Refinement of Safety-Related Hazards into Verifiable Code Assertions

Ken Wong[1] and Jeff Joyce[2]

[1]Department of Computer Science, University of British Columbia
Vancouver, BC, Canada V6T 1Z4
tel (604) 822-4912 fax (604) 822-5485
kwong@cs.ubc.ca

[2]Raytheon Systems Canada, Ltd.
13951 Bridgeport Road, Richmond, BC, Canada V6V 1J6
tel (604)279-5721 fax (604)279-5982
jjoyce@mail.hac.com

Abstract: This paper presents a process for the stepwise refinement of safety code assertions from identified system hazards. The code assertions are intended for use in system safety verification. The development of the safety code assertions increases the feasibility of using code verification tools such as SPARK Examiner in the safety verification of large software-intensive systems. The process is demonstrated for a hypothetical chemical factory information system.

1. Introduction

An important step in an overall process for the safety engineering of a software-intensive system with safety critical functionality is the verification of the source code with respect to the identified hazards. One difficulty with performing the verification is the large "semantic gap" between the source code and the abstract "system level" concepts/language used to define hazards. To address this problem, this paper outlines a process whereby a safety-related hazard may be systematically refined into verifiable code assertions.

The approach to generating safety code assertions outlined in this paper was originally motivated by our interest in the possibility of using code verification tools such as SPARK Examiner [1] as part of the safety verification of a large software system. Chapter 18 of Leveson's seminal textbook on software safety [3] hints at the possibility of using such tools, but expresses concern about their practical feasibility. Clearly, it would be naive to expect that the safety verification task could be automated by simply feeding the source code for an entire system into a verification tool along with a representation of a safety-related hazard. If code verification tools are to be used in the safety verification of the system, it will be necessary to process the hazard and the source code into a form that the tools can accept as input.

One problem is that the safety-related hazard is likely to be expressed at a much higher level of abstraction, and in a different form, than the assertions expected as input by the code verification tool. The process outlined in this paper addresses this particular problem. There are other problems with using code verification tools for the safety verification of a large software system. The discussion of these problems lies outside the scope of this paper, but in a presentation of our overall approach to safety verification [7] we suggest how many of these other problems may be addressed. This has lead 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 are not used, the refinement of a safety-related hazard into a set of verifiable assertions would support other methods of static analysis such as manual inspection.

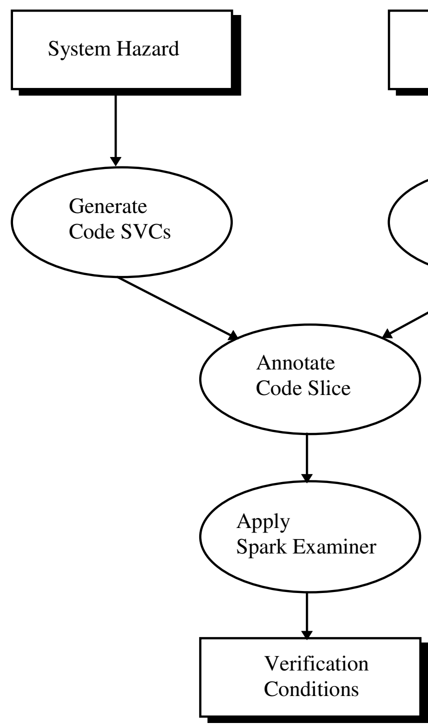

Figure 1: Overall safety verification process.

Our overall approach to the safety verification of large software-intensive systems is shown in Figure 1. This paper focuses on the “Generate Code SVCs” process which appears in the process bubble in the top left side of the figure.

The process for generating source code level “safety verification conditions” (SVCs) bridges the semantic gap between the hazard and source code though a series of refinement steps. This includes the development of SVCs at the system, design and source code level. It is assumed that the safety-critical code is either written in the SPARK Ada subset, or has been translated into SPARK through a process such as that described in [8]. The process for deriving the safety code assertions was developed during research conducted as part of the university-industrial collaborative project, formalWARE [6]. The process is illustrated in this paper with a hypothetical chemical factory information system.

Section 2 outlines some of the difficulties in bridging the semantic gap between the hazards and safety code assertions. The process for developing safety code assertions is presented in Section 3. The chemical factory information system is introduced in Section 4. Each step of the process is then applied to the chemical factory. Section 5 discusses the generation of system level SVCs for the chemical factory information system. The refinement of system level SVCs into design level SVCs is presented in Section 6. This is followed by a discussion of the refinement of design level SVCs into source code level SVCs in Section 7. Section 8 outlines the creation of the safety code assertions from the source code level SVCs. A summary and some conclusions are found in Section 9.

