Ensuring the Inspectability, Repeatability and Maintainability of the Safety Verification of a Critical System

Ken Wong
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada V6T 1Z4
kwong@cs.ubc.ca

Jeff Joyce, Jim Ronback
Raytheon Systems Canada Ltd.
13951 Bridgeport Road
Richmond, BC, Canada V6V 1J6
{jjoyce, jronback}@mail.hac.com

Abstract

This paper proposes an approach to the safety verification of the source code of a software-intensive system. This approach centers upon the production of a document intended to ensure the inspectability, maintainability and repeatability of the source code safety verification. This document, called a "safety verification case", is intended to be a part of the overall system safety case. Although the approach was designed for large software-intensive real-time information systems, it may also be useful for other kinds of large software systems with safety-related functionality. The approach involves the construction of a rigorous argument that the source code is safe. The steps of the argument include simplifying the safety verification case structure by isolating the relevant details of the source code, and reducing the "semantic gap" between the source code and the system level hazards through a series of hierarchical refinement steps. Some of the steps in a process based on this approach may be partially automated with tool-based support. Current research and industry practices are reviewed in this paper for supporting tools and techniques.

Keywords: system safety engineering, software safety, software verification.

1. Introduction

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. In this paper, we focus on one specific step of the overall safety engineering process, namely, "safety verification". While methods for performing "front-end" steps such as "hazard analysis" are relatively well understood and widely used, methods for performing safety verification are less well established. Several methods have been proposed as a means of verifying source code with respect to identified hazards, but there are few published discussions of how these techniques could be applied to anything other than relatively small, isolated subsystems.

This paper outlines an approach which is intended to strengthen the "back-end" of the overall safety engineering process for large software-intensive information systems. This approach centers upon the production of a document that records the safety verification of the source code. We have introduced the term "safety verification case" to refer to this document. Our use of this term 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 source code safety verification. The organization of the safety verification case provides a basis for the definition of a software verification process. In turn, this process could employ an appropriate combination of the methods, techniques and tools surveyed in this paper. The organization of the “safety verification case” is strongly influenced by our experience with large software-intensive systems. This organization provides a foundation for performing safety verification even when the code implicated by a particular hazard may not be conveniently contained in a relatively small, isolated subsystem.

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 adherence to manual 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 which 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.

There are a number of challenges to developing a safety verification case for large software-intensive systems. Software-implemented functionality has increased dramatically in almost all safety-critical sectors [STO96]. This is particularly true for information systems that receive, process and display critical data. Air Traffic Management (ATM) systems, for example, are being built with about 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. All these factors must be accounted for in the development of the safety verification case.

The structure for the source-code safety verification case proposed in this paper is most appropriate for hazards which are caused by the "normal" execution of the software, as opposed to hazards which may arise in the wake of a general, systematic failure of the system. The occurrence of a hazard caused by the normal execution of the software may be the result of a number of factors. One possibility is that the hazard may be the direct result of a behaviour *required* by an "unsafe" combination of functional requirements. Ideally, the existence of an unsafe combination of functional requirements should be discovered long before source-code safety verification is performed. A second possibility is that the occurrence of a hazard may be caused by an incorrect implementation of the functional requirements, i.e., the functional requirements were "safe", but a defect was introduced at either the design or coding level. A third possibility is that the occurrence of the hazard is the direct result of design or coding decisions made by the developers which were "not inconsistent" with the functional requirements, i.e., while not a direct result of behaviour *required* by the functional requirements, the hazard was the result of behaviour which was not explicitly *excluded* by the functional requirements. This third possibility may be the result of the software developer working from functional specifications which are incomplete or ambiguous.

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 a relatively high level of abstraction.

The output of the process is:

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

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

The safety verification case structure proposed in this paper is motivated by the authors' experiences with the development of a large real-time information system with safety-related functionality. The system consists of approximately 700K lines of production source code with the equivalent to millions more in over a dozen different COTS components. The system possesses an object-oriented, distributed software architecture. The concept of a safety verification case, as described in this paper, is designed for safety-critical information systems of this size and complexity.

This paper investigates the development of a safety verification case for software-intensive systems with modern software architectures. Safety verification is discussed in Section 2. The need for a safety verification case is presented in Section 3. The challenges of building a safety verification case for large software-intensive systems with modern software architectures are noted in Section 4. The structure of the safety verification case is outlined in Section 5. Some techniques for implementing the safety verification case are given in Section 6, as well as aspects of the process which may be automatable. Section 7 provides a summary of the paper.

2. Safety Verification of the Source Code

An important step in the overall safety engineering process is the verification of the source code with respect to identified hazards. As discussed in Chapter 18 of Nancy Levenson's seminal textbook on software safety [LEV95], 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 1508 [IEC95], and the US military standard, MIL-STD-882C [DOD93]. For example, the IEC 1508 defines a software safety validation phase which is designed to ensure that the safety-related software is compliant with the software safety requirements specification.

The discussion of safety verification within this paper 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 paper.

2.1 Safety as a Distinct Quality

Safety is a matter of ensuring that "bad" things do not happen, i.e., accidents which are unplanned events resulting in damage to property, death or injury. The safety engineering process is defined in terms of system hazards which are conditions of the system, along with certain environmental conditions, that will cause an accident [LEV95]. 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 is to display the "Present Position Symbol"

(PPS), the icon representing the aircraft, in an incorrect position [ELL96]. Figure 1 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. An incorrectly displayed PPS could lead an air traffic controller to give instructions to pilots that may contribute to a loss of separation.

