. The safety analysis of the source code provides direct evidence from the implementation that the hazards have been adequately mitigated.

### 2.6 Examples

The key worked examples in this dissertation include an ATM system and a Chemical Factory Management System. The research in this thesis is largely motivated by the author’s experiences with the development of the ATM system [KT94]. The other example, the Chemical Factory Management System, is a "toy model" largely based upon the ATM system, which was created to help illustrate the ideas in the dissertation.

The examples are software intensive safety-critical systems with object-oriented software architectures. In addition, the systems have a distributed software architecture, implemented in the Ada 83 programming language and integrated with a number of COTS products.

Both examples are representatives of an increasingly important class of safety-critical software systems, which Joyce refers to as "critical decision information systems" [Joy03]. These systems receive, processes and displays environmental data to operators who then make safety-critical decisions based on the information displayed by the system.

---

3 As quoted on page 489 of [Lev95].
2.6.1 Air Traffic Management System

The ATM system is an example of a safety-critical command and control system. Critical information is received, processed and made available to air traffic controllers.

2.6.1.1 Operational Concept

Controllers use information from the ATM system to help control air traffic. The controllers issue clearances to pilots indicating what altitude and routes they should be flying in order to maintain proper separation between the aircraft. The clearances will partly be determined from information like aircraft position, altitude and intended route of the flight.

In general, the information processed and displayed by the system includes radar data indicating aircraft position and altitudes, weather data including altimeter readings, and filed flight plans indicating the intended route and cruising altitude of the flight.

For example, radar data corresponding to an aircraft is displayed on the screen with a “Present Position Symbol” (PPS). Figure 2-6 shows the possible appearance of a PPS on a display. The display of a PPS is one means of displaying the position of an aircraft to an air traffic controller. In addition, in the PPS “datatag”, additional information like the aircraft altitude, speed and alerts is displayed.

![Figure 2-6: Part of an air traffic controller’s “situation display”.

2.6.1.2 ATM Safety

The ATM system is safety-critical in the sense that the purpose of such a system is to provide information to human operators who make critical decisions [ER96]. Loss of a critical air traffic control
function may require that Instrument Flight Rules (IFR) aircraft must be procedurally separated further apart using Visual Flight Rules (VFR), i.e., “see and be seen”. During this transition, hazardous loss of separation may occur.

For example, a typical ATM system hazard is to display the PPS in an incorrect position [ER96]. An incorrectly displayed PPS could lead an air traffic controller to give instructions to pilots that may contribute to a loss of separation.

A subtler ATM system hazard involves the incorrect display of altimeter settings by the system. An altimeter setting is a measure of atmospheric pressure and is used to calibrate the altimeter in an aircraft that indicates the altitude (above mean sea level) of the aircraft on the ground at the location for which the pressure value was determined. An invalid altimeter reading could potentially mislead a controller into believing that two aircraft are vertically separated when they are not, resulting in the controller failing to give corrective instructions to the pilots.

2.6.1.3 ATM Software Architecture

The ATM system has an object-oriented, distributed software architecture, which consists of up to a million lines of Ada source code, integrated with millions more in COTS products [KT94]. The ATM software architecture was designed to be scaleable and adaptable to a wide range of functionality and hardware platforms. A key design notion was “design for change”, which led to the adoption of a number of architectural principles.

Some characteristics of the ATM software architecture include [KT94]:

- **Distribution and Concurrency**
  The system is a set of communicating processes distributed over a computer network.

- **Parameterization**
  The use of startup and run-time parameters as well as generics (or templates), allows for software to be modified in a controlled and well-defined manner

- **Use of COTS products**
  Increasing, software development is making use of mature off-the-shelf products.

The following describes the “4+1 View Model” [Kru95] of the ATM software architecture.
The design view of the software shows the decomposition into a set of class abstractions. The classes capture the key domain information, as well as the common mechanisms and design elements across the different parts of the system.

The process view describes the concurrency and synchronization aspects of the design. At the highest level of abstraction, the process view of the architecture can be seen as a set of communicating processes (Ada programs) distributed across a set of hardware resources connected by a LAN.

The physical view describes the network of physical processing nodes.

The implementation view focuses on the organization of the software components in the software development environment. The software is partitioned into layers. Each layer is decomposed into subsystems and each subsystem is further decomposed into components (Ada packages and generics). The software development teams are then organized around the subsystems and layers.

<table>
<thead>
<tr>
<th>Layer 5</th>
<th>Human Computer Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>External Systems</td>
</tr>
<tr>
<td>Layer 4</td>
<td>ATC Functional Areas</td>
</tr>
<tr>
<td>Layer 3</td>
<td>ATC Classes</td>
</tr>
<tr>
<td>Layer 2</td>
<td>Support mechanisms (DVM):</td>
</tr>
<tr>
<td></td>
<td>communications, and so on</td>
</tr>
<tr>
<td>Layer 1</td>
<td>Common Utilities</td>
</tr>
<tr>
<td></td>
<td>Bindings, Low-level Services</td>
</tr>
</tbody>
</table>

**Figure 2-7: The ATM software architecture adapted from figure 5 of [KRU95]**

The ATM system architecture is organized into 5 layers, as shown in Figure 2-7, and into about 72 subsystems, with each subsystem containing from 10 to 50 components. Each layer defines a progressively more abstract machine depending only on the services of the lower layers. Domain-specific functionality is found in the upper layers and the domain-independent functionality in the bottom layers. Distributed object services and low-level services such as Pivot, a light-weight asynchronous thread scheduler, are provided by the Distributed Virtual Machine (DVM) layer [TC95], while COTS products and hardware-specific software are confined to the lowest layer.
The desire for a robust, maintainable software architecture, is achieved by using the class abstractions and layering to create a separation and encapsulation of various concerns such as the user interface, hardware interfaces and common mechanisms. In particular, it is possible to separate functional concerns of the application domain from the COTS products and the hardware environments.

2.6.2 Chemical Factory Management System

The Chemical Factory Management System is another example of a critical-decision information system in that information is displayed to factory operator who then makes safety-critical decisions based upon this information. This dissertation focuses on one aspect of the Chemical Factory Management System functionality - the display of vessel temperatures and the associated hazard.

2.6.2.1 Operational Concept

The chemical factory consists of a set of vessels filled with various chemical solutions that must be kept in equilibrium by monitoring the vessel temperature, pressure, vibration level and other vessel parameters. The Chemical Factory Management System receives, processes and displays the vessel information to factory operators, as shown in Figure 2-8.

![Diagram of Chemical Factory](image)

Operators monitor the vessel conditions and then make appropriate adjustments to the vessel parameters either manually through a control panel on the vessel or through a remote panel in the control room connected to the vessels.
For example, the vessel temperature must be kept below a specified maximum value otherwise there will be significant risk of a vessel fire or explosion. When the temperature is within a specified tolerance of the maximum value, the operator must take action to reduce the vessel temperature.

2.6.2.2 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 Management System, which runs on a central server and a set of workstations.

The server, where the vessel information is processed and maintained, receives the incoming sensor data over the LAN. The information is then broadcast to the workstations and displayed on their monitors. The broadcast mechanism is a highly reliable, asynchronous method of distributing information across the network.

2.6.2.3 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:

Requirement 356: Temperature Update.
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.

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

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.

```
Layer 5  | Human Computer Interface  
          |                  External Systems  
Layer 4  | Factory Functional Areas  
Layer 3  | Factory Classes  
Layer 2  | Support mechanisms (DVM): communications, and so on  
Layer 1  | Common Utilities  
          | Bindings, Low-level Services  
```

Figure 2-9: The layers of the static architecture from the implementation view.

2.6.2.6 Software Architecture

The chemical factory software architecture is based upon the ATM system architecture. In particular, the 4+1 View of the architecture is essentially the same for both systems:
1. The design view of the software shows the decomposition into a set of class abstractions.
2. The process view describes at the highest level of abstraction, the set of communicating processes distributed across a set of hardware resources connected by a LAN.
3. The physical view describes the network of physical processing nodes.
4. The implementation view describes the separation of the source code into layers (as shown in Figure 2-9), the layers into subsystems and each subsystem into components (Ada packages and generics).

The Unified Modeling Language (UML) [BRJ99] is used to describe the various views of the software architecture. UML is the emerging standard for specifying the object-oriented analysis and design of a software system. It is a graphical language and can be used to represent different views of the system, such as the design, process, physical and implementation views.

2.7 Summary

This chapter provided additional context for understanding the challenges modern software architectures pose to safety. The fundamentals of system safety were presented, including the impact of software on safety. In particular, certain attributes of modern software architecture will be shown in the next chapter to impact safety verification. The principles of the object-oriented approach to software development were reviewed, as well as the "safety verification case" that documents the safety verification. Finally, the two main examples used in this dissertation were introduced, along with a description of typical system hazards and their object-oriented software architectures.
3. **Software Safety Concerns**

This chapter will demonstrate why the traditional system safety paradigm of isolating safety-critical functionality is no longer tenable in the face of the increased size and complexity of modern software systems.

As discussed in Chapter 2, a traditional goal of software safety has been to simplify and isolate the safety-critical software as much as possible. The fact that safety concerns are not isolated in the modern software architectures is not necessarily a sign of bad design or an indictment of the OO approach to software. Given the size, complexity and integrated nature of the software, there is little disagreement among software engineers that a modular software architecture is required. The fact that the system must be maintainable and fault-tolerant, in addition to being safe, is also not in dispute. However, the result is safety might not be independent of other concerns that are encapsulated in software components.

The concept of safety concerns will be described in further detail in this chapter. The cross-cutting nature of safety concerns will be illustrated with a Chemical Factory Management System hazard. Additional examples of safety concerns in other systems are also presented.

### 3.1 Safety Concerns in the Source Code

This section expands on the notion of safety concerns in the source code.

#### 3.1.1 Safety as a System Concern

Safety is a system concern. The safety of a system is generally an emergent property that results from a potentially large number of individual design decisions (see the “Theories of Accidents” in Chapter 2). The property of being safe, or unsafe, is rarely the result of a particular line of source code. There potentially could be a single line of code that is the “sharp end” of a safety concern. However, ultimately there are a whole set of decisions that are responsible for the safety risk.

The term “concern” is borrowed from software engineering and means some aspect of the system that is of interest to the engineer. For example, a concern might be a generic system property like performance or structural integrity, or some aspect of system functionality like the user interface.

“Safety concerns” are aspects of the system that are related to safety. More precisely, a safety concern is a hypothesis about a specific combination of internal or external events that might lead to an occurrence of a hazard. In some cases, the concern may truly be a hypothesis – that is, an unproven conjecture. It is
also possible that a safety concern may be a record of a test result with safety implications. Not every safety concern is necessarily associated with an identified hazard, as some safety concerns may ‘fill in the cracks’ between identified hazards. A safety concern, or a group of related safety concerns, may eventually achieve a critical mass that warrants consideration as a new hazard.

Figure 3-1: Schematic of an ABS Braking System [VW04]

The identification of the components relevant to a safety concern is a key part of the safety process. During hazard analysis, the hazard is traced to the system subsystems and components. The hazard analysis involves examining the component interfaces and determining the ways in which the components might collaborate to contribute to a hazard.

For a non-software system, safety concerns can often be isolated to a single component or subsystem. By subsystem, we mean a part of the system which itself can be a system. For example, the braking system is a subsystem for an automobile. In particular, the subsystem can often be seen as a “functional slice” of the system, taking system input and creating system outputs. The braking system, along with the wheels, forms a functional slice within the automobile, as shown in Figure 3-1.

Extracting the functional slice is often possible with non-software systems as the interface between non-software components is usually simple compared to the potential complexity of the connections between
software components. For example, the disc brakes and brake lines serve the braking system and no other, and have a singular purpose.

When a mechanical engineer is tasked to perform safety verification of a braking system, this is likely to include the construction of a laboratory workbench model of the braking system. This model will be constructed from the components of the braking system. The model will be the same as an instance of the braking system installed in a production vehicle except that the rest of the vehicle has been "pruned" away. A major advantage of using a laboratory workbench model for part of the safety verification is convenience. As well, it may be easier to subject the laboratory workbench model of the braking system to extreme conditions that would be difficult or highly dangerous to create for a braking system installed in a vehicle.

3.1.2 Software Safety as a Cross-Cutting Concern

When software is involved, the hazard analysis continues with the tracing of the hazard to the set of relevant software subsystems and components that might contribute to the hazard. The safety analysis again involves examining the component interfaces and determining the ways in which the components might collaborate to cause the hazard.

As mentioned earlier in this dissertation, a traditional design goal of software safety has been to keep the safety-critical software as simple and isolated as possible. As discussed in Chapter 2, modern software practices involve separating and encapsulating software concerns in components. A modular software architecture could potentially be used to isolate the safety concerns. However, in order to achieve goals such as a fault-tolerance and maintainability, the software is typically partitioned along different lines, including separate subsystems for basic underlying services and mechanisms that support the application functionality.

In addition, the safety concern is often related to core system functionality for a critical decision information system, like the display of aircraft position for an ATM system or vessel temperature for the Chemical Factory Management System. To put the entire safety concern into a single subsystem would be tantamount to putting a large portion of the software functionality in one subsystem. This amount of software functionality would be unmanageable as the subsystem would be overly complex. Alternatively, if the core functionality was duplicated in a separate component for the safety concern, this would likely be more inefficient (e.g., greater demand for CPU utilization), expensive to maintain, and prone to errors. Ultimately, to derive the full benefit of modularity in mastering software complexity, the safety concern will typically involve a number of shared software components.
Furthermore, the software components related to the safety concern tend to be large and complex, where only a subset of component functionality is relevant to the safety concern. As well, safety-critical components are typically used outside the safety concern as well. For example, a software graphical user interface component used to display aircraft position would typically also be used to display other non-critical information.

3.1.3 Safety Concern Source Code

It was argued in Chapter 2 that the safety analysis of the source code was necessary in order to provide direct evidence from the implementation that the hazards have been adequately mitigated. In particular, source code that might contribute to the hazard must be identified and analyzed.

During the hazard analysis the following type of questions need be asked of the software [JF04]:

1. What compile-time constants or secondary sources of external data are involved in the processing of critical data?
2. Is the processing logic complete? For example, does it include provisions for when an optional element of the input data is not available?
3. Are different units of data used consistently? For example, we want to avoid the possibility of a defect like the defect that caused the failure of the Mars Climate Orbiter where English units (pounds) were used by code expecting metric units (newtons) [NASA99].
4. What could cause time-dependent data to be delayed during processing? For example, when the CPU is heavily loaded, could this result in a backlog of incoming system inputs to be processed?

These types of questions cannot be answered by simply examining component interfaces.

In order for the safety verification effort to be effective, the hazard analysis should focus on potential contributions of the source code to the hazard. In contrast, code analysis is often seen as a review of the source code conformance to coding standards and a search for source code anomalies. Though this type of inspection does serve an important quality assurance function, searching for anomalies is not a very efficient or effective means of identifying potential hazard causes. For example, as part of the ATM safety analysis, a code analysis tool was used to identify source code anomalies, such as unreachable code or uninitialized variables. The results were then analyzed for potential contributions to hazards. None of the anomalies uncovered were found to be safety-critical.
Source code related to a hazard might involve more than just the application level functionality, but also underlying services that are peripheral to the hazard. As well, the source code may be intermixed with a large amount of structural non-executable code. Eventually, if every possible code path was examined, a significant fraction of the software system could be deemed hazard-related.

3.1.4 “Long Thin Slice” Problem

When building the safety case for a software system, evidence must be provided that the source code does not contribute to the identified system hazards. As argued in Chapter 2, the safety verification case must include the documentation of the safety concern at the source code level, as well as the verification evidence that the safety concern source code does not contribute to a hazard occurring.

Ideally, it would be useful to be able to extract the safety concern for verification, as we could with the braking system in a vehicle. However, extracting the safety concern is much more difficult when it is not a standalone subsystem that can be executed separately. The difficulty in extracting and representing the safety concern for safety verification due to its cross-cutting nature, involving only a small amount of code from each component buried among a large amount of non-executable code, has been previously referred to as the “long thin slice problem” [Won98b].

For example, tracing the safety concern in the source code may involve looking at more than a thousand different locations in the source code merely to understand a sequence of actions of less than fifty steps corresponding to executable source lines of code. This searching may not present a problem to the developer who has been immersed for many months or years in the development of the software. The developer may be capable of tracing a sequence of actions across the long thin slice of code without looking up every relevant line of code from various source files. Other people, however, must also be able follow the same sequence of actions. These people include system certifiers, system maintainers and system developers intending to reuse the software in new systems. These people must also be able to evaluate the safety concern in order to carry out their own safety assessments of the system.

3.2 Example: Chemical Factory Management System Safety Concern

The Chemical Factory Management System is used to illustrate how safety is a cross-cutting concern. The architectural and the source code views of a Chemical Factory hazard is presented followed by a summary of the characteristics of the safety concern source code.

3.2.1 System Hazard
The following system hazard for the Chemical Factory Management System was given in Chapter 2:

**Hazard 3:** An invalid vessel temperature is displayed.

The receipt and display of vessel information, including the temperature, is core system functionality. There are system requirements that the vessel temperatures be displayed. Though the hazard could be eliminated by not displaying the temperature at all, this would result in the system not satisfying the system specification and would be unacceptable to the customer. As a result, it is not something that can be readily separated from the rest of the system functionality.

### 3.2.2 Architectural View of the Hazard

The safety-critical software can be described in terms of software structural elements, where the element in question depends on the architectural view of the software, e.g., processes, objects or components. This is a high-level view of the safety concern provides guidance as to where the safety concern source code is located. However, the architectural view of the hazard does not provide sufficient information for safety verification, which will require delving into the source code.

The development view focuses on the organization of the software components in the software development environment. As described in Chapter 2, the software is partitioned into five layers. Each layer is decomposed into subsystems and each subsystem is further decomposed into components (Ada packages and generics).

The static architecture reflects a fairly common separation of concerns seen in command and control systems. Layers are introduced to separate the large concerns, such as the user interface, external interfaces and lower level services.

The hazard-related software is scattered throughout all five layers, involving a large number of subsystems. For example, the receipt and display of vessel information involves the External System layer. The processing of the vessel information is performed in the Functional Areas layers and the Chemical Factory Management (CFM) Classes layers. All these layers are provided services by the Transport and Basic Services layers. As a result, source code related to the hazard is a cross-cutting concern that will involve a number of different layers and subsystem, as can be seen graphically in Figure 3-2. The solid arrow represents the executable critical code path through the system. The dotted arrows represent static dependencies for a particular code statement in the code path (e.g., a type definition).
The focus of the hazard analysis is on the basic application processing found in the upper layers of the software architecture. The contribution from the supporting services in the lower layers can be dealt with separately.

![Diagram of software architecture](image)

**Figure 3-2: The Hazard-Related Software as a Cross-Cutting Concern**

The key domain-specific components involved in the display of the vessel temperature are depicted in Figure 3-3. These include the relevant Ada packages and subprogram calls.

### 3.2.3 Source Code View of the Hazard

The architectural view of the software illustrates the key safety-critical software components and their high-level interactions. However, this view ultimately does not provide sufficient information for safety verification.

As discussed in Section 3.1.3, there are number of safety-related questions about the software that need to be addressed that can only be answered by looking at the source code. For example, an important question is whether stale temperatures will be removed from the display. A look at Figure 3-3 indicates that there is a software component responsible for monitoring temperature staleness. However, given the complexity of the processing of the sensor data, testing alone will not provide sufficient assurance that the mechanism will work in a safe manner. In particular, an inspection of the source code for the temperature display reveals a number of different code branches for the processing of the vessel temperature before it reaches the operator consoles.
In the following, a fragment of source code from the safety concern is examined to demonstrate some of the difficulties with identifying and documenting the safety concern source code. The source code is written in Ada 83. Classes are implemented with packages and generics, while subprograms are either procedures or functions.

**Figure 3-3: Temperature data flow highlighting the key components and procedures.**

### 3.2.3.1 Example Source Code within a Safety Concern

The hazard can be traced to lines of source code within each relevant component. In the following or discussion we focus on the subprogram “ProcessSensors”, which can be found in the “SensorServer” component shown in the upper-right hand corner of Figure 3-3. This subprogram processes information obtained from the sensors and then broadcasts the results to the operator consoles. As well, the subprogram schedules the “MonitorSensorStaleness” subprogram, which periodically checks if the sensor information has gone stale.
We will focus on the following portion from “ProcessSensors”, involving the vessel temperature, displayed in Figure 3-4. To trace the temperature value through this subprogram fragment involves the following.

The subprogram parameter value, “Readings”, contains the temperature value and is of type “Sensor.Object”. “Sensor” is a component in a different layer than the “SensorServer” component. The “Sensor.Object” type can be traced in a separate branch of code. It is ultimately derived from basic types defined in components in the bottom layer.

```pascal
procedure ProcessSensors(Readings : in out SensorReadingArray)
  is
  begin
    for I in Readings'Range loop
      Readings(I).Data.SID := CorrelateSensor(Readings(I).Code);
      if SensorStore.IsFound(Readings(I).Data.SID,
          CurrentSensors) = TRUE then
        SensorStore.Updated(Readings(I).Data.SID,
            Readings(I).Data, CurrentSensors);
      else
        SensorStore.Add(Readings(I).Data.SID,
            Readings(I).Data, CurrentSensors);
        SensorMonitor.Schedule(Readings(I).Data.SID);
      end if;
      Message(I) := Readings(I).Data;
    end loop;
    BC.Send(Message);
  end ProcessSensors;
```

Figure 3-4: Fragment of the “ProcessSensors” subprogram source code

The parameter, “SourceCode” is of type “SensorSourceCode” and is a generic parameter. The “ProcessSensors’ subprogram is part of the “SensorServer” component. The “SensorServer” component is an instantiation of the “Server” Generic, which provides the “SensorSourceCode” type,

```pascal
package SensorServer is new Server
  (SensorSourceCode => LM.SensorCode);
```

where “LM” is a renamed package,

```pascal
package LM renames LANMessage;
```
As can be seen, the “SensorSourceCode” type is derived from “SensorCode”, which is defined in the component “LANMessage”. The “SensorCode” type can be traced in a separate branch of code. It is ultimately derived from basic types defined in components in the bottom layer.

The storing and broadcast of the temperature is performed with the help of services provided by components in the lower layers.

1. The temperature is stored locally with the help of the subprograms “IsFound”, “Updated” and “Add”, which are part of the “SensorStore” component. This component is an instantiation of the “Store” generic and maintains a database of the current sensor readings. Tracing the temperature to this database would involve investigating the data storage service.

2. The monitoring of temperature staleness is performed with the help of the “Schedule” subprogram, which is part of the “SensorMonitor” components. This component is an instantiation of the “Schedule” generic. Tracing the monitoring of the temperature staleness would involve investigating the scheduling service.

3. The broadcast of the temperature value is performed with the help of the “Send” subprogram, which is part of the “BC” mode. This component is an instantiation of the “Broadcast” generic. Tracing the broadcast of the temperature, would involve investigating the broadcast service.

Ultimately, the temperature is received by the user interface with the help of the “ReadLAN” subprogram, which is part of the “UIManager” component. This component is where the temperature is further processed and displayed.

3.2.3.2 Use of Generics

For the components involved in the safety concern there are numerous dependencies between components, as well as an extensive use of generics. As mentioned previously, packages and generics are the key mechanisms in Ada 83 for implanting the class hierarchies.

For a given component, we may find the use of:

- lower-level mechanisms
- active objects
- implicit invocations
Components make use of these services by instantiating the appropriate generic. For example, class communication is achieved through implicit invocations, broadcast or a procedure call after the components has instantiated the generic with the procedure to be called. As well, the instantiations may be chained, the instantiated package instantiating other packages, to an indefinite depth.

### 3.2.4 Characteristics of the Safety Concern Source Code

A number of conclusions can be drawn from the analysis of the source code related to the safety concern.

Functionality that is irrelevant to the hazard is interleaved in the source code with source code relevant to the hazard. For example, the assumption was made (subject to validation) that the underlying services in the lower layers of the architecture would behave correctly and could be left out of the analysis. As can be seen in the source code fragments, there are frequent calls to the underlying services.

In general, if no assumptions were made and every line of code related to the hazard was traced, it is possible that a large percentage of the existing code base would be included in the safety concern. For example, tracing the hazard into each service call would reveal further dependencies on other services and components, so that each underlying service is a cross-cutting concern in its own right. This tracing could easily expand the safety concern source code until it is of unmanageable size, consisting largely of source code that is of only peripheral relevance to the hazard.

The use of generics to implement the class hierarchy adds a large cognitive overhead to program comprehension. As generics are used for almost all the components, large portions of the source code will simply be generic instantiations. To then understand a type or procedure call from an instantiated generic, the generic source code and the generic instantiation values, which are typically in source code files, must be kept in mind of the analyst. There is also a hierarchy of types, as types are defined within
each class abstraction. A single type used in an application component can be traced through a variety of components in different layers before the base types are reached.

Finally, the concurrent behaviour of the software is not easily determined from the source code. Following the source code, it is not a simple matter to determine when a new process is initiated or an asynchronous subprogram call is made. As a result, it may be difficult to determine, for example, the order in which source code is executed.

### 3.3 Other Examples

The cross-cutting nature of safety was demonstrated for the Chemical Management System, which is an example of an object-oriented, critical decision information system. However, cross-cutting safety concerns are likely to exist for any modular software system that has core control or display functionality that is safety-critical. As discussed in Section 2.1, software is increasingly being used to automate functions in a wide variety of safety-critical sectors, such as civilian and military aircraft, medical devices, automobiles, trains, petrochemical and nuclear plants. As well, the safety-critical software is becoming increasingly large, complex and integrated. As a result, scattered safety concerns will almost certainly appear in these systems, as software is involved with core system functionality and modern modular software approaches are employed.

For example, the Airbus A330/A340 adopts a “fly-by-wire” approach, where the primary flight controls involve the use of computers [Sto96]. The BMW 7 series of automobiles have computer-based control of the ignition, the fuel injection, the oxygen sensor and numerous ancillary functions [BMW04]. The fact that the software is safety-critical was demonstrated when an error in the BMW 745i engine software led to a recall of the automobile and reports of injuries from rear-end collisions that might have been caused by the error [BMW03].

This section describes additional safety-critical systems with cross-cutting safety concerns. These systems include an air traffic management system, radar data processing system and a blood pressure monitoring system.

#### 3.3.1 Air Traffic Management System

As discussed in Section 2, the Chemical Factory Management System is based closely upon an ATM system and the author’s experiences with the safety analysis of the system. The ATM system is also an object-oriented, critical decision information system.
For some of the ATM hazards, a safety analysis of the source code was performed. For example, as mentioned in Chapter 2, one of the standard air traffic management hazards addressed in ATM is the display of the aircraft position. This hazard involved fifteen objects implemented in over seven subsystems. The analysis did not include the COTS products or objects from the lower layers of the software architecture. It was determined that out of the approximately 700K lines of original source code, 1-2k lines of source code were relevant to the hazard [ER96].

However, it would not have been reasonable to isolate the hazard-related source code in a separate component given the software architecture, as the display of the aircraft position is core system functionality supported by all layers of the software.

3.3.2 Radar Data Processing System

The author was involved in the safety analysis of the radar data processing (RDP) system for a military air traffic management system. An RDP transforms incoming radar data into a set of tracks that can be displayed by an air traffic management system.