2. Development of Safety Code Assertions

The goal of this paper is to translate system hazards into a set of safety code assertions. A number of difficulties arise when attempting to express the system hazards in terms of code assertions.

2.1 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.

For instance, Section 4.2 of this paper uses the term "invalid temperature" in the definition of a sample hazard for a hypothetical chemical factory information system used as an example in this paper. Ultimately, the hazard must be understood in terms of global variables, sub-program parameters, constants, local variables and other elements of the source code. As revealed later in Section 7.2, the term "invalid temperature" corresponds to a relationship between specific data fields of the various messages passed between components of the software system which implements the chemical factory information system.

Sometimes there may be a high degree of “lexical similarity” between the terminology used to the define the system hazard and the identifiers used as names of elements of the source code implementation. For example, the source code may use a word such as “temperature” as the name of a data field in a message. 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 re-use.

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 "invalid" 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".

2.2 Time and Events

A system hazard often involves a relationship between events over a period of time. This temporal relationship may be apparent in the definition of the hazard or, as demonstrated in Section 5, may be revealed in the derivation of system level SVCs from the hazard definition. The temporality of a relationship is made explicitly by the use of phrases such as "...if the displayed temperature of vessel v is set to d at time t, then at some time no earlier than MAX_SYSTEM_PROPAGATION_TIME milliseconds before t ...".

To make use of tools such as SPARK Examiner, it is necessary to refine these temporal relationships into "non-temporal" assertions which are true or false at specific instants of time in the execution of the software system.

The refinement of temporal relationships into non-temporal assertions depends, in part, on the identification of causal relationships. For instance, it requires analysis of the software implementation to determine that an event such as "the displayed temperature of vessel v is set to d at time t" must have been *caused* by the occurrence of one or more internal events (either as combinations or as alternatives). By means of a search process, these events are chained backwards (in time) to other events. These causal relationships provide a basis for the refinement of the temporal relationships into "non-temporal" assertions.

2.3 “Backward” Expression

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

> *If pre-conditions on input*
> *then post-conditions on output*

The hazards, however, are often expressed in a “backward” fashion, beginning with the output, and then specifying the necessary conditions on the input:

If conditions on output
then conditions on inputs

The safety verification condition given as an example in Section 5 is an illustrative example of this "backwards" orientation.

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

If NOT(conditions on input)
then NOT(conditions on output)

This should place conditions on the input into the antecedent of the condition.

3. Process for the Construction of Safety Code Assertions

The safety code assertions are derived from a stepwise process that involves the development of system level, design level and source code level SVCs. If the refinement of the hazard into source code level SVCs is valid, then verification of the source code SVCs is sufficient for the safety verification of the hazard.

The inputs of the process are:

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

The safety code assertions are created for the hazard-related source code. In this paper it is assumed that a representation for the hazard-related source code has been created in the SPARK Ada subset [8].

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 refinement process involves the stepwise derivation of system level, design level and source code level SVCs from the hazard definition [5]. The source code level SVCs are translated into SPARK annotations. 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. They 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 [9].

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 "functional blocks" of code, and the system level SVCs are mapped into conditions on the functional block's input and output parameters.

3. **Generate source code level SVCs.** The design level SVCs are refined into source code level SVCs through analysis of the source code. The functional block parameters are identified in the source code, and the functional block SVCs are re-written in terms of the source code parameters. The resulting source code level SVCs may also be formalized [5].

4. **Generate safety code assertions**. The source code level SVCs are reformulated in a "forward" fashion and translated into SPARK annotations.

4. Example - Chemical Factory Information System

For illustrative purposes, we consider a hypothetical real-time safety-critical information system for a chemical factory. This is similar to other safety-critical information systems, such as air traffic management [2], in that environmental data is received, processed and displayed to human operators who make critical decisions.

4.1 System Description

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

4.2 Safety-Related Hazard

Following a process such as the system safety engineering process outlined in [3], we assume that the display of an "invalid" value as the temperature of a vessel is identified as a system hazard for the chemical factory control and monitoring system.

Hazard: *An invalid temperature, D, is displayed for vessel V at time T.*

It may be assumed that the identification of this hazard resulted from some earlier analysis which shows that the display of an invalid value as the temperature of a vessel, in combination with other conditions, could lead to a mishap such as a fire or explosion.