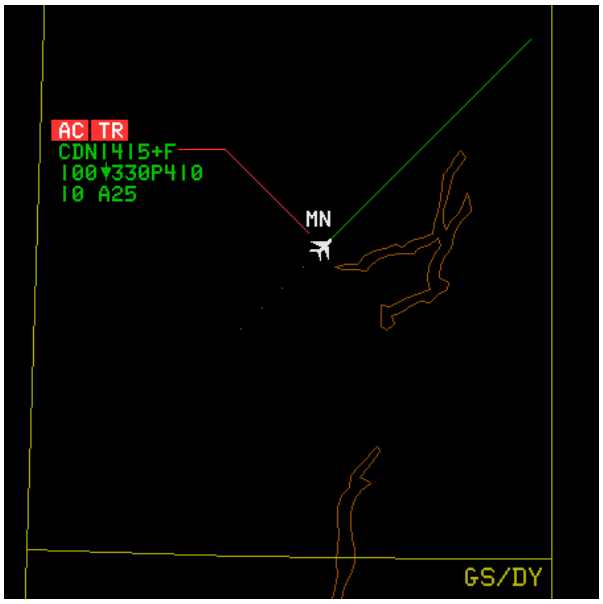

Figure 1: Part of an air traffic controller's "situation display" showing the "Present Position Symbol" (PPS) for an aircraft, i.e., the aircraft silhouette which indicates present position and heading.

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 which have been found to be hazardous, but are not explicitly excluded by the functional requirements.

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[1]:

> *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 commonalties 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 tradeoff is the realization that the safest aircraft are those that never leave the ground. This would certainly avoid the incorrect display of aircraft position. A less extreme, more realistic example, is the use of run-time method binding and 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.

[1] Page 182 of [LEV95].

In general, a system may satisfy the contractual requirements, be reliable, and yet be unsafe. There may be unintended functionality in the system, not specified by the requirements, that could be hazardous. The requirements themselves may not rule out or may even specify hazardous software behavior. As a result, safety verification must focus on analyzing the system with respect to system hazards.

2.2 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 involves 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 which are directly related to an identified hazard.

Similarly, it is reasonable to expect that the safety engineering process for a software-intensive system would include close scrutiny of source code specifically with respect to identified hazards. It is not enough to be 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.*[2]

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.

3. Safety Verification Case

The safety verification case is part of the overall system safety case which provides input into the decision to certify a system as "fit for use" before it is put into operational use. For example, many safety standards require a final detailed report on the safety of the system [STO96].

The safety verification case can play an important role in the life-cycle support process for the system. For example, the system will require periodic safety audits to ensure that the operational level of safety is maintained [LEV95]. In particular, it will be necessary to evaluate the impact on safety of any minor changes that may have occurred in the system, operations or the environment.

As well, the safety verification case can form the basis of the evaluation of safety when parts of the system are reused. For example, the implementation of the system may be adapted to produce a "next generation" of the system or to develop a family of systems as a generalization of the first existing member of the family.

In general, the people using the safety verification case will be different than the original developers. These people include the representatives of the authority that authorizes use of the system, system maintainers and system developers intending to reuse the software in new systems.

3.1 Repeatable Details

It will be necessary to regenerate the safety verification case after system modification or adaptation for a new use. The safety verification case should be captured with sufficient detail and precision to avoid having to repeat

[2] As quoted on page 489 of [LEV95].

as much as possible the efforts that went into constructing the original document. This will minimize the schedule, efforts and cost of developing the new safety verification case.

There is also a danger that budget or time constraints will not allow this process to be carried out to the same degree of thoroughness as the original effort. In particular, the process may overlook implicit or buried assumptions made during the original analysis. The ARIANE 5 failure [ARI96] is an example of software reuse where the original environmental assumptions were not specified for the new system. The catastrophic failure of this satellite resulted in direct costs of approximately $370 million. The Therac-25 accidents [LEV95] is an example of a system which reused software from an earlier system and replaced hardware interlocks with new software functions, without careful consideration of the impact on safety. These factors contributed to lethal overdoses of six people.

This danger of overlooking critical assumptions can be largely avoided if the thought process underlying the original safety analysis is well documented. The details should be sufficient for a new team of safety analysts to systematically regenerate the safety verification case. If a change has been made to the system which has an impact on safety, or if an assumption about the operating environment has changed, then this should be revealed to the team regenerating the safety verification case.

3.2 Rigorous Argument

The safety verification of a system will almost certainly require the use of "rigorous argument" where the term "argument" is used here in the dictionary sense of "a coherent series of statements leading from a premise to a conclusion".

The ability to construct a rigorous argument is a fundamental skill of many other professions where a high degree of certainty is required. For instance, a lawyer must build a case using rigorous argument to convince the jury that a particular outcome of a trial is "safe". The rigor of the argument laid out by the lawyer is ensured, in part, by judicial rules and by scrutiny of the opposing lawyer. Furthermore, the argument is recorded in the form of court transcripts to allow for its review at some future date.

