ASSESMENT OF SOFTWARE SAFETY VIA CATASTROPHIC EVENTS COVERAGE

Andrew J. Kornecki
andrew.kornecki@erau.edu
Department of Computing
Embry Riddle Aeronautical University
Daytona Beach, FL 32114
USA

ABSTRACT

As we depend more and more on software intensive systems, safety is of paramount importance. This paper attempts to address the often-neglected topic of safety assessment for software intensive systems. A thorough analysis of system hazards, and related catastrophic events, allows the developers to assess the system safety by identifying all potential catastrophic events and their impact on requirements, design, and other mitigation means. If we can present an argument that all hazards leading to catastrophic events have been handled, we de-facto establish a baseline for a "safe" system. This paper proposes classification of hazards and catastrophic events from the perspective of the software modules implementing the target system functionality.

KEY WORDS

software safety, software development lifecycle, safety critical systems

1. INTRODUCTION

The growing complexity of modern systems is due to the ever-increasing power of computing devices. The functionality that has been implemented in hardware is now often ported to software providing a variety of modifiable options and flexibility. However, the bad news is that the software complexity is the main source of potential hazards introduced by the software itself. These hazards are due to the system entering an inconsistent or unsafe state, which may occur in the case of a violation of mutual exclusion, lack of synchronization, corrupted communication, deadlock, etc. All these may lead to a situation known as a catastrophic event, which in turn may result in the system failure, if proper mitigation is not used.

The primary objective of software safety is to guarantee that software does not cause or contribute to a system reaching such a hazardous state. Since such a guarantee is rather difficult to accomplish, we will settle for second best. We strive to design the system in such way that:

- it detects and takes corrective action if the system reaches a hazardous state, and
- it mitigates possible damage in case a catastrophic event occurs.

In this paper we discuss the differences between reliability and safety, the impact and relation of software to the overall system safety, and introduce the concept of assessing the safety of a software intensive system through analysis of the hazards and potential catastrophic events. The measure of safety is defined as coverage of the complete set of system hazards, including the additional hazards introduced by the software itself.

2. RELIABILITY AND SAFETY

Reliability and safety are the major considerations for any high integrity system operations. There is a need to define and distinguish between these two closely related, nevertheless different concepts. Both reliability and safety are components of dependability [1] defined as a measure of the overall quality of the system - or the property of the system that justifies reliance on the system services. The typical dependability attributes, in addition to reliability and safety, are: availability, maintainability, confidentiality, security, integrity, etc. The dependability attributes are interrelated and their importance may vary.

A reliable system will assure continuity of its functions. Reliability defines the property of the system to meet its specified requirements. A system is thus reliable if it does what the developers said it would do. A deviation from the system requirements is treated as a failure. The failure can be a situation when the system does not accept the input, produces incorrect output, produces the output at the wrong time (too early, too late), or simply does not do what it "shall do" according to the requirements documents. Reliability is most often defined in terms of the probability that the system will be performing its

intended function, thus operating failure-free for a specified period (or a number of missions). Reliability is often identified by related measures like failure rate or its inverse, mean time to failure. Given appropriate data, various mathematical models allow us to derive a quantitative reliability estimate. The subject, despite being rather controversial when applied to software, has well-established literature [2, 3].

Safety is when we are protecting human health, life and environment. Safety is the property of a system that it will avoid hurting someone or cause property loss or damage. A system is safe if it does not cause the users to be injured or killed, does not cause damage or destruction to any connected equipment, does not cause a financial loss. The system that does not work (is not reliable) may be completely safe. Likewise, completely reliable system may still injure someone as it may produce an output that the developers never expected, thinking only in terms of the system functionality rather than considering potential hazards. The majority of real-time, embedded applications control some dedicated system, thus controlling the power that the system can dissipate. Systems with such characteristics are called safety-critical or safety-related. The representative literature positions are those of Leveson, Storey, Herrmann, Friedman/Voas, Gardiner, Bowen [4,5,6,7,8,9].