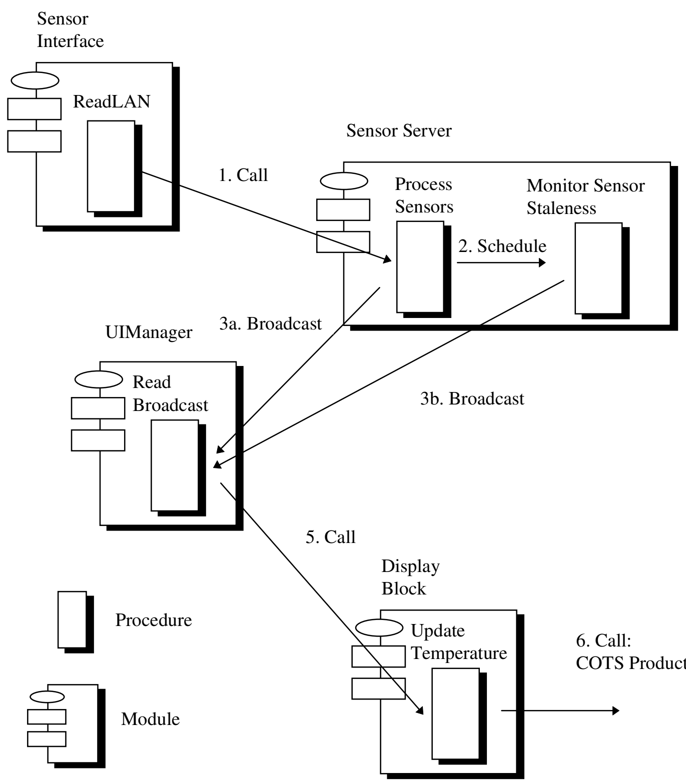

Figure 2: Temperature data flow highlighting the key modules and procedures.

4.3 Hazard-Related Source Code

The safety analysis will focus on a representation of the critical code, which is the output of a process that identifies the hazard-related code, and then translates it into the SPARK Ada subset [8]. The hazard-related source code modules and subprograms are shown in Figure 2. This includes code involved in the processing of sensor updates and the monitoring of stale sensor values.

5. System Level Safety Verification Conditions

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 [9]. The approach is similar to the concept of fault tree analysis (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) [4], 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 result of such an analysis for chemical factory hazard are five distinct system level SVCs. Full details can be found in reference [9]. The following is an example of one of the five system level SVCs:

> **Safety Verification Condition:**
> For all vessels, v, displayed temperatures, d, and times, t, if the displayed temperature of vessel v is set to d at time t, then at some time no earlier than 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.

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.

6. Design Level SVCs

The system level SVCs are refined into design level SVCs through analysis of the software architecture.

6.1 Hazard-Related Functional Blocks

Part of the process of producing a representation of the hazard-related code involves partitioning the relevant software into blocks which are invoked asynchronously. For example, this could be the result of code running in a process on a different computer, or which executes in a different "thread" on the same processor. The resulting "functional blocks" of code may be viewed as procedures with input and output parameters.

The hazard-related functionality for the chemical factory information system partitions into three such functional blocks. The monitoring of sensor staleness is one functional block, and the processing of sensor updates partitions into two functional blocks, LANToBroadcast and BroadcastToDisplay. For example, the LANToBroadcast block involves the reception and processing of LAN messages received from the external sensor monitoring system. LAN messages are read by the procedure `ReadLAN` and then processed by the procedure `ProcessSensors`, before being broadcast.

6.2 Functional Block SVCs

The system level SVCs are refined into SVCs for the functional blocks. An important aspect of the refinement is the separation of system functionality and timing issues, into separate functional block SVCs. This allows for the derivation of functional block pre- and post-conditions from the time-independent functional block SVCs.

The system level SVC introduced in Section 5 applies to the propagation of the temperature through the system, which is carried out by the LANToBroadcast and BroadcastToDisplay blocks, with a Broadcast mechanism providing the block communication. As a result, the system level SVC can be refined into design level SVCs involving these relevant functional blocks and the Broadcast mechanism, by extending the rigorous argument.

When performing the refinement, the functional blocks 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 functional block invocations and output. Furthermore, the functional block 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 functional block 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.

7. Source Code Level SVCs

The design level SVCs are refined into source code level SVCs, by identifying the functional block input and output parameters, along with any other relevant source code element. 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.

7.1 Functional Block Parameters

The input and output parameters of the functional block are determined from examination of the subprograms that make up the block. For example, the main thread of functionality for the LANToBroadcast block is contained in the `ReadLAN` and `ProcessSensor` procedures shown in Figure 2. Invocation of the `ReadLAN` procedure provides entry into the block 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 which are accessed by the ProcessSensor procedure. These are considered “global variables” of the functional block.

7.2 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`, reveals that updates from the different sensors are maintained in arrays of `Sensor Update` data records:

```
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`:

```
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 `Sensor Temperature`.

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

LANToBroadcast Source Code Level SVC:
For all Sensor Objects, s, in output BroadcastMessage, M,
if Sensor Object, s, contains the information SensorID, C, and PresentTemperature, 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.