Feng and Joyce implemented a toy model of an RDP with many of the same characteristics as a real RDP [JF04]. The RDP was developed with an object-oriented software architecture with the four most interesting classes being “Radar”, “Correlator”, “Track” and “Datablock”. The “Radar” object corresponds to a physical radar facility. The “Correlator” object matches incoming radar reports with existing tracks or initiates a new track. A “Track” object is created for each track, corresponding to an aircraft within radar coverage. Finally, the “Datablock” objects correspond to the representation of an aircraft on the display.

There are number of potential hazards for an RDP system, including the display of the incorrect altitude for a track. Feng and Joyce focus on this hazard and show that it cross-cuts the architecture. The altitude information is initially derived from the radar reports received by the system and then flows through each of the classes displayed in Figure 3-5.

3.3.3 Blood Pressure Monitoring System

The author helped implement a software safety process for a non-invasive, automated blood pressure monitoring system [VSM04]. The system consists of a monitor, set of cuffs, and power supply. The cuffs are placed around the patient’s arm, and the system then inflates and deflates the cuffs automatically, determines the blood pressure and heart rate, and displays the results on the monitor.
The monitor processes the blood pressure and heart rate readings with embedded software in a microprocessor. The source code is written in assembly and partitioned into modules. A basic hazard for a blood pressure monitor is the incorrect display of the blood pressure reading, which is core system functionality. As a result, the safety concern does involve a number of the source code modules.

Though the software architecture is not object-oriented, the safety concern is again cross-cutting, which will be true in general for any safety-critical control or information system implemented with a modular software architecture. The hazard-related functionality will typically coincide with core functionality and hence the critical code paths will traverse the software architecture.

![Figure 3-5: Radar Data Processor Class Diagram from [JF04]](image)

### 3.4 Summary

It was demonstrated in this chapter how safety is a cross-cutting concern that involves a number of different components in the software architecture. This is unlike non-software systems, where the hazard can typically be confined to a component or set of components forming a functional slice that can be independently analyzed. In particular, this is contrary to the traditional design goal of safety-critical software. This is not necessarily the result of bad design, but is due to modular software architectures and competing concerns such as robustness and maintainability.

In addition to being cross-cutting, the safety concern source code also has the following characteristics:
• For systems with object-oriented architectures, each software component relevant to the safety concern typically includes a large number of non-executable code statements. A significant portion of the non-executable code is necessary to support the class hierarchies. This non-executable code includes statements that specify the class dependencies and that re-export types from other components. The executable code statements are typically buried among all the non-executable code statements.

• Not all the code branches and not all of the code details are equally relevant to the hazard. Engineering judgment must be applied in eliding what is of greatest interest to safety. However, this implies assumptions are being made about the details that are left out which must also be documented and verified.

• For real-time systems, it is not easy to determine the concurrent software behaviour from the source code. Though this can be documented at the design level with active objects, it does not have the required level of detail.

The fact safety is a cross-cutting concern presents significant challenges to safety verification, which requires identification and documentation of the hazard-related source code. This issue has received little or no attention in the literature and there are no established industry practices for addressing it. In particular, existing software safety analysis techniques tend to focus on software requirements or design.

One issue is that the safety concern source code is unlikely to correspond to a functional slice that is easily extracted and tested. Another issue is that there is a large cognitive overhead in reviewing the source code. To understand a source line of code requires locating the relevant details in a number of different files and places in the source code. These details include type, variable and subprogram declarations. This search may involve a large number of files. A type declared in one of the upper layers of the software architecture is often derived from a base type in a lower layer, which has been re-exported and subtyped many times over. It is easy to imagine that the analysis of a given hazard could involve more than a hundred different locations in the source code, to document a sequence of actions involving less than ten distinct steps.
4. Foundations of Safety Concern Models

Safety verification involves following and studying the “footprints” of the safety concern in the source code. As defined in Chapter 3, a safety concern is a hypothesis about a specific combination of internal or external events that might lead to an occurrence of a hazard. In particular, the safety concern must be documented for system safety engineers, developers, certifiers, and other system stakeholders for assessing the impact of the source code on the hazard.

Documenting the safety concern involves creating a model of the source code corresponding to the safety concern. The safety concern model can be thought of as a plaster-of-paris cast of the footprints rather than the actual footprints themselves. It is in general much more feasible to create a plaster cast of footprints for study in the laboratory, as opposed to transporting the actual footprints.

This chapter describes the foundations of a safety concern model. First, the concept of a safety concern model is elaborated, followed by the desired model properties and allowed vocabulary for describing the models. A summary follows.

4.1 Safety Concern Models

This section develops further the notion of a safety concern model. In essence, safety concern source code corresponds to the critical code paths arising from various hazard scenarios. The safety concern models are models of the critical code paths.

4.1.1 Hazard Scenarios

As defined in Chapter 2, a hazard is a prerequisite to a mishap, such as an incorrect blood pressure reading display for a blood pressure monitor or brake failure for an automobile. A hazard scenario is a chronology of events that identifies one possible combination of internal and external hazard causes that results in a hazard. For a software system, the hazard scenario can be refined into a set of events occurring within or between a set of software components.

For the Chemical Factory Management System temperature hazard, a simple hazard scenario might be:

1. Vessel temperature increases until it exceeds the maximum allowable temperature.
2. The temperature displayed to the operator does not indicate the temperature has exceeded the maximum allowable temperature.
3. The operator takes no action to reduce the vessel temperature.
4. The vessel explodes.

A more complex example\(^4\) by Joyce involves an ATM hazard regarding the incorrect display of altimeter settings to controllers:

1. Flight A, on standard pressure, is en route at Flight Level 180.
2. Flight B, on altimeter setting, is flying in the opposite direction at 17,000 feet.
3. The flights are vertically separated, based on a rule that is only applicable for an altimeter setting of 2992 or greater.
4. The applicable altimeter setting falls below 2992 to 2962.
5. By some means other than the controller (e.g., automated broadcast of weather information), Flight B learns of the new altimeter setting, adjusts its altimeter setting to use the new altimeter setting and continues to fly at 17,000 feet.
6. The system fails to update the displayed altimeter setting such that the controller is unaware that Flight Level 180 is no longer usable, or that the previously applicable vertical separation is no longer applicable.
7. Flight A continues en route at Flight Level 180 that is now at an altitude of 17,700 feet.
8. Due to the display of a stale altimeter setting, the controller has been misled into concluding that separation exists when it does not.

Steps 1-5 of the above scenario establish the pre-conditions for the mishap. The hazardous behaviour occurs in Step 6, i.e., the system fails to update the displayed altimeter setting. Steps 7 and 8 document the safety consequences of this hazardous behaviour.

### 4.1.2 Critical Code Paths

The hazard scenarios can be further refined into critical code paths. A code path is a particular view of the code that generally involves a partially-ordered sequence of executable statements. In essence, a critical code path is the “footprints” of a safety concern in the source code.

A control flow is a particular kind of view of the code where the code path is ordered by time of execution. The code path may skip over portions of the code that are deemed to be irrelevant to the purpose at hand. In general, a control flow shows a segment of computation bounded by an initial event (e.g., receipt of a new input) and a terminal event (e.g., output of a corresponding response).

\(^4\) The following hazard scenario and discussion is taken from [Joy03].
A data flow is a particular kind of view of the code where the code path is ordered by the dependency of data computations based on previous data computations. Although a data flow view of the code must be consistent with execution order, it is generally a more abstract view of the execution of a system than a corresponding control flow.

Different kinds of flows will be particularly useful for handling different kinds of complexity in the analysis of a hazard. For example, a control flow view may be particularly useful for analyzing a behaviour that involves the implementation of complex decision logic. For example, in an ATM system this may be the decision logic that determines when a particular alert message is displayed on the air traffic controller's situation display. On the other hand, a data flow may be particularly useful in understanding the complex sequence of transformations performed by an ATM system to convert radar inputs (e.g., azimuth, slant range, Mode C altitude) into pixel positions on the situation display.

It is quite possible that the critical code path for one scenario is not exactly the same as the critical code path for another scenario for the same hazard. The two critical code paths may even be significantly different with a minimum of overlap between the two sets of subsystems that constitute the critical code paths of the two different scenarios. If the difference is significant, then it may be necessary to extract more than one model of the source code level implementation of the system.

### 4.1.3 Safety Concern Models

Safety concern models are the representations of the critical code paths. Building models is a well-established technique across engineering disciplines, taking advantage of the principles of decomposition, abstraction, and hierarchy [Boo94]. In this fashion, it allows the engineer to focus on the details of interest and abstract away the less significant details. The details can then be decomposed and arranged in a fashion that makes sense for the analysis.

For example, consider again the safety concern corresponding to the braking hazard for a car. A car manufacturer would likely build a model of the brake assembly to test it, rather than exclusively testing the brakes as part of the entire car. The brake assembly model would simplify end-to-end testing, stress testing ("shake and bake"), and other types of evaluations of the braking system.

The creation of models is also a central part of software development. The Unified Modeling Language (UML) [BRJ99] is the standard for object-oriented software development and provides a wide range of models at the requirements and design level.
In terms of system safety engineering, hazard analysis requires a model of the system. As noted by Levenson [Lev95], "Hazard analysis requires some type of model of the system, which may range from a fuzzy idea in the analyst's mind to a complex and carefully specified mathematical model". As previously noted, many software hazard analysis techniques have been developed involving the construction of models at the requirements and design level (e.g., work by Leveson and her colleagues [ML+97] or McDermid and his colleagues [FM+94], which is further described in Section 7).

![Figure 4-1: Extracting a model of the safety concern](image)

Safety verification of the source code will depend on a model at the source code level. A model of the source code will exist during safety verification, though it might be only the analyst's understanding of the code, i.e. a "mental model". The concept of mental models appears in the field of program understanding, especially as it relates to software maintenance and evolution [VV95]. However such a "mental model" will be difficult to construct and maintain due to the scattered nature of the source code details, as shown in Figure 4-1. Though a compiler might easily keep track of this type of information, it much more difficult for the human reader to keep in mind while attempting to understand the code. As well, the understanding will tend to be vague and imprecise, and there will be insufficient documentation of what the analyst has learned so that it will not be easy to communicate this understanding to others. Finally, the information will not be easily analyzed.

Ultimately, the goal is to have a model of the safety concern source code that is easily analyzable. If a critical code path code could be extracted in the same fashion as, for example, the braking system, it could then be similarly evaluated in isolation. It could be dynamically analyzed (executed) or analyzed with available software tools. This type of analysis is almost impossible to perform when the source code is scattered across the implementation of the software system and the relevant code paths are intertwined with code for unrelated functionality.
4.2 Safety Concern Model Properties

The following describes the desired properties for the model of a critical code path associated with a safety concern. The model should be a conservative approximation, sufficiently complete and tractable.

4.2.1 Conservative Approximation

The model should be a **conservative approximation** in the sense that every property of the model is also a property of the actual system, as long as any assumptions made during the extraction are true. The assumptions can be validated independently from the model. Conservative approximations are sometimes desirable as they are usually less complicated than the actual system.

By this definition we mean that anything that is provable from the model is also true for the actual system. The concept of conservative approximation is illustrated by analogy with the use of mathematical assertions to describe a number. The phrase "is equal to 3" is a very precise description of this number - in fact, there is only one number that satisfies this description. But there are other less precise descriptions such as the phrase "is less than four". Anything that can be logically proven about the latter statement can also be shown to hold for the former statement. Therefore, the statement "is less than four" can be considered to be a conservative approximation of "is equal to three". In this same sense, the model of the system should be a conservative approximation of the system.

4.2.2 Sufficiently Complete

The model should be **sufficiently complete** in the sense that the model contains enough information to interpret unambiguously every executable statement in the model for the purpose of safety. The model would be incomplete if there is omitted or ambiguous information so that it is not possible to determine if the resulting behaviour is safe. In an analogous manner, a requirements specification can be judged to be sufficiently complete if it is possible to determine whether the specified behaviour is safe [JL+91]. When assessing potentially hazardous software behaviour, for example, it might be sufficient to know whether a particular function returns a non-negative result without needing to know the actual value.

During the source code extraction, branches of source code will be pruned. Pruning source code will potentially leave out information necessary for following the critical code path. If the results of a function call, for example, are required to make sense of subsequent source code statements, assumptions must be added to the model to address this.
4.2.3 Tractable

To the extent that the model is represented by fragments of source code, the model should be tractable. One measure of this quality might be keeping the ratio of executable lines of code to non-executable lines "reasonable". Another measure might be limiting to a reasonable number the different places in the source code required to understand a single line of executable code. This is especially important if the verification depends on human comprehension of the model. If human understanding is required, "reasonable" might mean that there is no more than "7±2" lines of non-executable code, and no more than "7±2" places to look, for each executable line of code.

The safety concern typically includes a large number of non-executable code statements. This includes statements that specify the class dependencies and that re-export types from other components spread over multiple files. A reduction in this clutter and spread of non-executable statements is one of the goals of the modeling process.

4.3 Safety Concern Model Vocabulary

For describing models of cross-cutting safety concerns, the analyst needs to make a decision on the "vocabulary" to be used. Determining the model vocabulary is more than merely a matter of deciding on a set of words to be used in the construction of models. The establishment of this vocabulary constitutes a decision about the conceptual building blocks that may be used to construct models.

These concepts are never explicitly identified in many of the modeling methods used in engineering. Instead, the concepts are simply understood by intuition and familiarity. In the case of a physical model, such as a scaled model of a new aircraft wing design to be used in a wind tunnel, the physical properties of the modeling material (e.g., wood) existentially determine the modeling vocabulary.

However, in the case of building models that are only described with text and diagrams, there is a greater need to clarify what concepts can be expressed in the model. We call this the vocabulary of the model. For instance, a UML model might be a very good way of describing interactions between various subsystems of a spacecraft. But there are no particular provisions in UML for the concept of "mass", so we would claim that "mass" is not part of the UML vocabulary. Though "mass" could always be included in a UML description as a textual element, it is not a conceptual building block in the same sense that UML has explicit provisions for interprocess communication.

---

5 The number 7±2 was proposed by Miller as our capacity for processing information [Mil56].
The vocabulary of a useful modeling approach should be limited as the ultimate goal is to articulate the critical code path as clearly and simply as possible for analysis. As a result, the safety analyst might insist that the source code be reduced to its bare essence consisting of nothing more than, say, a sequence of assignment statements operating exclusively on 64-bit unsigned words. However, this approach is unlikely to capture all of the essential software features that might contribute to a hazard. As well, there is greater risk of translation error as the "semantic gap" grows between the model and source code.

Therefore the vocabulary must be rich enough to describe the all the essential software features. At the other extreme, the safety analyst might then decide that every software feature present in the source code should be made available in the model. The more "features" allowed into the vocabulary, the smaller the semantic gap will be between the model and the actual implementation. However, in that case the model could turn out to be as complicated as the original source code.

The key question then is what features of the source code needs to be represented in the model. We propose some guidelines in the following sections on what should or should not be included in the vocabulary of the model.

4.3.1 Component Structure

The biggest challenge to the analysis of the safety concern source code is the fact that it is thinly scattered among many components, with a large amount of non-executable source code (e.g., generic instantiations) to support the class structure.

It would be desirable to remove the component and class structure as much as possible, which in Ada is supported by packages and generics. The model vocabulary should avoid the use of packages and generics.

4.3.2 Types

The class structure also leads to type hierarchies, where new types are derived from types higher up in the class hierarchy. These type hierarchies should be reduced as much as possible. For types not part of the critical data, the type hierarchy should not be specified. Instead the type can be left unspecified or specified as some basic type. For types that are part of the critical data, the type hierarchies can be maintained and placed in a single location for easy reference.
As a result, the vocabulary must be able to represent the type hierarchies of the critical data. This includes the basic language types, as well the composite types, such as arrays and records. However, access types or pointers are very difficult to analyze, and should be eliminated from the model.

4.3.3 Control Structures

In order to model the control and data flows, and to reduce the semantic gap between model and source code, the analyst may include basic control structures and subprograms in the vocabulary. This includes conditions (if, case), loops (for, while), as well as procedures and functions.

However, unstructured control flows, such as exceptions and go-to statements, should be eliminated from the model. These uncontrolled flows again greatly complicate the analysis of the model.

4.3.4 Concurrent Behaviour

If the software consists of flows of control that execute concurrently, the model can be seen as a collection of software processes that interact through message passing and shared data. The message passing may be either synchronous (i.e., sender is blocked until the message is received by the recipient) or asynchronous (i.e., sender continues executing after sending a message regardless of when the message is received by the recipient).

4.3.5 Summary

Ultimately, engineering judgment will be required to determine what to allow in the model vocabulary. Of course, the vocabulary should not be the only control on complexity of the model, as the modellers must also make intelligent use of the vocabulary. Nevertheless, the vocabulary will have a big influence. The guiding principle is to reduce complexity of the model as much as possible, while still maintaining the key features relevant to the hazard. In other words, a model consisting of a sequence of assignment statements operating exclusively on 64-bit unsigned words will be acceptable if it captures sufficient information about the safety concern source code. The reality will be that a slightly larger vocabulary will likely be required, based upon the judgment of the analyst.

4.4 Summary

In this chapter we have laid down the foundations for building a code model of the safety concern. The creation of models is an essential tool in many engineering disciplines including software engineering. There are many approaches to creating software models at the requirements or design level. However, software models at the implementation level are rare. To specify the type of model required of the source code, we have described the desired model properties and the allowed model vocabulary. In particular,
we want a model that is approximately conservative, sufficiently complete and tractable. This ensures the model is appropriate for safety verification. The model vocabulary will have a large influence on what is included in the model and will be a control on model complexity. Ultimately the modellers must also make intelligent use of the vocabulary and use engineering judgment to determine what should be included in the model.
5. Constructing Safety Concern Models

The model of the safety concern should help address the difficulties created for safety verification by modern software systems. The characteristics of the safety concern source code described in the Chapter 3 include the following:

- Scattering of the safety concern source code over many components, including those of only peripheral interest to the safety concern.
- High cognitive overhead of numerous generics.
- Large type hierarchies defined over multiple components.
- Functionality that is intimately part of the critical code path, but whose details are peripheral or irrelevant to the hazard.
- Source code that executes concurrently in different processes and threads.

To address these issues, we propose an approach for constructing a model of the safety concern that consists of four basic steps. The model construction process is presented in a complete and self-contained form in Appendix B. The appendix can be used as a handbook for analysts planning to apply the technique without needing to refer to the rest of the dissertation.

This chapter introduces the model construction process using the Chemical Factory Management System as an example. Additional details on the translation step can be found in the case studies discussed in Chapter 6. A review is given of existing tools and techniques that can be used in support of creating the safety concern model. We then discuss how the code model might be maintained in the event of changes to the source code. The informal application of the process to the ATM system by Raytheon engineers is described that provides evidence of the usefulness of code models. A summary follows.

5.1 A Process for Constructing a Safety Concern Model

The inputs of the process are:

- **Hazard identification and analysis**: Includes identified system hazards, hazard scenarios, and hazard causes.
- **Software architecture and design**: Includes the software components, processes and logical design (e.g., class diagrams), as well as object-collaboration diagrams.
- **Software source code**: Includes the available developed source code.
The outputs of the process are:

- **Safety concern description** – Description of the hazard and the relevant software components, including hazard scenarios and the corresponding object-collaboration diagrams.
- **Safety concern model** – Representation of the extracted source code corresponding to the safety concern, including assumptions about the pruned code.

---

**Figure 5-1: Outline of Document Containing the Safety Concern Model**

Figure 5-1 displays an example outline of a document that contains a description of the safety concern, as well as a portion of a safety concern model corresponding to a critical code path. The document includes relevant background information on the hazard and hazard causes, such as the critical components and object-collaboration diagrams. As well, the document contains the extracted source code, along with assumptions about the pruned source code.

The steps of the process are the following:
1. **Flatten** - modify classes by making explicit all the attributes, methods, conditions, and invariants the class inherits from other classes. Expand parameterized classes (generics or templates) by replacing parameters with the actual parameters, changing the parameters names as necessary to avoid namespace conflicts.

2. **Fillet** – identify and extract the critical code paths. Prune the less relevant code branches and, if necessary, replace with something simpler or with an assumption of the expected behaviour.

3. **Partition** – identify the portions of the critical code path that execute in separate processes.

4. **Translate** – represent the source code in a notation that allows the irrelevant details to be omitted.

Though the steps are to be carried out sequentially, they may also overlap. These basic transformation steps can be carried out by a number of smaller sub-steps. The nature of these supporting sub-steps will partly depend on the programming language used to implement the software.

The translation of the safety concern source code can be performed using the same programming language, by either copying the extracted source code directly into a text document (supplemented with descriptive text in “informal English”) or by structuring the extracted source code so that it is executable. However, the source code could also be translated into a different linguistic context such as another programming language or a formal specification notation.

There are potential disadvantages in translating into a different linguistic context, such as the potential mistakes that might arise in the translation, as well as the addition effort required to perform the translation. In the case of a formal specification notation, the stakeholders will potentially have great difficulty in understanding the model, as the formal notation can be syntactically very different than the programming language.

As discussed further in Section 5.1.4.3, the usual motivation for using a formal mathematical notation is that it is typically more expressive than a programming language. As well, most formal notations have the support of tool-assisted reasoning. However, there can also be value in the translation process itself, as the act of translation forces a close inspection of the safety concern that can create a greater understanding of the source code and potentially uncover safety-related issues. In other words, “the journey is its own reward”, regardless if any analysis tools are applied to the final model.

5.1.1 **Flatten**
The class hierarchies are “flattened” to bring together the relevant lines of code and to reduce the amount of “non-executable” code. As mentioned earlier in Section 3.1.4, an attempt to analyze a cross-cutting concern might involve "looking at more than a thousand different locations in the source code merely to understand a sequence of actions of less than fifty steps corresponding to executable source lines of code". Even the task of understanding a single assignment statement could conceivably involve examining a dozen different source code files. The high cognitive overhead required to absorb this scattered detail can easily result in errors or lapses in rigor. Simply put, the purpose of flattening is to reduce the number of different places in the source code that must be considered to follow a critical code path. Even without carrying out the rest of the process described in Appendix B, the result of flattening will reduce the cognitive overhead of mentally combined scattered fragments. Clearly, flattening is beneficial if the safety engineer only needs to consider two or three different source code files, instead of a dozen different files, to understand a line of code.

```java
class A {
    protected int x;
    public A() {
        x = 5;
    }
    public void nextX() {
        x = (x + 1) * 2 - x - 1;
    }
}
class B extends A {
    protected int y;
    public B() {
        super(x);
        y = x * 2;
    }
}
class C extends B {
    protected int z;
    public C() {
        super(x, y);
        z = 3;
    }
}
```

**Figure 5-2: Class hierarchy in Java**

Object-oriented languages such as Java and C++ have explicit class mechanisms. A simple example of a class hierarchy is shown in Figure 5-2. Class flattening typically means in this case producing an equivalent class without any inheritance. This involves modifying the class by making explicit all the attributes, methods, conditions, and invariants it inherits from other classes. For example, class “C”, as displayed in Figure 5-2, would be flattened into class “Cl”, as displayed in Figure 5-3. This allows the analyst to see every feature a class possesses in one location. In general, the details of the flattening process will vary according to the programming language used to implement the software. Flattening of
software implemented in an object-oriented programming language is discussed further in Chapter 9 as an avenue of future research.

The focus of the flattening process in the dissertation will be on software implemented in Ada 83, the language used in the ATM example examined. Ada 83 does not have an explicit class mechanism. In particular, it does not support dynamic dispatching. Class hierarchies are implemented through packages and generics.

```java
class CI {
    protected int x;
    protected int y;
    protected int z;

    public CI () {
        x = 5;
        y = x*2;
        z = 3;
    }

    public void nextX() {
        x = (x + 1)*2 - x - 1;
    }
}
```

Figure 5-3: Class C Flattened into Class CI

For software implemented in Ada 83, flattening involves “expanding” each generic instance and removing the package structure. A representation of each generic instance is created by explicitly replacing each generic parameter for in the generic body with the actual parameter, where the actual parameter name is potentially modified to avoid namespace conflicts. The generic expansion can be done manually, though there is the potential for tool support to produce this representation.

For example, the “SensorServer” component is an instantiation of the “Server” Generic:

```java
package SensorServer is new Server
    (SensorSourceCode => LM.SensorCode,
     SensorRange => LM.SensorUpdateRange);
```

The instantiation of the generic provides the actual parameters “SensorCode” and “SensorUpdateRange”, which are from the “LM” package.

