

DETERMINING SOFTWARE (SAFETY) LEVELS FOR SAFETY-CRITICAL SYSTEMS

Doris Y. Tamanaha

dtamanaha@west.raytheon.com

Meng-Lai Yin

mlyin@west.raytheon.com

*Raytheon Systems Company
Loc. FU, Bldg. 675, M/S AA341
1801 Hughes Drive, Fullerton, CA 92834*

ABSTRACT

For safety-critical software-intensive systems, software (safety) levels are determined so that the appropriate development process is applied. This paper discusses issues of applying the results of fault tree analysis to software (safety) levels determination. In particular, the inconsistency problem, i.e., inconsistent software (safety) levels, is addressed and an approach is presented.

Keywords: Fault tree analysis application, safety-critical systems, software (safety) levels.

1. INTRODUCTION

For safety-critical systems, process requirements to develop the software need to be met. Several standards have been evolved which classify processes into levels, such as the RTCA/DO-178B “Software Considerations in Airborne Systems and Equipment Certification” [1] or the Software Engineering Institute Capability Maturity Model [2][3][4]. Applying the results of fault tree analysis to determine the software (safety) levels is proposed in this paper, since fault tree analysis has been widely used for safety-critical systems.

For large stringent systems, inconsistent software (safety) levels can occur. This is due to the various concurrent activities of different organizations, e.g., the software development group, the system architecture group, and the safety group. This paper describes the inconsistency problem, and the strategy and methods to deal with this problem. The goal is to ensure that appropriate software (safety) levels are applied to the developed software.

2. DETERMINING LEVELS

2.1 Software (Safety) Levels

Software (safety) levels determine the associated process to be followed by the software developers. There are several existing development processes defined based on software levels, such as the Capability Maturity Model (CMM) by the Software Engineering Institute (SEI), the ISO 9000 series of standards by the International Organization for Standardization [2][3][4], and the RTCA/DO-178B [1]. The discussion here focuses on the RTCA/DO-178B standard.

The RTCA/DO-178B “Software Considerations in Airborne Systems and Equipment Certification” [1] provides guidelines for the production of software for airborne systems and equipment [5]. In particular, five categories are identified for the failure conditions, i.e., catastrophic, hazardous, major, minor, and no effect. Five software (safety) levels are defined accordingly, i.e., level A, B, C, D and E. The software (safety) level determines the development effort that demonstrates compliance with certification requirements.

2.2 General Rules

The top-down methodology based on the fault tree models is fairly straightforward. The fault tree considers not only the events related to the software, but all the possible events that can cause the top event (root event). The methodology first determines the safety level of the top event, then follows the 2 general rules listed below: (1) For the events under an OR gate: the safety level of these events are the same as that of the top event of the

OR gate. (2) For the events under an AND gate, three cases are distinguished: (2a) If the event is associated with a monitoring function, i.e., that it monitors some other function(s), then this event has the same level as that of the top event of the AND gate. (2b) If the event is associated with a monitored function, i.e., its function is monitored by some monitoring function, then it can have a level lower than that of the top of the AND gate. The philosophy is that we believe the failure of this function can be detected and corrected by the monitoring function. (2c) If the events under an AND gate do not have the monitoring/monitored relationship, then they will inherit the same level as that of the top event of the AND gate. However, if these events are truly independent, then a level lower than the top event can be assigned. For the example shown in Figure 1, assuming the effect of the top event (Hazardously Misleading Information) is classified as level B according to RTCA/DO-178B. Then, the safety level for this HMI is B. If IE1 and IE2 are functions that have the monitored/monitoring relationship, e.g., IE1 is the monitored function and IE2 is the monitoring function, then IE1 has safety level D and IE2 has safety level B. If IE1 and IE2 are functions that have the monitored/monitoring relationship, e.g., IE1 is the monitored function and IE2 is the monitoring function, then IE1 has safety level D and IE2 has safety level B.