Similarly, the use of rigorous argument to verify the safety of a software system can be compared to the way that mathematicians may convince themselves (and one another) of the validity of mathematical conjecture. As suggested in a recently published textbook on safety-critical computer systems:

> *A safety case must include a rigorous argument for the safety of the system, and must demonstrate that it satisfies all its safety requirements. This will involve numerous steps that in some ways resemble the components of a mathematical proof. Each stage of this 'proof' must be carefully justified and any assumptions made explicitly. Any unjustified assumptions represents flaws in the safety argument, and hence the safety case.* [3]

Both of these kinds of rigorous argument have the important property that they can be recorded in a form that can be understood, reviewed, repeated (or re-constructed) by others who were not originally present in the courtroom listening to the courtroom lawyer, or who were not peering over the shoulders of the mathematician as a proof of the mathematical conjecture is worked out.

For a safety verification case, elements of the rigorous argument include (see Figure 2):

- The basic *claim* is an adequately mitigated hazard.
- The claim will be supported by statement of *facts* taken from the system requirements, architecture and the source code, as well as the system development and safety engineering processes.

[3] Page 365 of [STO96].

- The argument may be based on certain *assumptions* about the external environment, the compilation of the source code, the underlying operating system and hardware, and the COTS products.

- The assumptions and facts are linked to the claim through a series of *inference* steps.

The potential complexity of the rigorous argument means it may be necessary to hierarchically organize the argument in terms of a set of sub-claims. Each sub-claim and supporting argument can be documented separately. The sub-claims together support the basic claim.

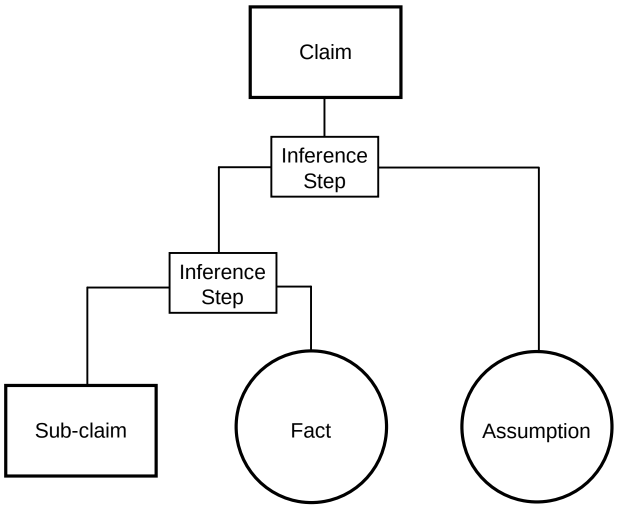

Figure 2: Rigorous Argument

The record of the argument must also be organized in a systematic fashion. For instance, each sub-claim, assumption and fact can be given a unique identification (e.g., “Assumption 2.4.2”). This is analogous to the way that physical evidence is itemized during a trial, as well as the way that a mathematician will usually employ a numbering scheme for assumptions, lemmas and other intermediate products in the process of writing out a proof.

Different types of inference rules can be used to carry out the argument. At one extreme the argument could consist of a rigorous, machine-checked mathematical proof using arithmetic and logical inferences. However, less formal, but rigorous forms of reasoning are also effective. The types of arguments can be classified as deterministic, probabilistic and qualitative [BIS98]. These arguments include, for example, testing, causal inferences, code inspection and demonstration of system compliance with recognized software standards. In general, the verification argument can consist of a mixture of formal/informal, manual/tool-assisted and forward/backward forms of rigorous reasoning.

4. Challenges to Developing a Safety Verification Case

There are a number of challenges to constructing a safety verification case for large software-intensive systems. One challenge is relating the hazard to the source code, as the hazard will usually be expressed in much different terms than the source code. Another challenge is that the hazard-related code will typically be found in a number of different software components.

4.1 The “Semantic Gap”

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

Sometimes there may be a high degree of “lexical similarity” between the terminology used to define the system hazard and the identifiers used as names of elements of the source code implementation. For example, an ATM term such as “PPS” may also appear in the source code as part of the name of an identifier. But in other circumstances, there may not be much lexical similarity between the definition of a hazard and the source code implementation of functionality directly related to this hazard. This may be prevalent in the case of a system which has been developed with an emphasis on abstraction and reuse. For example, it is possible that many of the source lines of code that are directly involved in generation of a PPS on the situation display are contained in parametrized class templates and do not contain any lexical "clues" (e.g., use of "PPS" as part of an identifier) of their relevance to the hazard. Obviously, this minimizes the usefulness of basic text searching tools since the use of a tool such as Unix "grep" to search for lines of source code that contain "PPS" may find very little of the source code which has a direct relevance to a hazard that concerns the PPS. But beyond the use of simple tool-based search strategies, the lack of lexical clues compounds with other sources of complexity when using manual methods to search for hazard related code.

In addition to the challenge of mapping the definition of a hazard to data elements of the source code such as global variables, sub-program parameters, constants, local variables and other elements of the source code, it is also necessary to refine abstract properties such as "incorrect position" into concrete relationships. Ultimately, these concrete relationships will be defined in terms of combinations of simple arithmetic and logical comparisons between data elements, e.g., "is equal to", "is less than".