The portion of the “Server” generic displayed in Figure 5-4 is translated into a flattened form in Figure 5-5.
For example, the actual generic parameter “SensorCode” has been renamed “LM_SensorCode” and substituted into the “Server” generic for “SensorSourceCode”. As well, the package structure has been removed and the various “Server” generic types have also been renamed to avoid namespace conflicts, e.g., “SensorReading” has been renamed “SensorServer_SensorReading”.

```plaintext
with Sensor;
generic
  type SensorSourceCode is (<>);
  type SensorRange is (<>);
package Server
--| This package processes the sensor readings:
is
  type SensorReading is
    record
      Data : Sensor.Object;
      Code : SensorSourceCode;
    end record;
  -->| The sensor information, along with physical sensor code
  type SensorReadingArray is array (SensorRange) of SensorReading;
  -->| The input sensor information, with physical sensor code
Figure 5-4: Fragment of the Server Generic

  type SensorServer_SensorReading is
    record
      Data : Sensor_Object;
      Code : LM_SensorCode;
    end record;
  -->| The sensor information, along with physical sensor code
  type SensorServer_SensorReadingArray is array (LM_SensorUpdateRange) of SensorServer_SensorReading;
  -->| The input sensor information, with physical sensor code
Figure 5-5: Flattened Source Code from the Server Generic

5.1.2 Fillet

Filleting involves slicing out the “flesh” of the safety-critical source code, carving around the “skeleton” of the static software architecture. In particular, filleting involves:

- **Identifying Software Hazard Causes and Scenarios**: The relevant object-collaboration diagrams and component hazard causes are determined.
- **Slicing the Source Code**: The source code is searched for code relevant to the safety concern, guided by the object-scenario diagrams and component hazard causes.
- **Pruning the Source Code**: The identified source code is “pruned”, as not all of the source code related to the safety concern needs to be included in the model.
As noted in Chapter 3, if every possible critical code branch was followed, it is possible that a large portion of the software system would then be deemed part of the safety concern, which would not be useful for the purposes of safety verification.

5.1.2.1 Identifying Software Hazard Causes and Scenarios

The results of the hazard identification and analysis are used to guide the search for the safety concern source code. The hazard identification provides the scope of the hazard. The hazard analysis of the software requirements and design identifies potential hazard causes and scenarios.

For the Chemical Factory Management System hazard, as mentioned in Chapter 2, a preliminary hazard analysis resulted in two basic hazard scenarios:

1. The vessel temperature display is updated with a stale or corrupted temperature value.
2. A displayed valid temperature value becomes stale.

The hazard scenarios provide the basis for the search for critical code paths.

As discussed further in Section 5.2.2.1, standard hazard analysis techniques include FTA and FMEA. The results of a hazard analysis of the software design and architecture include a set of components that are relevant to the hazard, as well as the potential hazard causes at the component level. The search for the safety concern source code begins with the identified components, guided by the component causes.

For an object-oriented design, the dynamic behaviour of the software can be described by object interactions, where an object is an instance of a class. The object-collaboration diagrams are useful for displaying the objects and their interactions corresponding to a hazard scenario and guiding the initial search for the critical code paths.

For example, for the Chemical Factory Management System, the first hazard scenario involves the display of temperature updates. The corresponding sequence of object interactions is shown in Figure 5-6. Sensor updates are received at the server through the “SensorInterface” class, before being sent to the “SensorServer” for processing. The processed temperature updates are broadcast to the workstations where they are processed by the “UIManager” and displayed on the screen through the “Display Block”.

The second hazard scenario involves the display of stale temperatures. The interaction diagram shown in Figure 5-7 graphically represents the update of a stale temperature as unavailable. The staleness
mechanism is implemented in the software by an object “SensorMonitor” on the server that starts a timer upon receipt of a temperature update. The object broadcasts a message to “UIManager” on the workstations to set the temperature value to “unavailable” if further updates are not received for the vessel within a certain time constraint.

From the object interaction diagrams, we are able to determine primary objects related to the safety concern:

- SensorInterface
- SensorServer
- ServerMonitor
- UIManager
- DisplayBlock
For the Chemical Factory Management System, the underlying implementation is simple and there is a one-to-one correspondence between the classes and the development components. Figure 3-3 displays the relevant components and subprograms. However, for more complex systems, such as the ATM system, this is not likely to be true. In particular, a number of different components might be used to implement a single class.

5.1.2.2 Slicing the Source Code

The critical code paths are identified and extracted from the source code. The search for the critical code paths can be performed in both a backwards and forward fashion. The critical code paths might involve control flows, data flows or a combination of the two flows. For the Chemical Factory Management System temperature hazard, a data flow would be useful in understanding the sequence of transformations performed by the system to convert a temperature sensor reading into a displayed temperature value. However, the displayed temperature reading is also affected by factors not directly related to the temperature value, such as the update of the displayed temperature as stale when a reading is determined to be stale. This portion of the critical path would involve a control flow.

The object-collaboration diagrams are used to guide the forward code search, from system inputs to system outputs. Each key component identified in the diagrams is examined in turn, beginning with the procedure call that provided entry into that component. Subsequent procedure calls are then traced within that subsystem until the path leads out to the next component in the interaction diagram. The diagram can be modified depending on what is uncovered in the search.

For example, the Chemical Factory receives a temperature update by an invocation of the subprogram “ReadLAN” in the “SensorInterface” component. The source lines of code that might affect the temperature are traced in this component until a call is made to the subprogram “ProcessSensors” in the “SensorServer” component. The search then continues in the “SensorServer” component.

The analysis can then be performed in a backward fashion, starting with the system output and identifying the lines of source code that might lead to that output. For example, the temperature is ultimately displayed on the screen by a call to the “SetTemperature” subprogram, which belongs to a COTS product. An invalid temperature display would be manifested with this subprogram call. The software can then be searched for code that might invoke this subprogram and so on, until potentially a system input is reached.
5.1.2.3 Pruning the Source Code

The code extraction involves removing irrelevant functionality from consideration in the following fashion:

- **Irrelevant Source Code**: The pruned source code does not contribute to a hazard occurrence and is eliminated from the model.
- **Unreachable Source Code**: The pruned source code is unreachable for a particular safety concern and is eliminated from the model.
- **Conservative Approximation**: The pruned source code potentially contributes to a hazard occurrence, but will be replaced by something simpler in the model that is a conservative approximation of the actual software.
- **Separate Validation**: The pruned source code potentially contributes to a hazard occurrence, but will be replaced by something simpler in the model and addressed separately.

5.1.2.3.1 Irrelevant Source Code

The pruned source code can be simply eliminated from the model if it is not relevant to the safety concern. For example, for class C1, the variable “z” might not have any impact on the hazard, so the variable “z” and assignment statement “z = 3” could be pruned.

```plaintext
procedure ReadLAN(Message : in LM.Object)
is
    Readings : SensorServer.SensorReadingArray;
begin
    for I in LM.SensorUpdateRange range 1..Message.NumUpdates loop
        Readings(I).Data.SensorQuality := Normalize(Message.Updates(I).InterpolatedState);
        Readings(I).Data.SensorTemperature := ConvertTemperature(Message.Updates(I).TemperatureEstab);
        Readings(I).Code := Code;
    end loop;
    SensorServer.ProcessSensors(Readings);
end ReadLAN;
```

**Figure 5-8: Subprogram “ReadLAN”**

For the Chemical Factory Management System, the subprogram “ReadLAN” is part of the critical code path for the temperature hazard. As shown in Figure 5-8, the subprogram involves a call to the subprogram “Normalize”. However, examination of the “Normalize” subprogram indicates that the
temperature value is not referenced in any fashion. As a result, with respect to the hazard scenario involving the corruption of the temperature, the “Normalize” subprogram can be eliminated from the model, as displayed in Figure 5-9.

5.1.2.3.2 Unreachable Source Code

Some portions of the critical code path could be deleted on the basis of being executed only in a particular state that can be shown to be unreachable in a particular safety concern. For example, if a system has distinct modes of operation, state machine diagrams (e.g., UML statecharts) could be used to help determine which portions of a critical code path corresponding to a particular state is unreachable. In general, simplification of state machines could be used to guide source code pruning, which is discussed further in Section 5.2.2.3.

```procedure Sensorinterface_Readlan(Message : in LM_Object)
  is
  Readings : Sensorserver_Sensorreadingarray;
  begin
    for I in LM_Sensorupdaterange range 1..Message.Numupdates loop
      Readings(I).Data.Sensortemperature := Sensorinterface_Converttemperature(Message.Updates(I).Temperatureestab);
    end loop;
    Sensorserver_Processsensors(Readings);
  end Sensorinterface_Readlan;
```

Figure 5-9: Flattened and filleted subprogram “ReadLAN”

5.1.2.3.3 Conservative Approximation

The pruned software can be replaced in the model by something simpler, which is a conservative approximation of the actual software. The principle of conservative approximation is essentially an application of Occam's razor, i.e., the model constructed should be the simplest one possible that still represents the desired behaviour.

For example, for class “C1”, "y = x * 2" can be replaced with "y = some even number larger than the current value of x" or "y = some even number" or "y = some number larger than y" or even just "y = some number". Each of these alternatives is conservative approximation to the actual code, i.e., anything that can be proved from the conservative approximation is also true of the original code.
5.1.2.3.4 Separate Validation

The pruned software can be replaced in the model by something simpler that assumes the correct desired behaviour for the pruned software. The assumption concerning the behaviour of the pruned software can be validated independently. For example, the class “C1” has the method “nextX”, intended to increment the value of “x”. Rather than include the entire function in the code model, the function can be replaced by something simpler that returns the correct value of “x” for the scenario under consideration.

In general, it is useful to separate in the analysis the behaviour of the application level processing and the reliability of the low-level services. Low-level services include code libraries, communication and concurrency mechanisms, abstract data types and COTS products.

For the Chemical Factory Management System, the subprogram “ProcessSensors” (as shown in Figure 3-4) makes use of the “Store” mechanism, which provides a database of sensor readings. The “Store” function “IsFound” determines whether a sensor reading corresponding to a particular sensor exists in the database. For the purposes of the analysis, the “IsFound” procedure can be replaced by something simpler, such as always returning “True”, as shown in Figure 5-10. The reliability of the “Store” mechanism can then be validated separately, allowing the safety concern model and the subsequent analysis to focus on the processing of the temperature.

```plaintext
function Sensorstore_Isfound(K : Sensor_Sensorid; S : Sensorstore_Object) return Boolean
is
begin
  return True;
end Sensorstore_Isfound;
```

Figure 5-10: Pruned function “IsFound”

5.1.3 Partition

Partitioning involves identifying the different processes involved in the critical code path and their method of communication. The processes (including threads) correspond to different flows of control in the system. Inter-process communication can be either asynchronous or synchronous.

Active objects can be used to model the threads and processes corresponding to different flows of control. The main active objects are shown in Figure 5-11 for the Chemical Factory Management System. The processing of the incoming sensor updates occurs on the server. The updates are broadcast to the various workstations for display. There is a separate thread on the server for monitoring the staleness of
the temperature. In this instance, there is not a one-to-one correspondence between active objects and classes.

The relevant processes and the corresponding source include:

- **LANToBroadcast**: From receipt of a LAN Message by the system to a broadcast to the operator consoles. This includes source code from the SensorInterface and SensorServer components.
- **BroadcastToDisplay**: From receipt of a broadcast by the operator consoles to the display of the temperature on the display. This includes source code from the UIManager and DisplayBlock components.
- **MonitorSensorStaleness**: The subprogram monitoring the staleness of the sensor reading. This is taken from the SensorMonitor component.

![Figure 5-11: Active objects used to represent processes and threads.](image)

The concurrent behaviour of the software could potentially be formally modeled. For example, state machine diagrams (e.g., UML statecharts) could be used to represent the process interactions. However, the details of how this might be done are beyond the scope of this dissertation. Potential techniques for modeling real-time systems are reviewed in Section 5.2.3. Future work in modeling the processes in terms of functional blocks is described in Chapter 9.

5.1.4 Translate
The final step of the process is to translate the critical code path into a form where the irrelevant source code details have been eliminated, including a representation of the pruned source code. Potential notations for the representation include "informal English", a programming language, or a formal mathematical notation. Chapter 6 presents additional details on how to specify a code model in each of these notations.

5.1.4.1 Informal English

A simple technique for representing the critical code paths involves extracting the relevant source code and annotating the code fragments in English. The annotations indicate the code’s relevance to the hazard, as well as assumptions about the pruned code. The annotations can be supplemented by figures such as object-collaboration and component diagrams that display the important components and data flows.

The greatest virtue of an informal language notation is that it is easily understood and communicated to others. Another virtue is its unbounded flexibility. It obviously does not require any special knowledge or tools on behalf of the analyst or other stakeholders in order to create or understand the model.

The limitations of such an approach are that the model is imprecise and not easily analyzed. The model will not be executable or amenable to any type of tool-based analysis. There will be no method of confirming that all assumptions about the pruned code have been captured.

5.1.4.2 Programming Language

A potentially more useful technique than informal English is the representation of the model in an executable format, i.e., a programming language:

a) Use the same language as the source code, or perhaps a "safe subset" of this language;
b) Use a different programming language, e.g., a functional programming language or
c) Use a specialized programming language that provides explicit support for reasoning about software behaviour.

The obvious benefit of an executable model is that it could be subject to dynamic analysis (e.g., testing). The advantage of using the same programming language is that the resulting model will be in a familiar notation (at least to those familiar with the source code).
The key limitation of this technique is that stubs would need to be created for the pruned subprograms called within the model. However, the need for program stubs is the same limitation inherent in unit or integration testing. Similarly, the model will be limited to the same type of analyses that can be performed on the original system.

If a different programming language were used, translation can be considered to be a verification exercise in itself in addition to testing of the final model. As well, in the case of a functional programming language, the functional language paradigm discourages "side effects", especially if one uses a purely functional approach. The advantage of avoiding side effects is that it makes the data transformations explicit and more readily analyzed. However, the limitation of using a different language is the much greater overhead in performing the translation and the notation will likely not be as familiar to stakeholders.

There are programming languages that also support reasoning about software behaviour. For example, the SPARK language has tool support for static code verification. The limitation is that there is again the additional overhead in translating the source code into a different programming language and the use of potentially unfamiliar proof tools.

5.1.4.3 Formal Mathematical Notation

Another specification approach would be use of a formal mathematical notation. Benefits of using a formal notation include the fact that formal notations are usually more expressive than a programming language, the translation effort itself is a form of code analysis, and there is the potential for tool-assisted reasoning about the model.

The limitation of mathematical notations is their lack of familiarity to most developers, and, typically, their syntactic distance from the source code. Use of these methods would likely require training for the analysts and other stakeholders who make use of the models.

Having a more expressive notation is particularly useful for specifying conservative approximations of the pruned code. Descriptions in a programming language are always algorithmic. A key advantage of a formal mathematical approach is the ability to describe behaviour in declarative rather than algorithmic terms. For example, we could describe the square root function with a formal mathematical notation by simply stating that it is a function that takes an input \( N \) and produces an output \( M \), such that \( M \times M = N \).

5.2 Supporting Techniques and Tools
The process for creating a model of the safety concern can potentially be supported by a variety of existing software and safety tools and techniques. Though none of these tools or techniques alone is sufficient for creating a safety concern model, they can support one or more steps in the model creation.

An alternative to existing tools is the possibility of building a tool that supports the code model creation using the semantic and syntactic information available in a compiler environment [Coo97]. In the case of Ada environments, the Ada Semantic Interface Specification (ASIS) [ACM97] provides access to compiler-generated information through an interface that is at the same semantic level as the Ada language [Ehr94]. This simplifies the construction of tools for specialized purposes, such as analyzing the differences between subsequent system builds, including changes in the system parameters [CSJ97]. For non-Ada environments, there exist tool support such as JANUS for parsing source code implemented in different programming languages to build an in-memory knowledge-based abstract syntax tree (KBAST) model of the entire system [ND01].

5.2.1 Flattening Tools and Techniques

This dissertation focused on software implemented in Ada 83. The flattening process was primarily a matter of "expanding" generics by manually replacing generic parameters with the actual parameters for the different instances of the generics, where the parameters are renamed to avoid namespace conflicts. There is potential for automating the process, though the author is not aware of any existing tools. It is possible that a flattening tool could be built using ASIS.

For object-oriented languages, class flattening typically means producing an equivalent class by making explicit all the attributes, methods, conditions, and invariants it inherits from other classes. Flattening is also used in subject-oriented programming [OK+96]. Tools that display a flattened form of a class are available for some object-oriented languages. For example, the EiffelStudio software development environment displays a "flat form" of a class written in the Eiffel programming language [Eif04]. More commonly, software development environments will typically have a class browser that display all the methods for a class, including those inherited from a parent class.

5.2.2 Fillet Tools and Techniques

Techniques for identifying critical code paths include hazard analysis and source code searching methods. In addition, some of the techniques described in Chapter 8 for creating safety concern models, like program slicing and software visualization tools, could also be potentially used.

5.2.2.1 Hazard Analysis
There are a number of hazard analysis techniques [IW90] available that potentially could be used to search for hazard-related software. For example, standard techniques include FTA, FMEA, Hazard and Operability Studies (HAZOP):

- FTA begins with the hazard as the “top event” and then the events that cause the hazard are determined, combined using logical operations such as “AND” and “OR”.
- FMEA is typically used to examine the consequences of failures in system components, beginning with a component fault and then tracing the effects forward to system outputs.
- HAZOP examines the consequences of deviations from the system design with the help of guidewords such as “NO” and “REVERSE” to identify potential deviations.

These techniques were originally designed for non-software systems, though there have been attempts to adapt them to software.

At the software specification level, Lutz and Woodhouse have extended FMEA to software in a method they call Software Failure Modes and Effects Analysis (SFMEA) [LW96]. SFMEA and an adapted version of FTA were then applied to specifications of spacecraft software modules. In particular, a set of guidewords appropriate to software modules was devised and documented in a Data Table and an Event Table. Levenson and colleagues have also developed a suite of tools and techniques for analyzing software specifications, including a variation of HAZOP called Deviation Analysis [ML+97]. In particular, they propose building a state-machine model of the system requirements that can then be analyzed with their toolset.

At the software design level, McDermid and colleagues have derived a set of analysis techniques aimed at integrating design and analysis [FM+94]. This includes a modified version of HAZOP called Software Hazard Analysis and Resolution (SHARD) for performing a structured exploratory safety analysis of a software design. Guidewords for SHARD are derived from the field of software failure classification. As well, they have developed a notation called Failure Propagation and Transformational Notation (FPTN), which represent the software as a set of modules and their failure modes, with the modules connected by the potential failures that propagate between them. FPTN is intended to overcome some of the limitations of FTA and FMEA when applied to software. For example, software fault trees tend to become large and intractable.
As noted previously, software hazard analysis techniques are useful for analyzing the software at the requirements or design level. These methods, however, are not well suited for determining which lines of code may contribute to a given hazard.

5.2.2.2 Code Searching Tools

Though the search for hazard-related code is primarily manual, the complexity of the code means that tool support is important. There is a variety of tools which could be used to expedite the process of searching source in either a forward or backwards direction.

Basic lexical searching tools like the unix utility "grep" can be used to search for regular expressions in the source code. Typically, software development environments provide additional support for component cross-referencing which would be useful when tracing critical code paths across component boundaries. For example, the Rational Apex Ada editor offers interactive hypertext Ada browsing [RAD04]. This includes links to type declarations and to places where a given type is used. Cross-referencing capabilities will be useful when tracing the critical code paths and uncovering static dependencies on other lines of source code.

Some software development environments are integrated with design tools in support of “round-trip engineering”, for going from the design to the source code and back again. For example, Rational Rose XDE Developer [RRX04] supports UML and roundtrip engineering with Java or C++. If the hazard-related object-collaboration diagrams have been identified, these tools can be used to help with the initial forward search of the relevant components.

As with flattening, another potential approach would be to create a search tool that queries the semantic and syntactic information available in a compiler environment.

5.2.2.3 Simplifying State Machines

If the system has distinct modes of operator or states, then state machines could be useful design documentation as a guide to pruning. As suggested in Section 5.1.2.3.2, portions of source code might be deleted on the basis of being only executed in a particular state that is shown to be unreachable in a particular safety concern.

There is an algorithmic procedure used by digital circuit designers to simplify a state machine by merging states - with precise rules that govern when two states can be merged. Similarly, software hierarchical state machines can be simplified and sliced according to a scenario [HW97]. The slicing of a
state machine representation of the software requirements could be used to guide what source code to prune.

5.2.3 Partitioning Tools and Techniques

The partitioning step involves identifying the different processes in the system. A formal model could then be created of the processes and their interactions. In particular, there are a number of different approaches that can be used for the specification and analysis of real-time systems that include formal logics and algebras, as well as graphical language techniques with formally defined semantics [Ost92].

There are a large variety of temporal and real-time logics available for formally specifying and verifying real-time systems [Gup92]. 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]. Process algebras are used for analyzing concurrent behaviour, including constructs for specifying parallel and sequential composition of systems [Bae04].

Graphical language approaches include Petri Nets [LS87], Statecharts and Statechart variants such as RSML [LH+94]. Statecharts are also used as a modeling technique in UML [BRJ99].

5.2.4 Translating Tools and Techniques

Three approaches were suggested in Chapter 5 for representing the critical code paths, including informal English, a programming language, or a formal mathematical notation.

5.2.4.1 Language Subsets

The safety concern could be represented in a language subset of the original programming language. For example, there are the MISRA C [MIS04] and SPARK [Bar97] language subsets.

The use of SPARK to represent a safety concern is discussed further in Chapter 7 of this dissertation.

Another alternative are functional languages like ML, which are also higher order with functions that can modify and combine existing functions. Functional languages have been used to create executable specifications and to prototype implementations.

There is a strong syntactic and semantic correspondence between S and ML. Of course they are quite different in that S is a non-executable mathematical notation, while ML is an executable language. ML does not provide quantification, nor does it provide anything analogous to Hilbert’s epsilon operator. But aside from these differences, the fact that we can use S to represent a safety concern (as demonstrated for
the ATM system) is a strong indication that ML could similarly be used to represent a safety concern. The advantage of a safety concern model represented in ML is that it would be executable and could be used as a basis for dynamic verification.

5.2.4.2 Formal Specification Notation

There is a wide range of formal mathematical notations typically intended for requirements specification [Win90]. Some of the more popular notations, such as “Z”, involve special symbols or formatting. As well, use of the notations to create a specification often means building up an elaborate model of the functionality using basic elements of discrete mathematics.

The notation used to represent the safety concern model must be able to capture the object-oriented features in the programming language. As a result, a potential alternative to the standard specification notations would be an object-oriented formal notation, such as one of the object-oriented extensions to Z [SBC92]. These notations are intended for the formal specification of object-oriented designs.

However, as will be shown in Section 6.3, a notation based on a typed predicate logic is adequate for specifying an object-based language. In fact, there are advantages in using a semantically simpler notation, without the object-oriented design constructs, which can mimic the desired language constructs. Notations based on higher-order logic include “S” [JDD94], the “term language” of the HOL system [GM93], and the PVS specification language [OS+95].

5.3 Maintaining the Code Model

The execution of the code model extraction must be recorded in a form that would allow the process to be repeated, while providing traceability between the model and the source code. The repeatability of the process is particular important for maintaining the model in the event of source code changes.

The simplest method of recording the extraction of a model is a textual description of the decisions made during the extraction. The description would include:

- The originating components and subprograms for each element in the model.
- The pruned components and subprograms, as well as an explanation for their exclusion from the model.
The description could be supplemented with observations such as: "Inspection of the source code revealed that line 872 of the file ‘mloop.ada’ contains the only executable statement which could cause variable M to be changed from its initial value.”

It would be particularly useful if generation of the model was automated or partially automated in some fashion. Some alternative techniques for recording the code model extraction are suggested as future work in Chapter 9.

5.4 Validation – ATM System

Raytheon engineers informally applied the code model construction process to several hazards during the safety analysis of the ATM system and a related military system, Military Automated Air Traffic System (MAATS). As noted by Joyce, the Safety Authority on both projects, “In several instances, the model of a safety concern obtained by this process was particular useful in finding potential sources of risk in source code” [Joy04].

One particular example involved the potential display of “stale” altitude data for a flight, due to the interaction of a software mechanism with another involved in altitude data processing. The problem was at the source code level, so would not have been revealed by an analysis of the requirements or design. As well, a routine code quality check (e.g., searching for uninitialized variables) of the source code would not have identified the problem, as the issue was not a matter of the source code failing to adhere to coding standards. In fact, the problem was not a “bug” in that neither of the software mechanisms was defective. Though system test results did show symptoms of this problem, it was the analysis of the source code model that pinpointed the problem and the particular software mechanism that was the root cause.

The experiences of the Raytheon engineers provide evidence of the effectiveness of using code models to supplement existing safety verification techniques. The problems uncovered by the code models were not identified by the other analysis techniques employed, such as code reviews, requirements analysis and system testing.

5.5 Summary

We have proposed the construction of safety concern models to address the challenges posed by modern software architectures to system safety. In particular, we can no longer assume that verifying the safety of the source code involves tracking the footprints of the safety concern in a single “backyard”, but might involve a number of sites in a variety of neighbourhoods in different cities. The safety concern model
removes the need to traverse these footsteps more than once across the system, by placing them in a single location for analysis.

In this chapter we presented an approach for creating a model of the safety concern designed to address the issues raised by the cross-cutting nature of safety concerns. The four basic steps include flattening the class hierarchies to reduce the amount of “non-executable” code, filleting the source code to prune out the irrelevant details, partitioning the extracted source into separate processes, and translating the extracted safety concern source code into another notation. The process is not tied to any particular tool, notation, or method. Instead it provides a general framework for what needs to be done to arrive at the safety concern model.

The usefulness of code models was validated by their use by company engineers during the safety analysis of the ATM system and a related military system, which uncovered safety-related issues not identified by other analysis techniques. In the next chapter, the effectiveness of the process is further evaluated when different translation notations are used.
6. **Examples of Representing Safety Concerns**

This chapter addresses the question of whether it is possible to construct a "useful" model of a cross-cutting safety concern. Earlier in this dissertation, Chapter 3 described the problems created for safety verification when a safety concern is scattered among a number of different components. Chapter 5 then proposed a process for constructing a model of the source code that was intended to address these problems. The usefulness of code models was validated by the experiences of engineers using code models during the safety analysis of the ATM system. The purpose of this chapter is to offer further evidence that the process for constructing a safety concern model can really be used to solve the problems posed by cross-cutting safety concerns.

The usefulness of the model will depend on how amenable the model is to analysis. As described in Chapter 2, safety verification can involve either static analysis or dynamic analysis (e.g., testing). This potentially includes a spectrum of techniques. For static analysis, it can be as simple as an inspection of the source code in order to address specific questions (e.g., what compile-time constants affect this output?).

In order to evaluate the effectiveness of the safety concern models, we addressed the following questions when applying the approach to example systems:

- **How tractable is the resulting model?** An important goal of constructing a model is to reduce the cognitive overhead of scattered concerns and the large amount of non-executable code due to class hierarchies.

- **How analyzable is resulting model?** Ultimately, the model is used for safety verification and must be analyzed in some fashion. Questions include how amenable the model is to inspection, whether the analysis can be performed dynamically (i.e., whether the model executable) and whether there are tools to support the analysis.

- **How faithful is the resulting model?** That is, to what extent does the selected translation technique ensure that the model is a faithful representation of the actual source code? Conversely, what opportunities does the technique create for ambiguity or misinterpretation?

- **How complete is the resulting model?** This not only means capturing the relevant code details, including type declarations and subprograms, but also specifying the pruned source code.
• **How usable is the approach and resulting model?** In particular, are special training and tools required to create, understand and analyze the model?

Simply put, the overriding question is whether the resulting safety concern model is something coherent that can be understood by the relevant stakeholders and effectively analyzed during safety verification.

The approach to creating a safety concern model was evaluated by two substantial case studies involving the Chemical Factory Management System and the ATM system. As described in Chapter 2, the ATM System is a large real-time, command and control system with an object-oriented software architecture. The Chemical Factory Management System is a “toy model” of a command and control system with a software architecture similar to the ATM system.

The rest of the chapter describes the creation of safety concern models using different notations including informal English, a programming language, and a formal mathematical notation. Our discussion of these different approaches is based on the models documented in the appendices. The effectiveness of each approach is evaluated with the main emphasis on translation step. However, as the choice of notation does influence how the earlier steps are performed, each case study will comment on all steps of the approach.

**6.1 Informal English**

This section describes the creation of a safety concern for the ATM System with extracted source code, explanatory English, and object-collaboration diagrams. This technique is likely to be the "status quo" in industry, i.e., what a safety engineer is likely to do today in the absence of any specialized technique. As previously discussed, this was the approach taken by engineers in the original safety analysis of ATM system.

With the informal English technique, the excavation of the code proceeds with the identification of the relevant subprograms and supporting data types. The source code is then “cut-and-pasted” into a word processing document and supplemented with an informal commentary in English. An example of a code model in informal English for the Chemical Factory Management System can be found in Appendix B.

The remainder of Section 5.1 describes the representation a safety concern model using informal English. This is followed by a description of the experience of applying this approach to an ATM System. The evaluation results are described in the summary of this section.
6.1.1 Flattening

Flattening the class hierarchies involves extracting the relevant source code from the generic and package bodies. In essence, the package and generic structure is eliminated.

In addition, the generic instantiations, which defines what is substituted in for the generic parameters, are also extracted and included in the model. This allows for convenient reference to the instantiation parameters when analyzing the generic source code.

For example, the instantiation of the "Interface" generic to create new component "SensorInterface" is defined as follows:

```view
package SensorInterface is new Interface
(SensorCode => SensorCodeEstab);
```

The generic instantiation (as shown above), along with the generic body, would be copied verbatim into the model, e.g., the generic body is not modified by the replacement of "SensorCodeEstab" with "SensorCode". Instead, the analyst can refer to the generic instantiation for the actual generic parameters during analysis of the generic source code.

6.1.2 Filleting

During filleting, the relevant subprograms and type definitions are extracted and copied into the model. The focus is on the subprograms that are explicitly part of the critical code path and the types involved with the critical data. For instance, the critical code path might involve a data flow that includes a series of subprogram calls, with the hazard-related data passed from subprogram to subprogram.

The relevant subprogram bodies are extracted from the package and generic bodies and then copied into the model. Similarly, the declarations of the data types corresponding to the variables maintaining the critical data are also copied. For traceability, supplementary text is added that describes the originating packages and generics.

For example, the subprogram "UpdateTemperature" from the package "DisplayBlock" is involved in the display of the temperature, which is a critical data item.

