Measures and Risk Indicators for Early Insight Into Software Safety

Dr. Victor Basili
University of Maryland and Fraunhofer Center - Maryland

Kathleen Dangle and Linda Esker
Fraunhofer Center - Maryland

Frank Marotta
U. S. Army Aberdeen Test Center

Ioana Rus
Honeywell Aerospace

Software contributes an ever-increasing level of functionality and control in today's systems. This increased use of software can dramatically increase the complexity and time needed to evaluate the safety of a system. Although the actual system safety cannot be verified during its development, measures can reveal early insights into potential safety problems and risks. An approach for developing early software safety measures is presented in this article. The approach and the example software measures presented are based on experience working with the safety engineering group on a large Department of Defense program.

The purpose of the system safety process is to identify and mitigate hazards associated with the operation and maintenance of a system under development. System safety is often implemented through an approach that identifies hazards and defines actions that will mitigate the hazard and verify that the mitigations have been implemented. The *residual risk* is the risk remaining when a hazard cannot be completely mitigated. The goal of the system safety process is to reduce this residual risk to an acceptable level, as defined by the safety certifier. Cost is a consideration in determining the level of acceptable residual risk.

As software contributes an ever-increasing level of functionality and control in today's systems, the system safety process must scrutinize software-specific components of the system. Software can contribute to system safety as both a hazard source and hazard mitigation. Software is not intrinsically hazardous, but plays a role in safety in many systems where it:

- Causes hardware to perform unsafe actions.
- Directs an operator to perform unsafe actions.
- Guides an operator to make unsafe decisions.
- Mitigates hazards.

In this article, we define a measurement approach that provides early visibility into the implementation of the software safety hazard process, assessing the level of consistency and discipline that is applied to the process for identifying and mitigating software-related hazards. Early process visibility assists safety engineers in detecting breakdowns in the process, asking the right kinds of questions, and making timely decisions that will improve the resulting system safety. This early visibility is important as mitigations typically affect system requirements and design; making these decisions late in the system development lifecycle can be cost-prohibitive. The proposed measurement approach identifies risks resulting from the application of the safety hazard analysis process (or lack thereof) by performing process checks, and assesses the *potential* for achieving a safe system. It is important to note that this approach does not provide for an evaluation of the system's safety.

> *"Early process visibility assists safety engineers in detecting breakdowns in the process, asking the right kinds of questions, and making timely decisions that will improve the resulting system safety."*

This article begins by defining terms and documenting our assumptions. We then describe our approach for defining specific safety measures in the context of an existing environment and provide some examples.

Terminology and Key Concepts

A *hazard* is any real or potential condition that can cause injury, illness, or death to personnel; damage to or loss of a system, equipment, or property; or damage to the environment. Key terms associated with hazards and their management are:

- **Causes.** What can make the hazard occur.
- **Controls.** Mitigation actions whose purpose is to minimize the chances of a hazard occurring.
- **Verifications.** Some assurance, like safety test cases, that the hazard has been controlled.

A hazard is *open* if at least one of its causes is open; a cause is *open* if at least one of its controls is open; a control is *open* if at least one of its verifications is open. A hazard is *closed* when all the controls for all its causes have been implemented and verified.

A *safety-related requirement* is a requirement whose purpose is to control a hazard. One hazard might be addressed by several requirements (e.g., one hazard may affect several parts of the system), or one requirement might address several hazards (e.g., a central control or communication system may mitigate hazards from multiple nodes).

A *hazard tracking system* (HTS) is a repository of identified system hazards and their associated causes, controls, and verifications. Within the HTS, causes should be related with the system element causing the hazard, controls should be related with the requirement(s) controlling or mitigating the hazard, and verifications should be related with the hazard cause and the test verifying that the hazard is controlled.

A hazard is defined as a software-related hazard if it has at least one software cause or one software control. A software safety-related requirement is a software requirement that can create or contribute

to a hazard in the system or is defined to control or mitigate a hazard.

An example of a system hazard description that has a software-related cause is as follows:

- **Accident/Mishap.** Undesired and unplanned event that results in a specified level of loss (e.g., two planes collide).
- **Hazard/Description.** State that leads to an accident (e.g., guidance system may malfunction).
- **Hazard Cause.** The action causing the hazard to occur (e.g., a miscalculation of the projected trajectory.
- **Hazard Control or Safety Requirement.** Mitigation via a requirement or set of requirements whose purpose is to minimize the chances of a hazard (e.g., multiple computations of the projected trajectory are computed and poled).
- **Verification.** An assurance that the hazard has been controlled (e.g., safety test cases).