4.2 The “Long Thin Slice”

Safety verification of the source code requires the identification of the critical code paths that lead to a hazardous software output. The code path can be viewed as a sequence of steps where each step corresponds to an executable line of source code. For software-intensive information systems, a typical execution sequence includes modules from a number of different components involving the user interface, application domain and hardware interfaces. However, the sequence may involve only a small amount of code from each component, forming a “long thin slice” of software.

For systems with object-oriented architectures, each software component in the long thin slice 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. To locate a given step in the critical code path involves searching for the corresponding executable code statement among all the non-executable code statements.

To understand a given step in the critical code path 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. For example, a type declared in one of the upper layers of the software architecture could be 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.

A developer who has been immersed for many months or years in the development of the software may be capable of tracing a sequence of actions across a "long thin slice" of code without looking up every bit of detail from various source files. But others, besides the original developers, must also be able to follow the same sequence of actions in order to conduct their own safety assessment of the system.

5. Structure of the Safety Verification Case

A structure for the safety verification case for the system source is proposed. The structure is based on a process which bridges the semantic gap between the system level hazards and the source code in a series of refinement steps, and which isolates the relevant code details.

As shown in Figure 3, the main parts of the safety verification case are:

1. step-wise refinement of the hazard into one or more source code level “safety verification conditions” (SVCs);

2. step-wise extraction of models of one or more critical code paths;

3. step-wise verification that each of the source code level SVCs (obtained from refinement documented in Part 1) is satisfied by the model of the source code implementation (obtained as a result of the extraction documented in Part 2).

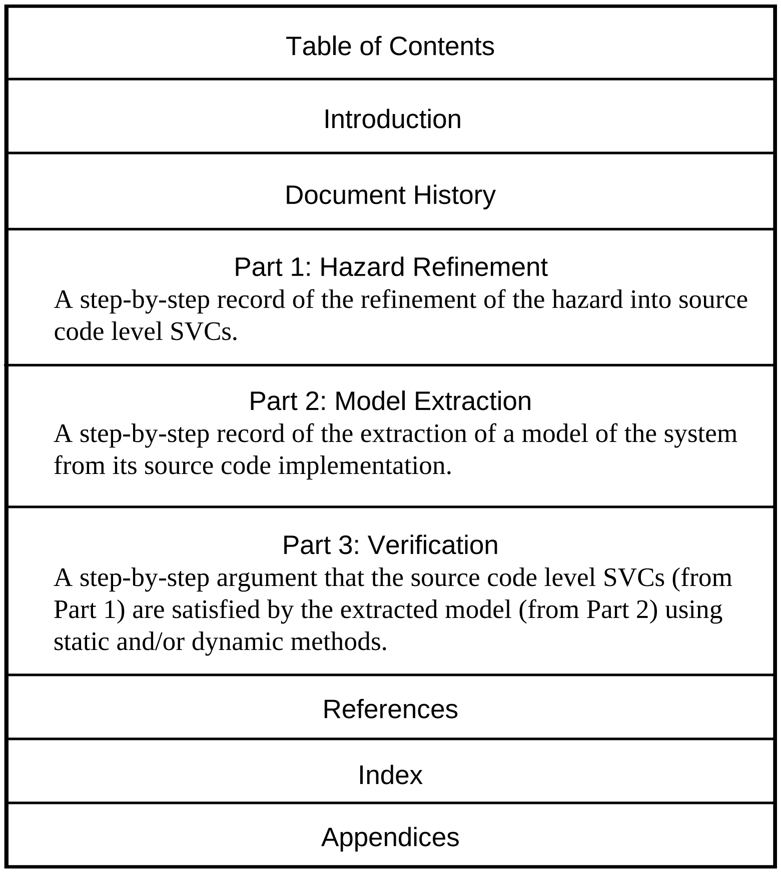

Table of Contents

Introduction

Document History

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

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

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

References

Index

Appendices

Figure 3: Structure of the Safety Verification Case

The structure of the safety verification case is described in greater detail in the rest of Section 5. Techniques for implementing the safety verification case are presented in Section 6.

5.1 Part 1: Hazard Refinement

The "Hazard Refinement" part of the safety verification case contains a set of source code level "Safety Verification Conditions" (SVC) along with a record of the execution of the process used to obtain these SVCs as a refinement of the hazard definition. As explained later in Section 6.1, this refinement may be performed in an *ad hoc* manner or it may involve a more rigorous approach such as an approach based on Fault Tree Analysis (FTA) or a related technique. In either case, both the execution of the process and its results must be recorded in a form which would allow the process to be repeated.

Each of the source code level SVCs derived from the definition of the hazard will be recorded in this part of the safety verification case and given a unique identification which may be used elsewhere in the safety verification case to reference the SVC.

An example developed by Wong [WON98a] shows how a system level hazard may be refined into a set of source code level SVCs. As illustrated by Wong's example, the refinement of a hazard into a set of SVCs will yield a combination of SVCs which may be:

- 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 source code level SVCs may be expressed as informal English statements. Alternatively, some of the SVCs may be expressed in a "codified form" using a notation that can be parsed by a software tool. As demonstrated in [WON98a], the use of a codified form may be effective as a means of ensuring the precision of the SVCs as well as avoiding ambiguity and other specification related problems associated with the exclusive use of informal English. The specification of SVCs in codified form such as the SPARK annotation notation [BAR97] may also allow the use of verification tools such as SPARK Examiner [BAR97].