A rigorous approach to safety has been proposed by Kelly with the “safety case” concept [10]. Rather than relying on prescriptive standards and regulations, the burden is shifted to the developers who are required to construct and present arguments that their systems attain an acceptable safety level. The arguments and related supporting evidence constitute the “safety case". The authors of the “asymmetric” approach to the safety critical software [11] attempted to define the most important areas where software assessment should be concentrated, suggesting an asymmetric allocation of resources. Another project dealing with the issue of safety assessment [12] involved building formal models and integrating diverse evidence to provide quantitative safety arguments. The research studied fuzzy sets, Bayesian belief nets, and various probability models. A related study dealt with issue of the context for software safety assessment [13] recognizing the fact that software is deterministic and its behavior depends not only on the inputs but also the environment in which software operates.

It is a common view for engineers involved in reliability studies that reliability and safety are identical. However, many accidents may happen without evident system failure resulting from a combination of environmental events, procedural mistakes, and system faults. One needs to emphasize that reliability is a bottom-up activity focusing on system failures while safety is top-down approach concentrating on system hazards.

3. SOFTWARE SAFETY ANALYSIS

With full understanding that safety is a critical system issue, there is still a need to analyze the system safety from the software viewpoint. The system Preliminary Hazard Identification (PHI) and the subsequent Preliminary Hazard Analysis (PHA) are the starting point for software safety analysis. They are designed to identify and categorize the hazards or potential mishaps that may result from system operation. Domain specialists and safety engineers must identify as many hazards as possible. Classifications of the hazard severity may range from *catastrophic* to *critical* to *marginal* to *negligible*. Also, the frequency of hazards can be categorized as e.g. *frequent*, *moderate*, *occasional*, *remote*, *unlikely*, or *incredible*. Depending on this classification, appropriate measures to handle the hazards are undertaken.

Once the hazards have been determined, we consider various faults, events, and parameter deviations. The principal objective of Software Fault Tree Analysis (SFTA) is to show that the software logic will not produce system safety failures. We determine all possible environmental conditions and events, which could lead to these software-induced failures. The basic procedure is to assume that the software output (lack thereof, wrong timing) caused a safety violation and then work backward to determine the possible reasons this output is produced.

The initial system safety analyses, conducted during the system requirements phase when the role of software is being defined, begins with identification of hazards associated with a particular design concept and/or operation. These initial analyses and subsequent system and software safety analyses identify when software is a potential cause of a hazard or when it will be used to support the control of a hazard. Such software is classified as safety-critical, which then makes it subject to software safety analysis.

Software Safety Analysis is the development activity designed to identify the software components (programs, modules, routines, functions, objects, data structures) that are critical to system safety, and thus must be examined in depth. These software components are identified as safety critical. It should be noted that the entire software must be analyzed at least to the extent necessary to determine its impact upon the safety critical components. All programs, routines, modules, tables, or variables which control or directly/indirectly influence the safety critical code shall also be classified as safety critical. Additionally, some or all of the software tools (e.g., compilers, support software, etc.) may also have to be designated as "critical". All safety critical software elements will be analyzed to the source/object code level by the follow-up software hazardous effect analysis.

It is clear that software by itself is not hazardous. However, some software components may be considered

hazardous when interfaced with a certain type of system. A typical example of what we call safety critical software is software controlling and monitoring some undesired or uncontrolled release of energy restricted by hardware components. Another example is software that provides indirect control or data for safety critical processes, thus leading to potentially erroneous decisions by human operators. The Software Hazardous Effects Analysis (SHEA) is the activity, which enables us to identify the *safety significant* and *safety critical* software at the system component/unit level. We use the SHEA to identify hazards associated with the role of a software item in performing the functions to deduce the possible causes, to identify safety requirements for design and test, and to control the hazard causes.

Considering the above, the starting place for the analysis is the code responsible for the output. We need to determine the current values of the code variables and then backtrack deducing how the program reached this part of the code. To be able to associate appropriate software components with identified hazards, it is critical to have specific and traceable safety requirements. The level of detail required for such analysis may be limiting factor. It is recommended for use only for the software components directly responsible for very critical and potentially catastrophic events.

The industries dealing on an everyday basis with software safety address the issues of software hazard risk assessment through appropriate standards: military (MIL-STD-882D, DEF00-55 and 56), aviation (DO178B), aerospace (NASA-STD-8719.13A), nuclear (IEC60880, ANS ANSI/IEEE 7.4.3.2), and programmable electronic (IEC61508) standards. The website http://www.12207.com/ provides a list of standards applicable to medical devices (ANSI/AAMI SW 68, FDA Guidance 1-3, IEC60601-1-1 and-4, ISO13485, ISO14971).