```view
procedure UpdateTemperature(This : Object) is begin
  if (This.Available = TRUE) then
    GRU.SetTemperature(This.Display, This.PresentTemperature);
  else
    GRU.SetAvailability(This.Display, This.Available);
  end if;
end UpdateTemperature;
```
The temperature is held in a “DisplayBlock” object also from the “DisplayBlock” package,

```plaintext
type Object is record
  Active : Boolean;
  Available : Boolean;
  Interpolated : Boolean;
  PresentTemperature : DisplayTemperature;
  Display : DisplayID;
end Record;
```

Other packages such as “Sensor” similarly contain declarations for types named “Object”. This is part of
a naming convention intended to support the use of Ada 83 to implement an object-based system.

The body of “UpdateTemperature”, along with the declarations of the “DisplayBlock” object and
“DisplayTemperature” types would be copied from the source code. A textual note would be added that
these are from the “DisplayBlock” package.

Subprograms determined to be outside the critical code path are not extracted, even if they are called
from within the path. Instead, assumptions about the pruned code are documented. The assumptions are
in the form of explanatory text that describes which particular source code branches have been pruned
and the reasons for having done so.

For example, the processing of the sensor readings involves processing (e.g., “normalization”) of data
known as the Sensor Quality. The Sensor Quality is not relevant to the temperature hazard and can be left
out of the safety concern model. The pruning of the Sensor Quality processing is noted as an assumption
in the appropriate section of the model:

B.3.1.2 Assumptions
The normalization of the Sensor Quality is not extracted, as it is not relevant to the processing of
the temperature value.

6.1.3 Partitioning

The different processes involved in the critical code path are identified along with their method of
communication, e.g., an asynchronous or synchronous procedure call. A textual description is provided in
the model. Diagrams, such as object scenario diagrams with active objects, can be used to help illustrate
the processes and their interactions. For each process, a list of the relevant packages and subprograms is
documented.

6.1.4 Translating
For each subsystem, the relevant packages and data types are copied from the source code. An overview is provided of the critical code paths. Then, for each package, the following are also documented in the model:

- Overview – a description of the extracted procedures
- Assumptions – a description of the pruned source code with explanations
- Subprograms – the subprogram bodies cut and pasted from the source code

The model is documented in a text file or word processing document.

6.1.5 Example: Air Traffic Management System

We applied the approach with informal English to a safety concern related to a hazard for an Air Traffic Management System.

During the flattening step, it was a challenge to document the static dependencies of the source code, i.e., what packages the source code originated from. For example, in one situation a generic was defined in one package, instantiated in a second package, with one of the instantiation parameters defined in a third package, and the generic instance used in a fourth package. Explanatory text was added to indicate which package was relevant to each piece of source code.

For filleting, the source code was available in HTML format and could be examined with a web browser. Source code was extracted from six subsystems spread over the top three layers of the software architecture. Approximately 1-2 k source lines of code were extracted.

Most of the code path of the track update had been written to execute synchronously for performance reasons. Situations where this did not occur included:

- The broadcast of data by the server resulting in the asynchronous invocation of a callback subprogram on the client side
- User interface callback subprograms
- Subprograms scheduled to be invoked in a periodic fashion, such as those intended to monitor data staleness

The relevant source code was copied into a Microsoft Word document. The document consisted of explanatory text supplemented by fragments of source code copied directly from the software. For the six
subsystems, the extracted source code and explanatory text involved approximately 120 pages. Although this may seem to be a large volume of code in a model that it is intended to focus narrowly on a particular safety concern, it represents less than 1% of the overall developed code of the system.

In the course of creating the safety concern model, we raised questions about the hazard causes and assumptions that appeared in the hazard definition and hazard analysis. This led to the identification of new hazard scenarios and the refinement of the existing ones.

6.1.6 Informal English: Summary and Discussion

In order to evaluate the effectiveness of the informal English technique, we now address the questions posed at the start of the chapter, i.e., whether the resulting safety concern model is tractable, complete, faithful, analyzable, and usable.

6.1.6.1 Tractable

Compared to the original system source code, having the safety concern source code gathered together in one place was a great simplification. The data type declarations and the generic instantiations were all conveniently in the same document near the source code.

6.1.6.2 Complete

Assumptions were easy to state in the informal English model and were clearly identified in a separate section of the document. However, it is difficult for us to determine if the resulting assumptions describe all the source code that was pruned and accurately represented the expected behaviour. In particular, there is no simple method of doing consistency checks between the model and source code that might reveal missing assumptions.

6.1.6.3 Faithful

The model consists of source code copied from the implementation into a document without modification. These source code fragments are obviously an accurate reflection of the actual source code. However, the supplementary text describing the critical code path, processes, and subprograms could be inaccurate or misleading.

6.1.6.4 Analyzable

Having the scattered safety concern source code in one place greatly simplified inspection of the source code. As well, it is useful to have explanatory text that describes the data flow and the reasons the subprograms were pruned.
However, as the generics were not expanded, there is the cognitive overhead of remembering the instantiation parameters when examining the generic source code. As well, it is difficult to follow and understand the concurrent behaviour of the processes. Ultimately, the model is not analyzable in any fashion other than inspection. In particular, no tool support is available.

6.1.6.5 Usable

The informal English approach required no additional skill beyond knowledge of the system and source code.

6.2 Use of a Programming Language as a Modeling Language

Chapter 5 described several alternatives to the use of informal English for representing a safety concern model. One of these alternatives is the use of a programming language as a modeling language. This might be the same language used to implement the system, or it might be a different language. As suggested in Chapter 5, one advantage of using a programming language is an executable model that can be tested.

To investigate this alternative, we experimented with the use of the programming language Ada to create a safety concern model for the Chemical Factory Management System. In this instance, we choose to use the same programming language used to implement the system, namely, Ada. Using the same language for both implementation and modelling obviously reduces the amount of translation effort. Nevertheless, our experience with this approach shows that the construction of the model is not just a simple cut-and-paste exercise.

In this section, we describe the four steps of the approach when using Ada as the choice of notation. We then present the results of applying the approach to the Chemical Factory Management System. This is followed by a summary and discussion. The resulting model can be found in Appendix C.

6.2.1 Flattening

Flattening involves removing the generic and package structure. As well, flattening involves replacing the generic parameters with the actual parameters for a given instance of a generic, where the names of the parameter are modified to avoid potential namespace conflicts. Unlike with the informal English technique, the analyst will not need to perform the parameter substitution during analysis of the model.

For example, the package “SensorServer” is an instantiation of the generic “Server”:

```ada
package SensorServer is new Server
    (SensorSourceCode => LM.SensorCode);
```

89
The generic parameter "SensorSourceCode" is set to the type "SensorCode" belonging to the package "LM". The flattening process involves replacing all instances of "SensorSourceCode" with "LM_SensorCode" in the generic body for this instance of the generic.

In general, the relevant source code can be extracted from the packages and generics. As the packages are extracted the component names can be added to the subprogram and type names to avoid potential name conflicts. For example, when "SensorCode" is extracted from the package "LM", it is renamed "LM_SensorCode" in the model.

6.2.2 Filleting

The subprograms that are part of the critical code path are extracted from the source code, which includes their declarations, bodies and relevant data types. As before, removal of the package structure involves renaming the subprograms and types by prefixing the name with the component name to avoid potential name conflicts. Comments are inserted into the model that indicates the originating package or generic.

Subprograms that are not part of the critical code path are left out of the model. However, if the pruned subprogram is explicitly called within the critical code path, it is necessary to include the subprogram declarations, along with all the types found in the subprogram parameters, in order for the model to compile. For the model to properly execute, a stub subprogram is required for each of the pruned subprograms.

As part of the pruning, assumptions are made about the behaviour of the pruned subprograms. Stub subprograms are constructed to correspond with this assumed behaviour, which is potentially "null" (i.e., performs no action).

6.2.3 Partitioning

The software processes could potentially be modeled as Ada tasks or by extraction of the concurrency and synchronization mechanisms used in the original system. However, multiple flows of control can greatly complicate the model and make it more difficult to analyze.

If the concurrent software behaviour is not relevant for a given critical code path, then the entire model can be written as a single process. In that case, comments can still be inserted in the model to indicate which portions of the model correspond to source code that would execute in separate processes in the actual implementation.
6.2.4 Translating

The extracted source code is translated into Ada, which means removing the non-essential language features that obscure the critical code path and makes the model difficult to test or analyze. This includes the removal of generics and packages during the flattening stage. Concurrency mechanisms, such as Ada tasks, can be eliminated so that the model executes synchronously. Exception handling can also be removed or replaced, possibly by stub functions.

6.2.5 Example: Chemical Factory Management System

We constructed a safety concern model in Ada corresponding to the temperature hazard for Chemical Factory Management System. The complete model can be found in Appendix C.

The generic parameters were replaced with the actual parameters for a given instance of a generic, where the names of the parameter are modified to avoid potential namespace conflicts.

Subsequent calls to supporting services from the bottom two layers of the software architecture were pruned. For example, the subprogram “Send”, which was part of the Broadcast generic, and subprograms from the “Store” Generic were not extracted. Exception handling was implemented as a lower level service as opposed to using the Ada exception handling mechanism. This was also pruned.

For the Chemical Factory Management System, concurrency and synchronization mechanisms were implemented as a lower level services as opposed to using Ada tasks. The critical code path consisted of two main processes, along with the “MonitorSensorStaleness” thread, as displayed in Figure 5-11.

However, for this critical code path, the emphasis was on the corruption of the temperature value, so the concurrent behaviour was not relevant to the hazard and not modeled. Instead, comments were inserted into the model indicating which subprograms belonged to which process in the actual software.

Ada packages were also removed during the translation, resulting in a set of global data structures and subprograms. Many of the packages contained only a small amount of relevant code. The entire model was then placed into a single Ada package, with the data types and subprogram declarations in the specification, and the subprogram bodies in the package body. Figure 6-1 displays the skeleton of the package body. The complete package specification and package body for the model can be found in Appendix C.

6.2.6 Ada Approach: Summary and Discussion
We were able to extract an executable model of the safety concern in Ada for the Chemical Factory Management System. We now evaluate the effectiveness of the model.

\begin{verbatim}
package body hazard3model
is
  [Stubs for pruned subprograms]
  [Extracted subprograms with variables and types renamed]
end hazard3model;
\end{verbatim}

Figure 6-1: Skeleton of the package body for the Ada model

6.2.6.1 Tractable

With Ada as the choice of notation, it was possible to expand the generics and to remove the package structure. As a result, a large amount of non-executable source code was eliminated from the model.

As well, all the relevant source code was gathered into a single package corresponding to the three processes. Only the subprograms that lie along the critical code path were included in the package bodies. Having the critical code path in a single package greatly reduced the cognitive overhead in inspecting the source code.

6.2.6.2 Complete

For any pruned subprogram directly called in the critical code path the declarations were extracted in order for the model to compile. For the model to properly execute, it was also necessary to create stubs for the pruned programs, though in most instances, “null” subprograms were sufficient. The subprogram declarations and stubs act as placeholders to indicate what source code had been pruned to create the model.

6.2.6.3 Faithful

As the same programming language was used in the model as in the original system, the syntax of the source code model is obviously an accurate reflection of the actual source code.

One potential source of discrepancies between the model and the source code would be due to the renaming of the types when the package structure was removed. This renaming can be quite extensive, as typically many of the types and variables come from different packages and are referenced by the component name. As a result, this also increases the effort in creating the model. The trade-off is that by removing the package structure the model is much more tractable and easily analyzed.
In the cases where mechanisms are removed, like the tasking or exception mechanisms, there is the potential that the mechanism will not be accurately represented in the model. The model obviously is most effective for critical code paths where this behaviour is not essential, like in the corruption of the temperature value. In general, for the pruned subprograms, there is the potential for inaccuracy in the model with the construction of the stubs.

6.2.6.4 Analyzable
The greatest benefit of a model in a programming language is that it can be compiled and executed. The compiler checks acts as a consistency check on the model, while an executable model means it can be tested.

In terms of static analysis, inspecting the source code is simplified as all the relevant subprograms can be found in a single package body. As well, all the relevant data types are in the package specification.

6.2.6.5 Usable
Creating the model should not require anything more than knowledge of Ada. The key difficulty is the creation of stubs for the pruned subprograms, especially the replacement of any eliminated mechanisms. In the latter case, the Ada model might not be the most appropriate choice unless there is an effective method of replacing or including those mechanisms.

However, this is the same difficulty confronted by unit or integration testing. The only difference in this case is that complete subsystems or components are not being used, but fragments from each. However, stubs are required in each case. There is potential that the stubs and drivers created for the unit and integration testing can be leveraged here. In any case, it is again the criticality of the hazard that will determine what effort should be exerted in testing the safety concern model.

6.3 Formal Specification in S
In this section, a notation based upon higher order logic is used to represent the safety concern. With this notation, it is possible to mimic the syntax of the source code as closely as possible, including such elements as data types, classes, control structures and subprograms. This leads to a more readable, traceable model for developers and analysts. However, the more expressive aspects of the notation allow details of the code paths to be modeled as much as necessary. In particular, the pruned source code can be simply specified.
The major difference from using a programming language is the fact that a formal notation is not constrained to only describe functionality in executable terms. A more expressive notation is particularly useful when pruning because we can replace large chunks of code with a relatively simple declarative description that may not be executable, but specifies as much as necessary for the purpose of analyzing the concern.

A key aspect of this technique is the use of uninterpreted types and functions to represent data types and classes, simplify type hierarchies, and to denote simplifying assumptions about the code. In particular, uninterpreted types can be used to represent classes, while maintaining syntactic similarity with the code. Uninterpreted functions can be used to specify software functionality when the details of the functionality are not necessary to the safety analysis.

It is possible to perform analyses of the model with simple lightweight tools. For example, all formal specification notations have a formally defined syntax and many of these notations also have semantic validation rules analogous to the semantic validation rules of strongly typed programming languages. For some formal specification notations, there are lightweight tools called “type checkers” that automatically check whether a formal specification conforms to these rules. In the same way that a compiler helps a software developer avoid a variety of errors in the source code, the use of a type checker can be very beneficial as a means of avoiding errors in the construction of a safety concern model.

If additional analysis is required, the model can also be input into other tools, such as theorem provers. However, program verification, as exemplified by the use Hoare proof rules to prove the correctness of program [Hoa69], is not necessarily our motivation for suggesting the use of a formal notation. Though using a formal specification notation does open the door to the possibility of doing formal proofs, there are good reasons for using a formal specification notation even if the analyst does not have any interest or intention in performing a formal proof.

The formal specification notation used to specify the safety concern, “S”, is based on higher order logic. This notation was developed at the University of British Columbia to serve as a foundation for a variety of different approaches to formal specification [JDD94].

The specification approach and notation is described in greater detail in the following sections, including how each element of the Ada language can be specified in S. An overview is given of the results of applying this approach to an ATM safety concern. A summary and discussion follows. A portion of the resulting model in S can be found in Appendix D.
6.3.1 Specification Approach

The specification of the code model in S involves a lexical approach and the use of partial specifications. Day has applied these techniques to software specifications [Day98], while Donat has applied them to test case generation [Don98a].

The representation of the critical code paths follows as closely as possible the lexical style of the source code, including such elements as data types, control structures and subprograms. A lexical style leads to a more readable model for developers and other stakeholders already familiar with the source code or the programming language. This also supports traceability of the model, as it simplifies direct comparisons with the original code.

Details of the critical code path are only modeled as much as necessary. The basic code structure in terms of functions and procedures is maintained in the specification. Maintaining the code structure is useful as different developers are typically responsible for different parts of the code. Also, this simplifies the analysis if particular individual segments in the code are of interest.

The use of partial specifications is in contrast to many formal specification approaches that attempt to minimize ambiguity by building a model of the desired behaviour from basic elements of discrete mathematics. For example, this approach was used when specifying a library system in the Z notation [Spi88]. However, the result is that a large amount of specification infrastructure is required to fill the void between the level of abstraction required of the desired behaviour and the basic elements of discrete mathematics. In addition to the cost of producing this infrastructure, the effort and background knowledge required to wade through this infrastructure is an obstacle to anyone attempting to understand the formal specification. As a key motivation for creating the code model is to prune away irrelevant detail, it would be self-defeating to have instead a large amount of specification infrastructure that again obscures the details relevant to safety.

6.3.2 Flattening

Flattening involves expanding the generics and removing the package structure. As with the Ada specification technique, formal generic type parameters are replaced by the actual generic parameters, where the variables and types are renamed. For example, the defined “LM” object would be represented with the component name prefixed to the type name to avoid namespace conflicts, i.e., “LM_Object”.
In cases where there is a single instance of a generic, type abbreviations can be used to indicate the replacement of a generic parameter with the actual parameter. For example, the replacement of the actual parameter “LM.object” can be indicated by:

: Message_Set_Generic_Syntaxic_Format == LM_Object;

6.3.3 Filleting: Representing Pruned Source Code

Pruned source code can be specified as uninterpreted functions. If additional detail is required the function can subsequently be defined or an axiom provided about the expected behaviour.

Uninterpreted function can be used to represent arithmetic operations that do not need to be defined. For example, the following S declaration introduces a new uninterpreted function named “Log” that may be used as a placeholder in the S model for references to the (natural) logarithmic function in the Ada 83 source code:

Log : Float -> Float;

If the actual value of the logarithmic function is relevant to the safety concern, “Log” could be defined in terms of exponentiation (assuming that “Exp” has been previously introduced in the S model) as follows:

Log m := @n. Exp (n) = m.

The above definition may be paraphrased as “application of the function Log to a value m returns a number n such that Exp (n) is equal to m”. The constant “@” is a built-in constant of S. This constant is a higher order version of Hilbert’s ε-operator [GM93]. In general, an expression of the form “@n. P(n)” denotes a value that satisfies the predicate “P”, if any such value exists. The use of “@” in constructing safety concern models is often a convenient way of avoiding specifying extraneous detail.

6.3.4 Partitioning: Representing Concurrent Behaviour

The concurrent behaviour of the software is not relevant for the critical code paths under consideration in this section. Chapter 9 provides some additional details on how the processes could potentially be modeled in S in future research.

6.3.5 Translating: Representing Ada in S

The use of uninterpreted types and functions, as well as quantification, supports the lexical approach of representing the source code. For example, classes, along with their attributes and methods, can be modeled with uninterpreted types, while maintaining lexical similarity with the code.
S supports the use of uninterpreted types and functions. Other formal specifications, such as the Z notation also include provisions for uninterpreted types. However, Z does not provide for uninterpreted functions or, at least, they are not commonly used in Z specifications. In general, Z specifications are done quite differently then the approach taken here for creating the code model.

Joyce [Joy89] originally proposed using uninterpreted types and uninterpreted functions as a mechanism for describing abstract specifications. This mechanism was further explored by Day [Day98] in her approach to software specifications. The uninterpreted types and functions serve as a “placeholder” in the specification indicating a term that is not defined but is necessary for the specification. The use of uninterpreted functions allows the analyst to match the level of abstraction of the filleted source code and to avoid adding unnecessary details about the pruned source code.

With an analysis technique that supports the use of uninterpreted functions, we can carry out analysis before the specification is complete. Analysis results are then valid for any interpretation of the uninterpreted functions. If the safety analysis requires it, the details of the definitions of the uninterpreted functions can be added. As well, it is easy enough to check whether a specification includes any uninterpreted functions.

6.3.5.1 Data Types

Uninterpreted types are used to represent data values whose composition is not relevant to the formalization task. For instance, if the types “sensorID” and “temperature” appear in the source code, these can be represented as uninterpreted types by the following S type declarations:

```plaintext
: sensorID;
: temperature;
```

If a type corresponds to a class, it can still be represented as an uninterpreted type. For instance, a type named sensor is introduced by a type declaration,

```plaintext
: sensor;
```

to serve as the formal representation of the Sensor class.

An instance of a class is then represented as a constant of the corresponding type. For instance, an object “Sensor1”, which is an instance of the class “Sensor”, would be represented by,

```plaintext
Sensor1 : sensor;
```
In this style of S specification, attributes of a class are selectively introduced as selector functions. For example, if "SID" and "SensorTemperature" are attributes the "Sensor" class, they may be formally represented by a pair of S function declarations,

SID : sensor -> sensorID;
SensorTemperature : sensor -> temperature;

which declare "SID" and "SensorTemperature" to be functions that map values of type sensor to values of type "sensorID" and "temperature" respectively.

As means of achieving a degree of lexical correspondence with the source code, references to attributes of a class may be given using "dot notation". For example, the S expression

Sensor1.SensorTemperature

denotes the value of the "SensorTemperature" of the "Sensor" object denoted by "Sensor1". In S, dot notation is merely postfix application of a function to a value. For instance, "Sensor1.SensorTemperature" denotes the application of the function "SensorTemperature" to "Sensor1". It is semantically equivalent to "SensorTemperature (Sensor1)", i.e., prefix application of a function to a value.

S type abbreviations can be used to represent derived types. For example, if the "SensorID" type has been derived from the type "float13", this can be represented by the type abbreviation,

:sensorID == float13;

If the "SensorID" is a subtype, with additional constraints on the parent type, lemmas can be introduced to state the restrictions. However, these are only introduced as necessary.

S type abbreviations can also be used to help represent arrays. For example, the type abbreviation

: broadcastMessage == SensorRange --> sensor;

can be used to introduce "broadcastMessage" as a name for a function type that maps an array index to a sensor. As indicated by the above type abbreviation, the value of the array index is constrained to be a value of type "sensorRange". The formal representation of an array in this manner allows array references to be expressed in a manner that corresponds lexically to the array references in the source code. For example, the S expression,

Message (I).SensorTemperature
denotes the value of the “SensorTemperature” attribute of the Ith element of the array denoted by “Message”. In this example, “Message” is either a variable or constant of type “broadcastMessage” and “I” is a variable or constant of type “sensorRange”.

6.3.5.2 Control Structures

If statements can be straightforwardly represented in S:

```plaintext
if (NOT(Success))
  then (Basic_Data_Is_Valid = False)
  else (Basic_Data_Is_Valid = True))
```

Some “for” loops can be modeled with the help of quantification. For example, the following Ada “for” loop iterates through every value in the range of an array index:

```plaintext
for Index in Sensor_Array'Range loop
```

where “Sensor_Array'Range” gives the index range for the array “Sensor_Array”.

In S, the “for” loop can be represented with the quantifier “forall”:

```plaintext
forall Index. In_Sensor_Array_Range(Index)
```

where we have defined the predicate “In_Sensor_Array_Range” to indicate whether “Index” is within the index range of the array “Sensor_Array”.

6.3.5.3 Local Variables and Assignment Statements

An important difference between a formal specification notation such as S and a programming language such as Ada is that S does not include any concept of a state that can change dynamically as a result of executing assignment statements. However, we can still model this by means of S “let” expressions.

For example, the following Ada code involves the declaration and initialization of variable “Con”:

```plaintext
Con : Syst.Float13 := Eccentricity * Sin (Phi);
```

In S, we can define a constant “Con” through a “let” statement that represents the initial value of the Ada variable “Con”:

```plaintext
let Con := Eccentricity * Sin(Phi) in
(S statements)
```

In Ada, an assignment statement assigns a new value to a variable:
Con := ((1.0 - Con) / (1.0 + Con)) ** (Eccentricity / 2.0);

In S, the assignment statement can also be represented by a let statement:

\[
\text{let } C := \text{Eccentricity} * \sin(\Phi) \text{ in} \\
\text{(let } Con := ((1.0 - C)/(1.0 + C)) ** (Eccentricity / 2.0) \text{ in} \\
\text{(S statements)})
\]

A different S constant “C” was introduced to differentiate the initial value of the Ada variable “Con” from its subsequent value.

### 6.3.5.4 Subprograms

Subprograms are represented by functions. For example, consider the Ada function “Small”, with initialized local variable “Con”:

```ada
function Small(Phi : in Angl.Latitude) return Syst.Float13 is
    Con : Syst.Float13 := Eccentricity * Sin (Phi);
begin
    Con := ((1.0 - Con) / (1.0 + Con)) ** (Eccentricity / 2.0);
    return Tan ((90.0 - Phi) / 2.0) / Con;
end Small;
```

This subprogram can be specified in S as follows:

```s
Small (Phi:Latitude):=
    let C := Eccentricity * Sin(Phi) in
    (let Con := ((1.0 - C)/(1.0 + C)) ** (Eccentricity / 2.0) in
    Tan((90.0 - Phi)/2.0)/Con);
```

### 6.3.6 Example: Air Traffic Management System

A safety concern model using S was constructed for the ATM system. A portion of the resulting model is presented in Appendix D.

The ATM safety concern is similar to the one in Section 6.1, when we applied the informal English technique. As a result, the same filleted source code was used as the starting point. As before, approximately 1-2 k source lines of code was extracted from the software implementation.

The source code model was separated into three files as follows, along with the file sizes (which includes blank lines):

- Basic data types, including those from the lower layers (493 lines)
- Supporting transformations, including data types and subprograms, from the bottom layer (1122 lines)
- Core subprograms (473 lines)
For each specified subprogram and data type, their name as it appears in the source code, along with the originating package or generic, was noted in an S comment. Additional comments were added to the start of each file to provide an overview of the contents.

Once the mapping of Ada elements to S elements was understood, it was a fairly straightforward process to perform the translation. Figure 6-2 shows the actual source code for a procedure, "Compute_Stereographic_Xy". This procedure is part of the critical code path for a hazard concerned with the display of the aircraft position. Figure 6-3 shows the result of translating the source code for "Compute_Stereographic_Xy" into S.

```
procedure Compute_Stereographic_Xy
(The_Lat_Long : in Geot.Latitude_Longitude;
With_Projection : in Stereographic_Object;
The_Xy : out Proj.Xy_Coordinate_In_Metres;
Is_Valid : out Boolean) is
  Sin_Phi : constant Syst.Float13 :=
    Syst.Elementary_Functions13.Sin
    (Syst.Float13 (The_Lat_Long.The_Latitude) * 
     Unit.From_Degrees_ToRadians);
  Cos_Phi : constant Syst.Float13 :=
    Sqrt (1.0 - Sin_Phi * Sin_Phi);
  Cos_Long : constant Syst.Float13 :=
    Syst.Elementary_Functions13.Cos
    (Syst.Float13 (The_Lat_Long.The_Longitude) - 
     Syst.Float13 (With_Projection.Tangency_Long)) * 
     Unit.From_Degrees_ToRadians);
begin
  declare
    Rk : constant Syst.Float13 :=
      Syst.Float13 (With_Projection.Radius) * 2.0 /
      (1.0 + With_Projection.Sin_Phi * Sin_Phi +
       With_Projection.Cos_Phi * Cos_Phi * Cos_Long);
  Is_Valid := True;
  The_Xy := (X =>
    Unit.Metres13
    (Rk * Cos_Phi * 
     Syst.Elementary_Functions13.Sin
     (Syst.Float13 (The_Lat_Long.The_Longitude) - 
      Syst.Float13 (With_Projection.Tangency_Long)) * 
      Unit.From_Degrees_ToRadians)),
    Y => Unit.Metres13
    (Rk * (With_Projection.Cos_Phi * Sin_Phi -
       With_Projection.Sin_Phi * 
       Cos_Phi * Cos_Long)));
exception
  when Constraint_Error =>$
    Is_Valid := False;
    The_Xy := Proj.Nil_Xy_Coordinate_In_Metres;
end Compute_Stereographic_Xy;
```

Figure 6-2: Subprogram "Compute_Stereographic_Xy"

The greatest challenge to creating the model in S was deciding what source code to prune and determining how to represent the pruned source code in the model. For example, not only were subprograms pruned, but also sections of source code within a subprogram body that were determined to be outside of the critical code path.
For this critical code path, the concurrent behaviour of the source code was not relevant. As a result, processes and their interactions were not modeled. The model involved a synchronous sequence of steps.

The final model was then run through a type checker, for S notation. This tool, called “Fuss” [JDD94], was effective in catching inconsistencies and missing items in the model. However, it was also a challenge at times to properly capture the nuances of the S syntax in order to make the model acceptable to Fuss. For example, some effort was required to place the brackets correctly in a subprogram specification so that the order of precedence was understood by Fuss.