Figure 1 provides an illustration of the context for this example.

Several assumptions are made: (1) all hazards should be recorded in an HTS; (2) hazards are retired or have their associated risk reduced over time, but do not leave the HTS; and (3) closed hazards can become open hazards when a new cause is found. Although the approach does not prescribe a particular management or organizational structure, it is assumed that the safety and project organizations communicate and collaborate effectively in both evolving requirements and verifying mitigations. As the safety hazard analysis will impact requirements, design, code, and tests, it is assumed that the standard processes defined by the project for change management apply to artifacts impacted by safety hazard analysis.

The *level of rigor* (LoR) is the amount of requirements analysis, development discipline, testing, and configuration control required to mitigate the potential safety risks of the software component [1]. Each software component should be assessed and assigned an LoR for development. This refers to any mechanism put in place to treat specific requirements with special treatment, giving a piece of software higher levels of safety assurance and providing users higher confidence through greater discipline and process.

A *safety-related defect* is a defect that refers to a failure to comply with a safety requirement, an unexpected behavior that affects safety, or the recognition that a control has not been defined/implemented/verified. Safety-related defects should be traceable to a hazard or may generate a new hazard. Defects can be counted directly or they can be weighed by the set of related requirements or hazards they affect. A *software defect tracking system* (i.e., tool/database to capture software defects identified during testing) is used as the source of safety-related software defects.

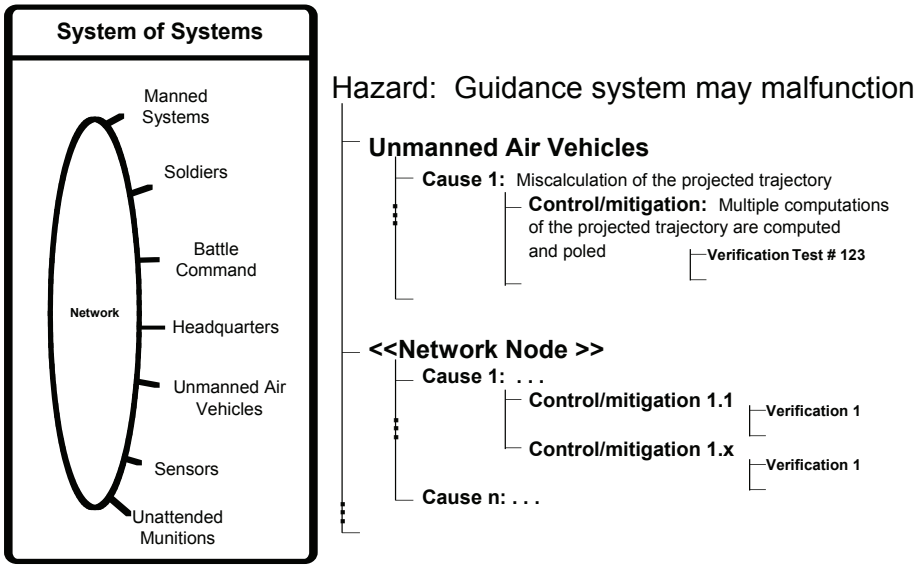

Figure 1: *Example of a Hazard, Cause, Control, and Verification*

Gaining Software Safety Visibility

Our goal in applying the proposed measurement approach is to provide software safety engineers visibility into the software safety process and to assist them in making judgments about the software safety process implementation and its execution. We identified five needs, and an associated inquiry area for each was defined:

1. **Software Safety Analysis Process.** Confirm that system and software requirements and development practices are in compliance with safety processes.
2. **Hazard and Mitigation Identification.** Ensure that the program is adequately identifying and documenting the appropriate information about a hazard (i.e., hazards, causes, and controls as defined by the software safety analysis process).
3. **Hazard Monitoring.** Ensure that sufficient actions are taken by analyzing and monitoring hazard causes, controls, and verifications over time (i.e., are the hazard controls being implemented, and verified).
4. **Appropriate LoR for Software Safety.** Balance risk with the cost of safety by identifying the appropriate software development LoR.
5. **Safety-Related Defects.** Identify whether any safety problems remain in the system for the safety assessment reports by identifying all outstanding safety-related defects.