To reduce the risks associated with the identified hazards we may use different approaches. In the order of preference we can:

- Modify the requirement specification
- Modify the design
- Integrate additional safety features
- Include additional warning devices
- Modify the operating procedures and training

Modification of requirements allows for early handling of hazards while the modification of operating procedures is an "after fact" activity – and is due to human imperfection. It is clear that the risk reduction is more effective and less costly when the hazards are discovered early in the lifecycle. The bad news is that for any system of reasonable complexity, it is likely that not all hazards can be predetermined. Consequently, additional activities may be required to assure system safety. These include, but are not limited to, partitioning and protection, safe kernels, watchdogs and procedural solutions. In addition, there is an increasing body of experience with application of formal methods and a model-based approach to support safety of the software-intensive product.

4. SAFE SOFTWARE

Software is assumed to be safe, if it is highly unlikely that it could produce an output that when interacting with the system would cause a loss of physical property, physical harm, and loss-of-life for the system that the software controls. The term "software safety" refers to a variety of development and assessment processes that attempt to accomplish this goal. However, software safety is a sub-problem of system safety. The objective of software fault tolerance techniques is to survive the faults during the run-time. By this measure, the fault tolerance can be treated as a sub-problem of system safety.

For each physical system we can identify a set of functions that it performs. For each function we may identify zero or more catastrophic events that must be prevented. If the software controls directly or indirectly a specific function, we must design the code in such way that the output will not lead to the undesired event. The objective of software safety assessment is to demonstrate the above. The weak points of this reasoning are that: (a) system safety (or a lack thereof) must be completely and unambiguously defined before software safety can be tackled, (b) the software "responsibility" must be completely and unambiguously defined. For example, a system placing incoming commands in a linked list for future execution may fail after the list exceeds some number of entries (for systems with memory limitation). This situation must be accounted for in the software development phase and additional code must be provided for remedial action.

Given the relationship between software components, one may use for example Fault Tree Analysis to assure that the software subsystem is safe. This popular graphical method starts with the top-event related to the identified hazard and works backwards to determine the causes. To guarantee that the entire software system does not introduce interference, we use concepts of protection via firewalls. Firewalls isolate critical requirements modules from modules that do not contribute to function affected by critical events. Non-critical software components must not affect the critical components in unpredictable fashion. Such integrity must be valid under both: normal and abnormal operation (i.e., when the faults manifest themselves).

5. SAFETY METRICS

There is no accepted agreement on software safety measures and metrics. The critical argument here is that safety is the system issue. The other argument is attributed to the fact that the safety is defined in the

negative: "do not harm". It is much easier to show that something happens than that something does not happen. Then, what can we accept as Software Safety Metrics?

