Looking at Code With Your Safety Goggles On

Ken Wong

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

Abstract. This paper presents a process for the refinement of safety-critical source code into a more tractable representation. For large software-intensive information systems, the safety engineering view of the system reveals a "long thin slice" of hazard-related software involving a number of different software components. The hazard-related software is documented in the system "safety verification case" which provides a rigorous argument for the safety of the source code. The refinement process creates a representation of the source code which isolates the relevant source code details. A hypothetical chemical factory information system is examined to illustrate aspects of this process and its significance.

1. Introduction

This paper presents a refinement process for the source code level implementation of a software-intensive safety-critical system. The output of the process is a model of the source code which is used in system safety verification. Details of the representation are documented in the system safety case.

The development process for software-intensive systems typically involves the refinement of functional requirements into the system design, and the refinement of the design into source code. The process often includes the creation of abstractions at each level to hide the implementation details of the lower levels. If the abstractions are sound, they can then be used with a high degree of confidence to implement the application functionality.

In the case of software-intensive systems with safety-related functionality, the development process will include the production of a safety case [16]. The safety case summarizes all the evidence that the system is safe to deploy. Included in the safety case is what we call the "safety verification case" [14] which documents the verification of the implementation with respect to previously identified hazards. The safety verification case is used to support future safety assessments of the system. These assessments are typically performed by people other than the original system developers.

This paper argues for the importance of including a representation of the safety-related aspects of the system implementation in the safety verification case. The future safety assessments of the system cannot rely with a "high degree of confidence" on the soundness of the software abstractions used to construct the system. The safety assessments will require implementation details in order to have assurance of the safety of the source code.

This paper focuses on large software-intensive information systems with modern software architectures. These systems may consist up to a million lines of production code with the equivalent of millions more in "Commercial-Off-The-Shelf" (COTS) products. Such systems are characterized by "long thin slices" of hazard-related source code involving a number of different software components. The safety verification case for these systems requires a representation of the long thin slice that isolates the relevant details.

The author has had the opportunity to examine in depth a large real-time information system with safety-related functionality through the formalWARE project [15]. The project provided access to the Ada-based implementation of the system, as well as to the safety engineering process used in system development. Some of the lessons learned from the formalWARE project are illustrated with a hypothetical chemical factory information system.

The chemical factory information system is introduced in Section 2, along with a system hazard. Section 3 discusses the differences between the safety engineering and software implementation views of the source code. Section 4 presents the need for a more tractable representation of the source code. Section 5 outlines the source code refinement process which is demonstrated on the chemical factory information system. Section 6 discusses the use of the source code representation in building a safety verification case. Section 7 provides a summary of the paper.

2. Chemical Factory Information System

For illustrative purposes, this paper focuses on a hypothetical information system for a chemical factory. The chemical factory information system is similar to other real-time information systems, like Air Traffic Management (ATM) systems, in that environmental data is received, processed and displayed to operators. The operators then make safety-critical decisions based on the information received from the system.

2.1 System Description

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

2.2 System Hazard

There are a number of potential system hazards for the chemical factory information system. Hazards are states of the system that may contribute to a mishap [9]. For example, one system hazard is as follows:

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

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

This hazard is analogous to a typical ATM system hazard, namely, the display of an invalid altitude value for an aircraft at some particular time.

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

Figure 1: The chemical factory information system static architecture.

2.3 Software Architecture

The chemical factory software architecture is based on an industrial Ada-based architecture for an ATM system [7]. The architecture can be described by a "4+1 View Model" composed of five different perspectives, or views, that include the logical, process, physical and development views. The fifth view is a set of scenarios that incorporate elements of the other four views. The software architecture is based on the use of abstraction, as well as decomposition and composition.

The logical design of the software is decomposed into a set of class abstractions. The classes capture the key domain information, as well as the common mechanisms and design elements across the different parts of the system.

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

The development view focuses on the organization of the software modules in the software development environment. The software is partitioned into five layers as shown in Figure 1. Each layer is decomposed into subsystems and each subsystem is further decomposed into modules (Ada packages and generics). The software development teams are then organized around the subsystems and layers.