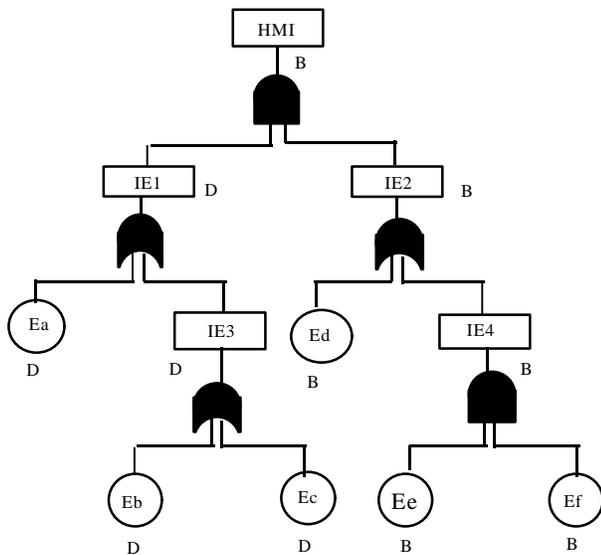


FIGURE 1. EXAMPLE FAULT TREE

For the basic event Ea and the intermediate event IE3, since IE1 has safety level D, they are marked as level D in accordance with rule 1 above. Moreover, for IE4, assuming the basic events Ee and Ef under the AND gate do not have the monitored/monitoring relationship. Thus, they are both marked as level B. For efficiency, minimum cut sets can be used as

assistance. Moreover, some engineering judgement is necessary when conflicts occurred [6].

2.3 The Process

A six-step process relates the fault tree analyses to the software activities is presented in Figure 2. The first three steps are the preliminary marking, whose results are recorded in a database called the Requirement Management System (RMS). The subsystem fault tree analyses and data flow analyses were performed as parts of the preliminary marking. The safety group conducted fault tree analyses, while the software people conducted the data flow analyses. The subsystem fault trees identify *software capabilities* that can cause a hazard. In other words, if a failure of a software capability contributes to a hazard, it is identified in the subsystem fault tree. Thus, the safety level for the software capability can be marked, based on the general rules described above. The marking results need to be integrated into the software development process. The model used for data flow analyses, referred to as the capability model, is marked for this purpose. Finally, the results are recorded into the RMS database.

When the preliminary marking is finished, the software development process moves to the stages of preliminary design and detailed design. It is during this time frame that the software level marking is refined and updated through extending the subsystem fault trees. Extending the subsystem fault trees is based on the information provided by the software preliminary design and detailed design. In software preliminary design, the CSCs (Computer Software Components) of each CSCI (Computer Software Configuration Item) are defined, as are the major data stores and interfaces among CSCs. Thus, the original subsystem fault tree can be extended to the CSC level. In the detailed design phase, the processes are further decomposed into Computer Software Units (CSUs) and functions. Hence, we can extend the fault trees to the CSU level.

Due to the characteristics of large systems that are composed of several organizations with different objectives, inconsistent software (safety) levels are expected. The results of this inconsistency are schedule costs and safety risks. Hence, the