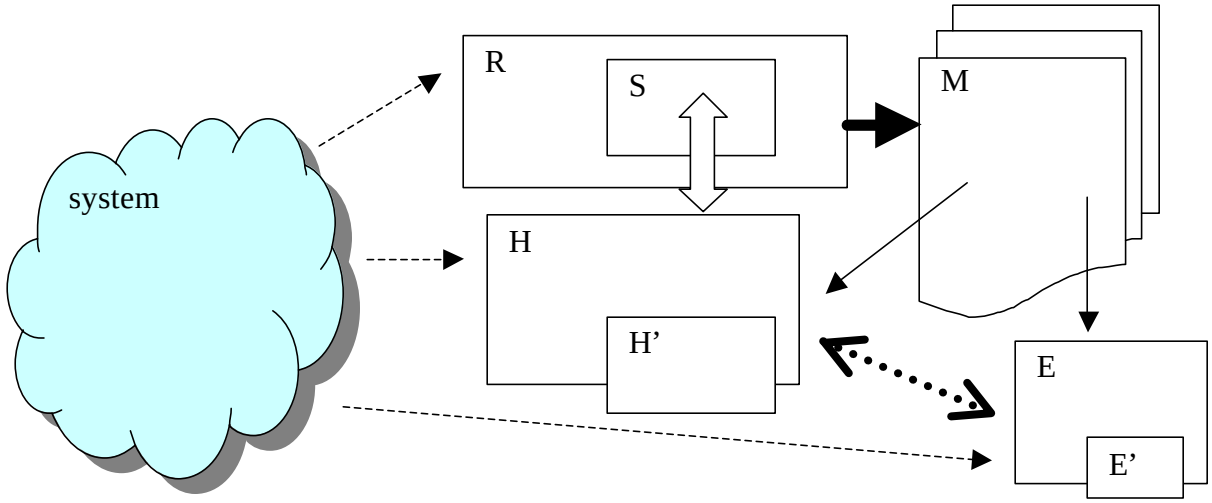

Figure 1: Relationship between system-software artifacts

We propose a software safety assessment related to the well-known concept of coverage. Assume that, as a result of system hazard analysis, we create a comprehensive set of all hazards (**H**) consisting of elements $\mathbf{h_1}$ to $\mathbf{h_n}$ and we identify the related set of catastrophic events (**E**) consisting of elements $\mathbf{e_1}$ to $\mathbf{e_m}$. Assume we have a complete set of software requirements listing both functional and the quality of service requirements that the software must implement (**R** – a list of $\mathbf{r_1}$ to $\mathbf{r_k}$). It is imperative that the set **R** has been created in such way that specific subset of **R** (say **S,** consisting of elements $\mathbf{s_1}$ to $\mathbf{s_p}$) can be directly linked to elements of **H**, i.e. there is a relation between **S** and **H.** The relationships between the sets are depicted in Figure 1.

The software/firmware of the system is implemented by a set of software components or modules (**M**). We review each module of the set **M** from a triple perspective:

(a) Does it implement an element of **S**?
(b) Does it contribute to the defined hazards producing an output leading to a catastrophic event in **E**?
(c) Does it contribute to any additional hazards (**H'**) and the resulting catastrophic events (**E'**)?

Case (a) is a situation when the requirements are derived from hazard analysis. In this case, the subset **S** is a part of the defined requirements. Meeting the requirements, which is a premise of system reliability, is in this case also a base for safety assurance. However, two remaining cases are pointing to the difference between reliability and safety. In case (b) we need to apply rigorous techniques for testing and verification. We should seek either design or procedural solutions to eliminate and/or reduce the hazard. In case (c) the lists **H** and **E** need to be modified (by augmenting them with the **H'** and **E'**, respectively). A re-design, leading to modification of the set **M,** may be necessary.

What are the rigorous techniques we postulate in case (b)? The simplicity of the module is one of the best defense lines. Reduced complexity makes the module both readable and testable while simplifying the possible interfaces between software modules. We need to identify what possibly can go wrong with the module code by analyzing such events as the deviation of the values, deviation of timing, lack of synchronization, corrupted communication, a potential for data loss, deadlock and live-lock. Another typical guideline relates to reduction of potential effect of common mode failures – a situation when failure occurs as a result of faults in different redundant modules. We generally accept avoiding single point of failure, where the failure of a single component may lead to the total system failure. We must not allow a single hardware fault (e.g. lack of data, reversed bit) to trigger an unrecoverable error. Other guidelines relate to the determinism of software execution by use of simple sequential structures, block-recovery redundancy and, if feasible, two-level architecture. Fault injection is an established and accepted technique that can be used for the identified safety critical modules.

In a situation (c) when the additional hazards are introduced, the objective would be either to eliminate them or to mitigate their effect. The elimination would rely on removing the causes of the hazard, which can be done by better understanding how hazards were introduced in the first place. For the hazards that cannot be removed, it is essential to mitigate them. This can be accomplished by adding additional components (monitors, watchdogs, and redundancy measures –

implemented either in hardware or software). On the hardware side, a concept known as safety core may be used. The safety core is a separate circuitry of limited functionality, designed to take over in a situation when the system failure prevents continuation of safe operation. A software solution may include a safety kernel – an independent programming module that monitors the state of the system to determine when potentially unsafe system states may occur or when transitions to potentially unsafe system states may occur. The safety kernel is designed to prevent the system from entering the unsafe state and return it to a known safe state. Obviously, the operations of safety kernel are separated from operation of the rest of the code. All the above-mentioned solutions, in principle, increase software complexity thus somehow contradicting the first rule of safe software: keep it simple.