Each layer defines a progressively more abstract machine depending only on the services of the lower layers. In particular, layer 2 is a Distributed Virtual Machine (DVM) providing services such as object distribution, a thread scheduler and tactical configuration [12]. COTS products are confined to the lowest layer.

3. Safety Engineering View of the Software

The safety engineering view of the software can be illustrated at the requirements and source code level. At both levels there are significant differences between the safety engineering and the software implementation views of the system.

3.1 Safety vs Functional Requirements

Safety engineering is often focused on what the system should **not** do, in contrast to software implementation, which is largely focused on implementing correct system functionality.

This difference is apparent when safety requirements are contrasted with functional requirements. Typical software functional requirements describe a "forward" view of system functionality, while safety requirements often involve a "backwards" look at the system.

For example, a functional requirement involving the temperature display of a vessel for the chemical plant information system, might look like this:

> *Upon receipt of a sensor update containing the information that a vessel V is at temperature D, the system shall update the displayed temperature of vessel V to D within S1 seconds.*

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

In contrast, a safety requirement focuses on system hazards, such as the invalid temperature display hazard defined in Section 2.2. The corresponding safety requirement might be stated as:

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

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

The safety requirement is not simply a reformulation of the functional requirement. In fact, it is possible to satisfy the functional requirement, while failing to satisfy the safety requirement.

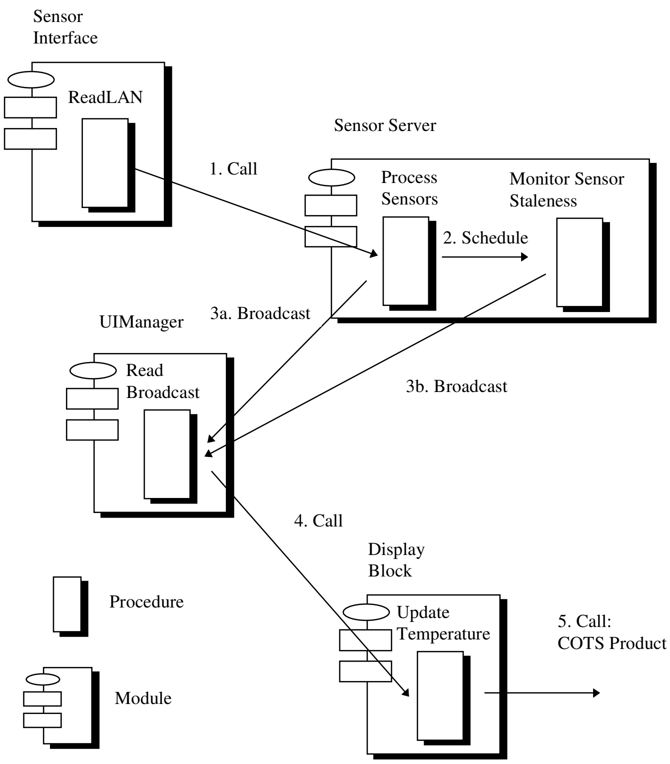

Figure 2: Temperature dataflow highlighting the key modules and subprograms.

3.2 The "Long Thin Slice" of Source Code

The software architecture uses abstractions which hide details that must be revealed for safety engineering. For software implementation, these details are relevant only to the team responsible for the module which implements a given abstraction. Safety engineering requires details from all the relevant software modules in order to evaluate the safety of the source code.

In particular, safety engineering must uncover code paths that lead to the hazard. A code path can be viewed as a sequence of steps where each step corresponds to an executable line of source code.

The relevant source code will typically be found in a number of different software components, involving only a small amount of code from each module. The result is a "long thin slice" of hazard-related software.

For the chemical factory information system, the modules related to the display of temperature data can be found in all five layers of the architecture. The modules from the upper layers of the static architecture are depicted in Figure 2.

Examination of a module from the long thin slice reveals a large number of non-executable code statements. A significant portion of the non-executable code is used to support the class hierarchies. These include statements that instantiate Ada generics, that specify module dependencies, and that re-export types from other modules. 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, as well as module renaming. 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.