8. Source Code Assertions

The source code level SVCs include functional correctness conditions which can be expressed as functional block pre- and post-conditions. These pre- and post-conditions can be translated into SPARK annotations.

8.1 Pre- and Post-Condition Style

The functional block SVC is re-written in a "forward" manner by taking the contrapositive form of the condition:

LANToBroadcast Source Code Level SVC (Contrapositive):
For all Sensor Objects, s, in output BroadcastMessage, M,
if NOT(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)
then NOT(Sensor Object, s, contains the information SensorID, C, and PresentTemperature, D)

The contrapositive form of the functional block SVC can be further refined by using “logical equivalencies”. For example, it is possible to move the “NOT” inwards, by using the appropriate logical equivalencies such as:

“NOT (there exists ...)” ≡ “for all (NOT ...)”

The refined contrapositive form of the source code level SVC is then:

LANToBroadcast Source Code Level SVC (Refined Contrapositive):
For all Sensor Objects, s, in output BroadcastMessage, M,
if for all SensorUpdates, u, in input LANMessage, L, with the information SensorCodeEstab, C1, which is not converted from SensorID, C, or TemperatureEstab, D1, which is not converted from D
then Sensor Object, s, does not contain the information SensorID, C, and PresentTemperature, D.

The contrapositive can be determined more precisely by first formalizing the source code level SVC [5]. For this source code level SVC, the entire SVC is a post-condition on the output. In other words, taking the contrapositive form of the source code level SVC did not uncover any pre-conditions.

8.2 SPARK Annotations

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

SPARK Examiner may be used to reduce the problem of verifying a "slice" of the code with respect to a source code SVC into 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 SPARK Simplifier and Proof Checker [1].

SPARK annotations make use of the SPARK Ada subset and appears in the code as Ada comments. However, they do not support quantification, so it is not possible to directly express the source code level SVC in the SPARK annotation language. Instead, a "proof function" is defined, `ConvertAll`, which has the same syntax as an Ada function, and the post-condition is expressed in terms of this proof function:

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

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

The proof function can defined as a proof rule:

```
ConvertAll(BM, LM, I, F) may_be_replaced_by
    for_all(u:integer, (I <= u and u <= F) ->
       not (for_all (s:integer, (I<=s and s <=F) ->
          (not(fld_SensorTemperature(element(BM, [u])) =
              ConvertTemperature(fld_TemperatureEstab(element(LM,[s])))
          or not(fld_SensorID(element(BM, [u])) =
              ConvertSensorCode(fld_SensorCodeEstab(element(LM,[s])))))
```

The proof rule is expressed in the FDL language, which is the required input for the SPARK Proof Checker. For illustrative purposes, the proof rule has been written in a form that closely mirrors the source code level SVC. If a proof of the conditions were to be attempted, it would be possible to re-write the rule in a form more conducive to the proof effort.

9. Summary and Conclusions

A method was presented in this paper for the informal systematic refinement of safety code assertions from system hazards. The 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".

The source code level SVCs (and corresponding SPARK annotations) 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.

Though the formulation of each step of the refinement involves informal arguments and statements of the resulting conditions, there may be value in partially formalizing some of the steps. For example, formalization of the source code level SVCs would contribute to the care and precision in which the source code elements are identified [5], and help ensure that the contrapositive form of the condition is correctly obtained.

10. 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 university-industrial collaborative project, formalWARE.

11. References

1. John Barnes, "High Integrity Ada The SPARK Examiner Approach", Addison Wesley Longman Ltd, 1997.
2. 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.
3. Nancy G. Leveson, "Safeware: System Safety and Computers", Addison-Wesley, 1995.
4. Nancy G. Leveson, Steven S. Cha, and Timothy J. Shimall, "Safety Verification of Ada Programs using software fault trees", *IEEE Software*, vol. 8, no. 7, pp. 48-59, July 1991.
5. Ken Wong, M.Sc. Thesis, Department of Computer Science, University of British Columbia, 1998.
6. http://www.cs.ubc.ca/formalWARE
7. Ken Wong, Jeff Joyce and Jim Ronback, "Ensuring the Inspectability, Repeatability and Maintainability of the Safety Verification of a Critical System", Department of Computer Science, University of British Columbia, TR-98-06, 1998.
8. Ken Wong, "Looking at Code With Your Safety Goggles On", in *Reliable Software Technologies - Ada-Europe'98,* Lecture Notes in Computer Science, Vol. 1411, Springer, 1998.
9. Jeffrey Joyce and Ken Wong, "Generating Safety Verification Conditions Through Fault Tree Analysis and Rigorous Reasoning", in *Proceedings of the 16th International System Safety Conference*, Seattle, Washington, September 1998.