If all the events from the modified set **E** are accounted for, we have achieved the required safety coverage. It is evident, that the proposed assessment is based on very meticulous hazard analysis leading to identification of the sets **H** and **E,** and subsequent mapping of the sets into the specific subset **S** of the software requirements (**R**), traceable into subsequent design (**M**). This traceability allows developers and testers to visualize the relations between system components and hazards, which in turn may lead to better understanding of the system safe operation.

6. CONCLUSIONS

The safety assessment process for software intensive system is an integral part of safety-critical development. Typically, dedicated safety engineers, with a system and/or hardware background, carry out the process. Hazard analysis is elaborated at the system level and then addressed in various aspects of the design by assigning the handling of specific hazards to hardware and software, as appropriate. The individuals involved should have an appreciation of the software impact on system safety and the potential hazards that the software may contribute (due to such situations as deadlock, lack of synchronization, corrupted data, or violation of mutual exclusion). The resulting potential catastrophic events must be added to the original, system-oriented list. Only then, a thorough analysis of the design and the assurance that the design choices will facilitate hazard mitigation and prevent the occurrence of the catastrophic events can we assess a measure of system safety. We need to stress again that only comprehensive system hazard analysis, extended to the software components, will allow developers to create a comprehensive list of hazard – the starting point for the proposed safety assessment. Historical safety experience, lessons learned, trouble reports, and accident and incident files are examples of techniques to help in the hazard identification. Typically, any organization developing safety critical products collect such data as components of a successful system safety effort. Many industries have published guidelines, checklists, standards, and codes of practice that may facilitate developing comprehensive hazard list.

The proposed approach attempts to give some quantifiable, albeit binary in nature approach to system safety: if all identified hazards and the related catastrophic events are handled by the design, the system is considered safe. Further work must be done to specify software hazard analysis process and evaluate variety of architectures supporting software intensive safety critical systems.

7. ACKNOWLEDGEMENT

The presented work originated during the author's sabbatical leave with the Cardiac Rhythm Management, Guidant Corporation, St.Paul, MN. The author would like to express appreciation to all those that helped him to understand the intricacies of true safety critical software. Particular credit is due to Nader Kameli and Conrad Sowder.

REFERENCES

1. B. Randell, J.C. Laprie, H. Kopetz and B. Littlewood, editors, *Predictably Dependable Computing Systems* (Springer-Verlag 1995, ISBN 3-540-59334-9)
2. J. Musa, A. Iannino, K, Okumoto, *Software reliability - measurement, prediction, application* (McGraw Hill, 1987, ISBN-0-07-044093)
3. H. Pham, *Software reliability* (Prentice Hall, Springer, 2000, ISBN-0-9813-0884-0)
4. N. Leveson, *Safeware - system safety and computers*, (Addison Wesley, 1995, ISBN-0-201-11972-2)
5. N. Storey, *Safety critical computer systems* (Addison Wesley Longman, 1996, ISBN-0-201-42787-7)
6. D. Herrmann *Software safety and reliability* (IEEE Computer Society, 1999, ISBN-0-7695-0299-7)
7. M.A. Friedman, J.M. Voas, *Software assessment: reliability, safety, testability* (John Wiley and Sons, New York, 1995, ISBN-0-471-01009-X)
8. S. Gardiner, *Testing safety-related software – a practical handbook* (Springer, 1998, ISBN-1-85233-034-1)
9. J. Bowen, M. Hinchey, *High integrity system specification and design* (Springer 1999, ISBN-1-85233-053-8)
10. T.P.Kelly, *Arguing Safety - A Systematic Approach to Safety Case Management*, PhD Thesis, Department of Computer Science Green Report YCST 99/05, University of York, England, 1999
11. S. A. Vilkomir, G. I Zhidok, Experience of Licensing of Software for Digital Safety Related Systems in Ukraine, Project Control for 2000 and Beyond, *Proceedings of ESCOM-ENCRESS 98*, Rome, Italy, 27-29 May 1998, 328-331

12. C. Garrett, S. Guarro, G. Apostolakis, The Dynamic Flowgraph Methodology for Assessing the Dependability of Embedded Systems, *IEEE Transactions on Systems Man and Cybernetics*, *25*, May1995, 824-840

13. N. Fenton, B. Littlewood, M. Neil, L. Strigini, A. Sutcliffe, D. Wright, Assessing Dependability of Safety Critical Systems using Diverse Evidence, *IEE Proceedings Software Engineering*, *145*(1), 1998, 35-39