For each area, readiness and visibility measures are defined, specifying different measurement details. A *readiness assessment* provides a preliminary view into the state of the safety process for software and checks that the data needed for the second type of measurement is available. *Software safety visibility* digs deeper by defining models, measures, and interpretations that provide information on the implementation of safety practices (or lack thereof) and points to safety-related risks and issues.

To minimize the overhead associated with data collection and analysis, a combination of a top-down goal/question/metric analysis and a bottom-up inventory of the data already collected by the organization is used to identify the measures that will be cost-effective and address management needs [2].

For example, to address software safety analysis, an investigation may be performed to determine whether there is a documented safety process that identifies requirements as safety-related and records that information in the requirements repository. If this is not true, then the program may have a problem and further measures that assume counting the number of safety-related requirements cannot be utilized. A sample set of key questions addressing the five inquiry areas for the readiness assessment are shown in Table 1. All readiness questions must be answered *Yes* to indicate that the appropriate measurements can be gath-

Inquiry Area	Readiness Assessment Questions
Software Safety Analysis Process	○ Is there a documented software safety process that identifies requirements as safety-related? ○ Are safety-related software requirements marked as such in the requirements repository?
Hazard and Mitigation Identification	○ Is there an (automated) HTS where software-related hazards, causes, controls, and verifications are recorded (and can be counted)?
Hazard Monitoring	○ Are hazards mapped back to their source (requirements) and controls mapped to requirements? ○ Are all the fields being entered into the HTS?
Appropriate LoR for Software Safety	○ Are the various levels of rigor identified and is the distribution rational?
Safety Defects	○ Are software safety-related failures/faults identified as such in the software defect tracking system? ○ Are safety-related test cases identified as such? ○ Are defect closures recorded?

Table 1: *Readiness Assessment Questions*

ered. *No* answers provide an early warning that software safety may not be properly addressed. In this case, the recommended action is to identify why the data is not available (root cause) and take an appropriate corrective action. The questions in Table 1 address problems in dealing with safety in general and software safety in particular.

While these *data readiness* questions seem simplistic, they can uncover a host of issues that may not be obvious unless the questions are asked explicitly. These questions expose some common problems in implementing a useable, cost-effective HTS and the overall hazard tracking approach:

- **Software Hazard Identification.** Safety-related requirements are not identified as such and hazard controls are not identified as software-related safety requirements, if they are. This can demonstrate inadequate attention to software safety.

Table 2: *Software Safety Visibility Needs*

Inquiry Area	Goal	Software Safety Visibility Questions
Software Safety Analysis Process	Check how well each organization, system, and integrator is addressing software safety in the system hazard analysis process.	○ Have a reasonable number of software safety-related requirements been identified?
Hazard and Mitigation Identification	Check if a reasonable number of software-related hazards, causes, controls, and verifications are identified.	○ Have a reasonable number of software safety hazards been identified? ○ Are causes, controls, and verifications being generated over time? ○ Does every cause have at least one control? ○ Does every control have at least one verification?
Hazard Monitoring	Check if software-related hazards (and hazard software components, i.e., causes, controls, and verifications) are identified and closed at an appropriate rate.	○ Have the number of open software causes/controls for hazards decreased over time?
Appropriate LoR for Software Safety	Check if the various software development groups are assigning reasonable levels of rigor to safety-related software.	○ Have the appropriate levels of rigor been allocated to software development?
Safety Defects	Check if software safety-related defects are being dealt with.	○ Have safety-related software defects been closed at a reasonable rate over time?

- **Hazard Traceability.** The HTS does not provide sufficient linkages among the requirements documentation system, the test plan, or to the defect tracking system. Hazards must be bi-directionally traceable to requirements, tests, and defects in order to verify complete coverage, determine comprehensiveness of the hazard analysis, and ensure that the hazard data represents the system accurately over time.
- **Data Integrity.** Hazards, causes, and controls may not be described in sufficient detail to be understood and verified. The information in the hazard tracking system must be accurate, clear, and specific in order to understand and track hazards throughout the development and deployment of the system.
- **LoR.** There may be difficulty in differentiating among different levels of rigor for the various software safety requirements and identifying, assigning, and tracking the appropriate LoR to specific software components that implement the safety-related requirement. Lack of proper LoR differentiation can lead to inadequate attention on high-risk hazards or too much attention on low-risk hazards. Additionally, the trade-off between higher levels of rigor and their associated higher costs must be considered in order to assess the *right* balance of LoR distribution. An LoR should be assigned and traceable from requirements through design to code.