The presence of a large amount of non-executable code and the dependence of a given executable line of code on a number of different modules are not necessarily signs of a poor design. In fact, they are a consequence of an object-oriented, modular and layered architecture which is designed to manage the complexity of a large software-intensive system. The result, however, is a long thin slice of source code which is not easily documented.

4. Documenting the "Long Thin Slice"

The safety engineering view of the source code will be captured in the "safety verification case" [14], which is part of the overall system safety case [16]. The safety verification case must provide evidence that the source code does not contribute to the identified system hazards. Part of the evidence includes documentation of the hazard-related source code.

Following the long thin slice may involve looking at more than a thousand different locations in the source code merely to understand a sequence of actions of less than fifty steps. This may not present a problem to the developer who has been immersed for many months or years in the development of the software. The developer may be capable of tracing a sequence of actions across the long thin slice of code without looking up every relevant line of code from various source files.

Other people, however, must also be able follow the same sequence of actions. These people include system certifiers, system maintainers and system developers intending to reuse the software in new systems. These people must also be able to follow the long thin slice in order to carry out their own safety assessments of the system.

5. Refinement of the Source Code

A more tractable representation of the long thin slice of hazard-related code can be created through refinement of the source code. Care must be taken that the representation accurately reflects the original source code.

5.1 Refinement Properties

The refinement of the source code into a model should be documented with enough detail so that it should be easy to determine the impact of any future changes to the source code on the model. The details should also be sufficient to ensure a repeatable process.

The refinement process may make use of simplifying assumptions. These assumptions must be stated explicitly. It must also be clear where each simplifying assumption is used in the refinement.

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 the simplifying assumptions are true.

The model should be **complete** in the sense that the model contains enough every 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 Refinement Process

The refinement process may be viewed conceptually as a sequence of transformations applied to a copy of the source code implementation together with a representation of the COTS products. The result of each refinement step is intended to be a conservative model of the implementation of the system.

The transformations may be carried out in an *ad hoc* manner relying on sound engineering judgment to ensure that each refinement step results in a conservative model of the implementation. To support repeatability of the safety verification case, the engineering judgment underlying each transformation must be carefully documented. The amount of documentation required to justify each step could be reduced by the use of a "standardized" set of syntax-oriented refinement rules whose soundness has been established.

Each transformation step could result in an executable model of the system in the sense that it can be compiled and executed. The refinement of the implementation into an executable model would be necessary if dynamic methods (i.e., testing) are to be used in the safety verification. But if the verification approach only uses static methods, then the refinement process can reduce size and complexity by allowing transformation steps that substitute executable aspects of the model with non-executable representations of system functionality.

These transformations might actually be applied directly to a copy of the implementation using basic editing tools. If the refinement process is based on a standardized set of transformation rules, then the entire refinement process could be documented by an executable script. It would then be possible to implement a software tool that executed the script and that checked that each transformation rule was used in a valid manner. Alternatively, these transformations may be simply be carried out "on paper" by describing the result of each transformation step.

The transformations are carried out until a tractable representation is created. This might mean that some qualitative criteria of “reasonable” has been satisfied. Alternatively, engineering judgment may be used to decide when the output of a given transformation is sufficiently tractable.

5.3 Refinement Steps

Four basic refinement steps have been identified for creating a tractable representation of the source code:

1. **Flatten** - collapse the class hierarchy.
2. **Fillet** - identify the hazard-related code.
3. **Partition** - decompose the hazard-related code into blocks.
4. **Translate** - represent the hazard-related code in a simpler notation.

Though the steps are to be carried out sequentially, they may also overlap.

5.3.1 Flatten

The class hierarchies are “flattened” to bring together the relevant lines of code and to reduce the amount of “non-executable” code. In Ada, class hierarchies are implemented with generics.

The class hierarchies are flattened by “expanding” each generic instance. A representation of each generic instance is created by substituting in the actual parameter values. It may be possible to build tools to produce this representation either directly or by extracting a compiler-intermediate form.

For the chemical factory information system, expanding the code results in, for example, a new module `SensorServer`, which is the `Server` generic with the parameter values substituted in for the parameters.