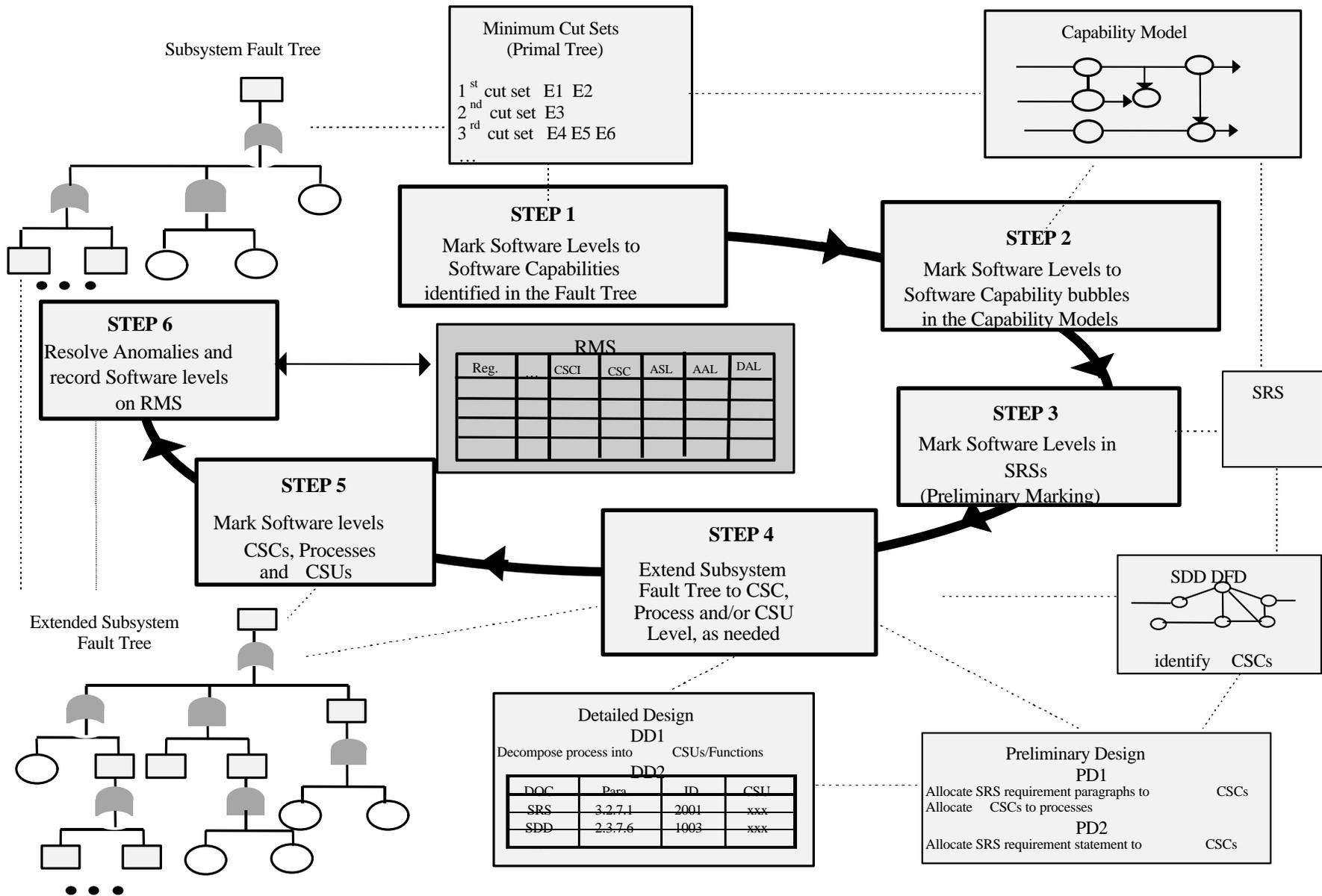


Figure 2. SOFTWARE (SAFETY) LEVEL DETERMINATION PROCESS

inconsistency needs to be resolved so that one accepted development process can be applied. The inconsistency problem is discussed next.

3. THE INCONSISTENCY PROBLEM

3.1 Inconsistent Views

Large programs entail several organizations that have impact on the software (safety) levels. Unfortunately, these organizations often have different views and responsibilities, which may conflict with each other, often as a result of changing requirements that affect revisions at multiple levels, e.g., within the architecture, software, or safety constraints. Hence, resynchronization is needed.

The three organizations that are related to the software (safety) levels are the software development, system architecture, and safety groups, as shown in Figure 3. The software group follows a process to develop the software. System architecture (with software representation) partitions the system and decides which software resides on which platforms. Usually, a single, consistent process is followed for the software developed on the same platform. The safety group analyzes the system and derives software (safety) levels as requirements.

If the problem of inconsistent software (safety) levels is not resolved, the software may not be developed appropriately. If the inconsistency is not resolved in a timely manner, schedule will be slipped, and cost will be increased. Moreover, if the inconsistency is not resolved correctly, the software development process can be inadequate. In short, the inconsistency problem needs to be resolved correctly and in a timely fashion in order for the system to be built.

4. THE APPROACH

4.1 The Basis

A basic philosophy we took is that the inconsistency is expected. Therefore, the existence of all the software levels shall be recorded. From there, the inconsistency can be identified. Only if we can recognize the inconsistency can the inconsistency problem be addressed and resolved.

There are two types of inconsistencies. The first type is referred to as the tolerable inconsistency where the software developers follow a process that exceeds the current process requirements. This tolerable inconsistency implies that developed software can be used in later phases when a higher level is required (software levels are interpreted as $A > B > C > D > E$, e.g., level A is higher than level B, etc.) The second type of inconsistency is intolerable, where the process that the developers follow does not meet the current process requirements for safety certification. Intolerable inconsistencies must be identified and resolved.