```
Compute_Stereographic_Xy(
    The_Lat_Long:Latitude_Longitude,
    With_Projection:Stereographic_Object,
    The_Xy:Xy_Coordinate_In_Metres,
    Is_Valid:Boolean) :=

    (let Sin_Phi := Elementary_Functions13_Sin
       (The_Lat_Long.The_Latitude * From_Degrees_To_Radians) in
        (let Cos_Phi := Sqrt(1.0 - Sin_Phi * Sin_Phi) in
            (let Cos_Long :=
                Elementary_Functions13_Cos
                (The_Lat_Long.The_Longitude - With_Projection.Tangency_Long)
                * From_Degrees_To_Radians) in
                (let Rk := (With_Projection.Radius * 2.0)/
                (1.0 + With_Projection.Sin_Phi * Sin_Phi +
                With_Projection.Cos_Phi * Cos_Phi * Cos_Long) in
                    forall (Constraint_Error:Exception).
                    if(NOT(Constraint_Error))
                    then(
                        (The_Xy.X = Rk * Cos_Phi *
                        Elementary_Functions13_Sin
                        (The_Lat_Long.The_Longitude - With_Projection.Tangency_Long) *
                        From_Degrees_To_Radians)) AND
                        (The_Xy.Y = Rk * (With_Projection.Cos_Phi * Sin_Phi -
                        With_Projection.Sin_Phi * Cos_Phi * Cos_Long)) AND
                        (Is_Valid = True))
                    else(
                        (The_Xy = Nil_Xy_Coordinate_In_Metres) AND
                        (Is_Valid = False))
                    )
    ));

Figure 6-3: Subprogram “Compute_Stereographic_Xy” translated into S
```

There were surprisingly few type conflicts when the model was checked by Fuss. The source code was written so that package names and renaming was done consistently and uniquely. As a result, the technique of pre-fixing type and variable names with the package alias was very effective. For Ada source code that does not follow good coding conventions, the flattening would likely have been more problematic.
6.3.7  S Approach: Summary and Discussion

The use of uninterpreted types and functions provide the key support for this approach. It is possible to achieve a large degree of lexical similarity with the source code syntax for a formal mathematical notation, i.e., a developer looking at the model should have little difficulty reading it and comparing it to the original code.

Translating the source code into S meant it was necessary to identify precisely the critical data types and the operations upon them. The type checker Fuss also provided a useful high-level consistency check on the model. During the translation process, a number of issues were questions, though none that were safety-related.

6.3.7.1 Tractable

With the S approach, it is possible to achieve a code model with “Ada-like” syntax. However, compared to the informal English approach, the model is much more clear, precise and concise. As well, as with the Ada technique, it is possible to expand the generics, so there was no cognitive overhead in remembering the actual generic parameters.

6.3.7.2 Complete

With S, pruned subprograms can be represented by uninterpreted functions. We can specify the function as much as necessary with assertions expressed in S that partially constrain the meaning of the uninterpreted function. As a result, we can avoid having to totally define an operation corresponding to an Ada subprogram while expressing what is required of the function for safety.

As well, the type checking performed by Fuss provides a useful consistency check on the final model. However, type checking does not ensure that all the necessary assumptions have been captured.

6.3.7.3 Faithful

It is possible to achieve a certain amount of lexical similarity between the S model and the source code, so that by inspection a certain amount of confidence can be achieved that the model does accurately reflect the source code.

However, there is always the potential that the corresponding line of S does not accurately reflect the source line of code. As well, there is the potential that that the pruned source code is not adequately specified.
6.3.7.4 Analyzable

The S notation has the support of various tools. The type checker Fuss can be used and provides a high-level consistency check on the model. However, as noted in Chapter 5, simply translating the safety concern source code into a different notation forces a close inspection of the safety concern source code. The translation helps create a greater understanding of the critical-code path.

There are potentially other tools that could be use to support the analysis including, for example, the symbolic functional evaluator, SFE [DJ99], and HOL, the theorem prover [GM93]. With these tools, more general properties about the model could potentially be verified. Future research involving the analysis of a safety concern model in S is described further in Chapter 9.

6.3.7.5 Usable

Though there was an initial learning curve, it became a fairly mechanical process to perform the translation once the Ada to S mapping was worked out. The most challenging aspects were specifying the pruned source code and writing the model in a form that was acceptable to the type checker.

Training would also be required for use of the S notation and the Fuss tool. In addition, if a theorem prover such as HOL is used to verify properties about the model, this would also require significant expertise and training. However, the use of a theorem prover is not the primary purpose of using the S notation and so is not necessary in order to gain benefit from creation of the code model.

6.4 Summary

In this Chapter, safety concern models were created with informal English, Ada, and S as the choice of notations. In each case, a coherent, simplified model of the critical code path resulted that was suitable for use for safety verification. In particular, the safety concern source code that was scattered over many components was gathered into one place, with a large amount of the non-executable source code removed. Examples of the results are presented in the appendices.

With the S notation, it was possible to achieve a code model with “Ada-like” syntax. Of course, with the Ada and the informal English notations, Ada syntax was actually used.

With the informal English approach, it was possible to place the generic instantiations close to the extracted generic source code for easy reference. With the Ada and S notations, it was possible to expand the generics, so there was no cognitive overhead at all in remembering the actual generic parameters for a particular instance of the generic.
The specification of pruned source code was very effective with the S approach compared to the Ada or informal English approach. For S, pruned subprograms were represented as uninterpreted types and functions, which could then be further refined if necessary. Procedure stubs were required for Ada.

S has the support of various tools for analysis. The type checker Fuss was used and provided a high-level consistency check on the model. Other tools include, for example, the symbolic functional evaluator, SFE [DJ99], and HOL, the theorem prover [GM93]. For the Ada technique, the model can be compiled and executed, allowing for testing of the model. The informal English technique does not have any tool support for analysis.

Compared to the informal English, a great deal more effort was required to create the specification in Ada and S. In particular, in most situations training would be required in the use of the S notation. The informal English and Ada approaches should not require any additional skill sets.

The notations, ranging from informal English to Ada to S, provide a range of techniques of increasing expressiveness, for specifying the pruned code, and analyzability, in terms of tool support. However, there is a trade-off between the expressiveness of the notation and the skill level effort required to use the notation. There is also increased likelihood for a “semantic gap” between the source code and the model syntax, though the increased tool support could potentially help mitigate this by detecting inconsistencies in the model. Ultimately, the appropriateness of a given technique will depend on practical considerations such as the amount of time and effort available, and the criticality of the hazard.
7. Extending the Safety Concern Model

In this chapter, we consider the possibility of enhancing a safety concern model to more directly incorporate knowledge of a particular hazard. In particular, we outline a process whereby a safety-related hazard may be systematically refined into source code level Safety Verification Conditions (SVCs) that supplement the code model.

The safety concern models constructed according to the process outlined in Chapter 4 already indirectly incorporates knowledge of a particular hazard. The models are the cumulative result of a series of decisions made by a safety engineer to distinguish what details of the source code should be copied into the model and what details can be excluded. These decisions are made on the basis of the engineer’s understanding of the hazard. Therefore, the resulting model is a reflection of the engineer’s knowledge of the hazard.

However, the safety concern model must ultimately be explicitly verified with respect to the hazard. In particular, Chapter 2 described a process for safety verification where the safety concern model is verified with respect to a set of SVCs. As shown in Figure 2-5 of Chapter 2, the left-hand side of the diagram involved the problem of refining system hazards into small granularity SVCs. A process for refining the hazard into source code level SVCs was proposed by the author in his M.Sc. dissertation [Won98a].

The purpose of this chapter is to suggest how detailed knowledge of a hazard can be explicitly represented in a safety concern model. In particular, we demonstrate two different approaches for the explicit representation of a source code level SVCs in a safety concern model. One approach involves creating code assertions from the SVCs using a formal description language called FDL, which is part of the SPARK methodology. This approach would allow for the use of verification tools based on formal logic to verify that the SVCs are satisfied by the representation of the source code in the safety concern model. The other approach involves the use of the S notation to represent SVCs. The use of S results in a more precise specification of the relevant software elements in the SVCs that should lead to a more effective static analysis of the source code.

The generation of code assertions is motivated by 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 book 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. For example, use of SPARK Examiner would require a code model and a set of verifiable code assertions expressed in the SPARK programming language.

The goal of expressing the SVCs in a codified form is to obtain explicit and precise specification of the relevant software components and attributes in the system level SVCs. The codified SVCs would hopefully lead to a more effective static verification of the source code. In fact, the act of codifying the SVCs in itself can be considered a form of static analysis as it involves a close inspection of the safety-critical code. One method of codifying the SVCs is specifying them in the notation S. With this notation, as was shown in Chapter 6, the relevant object-oriented features are captured in a simple way by representing types with classes and functions with class attributes.

In this chapter, the process for creating source code level SVCs is reviewed and then applied to the temperature hazard for the Chemical Factory Management System. A set of verifiable code assertions is then created using SPARK. As well, the codification of the source code level SVCs in S is described for the ATM system. This is followed by a summary and some conclusions.

### 7.1 Process for Creating Source Code Level SVCs

As detailed previously by the author [Won98a], source code level SVCs can be derived from a stepwise process that involves the development of system level, design level and source code level SVCs.

The inputs of the process are:

- **System hazards.**
- **Software architecture.**
- **Hazard-related source code.**

The output of the process is a combination of SVCs including:

- constraints on variable system parameters or constants - typically expressed as mathematical inequalities, e.g., "X must be refreshed at a greater rate than Y";
- functional correctness conditions on relatively small blocks of source code which lie directly in the critical code path - typically, a pre- and post-condition combination of assertions;
- partial specifications of "peripheral" aspects of the source limited to the minimal assumptions required to carry out safety verification - for example, a limit on the range of the value of the output of a subsystem which does not lie directly in the critical code path.

The steps of the refinement process are:

1. **Generate system level SVCs.** The system is analyzed with respect to the hazard to produce the system level SVCs. The SVCs are a set of system constraints that are sufficient to ensure that the identified hazards do not occur. The system level SVCs are generated from the use of a style of reasoning known as “proof by contradiction” to construct a rigorous safety argument.

2. **Generate design level SVCs.** The system level SVCs are refined into design level SVCs through analysis of the software architecture. The hazard-related software is partitioned into separate processes, and the system level SVCs are mapped into conditions on the process input and output parameters.

3. **Generate source code level SVCs.** The design level SVCs are refined into source code level SVCs through analysis of the safety concern model. The process parameters are identified in the safety concern, and the design level SVCs are re-written in terms of the identified parameters.

At this point, the source code SVCs can be used as an input to a code verification process. One possibility is conventional software testing.

### 7.2 Example: Chemical Factory Management System

The process for deriving source code level SVCs is applied to the temperature hazard for the Chemical Factory Management System.

#### 7.2.1 System Level SVCs

The system level SVCs are a set of constraints on the system that are designed to ensure that the system hazards do not occur. The system level SVCs are generated in support of a rigorous argument about the system safety [JW98]. The approach is similar to the concept of FTA in the sense that that it begins with an assumption that the hazardous condition has occurred and then work "backwards" to systematically cover all of the possible ways in which this condition might have arisen. Like Software Fault Tree Analysis (SFTA) [LCS91], a style of reasoning known as “proof by contradiction” is used to show that each disjunctive branch of the argument leads to a logical contradiction.
The argument involves first assuming that the hazardous condition exists. The analysis then proceeds in a stepwise manner by attempting to show that this assumption leads to a logical contradiction. The analysis of the hazard then branches 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. 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. Intuitively, the conditions are constraints on the system that are necessary to avoid the hazard.

The results of such an analysis for chemical factory hazard are five distinct system level SVCs [JW98]. The following is an example of one of the five system level SVCs:

<table>
<thead>
<tr>
<th>Safety Verification Condition:</th>
</tr>
</thead>
<tbody>
<tr>
<td>For all vessels, v, displayed temperatures, d, and times, t,</td>
</tr>
<tr>
<td>if the displayed temperature of vessel v is set to d at time t,</td>
</tr>
<tr>
<td>then at some time no earlier than MAX_SYSTEM_PROPAGATION_TIME milliseconds before t the system received a report from the external sensor monitoring system that the temperature of vessel v is d.</td>
</tr>
</tbody>
</table>

Figure 7-1: System Level SVC

MAX_SYSTEM_PROPAGATION_TIME is a constant that specifies the maximum time the system should take to display a vessel temperature on the screen after receiving a sensor update for that vessel.

7.2.2 Design Level SVCs

The system level SVCs are refined into SVCs for each software process (or thread) displayed in Figure 5-11. An important aspect of the refinement is the separation of system functionality and timing issues, into separate process SVCs. This allows for the derivation of process pre- and post-conditions from the time-independent process SVCs.

The system level SVC introduced in the previous section applies to the propagation of the temperature through the system, which is carried out by the "LANToBroadcast" and "BroadcastToDisplay" processes, with a Broadcast mechanism providing the process communication. As a result, the system level SVC can be refined into design level SVCs involving these relevant processes and the Broadcast mechanism, by extending the rigorous argument.

When performing the refinement, the processes can be viewed conceptually as a procedure with input and output parameters. The system level SVCs are then refined into separate functional and timing conditions, with the system events mapped onto the process invocations and output. Furthermore, the
process becomes the causal agent, i.e., it takes the input and creates the output, which replaces the timing sequence.

An example of a design level SVC for the “LANToBroadcast” process is:

**LANToBroadcast design level SVC:**
For all vessels, v, and broadcast temperatures, d,
if there exists the information that vessel v is d in the output broadcast message, M,
then there exists the information that the temperature of vessel v is d, in the input LANMessage, L.

Figure 7-2: Design level SVC for LANToBroadcast process

7.2.3 Source Code Level SVCs

The design level SVCs are refined into source code level SVCs, by identifying the process input and output parameters, along with any other relevant source code element from the safety concern model. In this section we provide a sense of the need for a safety engineer to "dig into" the source code in order to close the semantic gap between the definition of the hazard and the corresponding representation of the SVC in terms of the source code. The source code level SVCs then provide a direct link between the hazard and the safety concern model.

7.2.3.1 Process Parameters

The input and output parameters of the process are determined from examination of the subprograms that make up the process. For example, the main thread of functionality for the “LANToBroadcast” process is contained in the “ReadLAN” and “ProcessSensor” procedures. Invocation of the “ReadLAN” procedure provides entry into the process with a “LANMessage Object” as input. The “ReadLAN” procedure then invokes the “ProcessSensor” procedure, which broadcasts a “SensorServer BroadcastMessage” as output.

In addition to the input and output parameters for the “ReadLAN” and “ProcessSensor”, these procedures have access to package level variables which maintain state information for that package. For example, the “SensorServer” makes use of the “SensorStore Object” to maintain a set of sensor readings that are accessed by the “ProcessSensor” procedure. These are considered “global variables” of the process.

7.2.3.2 The Source Code Elements

The design level SVC refers to input LAN messages and output broadcast messages, which contains the vessel’s temperature obtained by the external sensor monitoring system.
Examination of the “LANMessage Object” in the safety concern model reveals that updates from the different sensors are maintained in arrays of “SensorUpdate” data records:

```plaintext
type LANMessage_SensorUpdate is
  record
    InterpolatedState : LANMessage_InterpolationRange;
    TemperatureEstab : LANMessage_Temperature_T;
    SensorCodeEstab : LANMessage_SensorCode;
  end record;
```

The “TemperatureEstab” field maintains the sensor temperature reading, and “SensorCodeEstab” field maintains the raw sensor code.

Examination of the “Broadcast Object” reveals that the updates are maintained in an array of Sensor Objects:

```plaintext
type Sensor_Object is
  record
    SID : Sensor_SensorID;
    SensorOperation : Sensor_Operation;
    SensorQuality : Sensor_Quality;
    SensorTemperature : Sensor_Temperature;
  end record;
```

The raw sensor code has been converted and stored as a “SensorID”. The raw temperature reading has been converted and stored as the “SensorTemperature”.

The design level SVC can now be re-written in terms of these model elements:

```plaintext
LANToBroadcast Source Code Level SVC:
For all Sensor Objects, s, in output BroadcastMessage, M,
if Sensor Object, s, contains the information SID, C, and SensorTemperature, D,
then there exists a SensorUpdate, U, in input LANMessage, L, with the information
SensorCodeEstab, C1, which is converted from SensorID, C, and
TemperatureEstab, D1, which is converted from D.
```

![Figure 7-3: Source code level SVC for LANToBroadcast process](image)

**7.3 Creating Verifiable Code Assertions**

The source code level SVCs can be used during safety verification process. For example, they could be used in support of code inspection or software testing. However, we now 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 a purely mathematical task of verifying a logical expression called a "verification condition". A code verification tool such as SPARK Examiner relieves the human analyst of the task of tracing through the code slice statement-by-statement. To complete the overall process, the verification
conditions must then be verified either by manual efforts or by use of tools such as the Simplifier and the Proof Checker [Bar97].

To prepare the code model for SPARK Examiner, the model must be first translated into the SPARK language. This includes the addition of code assertions, e.g., program pre- and post conditions that are derived from the SVCs.

7.3.1 SPARK

SPARK is a high-level programming language intended for high integrity applications. It has been used in a variety of application areas including avionics and railroad signalling. The SPARK language consists of a kernel that is a subset of Ada along with annotations that appear in the code as comments [Bar97].

There are two types of SPARK annotations, core and proof. The core annotations are used to support data flow analysis, while the proof annotations take the form of code assertions that support the formal verification of the software.

The SPARK kernel omits the following Ada features:

- Ada tasks, exceptions, generics
- Access types, goto statements, use package clauses

These features all create difficulties in proving that a program is correct.

There are three main steps in verifying the correctness of a SPARK program:

1. The verification engineer inserts proof annotations in the source code expressed in the form of assertions. These assertions may, for example, be pre- and post-conditions.
2. The annotated source code is processed by SPARK Examiner, a verification condition generator. If the source code has been properly annotated (e.g., assertions have been inserted where required), the verification condition generator will generate a set of verification conditions. These verification conditions are assertions of formal logic expressed in the Functional Description Language (FDL).
3. To complete the verification process, all of the verification conditions must be shown to be true. In many cases, these verification conditions may be relatively simple mathematical assertions and can be verified with little or no human assistance using automated theorem-proving.
techniques supported by the Simplifier and Proof Checker tools. Other verification conditions will be more problematic and will require human intervention.

Thus, the main role for human expertise in this process is the first step, i.e., annotating the source code with assertions. The second step is performed entirely by software and much of the third step benefits from automation.

7.3.2 Verifiable Code Assertions

The source code level SVC can be expressed as a SPARK annotation and incorporated into the safety concern model, assuming the safety concern model has been represented in SPARK. SPARK annotations make use of the SPARK Ada subset and appear in the code as comments. However, the SVC involves quantification ("for all"), which is not supported by the SPARK Ada subset, so it is not possible to directly express the SVC in a SPARK annotation. Instead, the SVC can be expressed in terms of "proof function" that is then separately defined in FDL, the same notation used for the generated verification conditions.

The SVC corresponds to a post-condition for the subprogram "SensorInterface_ReadLAN".

```plaintext
procedure SensorInterface_ReadLAN(Message : in LANMessage_Object);
  --# post ConvertAll(Message, BroadcastMessage,
  LANMessage_SensorUpdateRange'First,
  LANMessage_SensorUpdateRange'Last);
```

Figure 7-4: SPARK annotation

As shown in Figure 7-4, the post-condition can be incorporated as a comment in the model, i.e., prefaced by "--# post", where "#" indicates to SPARK Examiner that the comment is a SPARK annotation, while the keyword "post" indicates the annotation is a post-condition for the procedure. "ConvertAll" is a proof function that is used as a placeholder for the SVC.

The declaration of some proof functions is shown in Figure 7-4. Some proof functions such as "ConvertAll" are predicates, i.e., functions that return true or false, depending on the inputs. The use of a proof function as an annotation in a SPARK program serves as an assertion about the state of execution in terms of the inputs to the proof function. Other proof functions have an auxiliary role in helping to define other proof functions.

In particular, we have introduced the proof functions "ConvertSensorCode" and "ConvertTemperature" to help with later definition of "ConvertAll". "ConvertSensorCode" represents the conversion of a
“SensorCode” value into a “SensorID”, while “ConvertTemperature” represents the conversion of a “Temperature_T” value into a “Sensor_Temperature”.

A proof function declaration has the same syntax as an Ada function but does not need to be defined in the source code. The proof function acts like a placeholder, similar to an uninterpreted function in S. As shown in Figure 7-5, the proof function “ConvertAll” is declared in an Ada comment, where L is a “LANMessage_Object”, M is a “SensorServer_BroadcastMessage”, and “J” and “K” are the array index ranges for the “LANMessage_Object”.

As mentioned in Section 7.3.1, the output of the SPARK Examiner tool is verification conditions expressed in the FDL language, which, unlike the SPARK Ada subset, supports quantification. Proof functions can be defined in FDL and, for example, be input into the Proof Checker tool along with the generated verification conditions.

As a result, it is potentially useful to express the SVC in FDL. The source code level SVC shown in Figure 7-3 can be represented in FDL as follows:

```plaintext
--# function ConvertAll(L:LANMessage_Object; M:SensorServer_BroadcastMessage; J,K:LANMessage_SensorUpdateRange) return Boolean;
--# function ConvertSensorCode(SC : LANMessage_SensorCode) return Sensor_SensorID;
--# function ConvertTemperature(T : LANMessage_Temperature_T) return Sensor_Temperature;

Figure 7-5: Proof functions

ConvertAll(M, L, J K) may_be_replaced_by
for_all(m:integer, (J <= m and m <= K) and
for_some(l:integer,
let C = fid SensorTemperature(element(M, [m])) in
let D = fld_SID(element(M, [m])) in
(C = ConvertTemperature(fld_TemperatureEstab(element(L,[l])))) and
(D = ConvertSensorCode(fld_SensorCodeEstab(element(L,[l])))))

Figure 7-6: Proof function “ConvertAll”

where for FDL:

- “for_all” is the universal quantifier and “for_some” is the existential quantifier (with lower-case indicating bound variables), e.g.,
  - “for_all(m:integer, …)” means for all integers “m” such that …
  - “for_some(l:integer, …)” means there exists an integer “l” such that …
• "element" represents an array element, e.g., the m'th element of the "BroadcastMessage" array "M" is given by "element(M, [m])", which is a "SensorObject".
• "fld_<field name>" represents a record field, e.g., the "SensorTemperature" field of a "SensorObject" is given by "fld_SensorTemperature(element(M, [m]))"
• We have taken the liberty of using "let" expressions to improve the readability of the expression. This does not violate the integrity of the approach as they could, for example, be easily handled by a preprocessor that performs the appropriate textual substitutions

With the SPARK annotation and defined proof function, we have a direct representation of the hazard in the safety concern model.

7.3.3 Summary and Conclusions
There are challenges to using code verification tools such as SPARK Examiner as part of the safety verification of a large software system. One is the issue of cross-cutting concerns, which is addressed by the creation of safety concern models. Another 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. The process outlined in this chapter addresses this particular problem.

This has led us to conclude that code verifications tools such as SPARK Examiner may indeed be useful in the safety verification of a large software system. Even if such tools were 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.

7.4 Formal Specification of Source Code Level SVCs
The source code level SVCs can be translated into a codified form to support static analysis of the source code. For a system with an object-oriented software architecture, the choice of representation must be sufficiently expressive to represent the object-oriented features that are captured in the source code level SVCs.

7.4.1 Codified Form
A codified form involves a choice of representation with clearly defined rules of expression, along with validation checks that can be performed algorithmically. Though this could involve a formal syntax, the expressions could also be non-textual, such as data flow or object-collaboration diagrams.
There are a number of motivations for having a codified form of the source code level SVCs. For example, they would then be less subject to personal style and more amenable to systematic review by peers. Furthermore, if a machine-readable notation were used, there would be the potential for tool-based support.

As with the safety concern model, a formal notation based on a typed predicate logic, such as S, is adequate for specifying the conditions.

### 7.4.2 Chemical Factory Management System SVC

The formalized source code level SVC in S is given in its entirety as,

```plaintext
forall (M:broadcastMessage, l:sensorRange).
(let D := M(l).SensorTemperature in
(let C := M(l).SID in
if(M.isBroadcast)
then (exists (L:lanMessage, J:sensorUpdateRange).
  (L.isReceivedLanMessage)
  AND (D is_converted_temperature_of
       ((L.Updates)(J).TemperatureEstab))
  AND (C is_converted_sensor_code_of
```

**Figure 7-7:** Formalized source code level SVC for LANToBroadcast process

### 7.4.3 ATM System

For the safety concern model created in informal English for the ATM System, we constructed codified source code level SVCs. The source code level SVCs were first generated according to the hazard refinement process described in this chapter. The SVCs were then specified in S.

Specifying the source code level SVCs in S provided a focus for the examination of the source code. It was necessary to clearly and precisely identify the critical data types and the operations upon them. As only the functionality that impacted on the hazard was of interest, this also meant a precise specification of the safety-critical functionality of the relevant source code.

### 7.4.4 Summary and Conclusions

The source code level SVCs for a Chemical Factory Management System was codified in the S notation. Using a formal notation to represent the relevant object states resulted in a careful and disciplined examination of the source code. Writing down the states in a precise manner was a useful aid in identifying the relevant object attributes and relationships. In particular, identifying the initial and final object states led to questions regarding the relationship between them, and helped to uncover intermediate operations.
7.5 Summary and Conclusions

A method was presented in this chapter for the informal systematic refinement of system hazards into source code level SVCs. The SVCs were expressed in terms of elements of the safety concern model, providing a direct link between the safety concern model and the hazard. The SVCs were converted into code assertions in the form of SPARK annotations. Alternatively, the SVCs were codified in S.

The code assertions are the key program safety invariants, which can then be verified by means such as inspection, testing or code verification. Such a refinement of the hazard allows a tool intended mainly for "correctness verification", such as SPARK Examiner, to be used for “safety verification”.

Though the formulation of each step of the refinement involves informal arguments and statements of the resulting conditions, there was value in partially formalizing some of the steps. For example, formalization of the source code level SVCs contributed to the care and precision in which the source code elements are identified, and helped ensure that the contrapositive form of the condition is correctly obtained.