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

5.2 Part 2: Model Extraction

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

5.2.1 Hazard Scenarios

A hazard may occur in a variety of different ways. The various ways in which a hazard may occur is revealed by the results of the hazard refinement contained in Part 1 of the safety verification case. Each occurrence involves a sequence of events within the software system called a "scenario". These events occur within or between a set of software subsystems which we refer to as the "critical code path" for the scenario.

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 advantageous to extract more than one model of the source code level implementation of the system.

5.2.2 Representation of the Model

A variety of different formats may be used to represent the model of a critical code path extracted from the source code. One possibility is a textual description of the model annotated by an object scenario diagram [BOO94] and fragments of actual source code. Another possibility is the representation of the model in an executable format, i.e., in the same programming language used for the actual implementation or possibly in a closely related programming language. For example, the model of a critical code path for a system implemented in Ada might be represented in the SPARK programming language [BAR97]. SPARK is a subset of Ada which is amenable to verification with tools such as SPARK Examiner [BAR97]. While the production of the textual description may require less initial effort, the production of an executable model is more effective over the long term. An executable model would allow the use of software tools to partially automate the verification task by means of dynamic methods (testing) or static methods (e.g., code verification).

5.2.3 Model Properties

The model should be **conservative** 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 model should be **complete** in the sense that the model contains enough information to unambiguously interpret every executable statement in the model.

To the extent that the model is represented by fragments of source code, the model should be **tractable** in the sense that the ratio of executable lines of code to non-executable lines should be “reasonable”. Furthermore, the number of different places in the source code required to understand a single line of executable code should also be reasonable. 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.

5.2.4 Recording the Execution of the Extraction Process

The extraction of this model can proceed in either a bottom-up or top-down fashion. In either case, the execution of the process must be recorded in a form which would allow the process to be repeated. Either approach involves processes which can be recorded.

5.2.4.1 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 tracing backwards through the code by incrementally searching for the cause of an effect. Forward searching involves tracing a cause forward to its possible effects. For a large complex system, it is likely to be necessary to "prune" some branches of the search by introducing assumptions about the functionality of peripheral aspects of the system.

The simplest method of recording the extraction of a model by means of either forward or backward searching would be a textual description of 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."

A more rigorous representation of the extraction could be recorded in the form of an acyclic graph. In this form, each node in the web 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. 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 which may directly change the value of “X”. Additionally, there would be an arc to the line of source code which 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. There exist, however, computer-based, interactive visualization tools which 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 automatically generate source code for the model that could be compiled and executed.

The representation of the search of the source code in the form of a 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.

5.2.4.2 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 reduces 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 model of the implementation of the system.

The transformations may be performed at a variety of different levels of granularity and rigor. Section 6.2 describes five basic kinds of transformations. 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.

5.3 Part 3: Verification

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

The basic claim is the adequate mitigation of the hazard. The argument can be organized in a hierarchical fashion, involving sub-claims at the system and design level. These sub-claims will include the SVCs at those levels. Each SVC will be given a unique identification in Part 1.

Both Parts 1 and 2 will contain assumptions and statements of fact about the environment, system, design and source code. Each statement will also be given a unique identification in Parts 1 and 2.

The argument will consist of a series of steps linking the statement of facts and assumptions to the claims or sub-claims. The sub-claims, statements of facts and assumptions will be referenced by their unique identification assigned in Parts 1 and 2. The details of these steps will depend on the type of argument that has been employed. For example, if the inference step involves testing, the test plans, procedures and results will all be documented.

6. Processes, Techniques and Tools

Based on the description given in Section 5 of the structure of the safety verification case as a deliverable document, this section outlines a process for the development of this document. This section also describes a variety of techniques and tools which may be used as part of this process.

Figure 4 shows a process flow diagram that represents the development of the safety verification case. The left-hand side of this diagram represents the derivation of source code level SVCs from the definition of a hazard. The right-hand side represents the extraction of a model for one or more critical code paths for the hazard. At the bottom of the process flow diagram, the outputs of the hazard refinement and of the model extraction are used as inputs to the verification process. The output of the process is the safety verification case.

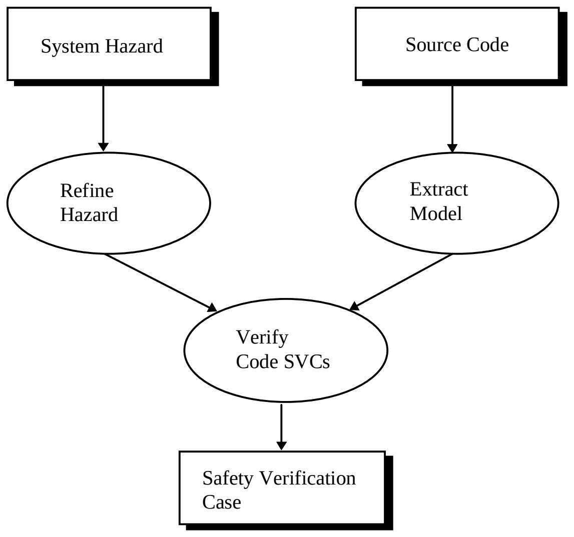

Figure 4: Safety verification process.