Many HTS problems are caused by an inadequate vision for the use of the HTS, such as when it is viewed as a storage repository rather than an analysis tool. It is important to make sure that (1) the HTS has adequate functionality, quality checks, and documentation; (2) there is traceability and synchronization among the various support systems (e.g., the HTS and the requirements management system and the defect tracking system); and (3) the quality of the data is monitored to minimize the need to scrub the data later on. The cost of not adhering to this advice is high rework costs and lower than desired system safety. Addressing these issues should simply be a part of the software safety development process.

Laying the Measurement Foundation

Once it is clear that the safety process has been established, deeper investigation of each inquiry area can be performed. An

Inquiry Area	Measure(s)	Model(s)	Response(s)
Software Safety Analysis Process	Percent Software Safety Requirements (PSSR) **PSSR =** # software safety requirements / # software requirements *100	***if*** **|PSSR - EPSSR| < e** ***then*** a reasonable number of software safety requirements have been identified ***where*** **the EPSSR =** the average of the PSSRs for all systems in the family, (*in line with other systems*) and **e =** σ (EPSSR) (i.e., standard deviation of the PSSRs used to calculate EPSSR) ***or*** **EPSSR** = #system safety requirements / #system requirements * 100, (*in line with system safety in general*) and **e** = 20% of EPSSR	PSSR not being within the range of EPSSR should indicate the need for a management action. For example, check into the safety hazard elicitation process and whether it is being applied right, investigate the reason why the system under consideration has such a small (or large) percentage of safety requirements, and develop a "get well" plan. If the value is too large, what are the cost and schedule implications of corrective actions?
Hazard Monitoring	Hazard cause/control closure evolution (HCCE) $HCCE_{i,3} = MA_{i+1,3} / MA_{i,3}$ ***where*** $MA_{i,3} = (X_{i-2} + X_{i-1} + X_i)/ 3$ is the moving average of the set of open causes (controls) at three consecutive time intervals.	If $HCCE_{i,3} \geq 1$ then the closure rate of hazard software causes/controls is not converging	If the number is ≥ 1 and it is not in the beginning phases of development, more effort should go into closing the hazard software causes/controls. If it is because the opens are increasing too fast (new hazards are being introduced, new causes for existing hazards), then investigate the reasons. If it is because the closes are not increasing fast enough, then investigate the reasons. Graphing the cumulative identified, open, and closed causes/controls provides good insight into the trends of these variables
Safety Defects	Count by priority of open safety-related software trouble reports at time i (COSRTR)	If COSRTR ≠ 0 then there are open defects that need further analysis	If all safety-related defects are not closed, then create a list of open defects, prioritize them, and investigate why they exist. This measure should be taken periodically starting at the beginning of test and up until safety assessment report delivery.

Table 3: *Some Examples of Software Safety Measures*

example set of software safety visibility goals and questions is presented in Table 2. When a readiness assessment question has been satisfied, the software safety visibility questions and measures throughout the life cycle of the program can be applied.

Establishing the measures requires more than identifying the data to be collected. Each *measure* is characterized in terms of the *question* it answers, the *model* used to interpret its values in order to answer the target question, the *response* that suggests the action to be taken based upon the answer to the question, and the *scope* of applying the measure. Table 3 presents examples of *models* and *responses* for three of the five inquiry areas[1].

For each model, assumptions were made about how the resulting measurements should be interpreted. An *expected value* and a *range* are selected for within which the actual is acceptable. The expected value can be derived by: (1) historical data from past programs, (2) prior data from the current program, (3) proxy estimate (i.e., comparison with something similar), or (4) expert estimate. The range of the expected values can be based on general distributions, or specific or related experience.

If the calculated value is not within the expected range, then there may be a problem. Expected values or ranges can be improved over time based upon the incorporation of new data into the model.

To illustrate these concepts, consider one measure proposed for the process area, PSSR, which is defined as *PSSR = # software safety requirements / # software requirements *100*. The model can be defined as:

***if* |PSSR – EPSSR| < e**
where EPSSR is the estimated value of PSSR, e is the acceptable threshold for deviation from the estimate, and (EPSSR -e, EPSSR +e) is the acceptable range,

***then* a reasonable number of software safety requirements have been identified.**

The key is to have good estimates for EPSSR and *e*. Ideally, historical data should be used and the estimated value and range (i.e., sigma, the standard deviation) is taken from a similar system or subsystem. However, there may be little historical data. In this case, proxies are identified for estimates[2].