The source code level SVCs (and their codified form) are not merely superficial re-formulations of the hazards. If the source level SVCs were used for a manual verification process, then we expect the analyst would be at a much greater advantage than if he/she attempted to perform the safety verification by inspecting the code directly in terms of the system level definition of the hazard.
8. Related Work

This chapter describes work related to modeling safety concern source code for safety verification. In general, there is little in the literature on the impact of cross-cutting concerns on safety verification or on the modeling of safety-critical source code. An overview of safety verification techniques is provided, followed by a review of the approaches explicitly focused on identifying and documenting concerns in source code.

8.1 Safety Verification Techniques

Safety verification involves either:

- **Dynamic analysis** that involves execution of the code. This includes testing and simulation.
- **Static analysis** that involves examination of the software without executing it. This includes informal techniques, such as code inspections, and formal techniques, such as analysis with code verification tools.

A review paper by Scott and Lawrence [SL94], as well as Chapter 12 in Storey’s text on safety-critical computer systems [Sto96], provide an overview of the standard software verification techniques that might be applied to safety verification.

8.1.1 Dynamic Analysis

Standard software testing techniques can be employed in support of safety verification, such as structural (white box) testing, functional (black box) testing, statistical testing, stress testing and regression testing [SL94]. In particular, safety testing is often viewed as verifying safety requirements [IEC95]. However, Joyce and the author have previously argued that requirements-based testing alone is not sufficient for safety testing and a hazard-driven approach is required [JW03].

In addition to standard testing techniques, additional hazards and hazard causes could potentially be identified through software fault-injection [VC+97]. In particular, Voas et al. have noted that fault injection is useful for investigating the effects of anomalous events caused by human factor errors or external failures. The events can be simulated by either modifying the code or forcing the state of the code to be modified when the code executes.
In some cases, it is difficult to determine if the resulting software output is “correct” or not. Knight et al. have addressed this problem by using “reversal checks” that takes the output of a computation, calculates the expected input, and compares the regenerated input to the actual input [KC+94].

It may be possible to exhaustively test certain limited safety properties. For example, Knight et al. described how a safety property involving an object location algorithm used in the imaging subsystem of a Magnet Stereotaxis System (MSS) was verified by exhaustive testing [KWW96].

In general, it will not be possible to verify safety through testing alone. This is the fundamental limitation of software testing, as noted in Dijkstra’s famous aphorism “testing can only reveal the presence of errors, never the absence” [Dij69]. The large number of possible software inputs and paths means it is not feasible to exhaustively test them all [PVP90]. As a result, testing can only provide a limited degree of assurance that the hazards have been mitigated.

8.1.2 Static Analysis

The limitations of testing mean that, when feasible, a static analysis of the source code should also be performed. Static analysis can use various informal or formal rules of reasoning that allow us to make a conclusion about a potentially infinite set of values. One example is inductive reasoning. There are other reasoning rules that similarly avoid the limitations of testing. Although the term “reasoning rules” sounds very formal, in fact, software developers use these rules everyday without necessarily thinking in formal terms. The chief limitation of static analysis methods is that they tend to be labour-intensive and expensive.

8.1.2.1 Informal

The most common form of static analysis is an inspection of the hazard-related code. There are many forms the inspection may take, though existing methods tend to focus on code quality checks [AC+03] or are hazard-analysis techniques that were not designed for software [IW95].

Software Fault Tree Analysis (SFTA) [LCS91] directly provides evidence of the absence of the hazard. FTA is performed at the source code level and begins by assuming a hazardous output from a 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 which implies that the source code cannot give rise to the hazard. The application of SFTA has typically been applied to relatively small systems. It would be
extremely difficult to apply the SFTA templates to software that is scattered across the architecture of the system. As a result, we can expect SFTA to be impractical for large systems unless the safety concern is isolated from the rest of the system.

Due to the labour-intensive nature of informal static analysis techniques, it is important that the analysis focus on the source code most relevant to the hazard, i.e., the safety concern source code.

8.1.2.2 Formal

There are mathematically based program verification techniques that exist which enable the proof that a program meets a formal, i.e., mathematical, specification [BS93]. Some of these techniques have been applied in industrial examples [GCR94].

Typically, program verification is associated with the use of Hoare logic to verify that a program satisfies its specification [Hoa69]. For example, the SACEM railway signalling system developed for the Regie Autonome des Transports Parisiennes (RATP) used both Hoare Logic and the B method [GCR94]. The safety-critical procedures were verified with Hoare Logic, while B was used to create a formal “re-expression” of the informal specification and provided a check on the proof annotations. However, except for the simplest programs typically in toy programming languages, it is very difficult to verify programs with Hoare logic.

Alternatively, the Darlington Nuclear Generating Station (DNGS) used the Software Cost Reduction (SCR) tabular-style specification, to ensure that two different software safety shutdown systems worked on demand and had no hidden functionality [GCR94]. However, the verification efforts were extremely labour intensive for a relatively small program.

The basic limitation of formal program verification approaches, besides being extremely labour intensive and expensive, is that they focus on program correctness as opposed to safety. As was argued in Chapter 2, safety is a property distinct from correctness. In particular, a correct system can still be unsafe.

A formal verification approach that does focus on safety involves the formal specification and verification of the system safety requirements. For example, Z was used to formally specify a Magnet Stereotaxis System (MSS) [KK93], with Knight and Kienzle first developing a set of software-safety specifications. In another case, Dutertre and Stavridou use the PVS verification environment [OS+95] to specify and verify avionics control system requirements that included safety requirements [DS94].
However, these projects were concerned with verifying the safety of the requirements and not the source code.

Given the limitations of formal program verification, Rushby has proposed the selective application of formal verification in areas where other techniques are inadequate, such as critical algorithms or safety properties [Rus95]. For example, a critical Byzantine fault-tolerant algorithm was formally verified [RV93].

The use of a formal verification technique would only supplement existing verification techniques, such as testing, as the safety properties that can be verified tend to be much more limited than what is required for safety verification, i.e., demonstrating the absence of a hazard. In particular, the method should focus some limited, relevant aspect of the source code, as captured in the safety concern model.

8.2 Representing Concerns

The problem of cross-cutting concerns has received a lot of attention, as evident, for example, in the aspect-oriented [KL+97] and hyperspace [OT01] approaches to software development. However, the focus of these approaches is addressing cross-cutting concerns during software design and implementation, and not on modeling existing concerns in the source code.

This section describes approaches that explicitly address representing concerns in the source code. This includes the textual documentation of delocalized plans [SP+88], the use of concern graphs to represent scattered concerns [Rob03], executable program slices for debugging [Wei84], and software visualization tools as an aid to program comprehension [SFM97]. In general, these approaches focus on the issue of documenting concerns at the implementation level for software maintenance purposes, as opposed to creating a model for analysis.

8.2.1 Textual Documentation

Studies were performed with professional programmers to investigate the impact of cross-cutting concerns on program comprehension during software maintenance [SL86]. In these studies, the problem is described as an issue of "delocalized plans", where a delocalized plan is some particular plan or intention of a programmer that is spread out over many components. The studies show that the likelihood of a software developer or maintainer correctly recognizing the software plan in a program decreases as the lines of code are spread out or delocalized.
One approach for addressing the issue of “delocalized plans” involved documenting the source code along with text that describes the plans of the developer, such as their design and implementation decisions [SL86]. For example [SP+88], the documentation could be arranged in a text document, with the left hand page containing the source code listing, while the right hand page contains the description of the causal interactions in the delocalized plans. Arrows can then be drawn from the description on the right to the code on the left.

This textual approach to documenting the delocalized plans could potentially be adapted to create a safety concern model in informal English. However, like the informal English technique, the resulting model has the same limitation of not being very analyzable.

8.2.2 Concern Graphs

Robillard proposed the construction of concern graphs to address the problems created for software evolution tasks by “scattered concerns” [Rob03]. The concern graph approach involves the creation of a simplified model of the concern that can be generated from the source code or some intermediate representation of the source code. The concern graph describes a subset of the program in terms of program elements and the relationships between them.

The concern graph is derived with the help of a “mapping function” that defines what program elements and relationships are to be retained in the model. For example, a mapping function that consists of functions as the only program element, and function calls as the only relationship between the program elements, would result in a program model that is its call graph. Subsets or “fragments” of the program model can then be selected to form concern graph. In this fashion, a model can be created of source code scattered over many components.

The model vocabulary is controlled by what program elements and relationships are included in the mapping function. As a result, there is a trade-off between the expressiveness of the model to represent details in the source code and simplicity, usability and scalability of the model.

The concern graph approach could potentially be used to represent the safety concern source code as an acyclic graph, as we proposed in Chapter 5. However, Robillard expressed reservations over the scalability of the method. In particular, for the mapping function and tool support he created for Java was limited to classes, interfaces, fields, and methods. Local (intra-method) elements were excluded due to the potential computational and memory costs, and on the grounds that such detail was unwarranted. As a
result, it is unknown how effectively concern graphs could model the source lines of code within a
method.

8.2.2.1 Program Slicing

The notation of a program slice was introduced by Weiser, which he claimed corresponded to the mental
model people maintained when debugging [Wei82]. In his original definition, the program slice is an
executable subset of the original program, and can be automatically determined from the source code
based on data flow and control flow analysis [Wei84]. With a basic slicing criterion involving a
statement of code and set of variables, the statically available information (i.e., without considering
program inputs) is used to determine all statements that affect the values of the selected variables at the
specified statement of code.

The slices introduced by Weiser can be seen as executable backward static slices [BG96]. Since Weiser’s
original definition, a wide variety of alternative approaches have been proposed [Tip95] that do not
require the slice to be executable, are static forward slices (e.g., determining subsequent statements
impacted by a specified statement and set of variable values), or involve dynamic slices (computed with
fixed inputs).

In terms of tool support, there are a variety of program slicers available [Hof95]. However, most of these
are research tools designed for “toy” languages. There are some commercial tools, such as the McCabe
Slice Tool, which provides some limited support for program slicing [STS97].

Program slicing could potentially be used to create an executable safety concern model in the same
programming language. In particular, it has been suggested that the program slice could be used for to
find all code that contributes to a variable in a safety-critical component [GL93]. This would be useful,
for example, when attempting to identify critical code paths associated with critical data, like the corrupt
temperature hazard scenario for the Chemical Factory Management System.

However, the automatic generation of the slice means that all code branches will be included, which
could potentially result in a model that involves a substantial portion of the software. For programming
slicing to create the desired safety concern model, there must be some way of flattening and pruning the
source code to eliminate irrelevant detail. As well, the slicing criteria are limited in what can be
specified. In particular, for hazard scenarios not clearly identified with critical data, like the stale
temperature hazard scenario for the Chemical Factory, it would be difficult to specify an appropriate
criterion. Finally, existing software tool support is very limited for performing slicing on a commercial piece of software written in a programming language like Ada.

8.2.2.2 Software Visualization Tools

There is a wide range of reverse engineering and software visualization tools that can be used for the navigation and presentation of software information [BE96]. One class of tools provide graphical representations of software structures linked to textual views of the program source code and documentation, with the goal of helping a maintainer form a mental model of the software [SFM97].

For example, the research tool Whorf [SBG92] was specifically designed to address cross-cutting concerns. It supports multiple views of the program such as source code listings, call-graphs, variable and function cross-references. Different instances of an object and lines of code can be highlighted in a particular colour in order to link the views and identify the concern source code.

Software Visualization Tools could potentially be used to help create a safety concern model in informal English, supplemented by the graphical representations of the higher-level software structures and hyperlinks to the source code. However, most of the existing tools were not designed to extract or highlight a safety concern. As well, the final model would not be easily documented or analyzed.
9. Summary and Future Work

There is an urgent need to provide safety engineers with methods and tools capable of managing the increasingly complex and integrated nature of safety-critical software. In particular, the goal of this dissertation is to strengthen the "back-end" of the overall safety engineering process for software-intensive safety-critical systems.

We might like to "turn back the clock" to when software was first used for safety-critical applications and, in most cases, safety-related code could be easily isolated from the rest of the system. But any approach based on this traditional paradigm is simply inadequate for the safety analysis of modern software. The increasingly complex and integrated nature of safety-critical software is fuelled by the consumer demand for increased efficiency, more automation and improved performance. Our understanding of these facts leads to the inescapable conclusion that any serious effort to build safety-critical software-intensive systems must abandon the traditional paradigm that insists on isolating the implementation of a safety concern from the rest of the system.

In its place, we have argued for a new paradigm that faces the reality of modern software – in particular, the reality of cross-cutting safety concerns. In support of this new paradigm, we have provided safety engineers with a methodology for extracting models of the safety concerns from the implementation of a system. These safety concern models may then used for the purposes of both documentation and safety verification.

We now summarize the results of the dissertation, including our contributions to the field of software system safety. Based upon the dissertation results, we offer a set of recommendations to stakeholders in safety-critical system development. We then discuss some avenues for future research. We finish by presenting some concluding thoughts.

9.1 Summary of the Contributions of the Dissertation

The following thesis statement was posited at the beginning of this dissertation:

The traditional system safety paradigm of isolating safety-critical functionality is no longer tenable in the face of the increased size and complexity of modern software systems. In a purely mechanical system, for instance, safety concerns typically form functional slices that can be conveniently partitioned from the rest of the system through design. Similarly, when software is involved, the conventional wisdom has been to isolate the software safety concern. However,
isolating safety-critical software is not always possible in many modern systems, especially large, software-intensive systems with a modular software architecture where the safety-critical functionality is interleaved with core system functionality. There are a variety of legitimate reasons why a safety concern is not easily separated at the source code level from the rest of the system. As a result, instead of attempting to impose a strict partitioning of safety from non-safety software, the safety engineer should use techniques to extract models of the safety concerns for safety verification.

The contributions of this dissertation can be summarized as follows:

1. A reassessment of the traditional system safety design principle that safety-critical functionality should be isolated.
2. An in-depth analysis of how object-oriented software can create difficulties for safety verification.
3. Developing the foundations for modeling a safety concern.
4. An approach for creating a safety concern model involving four basic steps.

9.1.1 Contribution #1: Safety as a Cross-Cutting Concern

The first contribution involves challenging the traditional notion that the safety-critical aspects of a software system should be isolated. This design principle has been stated in software safety papers [PVP90], standards [IEE94], and textbooks [Sto96]. The fact that modern software architectures can result in cross-cutting safety concerns is a paradigm shift for system safety and software engineers.

We examined a safety concern for a Chemical Factory Management System in detail. The system possessed a modular, layered architecture that is typical for information systems. We found that the safety concern was scattered among a number of different software components. This scattering of the safety concern is likely to be the rule for any information system with a modular architecture, as was seen with other systems, including a radar data processing system, an ATM system, and a blood pressure monitoring system.

9.1.2 Contribution #2: Safety Verification and OO

The second contribution is describing how certain aspects of software design and implementation designed to master software complexity can also complicate safety verification. Object-oriented design is a fundamental methodology that will continue to influence modern software development for the foreseeable future. There are genuine benefits to object-oriented design that might lead some system
stakeholders to assume that it is entirely beneficial to safety. However, this is not necessarily so. A contribution of this dissertation is to dispel that assumption and to indicate the challenges that object-oriented software poses to safety verification.

In particular, we demonstrated with the Chemical Factory Management System example how the notions of modularity and hierarchy in object-oriented approaches might lead to cross-cutting safety concerns. In addition, we observed a large amount of non-executable source code intended to support class hierarchies, safety-related functionality interleaved with other functionality, and the presence of concurrent software processes.

9.1.3 Contribution #3: Foundations of Safety Concern Models

The third contribution is laying out the foundations for constructing safety concern models. This includes both the model vocabulary and, in particular, a variety of different ways of representing these models. The representations include an executable model in the same programming language, which can be tested, and in a formal mathematical notation, which still maintain lexical similarity to the source code.

9.1.4 Contribution #4: An Approach to Creating Safety Concern Models

The fourth contribution is outlining a method for creating a safety concern model designed to overcome the challenges presented by modern software architectures. The method provides a practical framework within which existing tools and techniques can be employed to create the model.

The approach was demonstrated through application to both an ATM system and a Chemical Factory Management System. Safety concern models were created using informal English, Ada, and S. In each case, a coherent, simplified model of the critical code paths resulted that was suitable for use for safety verification. In particular, the safety concern source code that had been scattered over many components was gathered into one place, with a large amount of the non-executable source code removed.

9.2 Recommendations

Based upon the results of this dissertation, the following set of recommendations are made for the verification of safety-critical software systems:

- Identify the software components involved in a hazard scenario during the software high-level and detailed design.
- Identify the critical code paths and begin construction of the safety concern model during software implementation.
• Verify the safety concern model with respect to the source code level SVCs.

More generally, based upon the experiences of the author with the development of safety-critical software systems, the following set of recommendations are made:

• Develop hazard scenarios and system level SVCs during requirements specification and initial conception of the software architecture.
• Include hazard scenarios among the scenarios used for validation of the software architecture.
• Develop design level SVCs and incorporate into the design reviews.
• Develop source code level SVCs and incorporate into unit code reviews and tests.
• Include the safety concern model and design level SVCs in integration testing.
• Implement a hazard-driven safety test program that goes beyond testing of the safety requirements.
• Create a safety verification case that records the safety concern models, SVCs, and verification efforts.
• Log the impact of the hazards on the specification and architectural decisions that are made, as well as the impact of these decisions on the hazard.

9.3 Future Work

This section describes some of the different directions that can be taken with the safety concern model:

• Developing more effective methods for maintaining the code model.
• Overcoming the limitations of existing concern representation approaches for modeling safety concerns.
• Applying existing analysis tools and techniques to the safety concern model.
• Extending the safety concern model to address the modeling of concurrent processes.
• Encapsulating safety concerns during development.

9.3.1 Maintaining Safety Concern Models

There are potentially more effective methods for recording the execution of the code model extraction than a simple textual description of the process. In particular, it would be useful if some or all of the steps in the process could be automated. It would also be useful if the consistency between the model and the source code could be automatically checked. Different approaches are suggested as future work depending on whether the code extraction is carried out in a bottom-up or top-down fashion.

9.3.1.1 Bottom-Up Extraction
The model can be extracted in a bottom-up fashion through either a backward or forward search of the source code. In the case of a backward search, a critical code path is discovered by incrementally searching for the cause of an effect by tracing backwards through the code. Forward searching involves tracing a cause forward to its possible effects.

It was suggested in Chapter 5 that the simplest method of recording the extraction of a model would be a textual description. A more rigorous representation of the extraction could be recorded in the form of an acyclic graph. In this form, each node in the graph corresponds to a line of source code. Each arc between a pair of nodes in the web corresponds to a relationship between the two lines of source code. A variety of different kinds of relationships are likely to be recorded as arcs between nodes. Some of the arcs will represent cause-effect relationships while other arcs will represent "definitional" relationships. In particular, the cause-effect relationship could be a matter of control flow (e.g., statement X was executed because ...) or it could be a matter of data flow (e.g., variable was modified because ...). For instance, there would be arcs from a line of source code that contains a reference to a global variable "X" to all of the lines of source code that may directly change the value of "X". Additionally, there would be an arc to the line of source code that contains the declaration of this global variable.

It may be impractical to produce a readable representation of a large acyclic graph on a traditional medium, i.e., paper. Robillard's concern graph approach is a possible alternative [Rob03]. As well, there are computer-based, interactive visualization tools that could support the creation, maintenance and use of a graphical representation of the graph. It may be possible to create a hypertext link from nodes within the representation to corresponding locations in the source code. Depending on the completeness of this graphical representation and the kinds of relationships represented by the arcs, it may even be possible to generate automatically source code for the model that could be compiled and executed.

The representation of the search of the source code in the form of an acyclic graph allows the validity of the extracted model to be systematically checked. The model must be re-checked if the source code implementation of the system is changed. The validity of the extracted model can be systematically (perhaps even automatically) checked by verifying the relationships represented by each of the arcs in the model.

9.3.1.2 Top-Down Extraction

The top-down approach to the model extraction may be viewed conceptually as a sequence of transformations applied to a copy of the source code implementation. The transformations systematically
reduce the source code to a subset of the implementation that represents a critical code path. The result of each transformation is intended to be a conservative approximation of the implementation of the system.

The transformations may be performed at a variety of different levels of granularity and rigor. But in all cases, each transformation would be recorded as a distinct step. It is possible that an ad hoc set of transformations may be used to extract a model of the critical code path. In this case, the record of each step should include a careful explanation of the justification for the transformation. On the other hand, a previously established set of named transformations may be used to perform the extraction. In this case, the execution of the transformation process could be recorded in a more concise format by simply recording the name of the transformation used in each step. Many of the named transformation rules would likely be associated with "rules of use" which must be checked whenever a rule is used.

One possibility is to record the transformations in the form of an executable script. The execution of this script would replicate the actions of the software developer in using the tools of the development environment to edit a copy of the source code implementation of the system. In addition to performing actions such as deleting "irrelevant" portions of the code, the execution of the script must automatically check that each transformation step is applied in a manner consistent with the "rules of use" for each kind of transformation. This would allow a "fresh" version of the model to be automatically extracted from the source code implementation. If a change has been made to the implementation, and the change has an impact on the validity of the model, then either the freshly generated model will reflect the impact of this change or the execution of the script will fail because the "rules of use" will fail.

9.3.2 Constructing Safety Concern Models

This dissertation described the creation of safety concern models from software implemented in the Ada 83 programming language. The approach should also be applicable to software implemented in other programming languages.

As well, related work on tool-based approach for the selective extraction of implementation detail from source code has the potential to support the creation of safety concern models. However, current applications of these tool-based approaches have limitations that first need to be overcome.

9.3.2.1 Software Implemented in an Object-Oriented Programming Language

The approach for constructing a safety concern model is generally applicable to software with an object-oriented architecture. In each case, the flattening, filleting, partitioning, and translating steps should still be performed to create the model. However, there will be some differences in the details of the
application in the case of OO languages that support inheritance. OO languages include Ada 95, Java, C++, Eiffel and Smalltalk.

In particular, flattening of the software will involve a “vertical” collapse of class hierarchies, so that the inherited methods are explicitly present in the flattened class. Potential tool support for the flattening was described in Chapter 5. As well, searching for critical code paths prior to flattening is complicated by polymorphism and late binding of methods [Boo94].

9.3.2.2 Concern Graphs

As noted in Chapter 8, concern graph approach could potentially be used to represent the safety concern source code as an acyclic graph. However, Robillard’s application of the method to Java restricted the mapping function and tool support to classes, interfaces, fields, and methods [Rob03]. Local (intra-method) elements were excluded due to the potential computational and memory costs, and on the grounds that such detail was unwarranted.

Detail at the intra-method level is required for the safety concern models. In particular, the mapping function and tools would have to be extended to address the entire model vocabulary, including the elements of a particular executable line of code. As noted by Robillard, extending his method in this fashion raises issues of scalability and whether the computation costs will be too great [Rob03]. As well, there is also an issue of whether the source code details required for safety verification will be represented in the resulting code graph.

9.3.2.3 Program Slicing

As noted in Chapter 8, program slicing could potentially be used to create an executable safety concern model in the same programming language. However, the slicing criterion is limited to a statement of code, set of variables and, for dynamic slicing, a set of input values. The resulting automatic generation of the slice will likely include irrelevant branches of code, potentially involving a substantial portion of the software.

We require some mechanism for pruning the branches of the source code to eliminate irrelevant detail. One potential approach is to provide user interaction during the program slicing. For particular code branches, the user would have the opportunity to mark a branch as irrelevant or conversely to highlight code branch as relevant.

9.3.3 Analyzing Safety Concern Models
The safety concern model is ultimately to be analyzed for safety verification. For the safety concern model specified in a programming language, there are the standard software safety verification techniques that were reviewed in the Chapter 8. In particular, for a model in SPARK, there is potential of completing the verification with the help of code verification tools.

For a model in S, there are several potential alternative techniques for analyzing the model. One obvious approach is the use of a theorem proving system such as HOL, as S is a syntactic variant of the object language of HOL [JDD94]. In particular, HOL could be used to derive safety properties from the model. However, Day et al. have also proposed a set of lightweight tools for analyzing specifications expressed in higher-order logic that eliminate the overhead required in interacting with a conventional theorem prover [DDJ00]. For example, two tools that might be of value in analyzing the code model include one for performing symbolic functional evaluation and another for semi-automatically generating a set of test cases.

9.3.3.1 Tool-Based Code Verification

Chapter 7 described the construction of verifiable code assertions for a safety concern code model written in SPARK. SPARK Examiner can then be used to generate verification conditions from the model when supplemented with the code assertions. To complete the verification, 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]. For example, this verification could be performed on the verification conditions produced from the safety concern model for the Chemical Factory Management System.

However, even with the help of the tools, the process of verifying the verification conditions can be quite non-trivial. In particular, specifying the pre- and post-conditions after the software has been written can result in bloated annotations and intractable verification conditions for industrial software systems. It would be interesting to determine what type of annotations and verifications result from a safety concern model created in SPARK for a large software system such as the ATM system.

9.3.3.2 Symbolic Functional Evaluation

A model in S cannot be executed in the same fashion as a model in a programming language. However, Day and Joyce have proposed a method called symbolic functional evaluation (SFE) for evaluating a specification in higher order logic in a manner similar to the evaluation of a functional program [DJ99]. The following description of SFE is largely taken from Day and Joyce’s paper.
SFE is an extension of an algorithm for executing functional programs to evaluate expressions in higher order logic. SFE carries out logical transformations such as expanding definitions and simplification of built-in constants when uninterpreted constants and quantifiers are present, both are likely to be true for a code model in S. In addition, there are different levels of evaluation that can serve as natural "stopping points" when determining how far to evaluate the arguments of uninterpreted functions. There is also tool support for evaluating specifications in S, i.e., the research tool "Fusion".

The code model in S for the ATM hazard involves the potential corruption of the aircraft position. The approach then would be to specify some initial "input" values for the aircraft position and then attempt to apply the Fusion tool to the enhanced model. If the model could be fully evaluated, the result would be the "output" aircraft position values. In the presence of uninterpreted functions, the output of symbolic evaluation will likely be an expression rather than a value. The output expression would describe the output of a function of the inputs. This output expression could be extremely helpful in answering the kinds of questions described in Section 3.1.3 of this dissertation. For example, the output expression would immediately reveal what compile-time constants were involved in the processing of critical data.

9.3.3.3 Test Case Generation

Though the code model in S cannot be executed and tested as such, there is the possibility of partially automating the process of generating a set of test cases from the model. In particular, Donat has proposed an algorithm for automatically generating a set of test descriptions or "test frames" from a formal mathematical specification, which can then be manually turned into steps in a test procedure [Don98a]. The proposed algorithm uses well-defined rules of logic and is based upon specific, precisely defined coverage criterion.

For example, in one particular application of the algorithm by Donat, an S specification was put into stimulus/response form, domain knowledge was used to document dependencies between the conditions and a "Test Case Generation" (TCG) tool was used to generate a set of test frames from the specification [Donat98b]. The coverage criterion that was used essentially required that every condition that appeared in the specification also appeared in at least one test step. Each resulting test frame consisted of a set of "stimuli" and "response" conditions. Providing explicit values for the variables in the test frame creates the corresponding procedure test step.

In a similar fashion, the code model in S could also potentially be used as input into the TCG tool. One issue is that the code model is not in the appropriate form. Another is that an unmanageable number of potential test frames might result. As a result, some appropriate portion of the model would first have to
be extracted and put into stimulus/response form. For example, if there were a set of conditions in the code which needed to be tested. Then an appropriate coverage criterion would need to be specified, depending on how thorough the coverage is required to be. The result would be a set of test frames that could then be converted into a set of test steps for the safety concern.