6.1 Techniques and Tools for Hazard Refinement

The derivation of the source code level SVCs may be performed in an *ad hoc* manner relying on engineering judgment to determine what should be verified in the source code. However, undue confidence should not be placed in the effectiveness of engineering judgment for discovering all the SVCs necessary for the mitigation of a hazard. Wong [WON98a] describes an example where some of the SVCs derived from the definition of a system hazard were unlikely to have been discovered if only *ad hoc* methods had been used.

Aside from following a purely *ad hoc* approach, the refinement of a hazard definition into a set of SVCs is likely to involve a form of Fault Tree Analysis (FTA) or a related technique. FTA is often used during hazard analysis to reveal some possible hazard causes. There are other related techniques which can be used to identify combinations of assumptions, causes and other conditions that can provide a starting point of a rigorous argument that the hazard has been mitigated.

FTA begins with the identified hazard and then works backward to uncover all possible causes. The analysis begins with the hazard as the “top event” and then determines the intermediate events that cause the hazard which are combined using logical operations such as “AND” and “OR”. The analysis can be performed in a top-down fashion beginning at events at the system level, to events at the software component level, down to events at the source code. SVCs can be constructed to mitigate the hazard causes at each level. The analysis continues until the “root” hazard causes are uncovered or until adequate mitigation has been established for an intermediate cause.

Software safety requirements, for example, were derived from a FTA of a Magnetic Stereotaxis System [KNI93]. First, the system causes of the hazard were identified. Next, the software functions intended to reduce the likelihood of the hazard occurring were examined. Finally, the consequences of the failure of these software functions were determined, leading to a set of constraints on the functions.

Another method for generating SVCs has been suggested which is based on FTA analysis and guided by rigorous logical reasoning [WON98a]. The method is similar to 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 the possible ways in which this condition might have arisen. Like Software Fault tree Analysis (SFTA) [LEV91], a style of reasoning known as “proof by contradiction” is used to show that each disjunctive branch of the tree leads to a logical contradiction. However, Wong’s method is not tied to the semantics of a particular programming language and moreover, it is intended to serve as a method for the generation of SVCs in preparation for the actual verification -- unlike SFTA which is generally tied to the semantics of a particular programming language and is meant to be used directly to perform the verification task.

6.2 Techniques and Tools for Model Extraction

Section 5.2 briefly described two different approaches to the extraction of a model of a critical code path from the source code implementation. A bottom-up approach involves either a forward or backwards search of the source code. A top-down approach involves the sequential transformation of a copy of the source code implementation into the desired model.

6.2.1 Bottom-Up Extraction

There is a variety of tool support which may be used to expedite the process of searching source in either a forward or backwards direction.

Software development environments typically support various forms of code cross-referencing. The Rational APEX Ada editor, for example, offers interactive hypertext Ada browsing [RAT98a]. 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 execution sequences and uncovering its dependencies on other lines of source code. The Ada Analyzer [RAT98b], for example, traverses compiled Ada units, locates constructs according to some specific selection criteria, and places the relevant information in a hypertext table [RAT98b]. Data flow analyzers [DAI94] could be used to directly trace the critical data flows. The source code associated with a data flow can be extracted with a program slicer. Most program slicers, however, are research tools designed for “toy” languages [HOF95]. There are some commercial tools, such as the McCabe Slice Tool, which provides some support for program slicing [STS97]. Program slicing for object-oriented programs is particularly difficult due to the run-time binding of methods and complex object-interaction graphs. Program Explorer performs dynamic object slicing for software written in C++ [LAN97].

Another approach would involve the use of a tool that queries the semantic and syntactic information available in a compiler environment [COO97]. In the case of Ada environments, the Ada Semantic Interface Specification (ASIS) [ASI97] 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. Tools can be developed, for example, to analyze the differences between subsequent system builds such as any changes in the system parameters [CEL97].

Software visualization tools [BAL96] for creating representations of source code structure and runtime behavior may also be useful in searching code.

6.2.2 Top-Down Extraction

The top-down extraction of the model will involve a series of transformation steps isolating the relevant details of the critical code paths, while eliminating the irrelevant code.

Wong [WON98b] describes five basic kinds of transformation steps:

1. **Flatten.** Each executable statement in the critical execution sequences will depend on other lines of source code, which may be located in a number of different components which are organized into class hierarchies. These hierarchies should be "collapsed" to bring together the relevant lines of code and to reduce the amount of "non-executable" code.

2. **Fillet.** The critical code path, along with any supporting source code, is isolated.

3. **Filter.** The lower-level services (e.g., which belong to the lower layers of the architecture) are replaced with some simplifying assumptions. These services may then be analyzed separately.

4. **Partition.** The basic software components of the model are identified. For example, the model can be partitioned into "functional blocks" [WON98b] that execute in different processes or threads (light-weight processes).

5. **Translate.** The source code is represented in a simpler form involving, for example, a textual description of the model. Alternatively, the model could be represented in executable form expressed in the same programming language as the original source code or in a closely-related programming language.

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.