5.3.2 Fillet

Filleting involves identifying the source code which may contribute to the hazard. For the chemical factory information system, the modules and subprograms shown in Figure 2 provide a convenient starting point for the search. Further examination of the software architecture and the source code reveals that an invalid temperature has a number of potential causes, such as the corruption of the vessel temperature value, a stale sensor reading or a miscorrelation of sensors to vessels. The modules and subprograms that implement these features are identified and the relevant code is extracted.

There are a number of hazard analysis techniques [5] available which are designed to uncover system faults. These techniques include Hazard and Operability Studies (HAZOP), Failure Modes And Effects Analysis (FMEA) and Fault Tree Analysis (FTA), among others. These techniques were originally designed for mechanical systems, though there have been some attempts to adapt FTA [10, 17], HAZOP [17,19] and FMEA [17,18] to software. These techniques are useful for identifying software faults at the requirements or design level. These methods, however, are not well suited for determining which lines of code may contribute to a given hazard.

At the source code level, the search for hazard-related source code is a manual effort. The results of the hazard analysis at the requirements and design level can be used to guide the search. The search can be conducted in either a forward or backward fashion [9]. A forward search is particularly appropriate for tracing known hazard scenarios. A backward search is appropriate for identifying new scenarios.

Though the search for hazard-related code is primarily manual, the complexity of the code means that tool support is important. Software development environments, for example, provide support for module cross-referencing which is useful when tracing hazardous code paths across module boundaries. There are also other static code analysis tools, such as data flow analyzers, which would be useful in identifying critical data flows [2]. In particular, there are program-slicing tools that can be used to extract all code connected to a critical variable. Though most of these are research tools [4], there are some commercial tools that do a limited form of program slicing [11]. Program slicing for object-oriented programs is particularly difficult due to the run-time binding of methods and complex object-interaction graphs, though there is a tool that performs a limited form of object slicing [8].

5.3.3 Partition

Partitioning involves decomposing the filleted code into blocks which execute in different processes or threads (light-weight processes). Each “functional block” of code can be viewed as a procedure with input and output parameters. Partitioning the code along dynamic lines allows for the examination of certain code properties, such as the effect of block execution order on safety.

The hazard-related code is partitioned by tracing subprogram calls until a subprogram invocation results in a new process or thread. For the chemical factory information system, there are three hazard-related functional blocks: the reading and processing of a LAN message, the monitoring of stale sensor data, and updates to the temperature display.

In addition, the functional blocks can be further decomposed into existing static abstractions, such as subsystems or modules. Since system development makes use of these abstractions it would ease the fit of the safety analysis with the rest of the software development program. Unit testing and code reviews, for example, could be leveraged for the safety analysis of the functional blocks.

5.3.4 Translate

The final step of the process is to translate the hazard-related code into a simpler form. For the chemical factory information system, the hazard-related code is translated into the SPARK Ada subset [1]. SPARK is supported by program analysis and verification tools that could be used in the safety verification.

SPARK does not support generics, which are removed during the flattening stage. Though SPARK does support Ada packages, these are removed during the translation, resulting in a set of global data structures and subprograms. Many of the hazard-related modules contain only a small amount of relevant code, so the module structure is not needed for the safety analysis. In general, removal of the module structure may involve re-labeling the module subprograms and types to avoid name conflicts.

Two other approaches to representing the hazard-related code include informal English and formal mathematical notations.

The simplest approach involves extracting the hazard-related code, and annotating the code fragments in English. The annotations indicate the code's relevance to the hazard and can be supplemented by figures such as object-scenario and module diagrams to indicate the important modules and data flows [3]. The limitations of such an approach is that that the model is imprecise and not easily analyzed. The greatest virtue of a informal language notation is that it is easily understood and communicated to others.

A more precise approach would be to use a formal mathematical notation [13], which is usually more expressive than a programming language. For example, SPARK includes an annotation notation that uses the SPARK Ada subset, but is limited by not supporting quantification (i.e., "for all ..." and "there exists ..."). There are alternative machine-readable specification notations that are more expressive and which also have tool support that can aid analysis. The limitation of mathematical notations are their lack of familiarity to most developers, and their syntactic distance from the source code.