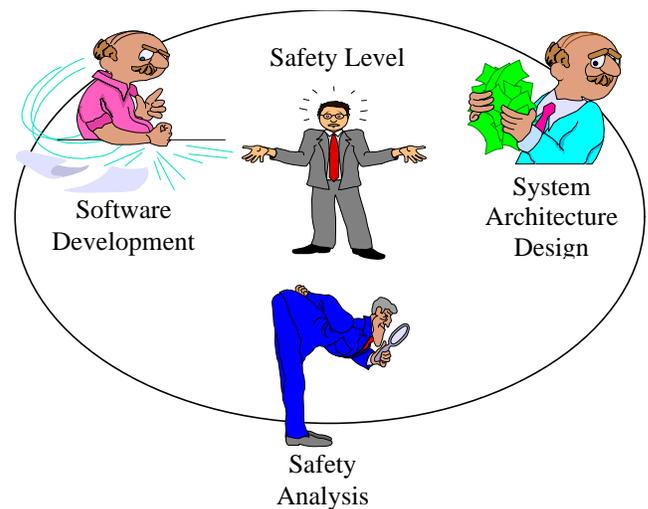


FIGURE 3. DIFFERENT VIEWS OF SOFTWARE (SAFETY) LEVELS

4.2 The Process of Managing Inconsistent Software (Safety) Levels

To manage the inconsistent software (safety) levels, three major steps are proposed: (1) maintaining software (safety) levels associated with different organizations, (2) cross-checking the recorded software (safety) levels and identifying the inconsistency areas, and (3) resolving the inconsistency by involved organizations and engineers.

A process is defined to manage the inconsistency problem, as shown in Figure 4. The process is iterative, since the system development itself is iterative. A central repository, e.g., the Requirement Management System database, is maintained as which is the center of the process. Configuration

control of the database is necessary to ensure the overall consistency of the software (safety) levels. Cross checking different software (safety) levels will identify the inconsistent areas. To resolve the problem of inconsistency involves the teamwork efforts of software engineers, system engineers and safety engineers.

4.3 Maintaining the Software (Safety) levels

Because inconsistency is expected, all the differences can be managed. The strategy is to record all the software levels resulting from different organizations, recognize any inconsistency, and deal with the intolerable inconsistency problems. The different software (safety) levels (due to the various organizations) are recorded in the Requirements Management System database. Three different software (safety) levels are maintained, the “Assigned Software Level” (ASL), the “Assessed Architecture Level” (AAL), and the “Development Assurance Level” (DAL). A detailed description of each of the levels is addressed below. The goal of managing the inconsistency is to assure that $ASL \leq AAL \leq DAL$ (software levels are interpreted as $A > B > C > D > E$, i.e., level A is higher than level B, etc.).

The ASL focuses on the severity effects and hazard mitigation. The ASL is assigned based on the system safety assessment results, e.g., the fault trees, as described in Section 2. Note that this ASL serves as the minimum acceptable software (safety) level, as it is based solely upon the referenced safety analysis and functional mitigation. The ASL is implementation independent. The AAL is used to reflect design constraints from architectural allocation of software capabilities. To prevent software developed at a low level process from corrupting software developed at a higher level process, several mechanisms are considered, such as the firewall concept. However, to reduce the complexity of the inconsistent process throughout the whole system, a “safety system high” concept is used. The AAL is determined based on the *highest* severity level of software assigned to that platform. For example, to simplify the development process management, all software developed on a platform certifiable to level B is developed to level B, even if the software has an assigned safety level or ASL of D.

The incremental strategy has been widely used for large safety-critical systems. Not only because a program needs to improve as the equipment and technology improve, but also because the safety concern is changing as the phases proceed and the system becomes operational in a production sense. To prevent rework efforts as much as possible, the DAL can be used to demonstrate the achievement of compliance to final phase requirements. This DAL is committed to by software development and is a development strategy to meet or surpass the current AAL requirement. To accommodate the incremental strategy, a separate set of the three software (safety) levels is maintained. An advantage of maintaining different software (safety) levels is that it helps an individual organization to focus on its own tasks. In particular, the software development team only needs to focus on the DAL and develops the software to the assigned DAL. It is the safety engineers responsibility to assure the relationship of $ASL \leq AAL \leq DAL$.

5. RESULTS

5.1 Identified Anomalies

Anomalies are the intolerable inconsistencies. This section describes the anomalies that are identified during the implementation of the process, and a general process of how to resolve these anomalies. Two types of anomalies are recognized, e.g., internal and external. The internal anomalies are due to the sharing of the same software requirement by different software configuration items. For example, a system service function may be used in several subsystems that are on different platforms. Different failure effects may be estimated, since different safety concerns are applied to different subsystems. As a result, different ASLs are assigned for the same software requirement. The internal anomalies can be resolved by assigning the *highest* software (safety) level to the software being concerned. In other cases, similar design may be used on platforms at different certification levels. In those cases, the safety level for the requirements allocated to the similar design carries a dual designation, say “B/D”. It is then understood that