Some of the tools described in the bottom-up extraction of the model can be used to support the top-down extraction. Searching tools can be used to help carry out the filleting and to help identify the "functional blocks". Tools could be custom-built to automatically carry out the flattening and translation steps.

6.3 Approaches to Verifying the SVCs

The inference steps will include a combination of:

- **Dynamic analysis** involves execution of the code. This includes testing and simulation.

- **Static analysis** 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 (e.g., SPARK Examiner [BAR97]).

System, integration or unit testing can be performed to verify the source code level SVCs. The type of testing needed will depend on the type of software component constructed during the refinement of the source code. The component may, for example, involve a number of software subsystems.

In addition to standard testing techniques, violations of the system level SVCs can be examined through software fault-injection [VOA97]. In particular, fault injection is useful for investigating the effects of anomalous events caused by human factor errors or external failures. These events are simulated by either modifying the code or forcing the state of the code to be modified when the code executes.

It may be possible to exhaustively test certain source code level SVCs. For example, a safety property involving an object location algorithm used in the imaging subsystem of a Magnet Stereotaxis System (MSS) was verified by exhaustive testing [KNI96].

In general, it will not be possible to completely verify SVCs through testing. The large number of possible software inputs and paths means it is not feasible to exhaustively test them all [PAR90]. As a result, testing can only provide a limited degree of assurance that the system level SVCs are satisfied.

The limitations of testing mean that, when feasible, a static analysis of the source code should also be performed. The chief limitation of static analysis methods are that they tend to be labor-intensive and expensive.

The simplest 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 the correctness of the code [SCO94], or are hazard-analysis techniques that were not designed for software [IPP95]. It is important that the focus of the adopted inspection method should be the verification of the source code level SVCs.

Software Fault Tree Analysis (SFTA) [LEV91] 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 have typically been applied to relatively small systems, and it is not clear how well the technique will scale. It may be feasible to use SFTA to help verify the representation of the hazard-related code with respect to source code level SVCs.

There are mathematically-based program verification techniques that exist which enable the proof that a program meets a formal, i.e., mathematical, specification [BOW93]. For example, 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 [GER94]. However, the verification efforts were extremely labor intensive for relatively small programs.

It has been suggested that formal verification of select safety properties is a more reasonable application of formal verification [RUS95]. For example, there exist code verification tools such as SPARK Examiner [BAR97] which verifies a program with respect to a set of code assertions (i.e., pre- and post-conditions). One possible approach to verifying the source code level SVCs is to first refine them into a set of verifiable code assertions. If the source code is then refined into a set of software components expressed in the SPARK Ada subset, it may then be possible to verify the components with respect to the code assertions using the SPARK tool set.

7. Summary

In this paper, we introduce the term “safety verification case” to refer to a deliverable document that, as part of the overall safety case, records the safety verification of the source code. The safety verification case is documented in sufficient detail so that people other than the original developers can inspect, maintain or repeat the verification. These people include certification authorities, system maintainers and system developers intending to reuse the software in the next generation of the system.

Large software-intensive information systems with modern architectures present a number of challenges to the development of the safety verification case. One challenge is bridging the “semantic gap” between the source code and the high-level language used to express the system hazards. Another challenge is the capture and documentation of the “long thin slice” of source code associated with each hazard.

This paper has proposed a structure for the safety verification case which addresses these challenges. The structure is particularly appropriate for hazards involving the display of critical data. These type of hazards are typically found in real-time information systems. The structure has three main parts which record the refinement of the identified system hazards into source code level “safety verification conditions” (SVCs), the extraction of models of the critical code paths and the verification of the code models with respect to the source-code level SVCs.

The paper has described how methods ranging from *ad hoc* engineering judgment to rigorous "proof by contradiction" techniques may be used to refine identified system hazards into source code level SVCs. The paper has also described both a "bottom-up" and a "top-down" approach to the extraction of a model of a critical code path. Both dynamic and static approaches to the verification step of the process were also described. In support of these approaches, the paper has identified a variety of specialized techniques and tools that may reduce the amount of effort required to develop the safety verification case as well as increase the accuracy, rigor and thoroughness of its content.

8. Acknowledgments

This work was partially supported by B.C. Advanced Systems Institute, Raytheon Systems Canada Ltd, and MacDonald Dettwiler. This work is a component of the formalWARE university-industrial collaborative project [FOR98].

9. References

ARI96 ARIANE 5 Inquiry Board, "ARIANE 5 Flight 501 Failure Report by the Inquiry Board", Paris, July 1996.

ASI97 Association for Computing Machinery (ACM) Special Interest Group in Ada (SIGAda) ASISWG/ISO/IEC JTC 1 SC 22/WG 9 ASISRG, ASIS Working Draft, Version 2.0.p, 25 August 1997.

BAL96 Thomas Ball and Stephen G. Eick, "Software Visualization in the Large", *IEEE Computer*, April 1996.

BAR97 John Barnes, "High Integrity Ada The SPARK Examiner Approach", Addison Wesley Longman Ltd, 1997.

BIS98 Peter G. Bishop and Robin E. Bloomfield, "A Methodology for Safety Case Development", in Safety-critical Systems Symposium, Birmingham, UK, February 1998.

BOO94 Grady Booch, "Object-Oriented Analysis and Design with Applications (Second Edition)", Benjamin/Cummings Pub. Co., Redwood City, California (1994).