One possible proxy is to use system safety requirements as the benchmark for software safety requirements. We can let the range be defined by some percentage around that value that provides initially acceptable limits. Once the program is under development, early data can be substituted on the program for these proxies. Thus:

EPSSR = #system safety requirements / #system requirements *100

and e = 20 percent of EPSSR.

The model is interpreted by defining a *response* if the resulting value is not within range. For example, if PSSR is not within the range *e* of EPSSR, it indicates the need for management action. One example would be to check into the safety analysis process and whether it is being appropriately applied, investigate the reason why the system under consideration has such a small (or large) percentage of safety requirements, and develop a "get well" plan.

In defining these measures, existing data sources (e.g., hazard tracking database, requirements management repositories, and defect tracking systems) and processes (e.g., safety analysis processes) were leveraged. This can be done provided that the assumptions upon data collection listed in the Terminology and Key Concepts section are true. The derived measures in Table 3 can be graphically represented (e.g., as evolution over time), as appropriate, for the analysis results on the questions it helps to answer. Key issues for determining software safety visibility are: (1) selecting the right subset of measures, (2) defining appropriate thresholds, (3) determining appropriate management responses, and (4) providing user-friendly reports and actionable responses; all of these issues are program-dependent.

The safety measures collected by a program form the beginning of an experience base, which creates a historical base across current programs within a program and for future programs. To date, there is very little data on which to calibrate the models. It is hoped that programs will start collecting data so that more knowledge can be obtained and software safety measures baselines can be established.

Conclusion

The methodology presented here should be tailored to fit the context of the organization; it is not intended to imply a correct answer or an *all or nothing* approach. The areas of inquiry and the measures can be adjusted appropriately; however, as a minimum, any program dealing with safety should at least address the readiness questions.

Gaining visibility through objectives measures into software safety has become increasingly important for today's software-intensive programs. Although software safety measures cannot determine whether a system is safe, they can provide valuable indicators of problems and risks that give management critical knowledge for making timely and well-informed decisions.◆

References

1. Radio Technical Commission for Aeronautics, Inc. "Software Considerations in Airborne Systems and Equipment Certification." RTCA DO-178B. 1 Dec. 1992.
2. Basili, V., and D. Weiss. "A Methodology for Collecting Valid Software Engineering Data." IEEE Transactions on Software Engineering Nov. 1984: 728-738.

Notes

1. The *question* is omitted from this table due to space limitations; for *scope*, we assume that these measures apply to the entire system.
2. This argues for the need to accumulate data on programs, not just for the good of the current program but for use in future programs.

Additional Reading

1. Joint Software System Safety Committee. Software System Safety Handbook. Dec. 1999.
2. MIL-STD-882. DoD Standard Practice for System Safety. 10 Feb. 2000 <http://safetycenter.navy.mil/instructions/osh/milstd882d.pdf>.

About the Authors

Victor Basili, Ph.D., is a professor of computer science at the University of Maryland and Chief Scientist at the Fraunhofer Center - Maryland. He works on measuring, evaluating, and improving the software processes and products.

E-mail: basili@fc-md.umd.edu

Frank Marotta is a mathematician at the U.S. Army Aberdeen Test Center and has more than 20 years experience in software testing of Army weapon systems, test optimization, and software metrics.

E-mail: frank.marotta @us.army.mil

Kathleen Dangle is a division director at the Fraunhofer Center - Maryland where she works with organizations to implement software management-related improvements, such as software measurement, acquisition, Capability Maturity Model® Integration-based processes, and helps create learning organizations.

E-mail: kdangle@fc-md.umd.edu

Ioana Rus is a member of the Honeywell Aerospace Software Center for Excellence. She works in process modeling and simulation, empirical studies, software dependability, and measurement.

E-mail: rus.ioana@gmail.com

Linda Esker is a senior engineer at the Fraunhofer Center - Maryland where she provides expertise to government programs in program management, software development, as well as metrics definition and analysis.

Phone: (301) 403-8967
E-mail: lesker@fc-md.umd.edu

® Capability Maturity Model is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.