9.3.3.4 Theorem Proving

Though the possibility of doing Hoare style proofs with the code model might prove to be too difficult or not relevant to safety, there is also the possibility of using theorem proving or model checking to derive certain safety properties from the model. For example, as noted by Joyce et al. it is relatively straightforward to translate an S specification into a form suitable as input into the HOL theorem proving system [JDD94].

The following approach could be attempted with the code model:

1. Translate the S representation of the safety concern model into input for HOL.
2. Formulate in HOL a safety property that should be derivable from the model.
3. Derive the safety property as a logical consequence of the model using HOL.

For example, the safety concern model that was created for an ATM hazard in Chapter 6 involves the corruption of the aircraft position. A simple safety property might be "If an update is made to the situation display for a particular track, then the system must have previously received a track update for this particular track". The safety property could then be formulated in HOL and an attempt made to derive this property from the model.

9.3.4 Extending the Safety Concern Model

The software processes could be further modeled as a set of functional blocks that potentially could be more effective for analysis of timing and concurrency issues. Below is a sketch of how this might be done. Future work would involve further clarifying the functional block model and applying it to an example.

The model can be seen as a collection of software processes that interact through message passing and potentially shared data. The message passing may be either synchronous (i.e., sender is blocked until the message is received by the recipient) or asynchronous (i.e., sender continues executing after sending a message regardless of when the message is received by the recipient).
Each process consists of a set of function blocks, where each functional block maps an input message to a set of output messages. In this abstraction, a functional block is executed instantaneously. However, the duration of time between sending a message (as an output of a functional block) and its receipt (as an input of a functional block) is finite, but unknown and variable. To be a valid model, the functional block must not be used as a model for fragment of code that may fail to terminate.

Time passes between executions of functional blocks. It is assumed that the execution of functional blocks is serialized, i.e., only one block is executed at a time. When more than one block is "ready" to be executed (i.e., more than one block has a waiting input message), no assumption may be made about the execution order.

The functional blocks and their interactions could then be modeled in S. For example, a functional block FB1 that responds to the receipt of a "msg1" message might look like the following:

```s
FB1:
Read (Msg1 (x,y));
a := if y then x else 3;
b := a + 2;
Write (Msg3 (b));
```

Figure 9-1: Example functional block source code

The following definition defines a constant "FB1" in S that represents the behaviour of FB1:

```s
FB1 :=
forall x y t.
if (msg1 x y) is_received_at_time t then
let a := if y then x else 3 in
let b := a + 2 in
exists n. (msg3 b) is_sent_at_time (t+n);
```

Figure 9-2: Example S specification for functional block behaviour

The "exists n" quantification is a way of saying "at some future time t or later".

9.3.5 Encapsulating Safety Concerns

This dissertation described the difficulties of extracting a cross-cutting safety concern for analysis. It would be of immense benefit if the safety concern could be encapsulated during development. Furthermore, it would be especially useful if the safety concern could later be extracted and executed independently, as is often possible with non-software safety concerns. In this fashion, the central problem that cross-cutting concerns pose to safety verification would be addressed.
For example, it might be possible to use existing mechanisms such as Aspect-Oriented Programming (AOP) [KL+97] to encapsulate the safety concern. However, initial investigations by Joyce and Feng [JF04] indicate that a safety concern does not naturally correspond to an aspect. The key difficulty appears to be that a safety concern can correspond to core functionality, while aspects tend not to be units of the system's functional decomposition [KL+97]. It is possible that alternate methods, such as the hyperspace approach [OT01], might be more suitable.

9.4 Conclusion: Software Safety Verification and the Way Forward

We have seen in this dissertation that the "footprints" of a safety concern are quite likely to be scattered across the implementation of a complex software system. The fictional detective Sherlock Holmes never expected the footprints of his quarry to be conveniently located in one place, and likewise, safety engineers must be prepared to deal the likelihood that important details about a safety concern must be widely distributed across the source code in spite of traditional wisdom that these traditional details should be conveniently located in one place.

Many, if not most, system safety engineers are truly unsure about how to perform safety verification of a software-intensive system. Some are unable to think of anything more than taking extra care in verifying safety requirements and safety-related requirements. But this is not enough because the safety requirements might not be sufficiently complete or there could be unintended functionality in the system. Some turn to statistical measures such as measures of defect detection in safety-related code. But this is not enough as hazards might arise from factors other than software defects. Some use inspection techniques to search for possible weakness in the code such as uninitialized variables. But this is not enough because hazardous behaviour can arise in a myriad of ways beyond source code failing to satisfy coding standards. This dissertation provides the way forward.

Ultimately, this dissertation demonstrates that a paradigm shift is required on how system safety engineers view and verify software-intensive systems. It is no longer reasonable to assume that safety concerns will be conveniently isolated from the rest of the system. The cross-cutting nature of the safety concern is a fundamental break from a traditional principle of system safety practices and forces a reconsideration of strategies used to design and verify safety-critical software.

The fact that safety is a cross-cutting concern should be recognized and addressed upfront during software design. In particular, modeling a safety concern can viewed as part of a more fundamental issue of keeping track of all of the decisions that influence the architecture of a software system. While this dissertation has focused exclusively on safety concerns, any realistic approach to the development of a
complex software system must simultaneously consider a variety of goals including performance, availability and security. Because these goals may conflict, it is important to understand decisions about one goal in terms of its impact on other goals. Potential future research would involve exploring the possibility of providing links between a safety concern model and an overall structure that keeps track of architectural decision. This dissertation provides the first step toward that goal.
Bibliography


Appendix A. Formal Specification Notation S

The formal specification notation used, "S", is based on typed predicate logic. This notation was developed at the University of British Columbia to serve as a foundation for a variety of different approaches to formal specification [JDD94].

S is a syntactic variant of the HOL object language, with some additions. As noted in Chapter 5, a notation based on higher order logic provides the following:

- Uninterpreted types and constants
- Generic functions, where it is not necessary to assume any such properties as partial, total, injection, surjection, or bijection
- higher-orderness, i.e., the ability of functions to take functions as arguments
- Quantifiers, such as a “for all” and “exists”

S is plain text. Instead of using symbols such as "!" and "?" for universal and existential quantification, in S the keywords “forall” and “exists” are used.

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.
Appendix B. Process for Constructing a Safety Concern Model

B.1 Introduction
This document specifies a process for constructing a model of the source code related to a safety concern. This process description assumes the existence of other safety processes such as Hazard Identification, Hazard Analysis and Safety Verification. In particular, the resulting model is an input into the Safety Verification process.

B.2 References


B.3 Definitions and Abbreviations

B.3.1 Definitions
The following definitions are primarily based upon the system safety standard MIL-STD-882C [DOD93]:

- **Mishap (or Accident)**: An unplanned event or series of events resulting in death, injury, occupational illness, or damage to or loss of equipment or property, or damage to the environment.

- **System Hazard (or Hazard)**: A state of the system that, possibly in combination with environmental conditions, leads to a mishap.

- **Hazard Analysis**: The identification of hazards and hazard causes.

- **Hazard Cause**: An internal (i.e., system) or external (i.e., environmental) condition that, possibly in combination with other internal and environmental conditions, leads to a hazard.

- **Hazard Scenario**: A chronology of events that identifies one possible combination of internal and external hazard causes that results in the occurrence of a hazard.
- **Safety Concern**: A hypothesis about a specific combination of internal or external events that might lead to an occurrence of a hazard.

- **Critical Code Path**: A particular view of the code that generally involves a partially ordered sequence of executable statements that leads to a hazard occurrence.

- **Safety Concern Model**: Representation of a critical code path corresponding to a system hazard.

- **Safety**: Freedom from those conditions that can cause mishaps.

The distinction between mishap, hazards and hazard causes is not always clear from the definitions. Figure 1 provides an overview of the relationship between them.

![Figure 1: The relationship between hazards, hazard causes and mishaps.](image)

For a blood pressure monitoring system, an example of a mishap is the improper treatment of hypertension/hypotension. A hazard would be the display of an incorrect blood pressure reading. An internal hazard cause, at the system level, would be a miscalculation of the blood pressure reading. The hazard cause could be further traced to a software cause, such as the software corruption of a sensor reading. An external hazard cause would be an improper adjustment of the pressure cuff around the patient’s arm.

The resulting hazard scenario might then be:

1. A blood pressure reading is received.
2. The reading is corrupted by the system.
3. An incorrect blood pressure is displayed.

During hazard analysis, the hazard scenario can then be refined into a sequence of events within or between a set of subsystems and subsystem components.
B.3.2 Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COTS</td>
<td>Commercial-Off-The-Shelf</td>
</tr>
<tr>
<td>FMEA</td>
<td>Failure Modes and Effects Analysis</td>
</tr>
<tr>
<td>FTA</td>
<td>Fault Tree Analysis</td>
</tr>
<tr>
<td>HAZOP</td>
<td>Hazard and Operability Studies</td>
</tr>
<tr>
<td>OO</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>SLOC</td>
<td>Source Lines of Code</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
</tbody>
</table>

B.4 Roles and Competencies

An analyst will perform the construction of the safety concern model. The model construction will require competencies in the following areas:

- **Operational and Domain**: The safety implications of the source code cannot be understood separately from the overall system operation. In particular, understanding of the domain will be required to help assess whether particular software outputs are relevant to safety.

- **System Safety**: The hazard identification and analysis are required input for model construction. The model construction is guided by the relevance of the software to the system hazard.

- **Software Architecture**: Prior to the search for the safety concern source code, it is necessary for the analyst to understand the overall software architecture. In particular, the components and processes involved with the different hazards must be identified.

- **Software Implementation**: The search for source code will involve an understanding of the programming language, software development environment, and the static software architecture.

The support of other stakeholders, such as domain experts, system safety engineers, software architects and software developers can help provide the necessary competencies.

B.5 Process Inputs

The inputs of the approach are:

- **Hazard identification and analysis**: Includes identified system hazards, hazard scenarios, and hazard causes.
- **Software architecture and design**: Includes the software components, processes and logical design (e.g., class diagrams), as well as object-collaboration diagrams.
- **Software source code**: Includes the available developed source code.

A part of the system safety process involves identification and analysis of system hazards [Lev91], which should precede model construction. The safety concern model is constructed with respect to a system hazard. The analysis of the software requirements and design should be analyzed for potential hazard causes. There exist a variety of techniques [IW90] to support hazard analysis including Hazard and Operability Studies (HAZOP), Failure Modes And Effects Analysis (FMEA) and Fault Tree Analysis (FTA).

**B.6 Process Outputs**

The outputs of the approach are:

- **Safety concern description** – Description of the hazard and the relevant software components, including hazard scenarios and the corresponding object-collaboration diagrams.
- **Safety concern model** – A representation of the extracted source code corresponding to the safety concern.

Figure 2 displays an example outline of a document that contains a description of the safety concern, as well as a portion of a safety concern model corresponding to a critical code path. The document includes relevant background information on the hazard and hazard causes, such as the relevant components and object-collaboration diagrams. As well, the document contains the extracted source code, along with assumptions about the pruned source code.

**B.7 Process Steps**

The steps of the process include the following:

1. **Flatten** - modify classes by making explicit all the attributes, methods, conditions, and invariants the class inherits from other classes. Expand parameterized classes (generics or templates) by replacing parameters with the actual parameters, changing the parameters names as necessary to avoid namespace conflicts.
2. **Fillet** – identify and extract the critical code paths. Prune the less relevant code branches and, if necessary, replace with something simpler or with an assumption of the expected behaviour.
3. **Partition** – identify the portions of the critical code path that execute in separate processes.
4. **Translate** – represent the source code in a notation that allows the irrelevant details to be omitted.

Though the steps are to be carried out sequentially, they may also overlap. These basic transformation steps can be carried out by a number of smaller sub-steps.

---

**Figure 2: Outline of a Document Containing the Safety Concern Model**

**B.7.1 Flatten**

For systems with object-based software architectures, the class hierarchies are “flattened” to bring together the relevant lines of code and to reduce the amount of “non-executable” code. For example, Figure 3 displays a class hierarchy for classes A, B and C.

Object-oriented languages such as Java and C++ have explicit class mechanisms. Class flattening typically means in this case producing an equivalent class without any inheritance. This involves modifying the class by making explicit all the attributes, methods, conditions, and invariants it inherits from other classes. This allows the analyst to see every feature a class possesses in one location.
For example, class "C", as displayed in Figure 3, would be flattened into class "CI", as displayed in Figure 4.

As well, some OO languages have parameterized classes (e.g., C++ templates). Flattening involves "expanding" each class instance. A representation of each class instance is created by explicitly replacing each parameter for in the class body with the actual parameter, where the actual parameter name is potentially modified (e.g., type name prefixed with class name) to avoid namespace conflicts.

```java
class A {
  protected int x;
  public A() {
    x = 5;
  }
  public void nextX() {
    x = (x + 1)*2 - x - 1;
  }
}

class B extends A {
  protected int y;
  public B() {
    super(x);
    y = x*2;
  }
}

class C extends B {
  protected int z;
  public C() {
    super(x,y);
    z = 3;
  }
}
```

**Figure 3:** Class hierarchy in Java

```java
class CI {
  protected int x;
  protected int y;
  protected int z;
  public CI() {
    x = 5;
    y = x*2;
    z = 3;
  }
  public void nextX() {
    x = (x + 1)*2 - x - 1;
  }
}
```

**Figure 4:** Class C Flattened into Class CI

Tools that display a flattened form of a class are available for some object-oriented languages. For example, the EiffelStudio software development environment displays a "flat form" of a class written in the Eiffel programming language. More commonly, software development environments will typically
have a class browser that display all the methods for a class, including those inherited from a parent class.

**B.7.2 Fillet**

Filleting involves slicing out the “flesh” of the safety-critical source code, carving around the “skeleton” of the static software architecture. In particular, filleting involves:

- **Identifying Software Hazard Causes and Scenarios**: The relevant object-collaboration diagrams and component hazard causes are determined.
- **Slicing the Source Code**: The source code is searched for code relevant to the safety concern, guided by the object-scenario diagrams and component hazard causes.
- **Pruning the Source Code**: The identified source code is “pruned”, as not all of the source code related to the safety concern needs to be included in the model.

If every possible critical code branch was followed, it is possible that a large portion of the software system would then be deemed part of the safety concern, which would not be useful for the purposes of safety verification.

**B.7.2.1 Identifying Software Hazard Causes and Scenarios**

The results of the hazard identification and analysis are used to guide the search for the safety concern source code. The hazard identification provides the scope of the hazard. The hazard analysis of the software requirements and design identifies potential hazard causes and scenarios. The hazard scenarios provide the basis for the search for critical code paths.

Standard hazard analysis techniques include FTA and FMEA:

- FTA begins with the hazard as the “top event” and then the events that cause the hazard are determined, combined using logical operations such as “AND” and “OR”.
- FMEA is typically used to examine the consequences of failures in system components, beginning with a component fault and then tracing the effects forward to system outputs.

The results of a hazard analysis of the software design and architecture includes a set of components that are relevant to the hazard, as well as the potential hazard causes at the component level. The hazard
causes typically correspond to a set of hazard scenarios. The search for the safety concern source code begins with the identified components, guided by the component causes.

For an object-oriented design, the dynamic behaviour of the software can be described by object interactions, where an object is an instance of a class. Object-collaboration diagrams [BRJ99] are useful for identifying object interactions corresponding to the hazard scenarios. For simple OO systems, there is a one-to-one correspondence between the classes and the development components. However, for more complex systems, this is not likely to be true. For example, a number of different components might be used to implement a single class.

**B.7.2.2 Slicing the Code**

The critical code paths are identified and extracted from the source code. The search for the critical code paths can be performed in both a backwards and forward fashion. The components and component causes determined during the hazard analysis are used to guide the search for the critical code paths.

The critical code paths might involve control flows, data flows or a combination of the two types of flows. For example, for a blood pressure monitor, a data flow would be useful in understanding the sequence of transformations performed by the system to convert a blood pressure sensor reading into a displayed blood pressure value. However, the blood pressure display might be affected by factors not directly related to the blood pressure value, such as a timeout of the display. This portion of the critical path would involve a control flow.

The object-collaboration diagrams are used to guide the forward code search, from system inputs to system outputs. Each key component in the diagrams is examined in turn, beginning with the procedure call that provided entry into that component. The component hazard cause identified in the FTA or FMEA is used during the search of the component source code. Subsequent procedure calls are then traced within that subsystem until the path leads out to the next component. The object-collaboration diagram and hazard analysis are updated depending on what is uncovered during the search.

The analysis can then be performed in a backward fashion, starting with the system output and identifying the lines of source code that might lead to that output. For example, the procedure that causes the blood pressure to be displayed on the screen could be identified. The software can then be searched for code that invokes this procedure and so on, until potentially a system input is reached.
Tool support for the search includes basic lexical searching tools like the unix utility “grep” that search for regular expressions in the source code. Typically, software development environments provide additional support for component cross-referencing which would be useful when tracing critical code paths across component boundaries. These capabilities include links to type declarations and to places where a given type is used.

Some software development environments are integrated with design tools in support of “round-trip engineering”, for going from the design to the source code and back again. If the hazard-related object-collaboration diagrams have been identified, these tools can be used to help with the initial forward search of the relevant components.

**B.7.2.3 Pruning the Code**

The code extraction involves removing irrelevant functionality from consideration in the following fashion:

- **Irrelevant Source Code**: The pruned source code does not contribute to a hazard occurrence and is eliminated from the model.
- **Unreachable Source Code**: The pruned source code is unreachable for a particular safety concern and is eliminated from the model.
- **Conservative Approximation**: The pruned source code potentially contributes to a hazard occurrence, but will be replaced by something simpler in the model that is a conservative approximation of the actual software.
- **Separate Validation**: The pruned source code potentially contributes to a hazard occurrence, but will be replaced by something simpler in the model and addressed separately.

The pruned source code can be simply eliminated from the model if it is not relevant to the safety concern. For example, for class C1, the variable “z” might not have any impact on the hazard, so the variable “z” and the assignment statement “z = 3” could be pruned.

Some portions of the critical code path could be deleted on the basis of being only executed in a particular state that can be shown to be unreachable in a particular safety concern. For example, if a system has distinct modes of operation, state transition diagrams could be used to indicate when a portion of a critical code path corresponding to a particular state is unreachable.
The pruned software can be replaced in the model by something simpler, which is a conservative approximation of the actual software. For example, for class “C1”, "y = x * 2" can be replaced with "y = some even number larger than the current value of x" or "y = some even number" or "y = some number larger than y" or even just "y = some number". Each of these alternatives is conservative approximation to the actual code, i.e., anything that can be proved from the conservative approximation is also true of the original code.

The pruned software can also be replaced in the model by something simpler, which can be verified independently. In particular, it is useful to separate in the analysis the behaviour of the application level processing and the reliability of the low-level services. Low-level services include code libraries, communication and concurrency mechanisms, abstract data types and COTS products. The low-level services can be pruned from the model and validated separately.

For example, the class “C1” has the method “nextX”, intended to increment the value of “x”. Rather than include the entire function in the code model, the function can be replaced by something simpler that returns the correct value of “x” for the scenario under consideration. The correctness of the function would be subject to separate validation.

B.7.3 Translate
The final step of the process is to translate the critical code path into a form where the irrelevant source code details have been eliminated, including a representation of the pruned source code. Potential notations for the representation include “informal English”, a programming language, or a formal mathematical notation.

The notations provide a range of techniques of increasing expressiveness, for specifying the pruned code, and analyzability, in terms of tool support. However, there is a trade-off between the expressiveness of the notation and the skill level effort required to use the notation. There is also increased likelihood for a “semantic gap” between the source code and the model syntax, though the increased tool support could potentially help mitigate this by detecting inconsistencies in the model. Ultimately, the appropriateness of a given technique will depend on practical considerations such as the amount of time and effort available, and the criticality of the hazard.

B.7.3.1 Informal English
A simple technique for representing the critical code paths involves extracting the relevant source code and annotating the code fragments in English inside a text document.
The extracted source code includes:

- Instantiations of parameterized classes that are used in the critical code paths
- Extracted executable source lines of code
- Declarations of the data types corresponding to the variables maintaining critical data

The annotations include:

- Source of the extracted source code (e.g., originating procedure/function and source code file)
- Assumptions about the pruned source code
- Figures such as object-collaboration and component diagrams that indicate the important components and data flows

The outline of an example document is shown in Figure 2.

The greatest virtue of an informal language notation is that it is easily understood and communicated to others. Another virtue is its unbounded flexibility. It obviously does not require any special knowledge or tools on behalf of the analyst or other stakeholders in order to create or understand the model. The limitations of such an approach are that the model is imprecise and not easily analyzed. The model will not be executable or amenable to any type of tool-based analysis. There will be no method of confirming that all assumptions about the pruned code have been captured.

**B.7.3.2 Programming Language**

A potentially more useful technique than informal English is representing the model in an executable form using the same programming language. For example, the extracted source code is structured (e.g., into a C++ header and body files), supplemented with type definitions and stub functions, and linked to standard language libraries as necessary so that the model can be compiled. A driver function can be added to drive the execution of the model.

The obvious benefit of an executable model is that it can be tested. The key limitation of this technique is that stubs would need to be created for the pruned subprograms. As well, the model will be limited to the same type of analyses that can be performed on the original system.

Alternatively, the model could be represented in a different programming language:
1. Use a different programming language, e.g., a functional programming language or

2. Use a specialized programming language that provides explicit support for reasoning about software behaviour.

If a different programming language were used, translation can be considered to be a verification exercise in itself in addition to testing of the final model. As well, in the case of a functional programming language, the functional language paradigm discourages “side effects”, especially if one uses a purely functional approach. The advantage of avoiding side effects is that it makes the data transformations explicit and more readily analyzed. However, the limitation of using a different language is the much greater overhead in performing the translation and the notation will likely not be as familiar to stakeholders.

There are programming languages that also support reasoning about software behaviour. For example, the SPARK language has tool support for static code verification. The limitation is that there is again the additional overhead in translating the source code into a different programming language and the use of potentially unfamiliar proof tools.

B.7.3.3 Formal Mathematical Notation

Another specification approach would be use of a formal mathematical notation. For example, the formal specification notation “S” has been used to represent software written in an object-based programming language. S is based on higher order logic and was developed at the University of British Columbia to serve as a foundation for a variety of different approaches to formal specification [JDD94].

Benefits of using a formal notation include the fact that formal notations are usually more expressive than a programming language, the translation effort itself is a form of code analysis (i.e., “the journey is its own reward”), and there is the potential for tool-assisted reasoning about the model. The limitation of mathematical notations is their lack of familiarity to most developers, the potential for mistakes in the translation, and, typically, their syntactic distance from the source code. Use of these methods would likely require training for the analysts and other stakeholders who make use of the models.
Appendix C. English Model of Chemical Factory Safety Concern

This section contains the safety concern model corresponding to the hazard scenario involving the corruption of the temperature value by the system for the Chemical Factory Management System.

C.1 Processes

The relevant processes and the associated software components include the following:

- LANToBroadcast: Sensor Interface, Sensor Server
- BroadcastToDisplay: UI Manager, Display Block

The processes communicate with asynchronous broadcast messages.

C.2 Type Definitions

The following are the data types relevant to the processing and display of the vessel temperature.

The temperature is received and stored in a LAN Message object, as defined in package “LANMessage”:

```plaintext
type Temperature_S is range -1..1E4;
--| The temperature in celsius

type SensorUpdate is record
  InterpolatedState : InterpolationRange;
  TemperatureEstab : Temperature_S;
end record;
--| The contents of a sensor reading update

type SensorUpdateRange is range 1..MaxNumUpdates;

type SensorUpdateArray is array (SensorUpdateRange) of SensorUpdate;

type Object is record
  LMID : LANMessageID;
  LMSource : LANMessageSource;
  NumUpdates : SensorUpdateRange;
  Updates : SensorUpdateArray;
end record;
--| A LAN Message object
```

The temperature is parsed from the LAN Message object and stored in a Sensor object, defined in package “Sensor”. The Sensor object is part of an array of “SensorReading” records.

```plaintext
type SensorReading is
  record
    Data : Sensor.Object;
  end record;
```

The temperature is received and stored in a LAN Message object, as defined in package “LANMessage”:
The temperature is broadcast to the operator consoles as part of an array of Sensor objects, from package "Broadcast":

```
type Message is array (BroadcastRange) of Sensor.Object;
  --| The broadcast sensor information, with logical sensor code
```

The temperature is received at the operator console and stored in a Display Block object, from package "DisplayBlock":

```
type Object is record
  Active : Boolean;
  Available : Boolean;
  Interpolated : Boolean;
  PresentTemperature : DisplayTemperature;
  Display : DisplayID;
end Record;
```
C.3 Source Code

This section includes source code from the following software components:

- Sensor Interface
- Sensor Server
- UI Manager
- Display Block

C.3.1 Component: Sensor Interface

C.3.1.1 Overview

The "SensorInterface" component is an instantiation of the "Interface" generic:

```plaintext
package SensorInterface is new Interface
(SensorCode => SensorCodeEstab);
```

The temperature update is received through the "ReadLAN" subprogram. The temperature is then passed on to the "SensorServer" component through the "ProcessSensors" subprogram.

C.3.1.2 Assumptions

The normalization of the Sensor Quality is not extracted as it is not relevant to the processing of the temperature value.

C.3.1.3 Subprograms

ReadLAN procedure:

```plaintext
procedure ReadLAN(Message : in LM.Object)
is
  Readings : SensorServer.SensorReadingArray;
begin
  for I in LM.SensorUpdateRange range 1..Message.NumUpdates loop
    Readings(I).Data.SensorQuality
      := Normalize(Message.Updates(I).InterpolatedState);
    Readings(I).Data.SensorTemperature
      := ConvertTemperature(Message.Updates(I).TemperatureEstab);
    Readings(I).Code
      := Code;
  end loop;
  Sensorserver.ProcessSensors(Readings);
end ReadLAN;
```
C.3.2 Component: Sensor Server

C.3.2.1 Overview

The “SensorServer” component is an instantiation of the “Server” generic:

```plaintext
package SensorServer is new Server
  (SensorSourceCode => LM.SensorCode);
```

where “LM” is a renamed package:

```plaintext
package LM renames LANMessage;
```

The SensorServer component receives the temperature update through the “ProcessSensors” subprogram, which is then broadcast to the operator workstations with the Broadcast mechanism.

C.3.2.2 Assumptions

The Store and Broadcast mechanisms are not extracted as the lower level services are not part of the hazard analysis.

C.3.2.3 Subprograms

ProcessorSensors Procedure:

```plaintext
procedure ProcessSensors(Readings : in out SensorReadingArray)
in
  Message : BC.Message;
begin
  for I in Readings’Range loop
    Readings(I).Data.SID := CorrelateSensor(Readings(I).Code);
    if (SensorStore.IsFound(Readings(I).Data.SID, CurrentSensors) = TRUE)
      then
        SensorStore.Updated(Readings(I).Data.SID,
                           Readings(I).Data, CurrentSensors);
      else
        SensorStore.Add(Readings(I).Data.SID, Readings(I).Data,
                        CurrentSensors);
        SensorMonitor.Schedule(Readings(I).Data.SID);
      end if;
    Message(I) := Readings(I).Data;
  end loop;
  BC.Send(Message);
end ProcessSensors;
```

C.3.3 Component: UI Manager

C.3.3.1 Overview
The "UI Manager" component is an instantiation of the "Manager" generic:

```plaintext
class UIManager is new Manager
(ReadingRange => BroadcastRange);
<!-- Create a map of SensorIDs to Sensor Objects
```

The component receives the broadcast of the temperature updates with the "ReadBroadcast" subprogram. The component then updates the display with the "DisplayBlock" subprogram "UpdateTemperature".