CHA90 Stephen S. Cha, "Safety Verification on Software Design", Ph.D. Dissertation, ICS, Department of California, Irvine, June 1990.

BOW93 Jonathan Bowen and Victoria Stavridou, "Safety-Critical Systems, Formal Methods and Standards", *IEE/BCS Software Engineering Journal*, 8(4):189-209, July 1993.

CEL97 Celier, Vincent M., Drasko Sotirovski, Christopher J. Thompson, "Code-Data Consistency in Ada"; in *1997 Ada-Europe International Conference on Reliable Software Technologies, London, U.K. Proceedings*, Springer Lecture Notes in Computer Science #1251, pp. 209-216, June 1997.

COO97 C. Daniel Cooper, "ASIS-Based Code Analysis Automation", *Ada Letters*, Volume XVII, No. 6, November/December 1997.

DAI94 Gregory T. Daich, Gordon Price, Bryce Raglund, Mark Dawood, "Software Test Technologies Report", Test and Reengineering Tool Evaluation Project, Software Technology Support Center, August 1994.

DOD93 Department of Defence, "Military Standard 882C: System Safety Program Requirements", 1993.

ELL96 Bruce Elliott and Jim Ronback, "A System Engineering Process For Software-Intensive Real-Time Information Systems, in *Proceedings of the 14th International System Safety Conference*, Albuquerque, New Mexico, August 1996.

EHR94 Daniel H. Ehrenfried, "Static Analysis of Ada Programs", *Ada Letters*, Volume XIV, No. 4, July/August 1994.

FOR98 http://www.cs.ubc.ca/formalWARE/

GER94 Susan Gerhart, Dan Craigen and Ted Ralston, "Experience with Formal Methods in Critical Systems", *IEEE Software*, January 1994.

HAN96 Kirsten Mark Hansen, “Linking Safety Analysis to Safety Requirements - Exemplified by Railway Interlocking Systems”, Ph.D. Thesis, Technical Universi ty of Denmark, Department of Information Technology, August 1996.

HOF95 Tommy Hoffner, “Evaluation and comparison of program slicing tools. Technical Report”, LiTH-IDA-R-95-01, Department of Computer and Information Science, Linkping University, Sweden, 1995.

IEC95 International Electrotechnical Commission , “Draft International Standard IEC 1508: Functional Safety: Safety Related Systems”, Geneva, 1995.

IPP95 Laura M. Ippolito and Dolores Wallace, “A Study on Hazard Analysis in High Integrity Software Standards and Guidelines”, NISTIR 5589, National Institute of Standards and Technology, January 1995.

KNI93 John C. Knight, Darrell M. Kienzle, “Preliminary Experience Using Z to Specify a Safety-Critical System”, in *Proceedings of the Z User Workshop*, Edited by J.P. Bowen and J.E. Nicholls. Published by Springer Verlag, 1993.

KNI96 John C. Knight, Kevin G. Wika, and Shannon Wrege , “Exhaustive Testing As A Verification Technique”, Submitted to the *International Symposium on Software Testing and Analysis* , January 8-10, 1996 (ISSTA 1996).

LAN97 Danny B. Lange and Yuichi Nakamura, “Object-Oriented Program Tracing and Visualization”, *IEEE Computer*, pp 63 -70, May 1997.

LEV95 Nancy G. Leveson, “Safeware: System Safety and Computers”, Addison-Wesley, 1995.

LEV91a Nancy G. Leveson, “Software Safety in Embedded Systems”, *Communications of the ACM*, 34(2), pp 34-46, February 1991.

LEV91b Nancy G. Leveson, Steven S. Cha, and Timothy J. Shimall, “Safety Verification of Ada Programs using software fault trees”, *IEEE Software*, 8(7), pp 48-59, July 1991.

PAR90 David L. Parnas, A. John van Schouwen and Shu Po, "Evaluation of Safety-Critical Software", *Communications of the ACM*, 33(6), pp 636-648, June 1990.

RAT98a Rational Software Corporation, Rational Apex Ada Datasheet.

RAT98b Rational Software Corporation, “Ada Analyzer: A Technical Overview”, Technical Paper.

RUS95 John Rushby, “Formal Method and Their Role in the Certification of Critical Systems”, SRI International, Technical Report CSL-95-1, SRI 1995.

SCO94 John A. Scott and J. Dennis Lawrence, “Testing Existing Software For Safety-Related Applications”, Fission Energy and Systems Safety Program, Lawrence Livermore National Laboratory, UCLR-ID-117224, Revision 7.1, December 1994.

STO96 Neil Storey, “Safety-Critical Computer Systems”, Addison-Wesley, 1996.

STS97 “Slicer Tools List”, Software Technology Support Center, October 1997.

VOA97 Jeffery Voas, Frank Charron, Gary McGraw, Keith Miller and Michael Friedman, “Predicting How Badly `Good’ Software can Behave”, *IEEE Software*, 1997.

WON98a Ken Wong, M.Sc. Thesis, Department of Computer Science, University of British Columbia, 1998.

WON98b Ken Wong, “Looking at Code With Your Safety Goggles On”, to appear in *Proceedings of the 1998 Ada-Europe International Conference on Reliable Software Technologies*, Uppsala, Sweden, June 1998.