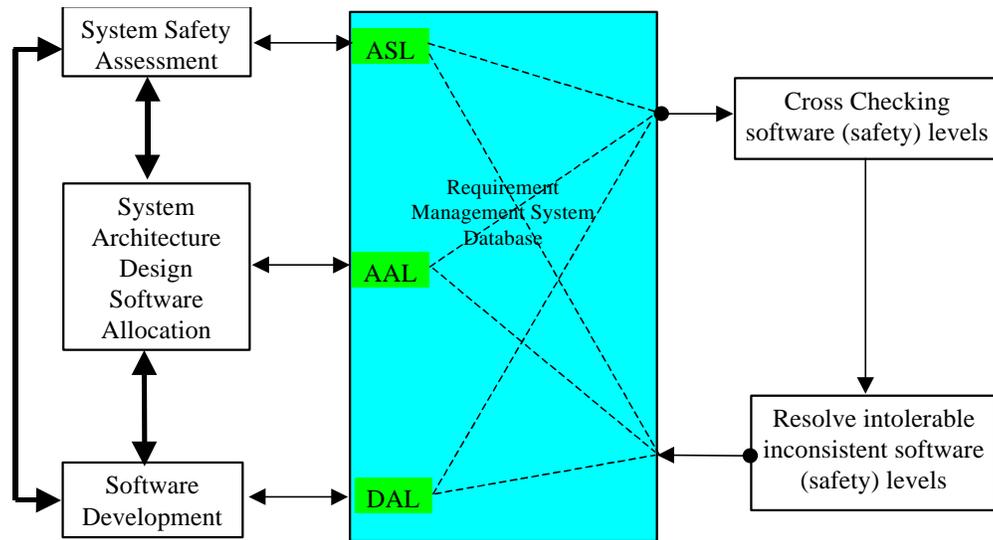


FIGURE 4. PROCESS OF MANAGING INCONSISTENT SOFTWARE (SAFETY) LEVELS

those requirements must be examined for two processes.

External anomalies are caused due to the inconsistency. The identified external anomalies are: (1) The DAL for a particular entry in the database is denoted as N/A (non applicable) or the value is missing, while the corresponding ASL (and/or AAL) has a level assigned. This occurs especially when COTS (Commercial Off The Shelf) products are used. (2) One of the software (safety) levels (AAL, or DAL) has two values assigned, while the other one has only one level. (3) The DAL is lower than the AAL and ASL.

5.2 Resolving Anomalies

The safety engineers, software development team leads, and system engineers are informed of the anomalies that have occurred. For each anomaly identified, corresponding safety engineers and software engineers work together to resolve the problem. Once the anomalies are resolved and a consensus is reached, a “Software Change Control Board” reviews and approves the request for level changes. This satisfies the issues of configuration control for the database. Software safety engineers assign the ASL and AAL changes that are reviewed internally by the software safety team. Moreover, software safety engineers participate on the Software Change Control Board with sign-off capability for all three software levels, i.e., ASL, AAL, and DAL. Recall that the goal of this process is to assure that $ASL \leq AAL \leq DAL$.

6. CONCLUSION

In this paper, we present a method of determining software (safety) levels based on fault tree analysis. The inconsistency problem resulting from the need to operate concurrent activities to meet schedules in building large, complex systems is addressed and a strategy of handling it is discussed. The software levels determined using this approach demonstrate the safety quality of a safety-critical system. Moreover, the approach is suitable for systems developed incrementally. Extensions being investigated are the relationship of software (safety) levels to other analysis approaches, e.g., safety-critical thread analysis and the use of software fault-injection techniques to harden the software itself, guided by the software level markings.

7. REFERENCES

- [1] *Software Considerations in Airborne Systems and Equipment Certification*, Document No. RTCA/DO-178B, prepared by Special Committee 167 of RTCA, December 1, 1992.
- [2] *A System Engineering Capability Maturity Model*, Version 1.1, System Engineering Capability Maturity Model Project, Carnegie Mellon University Software Engineering Institute, SECMM-95-01 CMU/SEI-95-MM-003, Nov. 1995.
- [3] *The Capability Maturity Model: Guidelines for Improving the Software Process*, Carnegie Mellon University, Software Engineering Institute, Addison-Wesley Publishing Company, 1995.

[4] Mark C. Paulk, "How ISO 9001 Compares with the CMM", IEEE Software, January 1995.

[5] *Global Position System: Theory and Applications*, Volume I and II, American Institute of Aeronautics and Aeronautics, Inc. 1996.

[6] G. Watt, "Phase 1 Software Level Marking Guidelines", WAAS SEN 5-2-5, 1997.