### C.3.3.2 Assumptions

The COTS procedure calls from the "Screen" package are not extracted as COTS products are not a part of the hazard analysis. The Display Block subprograms "CorrelateVessel" and "UpdateInterpolated" are also not extracted as they are not relevant to the processing of the temperature value.

### C.3.3.3 Subprograms

#### ReadBroadcast Procedure:

```plaintext
procedure ReadBroadcast (Message : BroadcastMessage)
is
  VID : Vessel.VesselID;
  Display : DB.Object;
begin
  for I in Message'Range loop
    VID := CorrelateVessel (Message(I).SID);
    if (Screen.IsFound (VID, ScreenDisplays) = True) then
      Display := Screen.Find (VID, ScreenDisplays);
      if Message(I).SensorOperation = Sensor.Create then
        Display.Active := TRUE;
        Display.Available := TRUE;
        Display.PresentTemperature := ConvertToDisplayTemperature (Message(I).SensorTemperature);
        DB.UpdateTemperature (Display);
        DB.UpdateActivity (Display);
      elsif Message(I).SensorOperation = Sensor.Updated then
        Display.Available := TRUE;
        Display.PresentTemperature := ConvertToDisplayTemperature (Message(I).SensorTemperature);
        DB.UpdateTemperature (Display);
      elsif Message(I).SensorOperation = Sensor.Delete then
        Display.Active := FALSE;
        DB.UpdateActivity (Display);
      end if;
      if Message(I).SensorQuality = Sensor.MaxQuality then
        Display.Interpolated := FALSE;
      else
        Display.Interpolated := TRUE;
      end if;
    end if;
  end loop;
end ReadBroadcast;
```
C.3.4 Component: Display Block

C.3.4.1 Overview
The “UpdateTemperature” subprogram updates the display with the vessel temperature.

C.3.4.2 Assumptions
The COTS products “GRU” subprogram calls are not extracted as COTS products are not a part of the hazard analysis.

C.3.4.3 Subprograms
UpdateTemperature Procedure:

```plaintext
procedure UpdateTemperature(This : Object) is
begin
  if (This.Available = TRUE) then
    GRU.SetTemperature(This.Display, This.PresentTemperature);
  else
    GRU.SetAvailability(This.Display, This.Available);
  end if;
end UpdateTemperature;
```
Appendix D. Ada Model of Chemical Factory Safety Concern

This section contains the safety concern model corresponding to the hazard scenario involving the corruption of the temperature value by the system for the Chemical Factory Management System.

D.1 Type Definitions

```
package Hazard3Model
is
   -- Source Code Model of Hazard 3 ("An invalid vessel temperature is displayed")
   -- Assumption: Lower level services have been pruned

   -- Module LANMessage
   -- renamed LM
   -- Package LANMessage

   LM_MaxNumUpdates : constant := 11;
   -- Maximum number of updates in a LAN Message
   LM_MaxSensorCode : constant := 25;
   -- Maximum number for the sensor code ID
   type LM_SensorCode is range 1..LM_MaxSensorCode;
   -- Each sensor has its own internal ID
   -- The ID may not be unique
   type LM_Temperature_S is range -1..1E4;
   -- The temperature in Celsius from 0 to 1000
   type LM_InterpolationRange is (Interpolated1, Interpolated2, NotInterpolated);
   -- The sensor data quality
   LM_MaxLANMessageID : constant := 25;
   -- Each LAN message has a unique ID
   type LM_SensorUpdate is record
      InterpolatedState : LM_InterpolationRange;
      TemperatureEstab : LM_Temperature_S;
      SensorCodeEstab : LM_SensorCode;
      end record;
   -- The contents of a sensor reading update
   type LM_SensorUpdateRange is range 1..LM_MaxNumUpdates;
   type LM_SensorUpdateArray is array (LM_SensorUpdateRange) of LM_SensorUpdate;
   type LM_Object is record
      LMID : LM_LANMessageID;
      NumUpdates : LM_SensorUpdateRange;
      Updates : LM_SensorUpdateArray;
      end record;
   -- A LAN Message object

   -- Module Sensor
   -- Package Sensor

   Sensor_MaxSensorID : constant := 20;
   Sensor_MaxQuality : constant := 255;
   subtype Sensor_SensorID is Integer range 1..Sensor_MaxSensorID;
   subtype Sensor_Quality is Integer range 1..Sensor_MaxQuality;
   type Sensor_Operation is (Sensor_Upated, Sensor_Create, Sensor_Delete);
   type Sensor_Temperature is digits 6;
   type Sensor_Handle is record
```
```plsql
SID : Sensor_SensorID;
end record;

type Sensor_Object is
record
    SID : Sensor_SensorID;
    SensorOperation : Sensor_Operation;
    SensorQuality : Sensor_Quality;
    SensorTemperature : Sensor_Temperature;
end record;

--| Module SensorStore
--| - Instantiated generic Store within Server generic

type SensorStore_Object is
record
    SID : Sensor_SensorID;
    StoredObject : Sensor_Object;
end record;

procedure SensorStore_Add(K : in Sensor_SensorID; I : in Sensor_Object; S : in out SensorStore_Object);
procedure SensorStore_Updated(K : in Sensor_SensorID; I : in Sensor_Object; S : in out SensorStore_Object);
function SensorStore_IsFound(K : Sensor_SensorID; S : SensorStore_Object) return boolean;

--| Module SensorInterface
--| - Instantiated generic Interface within Server generic

procedure SensorInterface_ReadLAN(Message : in LM_Object);
--# global in out BroadcastMessage, CurrentSensors;
--| This procedure reads a sensor message off the LAN
--| The message is parsed and sensor readings extracted
--| The readings are then passed along to the sensor server

function SensorInterface_ConvertTemperature(T : LM_Temperature_S) return Sensor_Temperature;

--| Module SensorServer
--| - Instantiated generic Server within Interface generic

SensorServerSensorMonitorPeriodicity : constant := 1;
--| The check for sensor staleness period

type SensorServer_SensorReading is
record
    Data : Sensor_Object;
    Code : LM_SensorCode;
end record;
--| The sensor information, along with physical sensor code

type SensorServer_SensorReadingArray is array (LM_SensorUpdateRange) of SensorServer_SensorReading;
--| The input sensor information, with physical sensor code

type SensorServer_BroadcastMessage is array (LM_SensorUpdateRange) of Sensor_Object;
--| The broadcast sensor information, with logical sensor code

procedure SensorServer_MonitorSensorStaleness(SID : in Sensor_SensorID);
--| This procedure monitors a sensor reading
--| and deletes the reading if it is stale

procedure SensorServer_ProcessSensors(Readings : in SensorServer_SensorReadingArray);
--| This procedure processes the sensor readings
--| The logical SensorID is determined from the physical sensor code
--| If a sensor reading exists for this sensor, an update is generated
--| otherwise a create is generated
--| and a monitor for sensor staleness is initiated

--| Module BC
--| - Instantiated generic Broadcast within Server generic

procedure BC_Send(M : in SensorServer_BroadcastMessage);
```

---

164
procedure SensorMonitor_Schedule(I : in Sensor_SensorID);

package Vessel
package GraphicsRUs
module DataBlock
module Screen
module UIManager
D.2 Source Code

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
package body Hazard3Model
is
CurrentSensors : Sensorstore_Object;
--| The map of SensorIDs to Sensor Objects
-- | SensorInterface
ScreenDisplays : Screen_Object;

--| Pruned Subprograms
-- | The source code for these subprograms have been replaced with stubs
-- | The assumption is that these supporting services will work correctly
-- | Additional Assumptions:
-- | The datablock corresponding to the correct vessel will be identified
-- | The values for the Vessel ID and Sensor ID are assumed known

V1D_Constant : Vessel_VesselID := 23;
DB_Object_Constant : DBjObject;
Sensor_Sensorid_Constant : Sensor_Sensorid := 1;

--| Process: LANToBroadcast
--| Module: SensorStore
procedure Sensorstore_Add(K : in Sensor_Sensorid; I : in Sensor_Object;
S : in out Sensorstore_Object)
is
begin
null;
end Sensorstore_Add;

procedure Sensorstore_Updated(K : in Sensor_Sensorid; I : in Sensor_Object;
S : in out Sensorstore_Object)
is
begin
null;
end Sensorstore_Updated;

function Sensorstore_Isfound(K : Sensor_Sensorid;
S : Sensorstore_Object) return Boolean
is
begin
return True;
end Sensorstore_Isfound;

--| Module: SensorInterface
function Sensorinterface_Converttemperature{T : LM_Temperature_S)
return Sensor_Temperature
is
Newtemperature : Sensor_Temperature;
begin
Newtemperature := Sensor_Temperature(T);
return Newtemperature;
end Sensorinterface_Converttemperature;

--| Module: BC
procedure Bc_Send(M : in Sensorserver_Broadcastmessage)
is
begin

166
function UIManager_ReadBroadcast(UIManager_BroadcastMessage(M));
end Be_Send;

--| SensorServer

function SensorServer_Correlatesensor(Ssc : LM_Sensorcode)
return Sensor_SensorId
is
begin
return Sensor_SensorId_Constant;
end SensorServer_Correlatesensor;

procedure SensorServer_Monitorsensorstaleness(Sid : Sensor_Sensorid)
is
begin
null;
end SensorServer_Monitorsensorstaleness;

---| Process: BroadcastToDisplay
---| Module: Screen

procedure Screen_Updated(K : Vessel_VessellD; I : DB_Object; S : Screen_Object)
is
begin
null;
end Screen_Updated;

function Screen_Find(K : Vessel_VessellD; S : Screen_Object) return DB_Object
is
begin
return DB_Object_Constant;
end Screen_Find;

function Screen_IsFound(K : Vessel_VessellD; S : Screen_Object) return boolean
is
begin
return True;
end Screen_IsFound;

---| Module: GRUS - GRU

procedure GRU_SetTemperature(Display : GRU_ID;
Temperature : GRU_DisplayField)
is
begin
if (Temperature /= -1)
then
Put(Integer(Temperature));
Put_Line(" ");
end if;
end GRU_SetTemperature;

procedure GRU_SetAvailability(Display : GRU_ID; Status : Boolean)
is
begin
null;
end GRU_SetAvailability;

---| Module: UIManager

function CorrelateVessel (SID : Sensor_SensorID)
return Vessel_VessellD
is
begin
return VID_Constant;
end CorrelateVessel;
function ConvertToDisplayTemperature (T : Sensor_Temperature) return DB_DisplayTemperature
  is
  begin
  return DB_DisplayTemperature (T);
end ConvertToDisplayTemperature;

-- Module: Datablock - DB
procedure DB_UpdateActivity(This : DB_Object) is
  begin
  null;
end DB_UpdateActivity;

procedure DB_UpdateInterpolated(This : DB_Object) is
  begin
  null;
end DB_UpdateInterpolated;

-- Process: MonitorSensorStaleness
-- Module: SensorMonitor
procedure Sensormonitor_Schedule(I : in Sensor_Sensorid) is
  begin
  null;
end Sensormonitor_Schedule;

-- Subprogram Bodies from Critical Code Path
-- - LANToBroadcast Process
-- | - Receives sensor update and broadcasts temperature to workstations
-- | - BroadcastToDisplay Process
-- | - Receives temperature update at workstation and displays temperature

-- Process: LANToBroadcast
-- Module: SensorInterface
procedure Sensorinterface_Readlan(Message : in LM_Object) is
  Readings : Sensorserver_Sensorreadingarray;
  begin
  for I in LM_Sensorupdaterange range 1..Message.Numupdates loop
    Readings (I).Data.SensorQuality := Normalize (Message.Updates(I).InterpolatedState);
    Readings (I).Data.SensorTemperature := Sensorinterface_Converttemperature (Message.Updates(I).Temperatureestab);
  end loop;
  Sensorserver_Processsensors(Readings);
end Sensorinterface_Readlan;

-- Module: SensorServer
procedure Sensorserver_Processsensors(Readings : in Sensorserver_Sensorreadingarray)
is
Message: Sensorserver_Broadcastmessage;
begin
for I in Readings'Range
loop
Message(I) := Readings(I).Data;
Message(I).Sid := Sensorserver_Correlatesensor(Readings(I).Code);
if (Sensorstore_Isfound(Message(I).Sid, CurrentSensors) = True) then
    Message(I).Sensoroperation := Sensor_Updated;
    Sensorstore_UpdateMessage(I).Sid, Message(I), CurrentSensors;
else
    Message(I).Sensoroperation := Sensor_Create;
    Sensorstore_AddMessage(I).Sid, Message(I), CurrentSensors;
    Sensormonitor_ScheduleMessage(I).Sid);
end if;
end loop;
Bc_Send(Message);
end Sensorserver_Processsensors;

--- I Process: BroadcastToDisplay

--- Module: UIManager
procedure UIManager_ReadBroadcast(Message : UIManager_BroadcastMessage) is
vid: Vessel_VesselID;
Display : DB_Object;
begin
for I in UIManager_BroadcastMessage'Range
loop
vid := correlateVessel(Message(I).SID);
if (Screen_IsFound(vid, ScreenDisplays) = True) then
    Display := Screen_Find(vid, ScreenDisplays);
    if Message(I).SensorOperation = Sensor_Create then
        Display.Active := TRUE;
        Display.Available := TRUE;
        Display.PresentTemperature := ConvertToDisplayTemperature(Message(I).SensorTemperature);
        DB_UpdateTemperature(Display);
        DB_UpdateActivity(Display);
        elsif Message(I).SensorOperation = Sensor_Updated then
            Display.Available := TRUE;
            Display.PresentTemperature := ConvertToDisplayTemperature(Message(I).SensorTemperature);
            DB_UpdateTemperature(Display);
        elsif Message(I).SensorOperation = Sensor_Delete then
            Display.Active := FALSE;
            DB_UpdateActivity(Display);
        end if;
        if Message(I).SensorQuality = Sensor_MaxQuality then
            Display.Interpolated := FALSE;
            else
            Display.Interpolated := TRUE;
        end if;
        DB_UpdateInterpolated(Display);
        Screen_Updated(vid, Display, ScreenDisplays);
end loop;
end if;

end loop;
end UIManager_ReadBroadcast;

--| Module: SensorInterface

procedure DB_UpdateTemperature(This : DB_Object) is
begin
  if (This.Available = TRUE) then
    GRU_SetTemperature(This.Display, This.PresentTemperature);
  else
    GRU_SetAvailability(This.Display, This.Available);
  end if;

end DB_UpdateTemperature;

end hazard3model;
Appendix E. S Model of an ATM Safety Concern

This section contains a portion of the safety concern model corresponding to an ATM hazard scenario involving the corruption of the aircraft position. The code path segment involves the projection of a latitude and longitude onto the stereographic projection plane. The source code is similar to that found in an actual ATM system.
E.1 Type Definitions

<table>
<thead>
<tr>
<th>Ada Source Code</th>
<th>S Model</th>
</tr>
</thead>
</table>

**Ada Source Code**

```ada
type Gnomonic_Or_Stereographic_Object is
  record
    Radius : Unit.Metres13 := 0.0;  -- Conformal radius at the tangency point
    Sin_Phi : Syst.Float13 := 0.0;  -- sin(Tangency_Lat)
    Cos_Phi : Syst.Float13 := 0.0;  -- cos(Tangency_Lat)
  end record;

subtype Gnomonic_Object is Gnomonic_Or_Stereographic_Object;
subtype Stereographic_Object is Gnomonic_Or_Stereographic_Object;

type Lambert_Conformal_Object is
  record
    N : Syst.Float13 := 0.0;
    Major_Axis_Times_F : Unit.Metres13 := 0.0;
    Rho_0 : Unit.Metres13 := 0.0;
  end record;

type Utm_Object is
  record
  end record;

type Ups_Object is
  record
  end record;

type Orthographic_Object is
  record
    Tangency_Lat_Long : Geot.Latitude_Longitude :=
  end record;
```

**S Model**

```
% This type doesn't exist in the code.
% It was created to avoid conflicts among the names of the object data fields.
% e.g., Centre_Long exists for both Lambert_Conformal_Object and UTM_Object
: Projection_Base;

: Gnomonic_Object == Projection_Base;
: Lambert_Conformal_Object == Projection_Base;
: Stereographic_Object == Projection_Base;
: Utm_Object == Projection_Base;
: Ups_Object == Projection_Base;
: Orthographic_Object == Projection_Base;

Nil_Lambert_Conformal_Object : Lambert_Conformal_Object;

% Data fields of Gnomonic_Object and Stereographic_Object
Radius : Projection_Base -> Metres13;
Tangency_Lat : Projection_Base -> Latitude;
Tangency_Long : Projection_Base -> Longitude;
Sin_Phi : Projection_Base -> Float13;
Cos_Phi : Projection_Base -> Float13;

% Data fields of Lambert_Conformal_Object
Centre_Long : Projection_Base -> Longitude;
N : Projection_Base -> Float13;
Major_Axis_Times_F : Projection_Base -> Metres13;
Rho_0 : Projection_Base -> Metres13;

% Data fields of UTM_Object
Centre_Long : Projection_Base -> Longitude;

% Data fields of UPS_Object
% Tangency_Lat : Projection_Base -> Latitude;
% Tangency_Long : Projection_Base -> Longitude;

% Data fields of Orthographic_Object
Tangency_Lat_Long : Projection_Base -> Latitude_Longitude;
Sin_Tangency_Conformal_Lat : Projection_Base -> Float13;
Cos_Tangency_Conformal_Lat : Projection_Base -> Float13;
Two_E0 : Projection_Base -> Metres13;
Four_E0 : Projection_Base -> Metres13;
Four_E0_Squared : Projection_Base -> Metres13;
Tangency_Vector : Projection_Base -> DPos2D_Vector;
Ux : Projection_Base -> DPos2D_Vector;
Uy : Projection_Base -> DPos2D_Vector;
Base_E0 : Projection_Base -> Metres13;

% "kind" has been appended to the names of the constants of
% Kind, to avoid conflicts with the names of the
% constants of Proj_Object.
: Kind := UndefinedKind | GnomonicKind | Lambert_ConformalKind
| StereographicKind | Universal Transverse MercatorKind
```
Geot.Nil_Latitude_Longitude;
Sin_Tangency_Conformal_Lat : Syst.Float13 := 0.0;
Cos_Tangency_Conformal_Lat : Syst.Float13 := 0.0;
Two_E0 : Unit.Metres13 := Unit.Nil_Metres13;
Four_E0 : Unit.Metres13 := Unit.Nil_Metres13;
Four_E0_Squared : Unit.Metres13 := 0.0;
--Vector fields
Tangency_Vector : DPos2D.Vector := DPos2D.Nil_Vector;
Ux : DPos2D.Vector := DPos2D.Nil_Vector;
Uy : DPos2D.Vector := DPos2D.Nil_Vector;
E0 : Unit.Metres13 := 0.0;
end record;

end record;

Obj Nil_Object : constant Obj : =
{Projection_Kind => Undefined,
Translation => Proj.Nil_Xy_Coordinate_In_Metres};
### E.2 Source Code

**Ada Source Code**

```ada
procedure Compute_Stereographic_Xy
(The_Lat_Long : in Geot.Latitude_Longitude;
   With_Projection : in Stereographic_Object;
   The_Xy : out Proj.XyCoordinate_In_Metres;
   Is_Valid : out Boolean) is

   Sin_Phi : constant Syst.Float13 :=
      Syst.Elementary_Functions13.Sin
      (Syst.Float13 (The_Lat_Long.The_Latitude) *
       Unit.From_Degrees_To_Radians);

   Cos_Phi : constant Syst.Float13 :=
      Syst.Elementary_Functions13.Cos
      ((Syst.Float13 (The_Lat_Long.The_Latitude) -
       Syst.Float13 (With_Projection.Tangency_Longitude)) *
       Unit.From_Degrees_To_Radians);

begin
   declare
      Rk : constant Syst.Float13 :=
         Syst.Float13 (With_Projection.Radius) * 2.0 /
         (1.0 + With_Projection.Sin_Phi * Sin_Phi +
          With_Projection.Cos_Phi * Cos_Phi * Cos_Long);
   begin
      Is_Valid := True;
      The_Xy := (X =>
         Unit.Metres13*
         (Rk * Cos_Phi *
          Syst.Elementary_Functions13.Sin
          (Syst.Float13 (The_Lat_Long.The_Latitude) -
           Syst.Float13 (With_Projection.Tangency_Longitude)) *
           Unit.From_Degrees_To_Radians)),
      Y => Unit.Metres13*
         (Rk * (With_Projection.Cos_Phi * Sin_Phi -
               With_Projection.Sin_Phi * Cos_Phi * Cos_Long)));
   end;
exception
   when Constraint_Error =>
      Is_Valid := False;
```

**S Model**

```ada
% Projection Functions
% Compute_XyFromPOS2D calls the appropriate projection function
% The projection functions projects the latitude and longitude
% onto the projection plane defined by the given projection object
% i.e., Gnomonic, Stereographic, Lambert Conformal, UTM/UPS, Orthographic

subprogram Compute_Stereographic_Xy(  
   The_Lat_Long:Latitude_Longitude,  
   With_Projection:Stereographic_Object,  
   The_Xy:XyCoordinate_In_Metres,  
   Is_Valid:Boolean) :=

   (let Sin_Phi := Elementary_Functions13.Sin
      (The_Lat_Long.The_Latitude * From_Degrees_To_Radians) in
   (let Cos_Phi := Sqrt((1.0 - Sin_Phi * Sin_Phi)) in
    (let Cos_Long :=
      Elementary_Functions13.Cos
      (The_Lat_Long.The_Longitude -
       With_Projection.Tangency_Longitude) *
       From_Degrees_To_Radians) in
     (let Rk := (With_Projection.Radius * 2.0) /
      (1.0 + With_Projection.Sin_Phi * Sin_Phi +
       With_Projection.Cos_Phi * Cos_Phi * Cos_Long) in
      forall (Constraint_Error:Exception).
      if NOT(Constraint_Error) then
         (The_Xy.X = Rk * Cos_Phi *
           Elementary_Functions13.Sin
           (The_Lat_Long.The_Longitude -
            With_Projection.Tangency_Longitude) *
            From_Degrees_To_Radians)) AND
         (The_Xy.Y = Rk * (  
            With_Projection.Cos_Phi * Sin_Phi -
            With_Projection.Sin_Phi * Cos_Phi * Cos_Long)) AND
         (Is_Valid = True))  
   else  
      (The_Xy = Nil_XyCoordinate_In_Metres) AND
      (Is_Valid = False))
```
procedure Compute_Xy (The_Lat_Long : in Geo.Longitude; 
With_Projection : in Proj.Object; 
The_Xy : out Proj.Xy; 
Is_Valid : out Boolean) is 
begin 
A_Xy : Proj.Xy := Proj.Nil_Xy; 
Successful : Boolean := False; 
begin 
case With_Projection.Projection_Kind is 
when Undefined => 
Event_Report (This_Event => Coco_Event.Uninitialized_Projection, 
With_Severity => Ehi.Controlled_And_Fatal; 
In_Program_Unit => "Compute_Xy"); 
The_Xy := Proj.Nil_Xy; 
Is_Valid := False; 
when Gnomonic => 
Compute_Gnomonic_Xy (The_Lat_Long => The_Lat_Long, 
With_Projection); 
if Successful then 
The_Xy := (A_Xy.X + With_Projection.Translation.X, 
A_Xy.Y + With_Projection.Translation.Y); 
Is_Valid := True; 
else 
The_Xy := Proj.Nil_Xy; 
Is_Valid := False; 
end if; 
when Lambert_Conformal => 
Compute_Lambert_Conformal_Xy (The_Lat_Long => The_Lat_Long, 
With_Projection); 
if Successful then 
The_Xy := (A_Xy.X + With_Projection.Translation.X, 
A_Xy.Y + With_Projection.Translation.Y); 
Is_Valid := True; 
else 
The_Xy := Proj.Nil_Xy; 
Is_Valid := False; 
end if; 
when Stereographic => 
Compute_Stereographic_Xy (The_Lat_Long => The_Lat_Long, 
With_Projection); 
end Compute_Xy;
The_Xy := A_Xy;
Is_Valid := Is_Successful;
if Is_Successful then
The_Xy := (X => A_Xy.X + With_Projection.Translation.X,
Y => A_Xy.Y + With_Projection.Translation.Y);
Is_Valid := True;
else
The_Xy := Proj.Nil_Xy_Coordinate_In_Metres;
Is_Valid := False;
end if;
when Universal_Transverse_Mercator =>
Compute_Utm_Xy (The_Lat_Long => The_Lat_Long,
With_Projection => Universal_Transverse_Mercator,
The_Xy => A_Xy,
Is_Valid => Is_Successful);
if Is_Successful then
The_Xy := (X => A_Xy.X + With_Projection.Translation.X,
Y => A_Xy.Y + With_Projection.Translation.Y);
Is_Valid := True;
else
The_Xy := Proj.Nil_Xy_Coordinate_In_Metres;
Is_Valid := False;
end if;
when Universal_Polar_Stereographic =>
Compute_Ups_Xy (The_Lat_Long => The_Lat_Long,
With_Projection => Universal_Polar_Stereographic,
The_Xy => A_Xy,
Is_Valid := Is_Successful);
if Is_Successful then
A_Xy := (X => A_Xy.X + With_Projection.Translation.X,
Y => A_Xy.Y + With_Projection.Translation.Y);
The_Xy := A_Xy;
Is_Valid := (A_Xy.X in Rdps_Plane_Side) and then
(A_Xy.Y in Rdps_Plane_Side);
else
The_Xy := Proj.Nil_Xy_Coordinate_In_Metres;
Is_Valid := False;
end if;
end Compute_Xy;
procedure Compute_Xy (The_2D_Coordinate : in POS2D.Object;
With_Projection : in Proj.Object;
The_Xy : out Proj.Xy.Coordinate_In_Metres;
Is_Valid : out Boolean) is
Successful : Boolean := False;
begin
  case With_Projection.Projection.Kind is
    when Orthographic =>
      Compute_Orthographic_XyFromPOS2D
      (The_2D_Coordinate => The_2D_Coordinate,
       With_Projection => With_Projection.
       Orthographic,
       The_Xy => A_Xy,
       Is_Valid => Successful);
      if Successful then
        A_Xy := (X => A_Xy.X + With_Projection.Translation.X,
                 Y => A_Xy.Y + With_Projection.Translation.Y);
        Is_Valid := (A_Xy.X in Rdps.Plane.Side) and then
                     (A_Xy.Y in Rdps.Plane.Side);
      else
        The_Xy := Proj.Nil_Xy.Coordinate_In_Metres;
        Is_Valid := False;
  end if;
  when Undefined | Gnomonic | Lambert_Conformal | Stereographic |
  Universal_Transverse_Mercator | Universal_Polar_Stereographic => Compute_Xy
  (The_Lat_Long =>
   POS2D.Latitude_And_Longitude_Of
   (The_2D_Coordinate),
   With_Projection => With_Projection,
   The_Xy => The_Xy,
   Is_Valid => Is_Valid);
end case;
end Compute_Xy;