6. Safety Verification Case

The output of the refinement process is a model of the source code which is documented in the safety verification case. The safety verification case provides a detailed rigorous argument for the safety of the source code with respect to identified system hazards. Safety verification of the model can be achieved through dynamic techniques, such as testing, or static techniques, such as code inspection.

Static techniques also include the use of program verification tools such as SPARK Examiner [1]. These tools, typically, are used to verify program correctness with respect to a set of code assertions (i.e., pre- and post-conditions). Doubts have been raised about the feasibility of using these tools in safety verification [9]. The refinement of the hazard into a set of verifiable code assertions [14] is one step toward increasing the feasibility of using these tools. The refinement of the source code into a more tractable representation is another important step toward the practical use of correctness verification tools like SPARK Examiner in safety verification.

7. Summary

The safety engineering view of the software differs from the standard architectural views. This is true for the large software-intensive safety-critical information systems investigated in the formalWARE project. These systems consist up to a million lines of source code, with the equivalent to millions more in COTS products. The safety engineering view of the software reveals "long thin slices" of hazard-related software. The long thin slice consists of small amount of code in a number of different software components, spread out throughout a software architecture that may consist of hundreds of components.

The long thin slice of hazard-related code must be documented in a concise fashion in the system safety case. Though the developers may be able to locate and understand the slice, others (i.e., system certifiers, maintainers and developers intending to reuse the software) must also be able to identify and analyze the slice. The fact that a simpler representation of the source code is needed is not necessarily an indication of a poor design. It is simply the documentation that others require to conduct an independent safety assessment given the size and complexity of the system.

This paper outlines the steps necessary to create a simpler representation of the critical code for the safety verification case. The representation is created by a refinement of the source code that isolates the relevant code details. Additional work, however, is required to develop more effective techniques and tools for carrying out these steps. In particular, it would be useful to have tools for automating the code expansion and translation. As well, there is a need for more effective tool support for backward searches of the code.

8. Acknowledgments

This paper is a product of research performed in collaboration with Dr. Jeffrey Joyce of Raytheon Systems Canada Ltd. 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.

9. References

1. John Barnes, "High Integrity Ada The SPARK Examiner Approach", Addison Wesley Longman Ltd, 1997.
2. 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.
3. 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.
4. 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.
5. 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.
6. International Electrotechnical Commission, *Draft International Standard IEC 1508: Functional Safety: Safety Related Systems*, Geneva, 1995.
7. Philippe B. Krutchen, "The 4+1 View Model of Architecure", *IEEE Software*, November 1995.
8. Danny B. Lange and Yuichi Nakamura, "Object-Oriented Program Tracing and Visualization", *IEEE Computer*, pp 63 -70, May 1997.
9. Nancy G. Leveson, "Safeware: System Safety and Computers", Addison-Wesley, 1995.
10. 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.
11. "Slicer Tools List", Software Technology Support Center, October 1997.
12. Christopher J. Thompson and Vincent Celier. "DVM: An Object-Oriented Framework for Building Large Distributed Ada Systems", *Proceedings of the TRI-Ada '95 Conference*, ACM, Anaheim, November 6-10, 1995.
13. Jeanette M. Wing, "A Specifier's Introduction to Formal Methods", *IEEE Computer*, 23(9), pp. 8 -22, September 1990.
14. Ken Wong, M.Sc. Thesis, Department of Computer Science, University of British Columbia, 1997.
15. http://www.cs.ubc.ca/formalWARE/
16. Peter G. Bishop and Robin E. Bloomfield, "A Methodology for Safety Case Development", in *Safety-critical Systems Symposium*, Birmingham, UK, February 1998.
17. P. Fenelon, J.A. McDermid, et al., "Towards Integrated Safety Analysis and Design", *ACM Computing Reviews*, 2(1), p. 21-32, 1994.
18. Robyn R. Lutz and Robert M. Woodhouse, "Experience Report: Contributions of SFMEA to Requirements Analysis", in *Proceedings of ICRE'96*, 1996.
19. Francesmary Modugno, Nancy G. Leveson, Jon D. Reese, Kurt Partridge, and Sean D. Sandys, "Integrated Safety Analysis of Requirements Specifications", in *Proceedings of the 3rd International Symposium on Requirements Engineering*, Annapolis, Maryland, January 1997.