Software in Safety-Critical Applications

Inga-Lill Bratteby-Ribbing; Defence Materiel Administration; Uppsala, Sweden

Keywords: software, safety, reliability, real-time systems

Abstract

What requirements and safety properties are relevant for software in safety-critical applications? Where do you find the causes for software failures leading to accidents?
These are some of the safety issues addressed in a handbook for software in safety-critical applications developed for the Swedish Armed Forces at the Defence Materiel Administration (FMV). The handbook includes general software safety requirements structured w.r.t. the parties involved in an acquisition, the personnel qualifications, the processes, the production environment and the product. This structure is useful at specification, verification and as a base for safety case argumentations for all types of safety-critical applications –not only defence systems.

This paper illustrates how different defects might be activated and how the effect of these can be prevented. It presents a categorization of software deficiences and emphasizes the importance of a safety view in the hierarchy of architectures. Finally the contents of the FMV software safety handbook is described.

Introduction

Terminology: An application is *safety-critical* if it has a potential to cause harm to people, property or environment. A primary concern is a striving for *safety* –freedom from accidents or losses (refs. 1-3). A *system safety* approach is adopted to achieve a system where mishaps are avoided, prevented or at least kept to a tolerable level.
Software safety is the discipline of system safety applied to the software processes, the people involved in these and the products. For a software product the safety properties will depend on the surrounding system, its environment and usage. A software component which in a specific context has an evidenced execution record free from accidents or failures will in another (even similar) situation not necessarily show the same behaviour. An evaluation of its safety properties has to be made with software in the intended context. This is why software reuse (and especially the inclusion of COTS) is risky in safety-critical applications.

Software safety and reliability: The discipline of software safey is a combination of techniques from traditional software engineering, reliability and system safety. Different types of reliability techniques are used to avoid, detect and eliminate faults and to provide specified functionality despite remaining faults (fault tolerance). With a combination of methods the absence of failures from the lowest component up to the topmost system level is demonstrated:

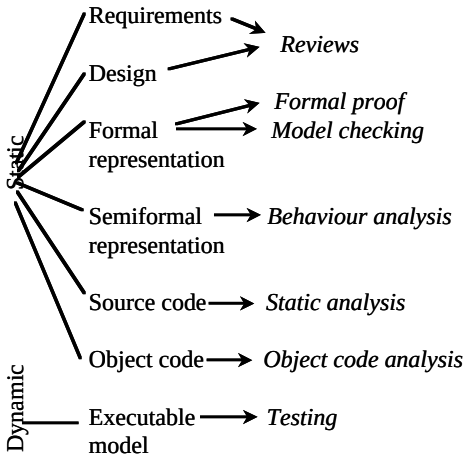

Figure 1 – Reliability verification

System and software safety are instead applied top-down, from system to component level, using various types of safety analysis techniques to find hazards which together with other conditions in the system and its environment could result in accidents. In this way not only a single root cause can be identified and prevented, but also combined effects from several parts, where each may have a previous record of a reliable behaviour. The priority order in both cases is 1) to eliminate root causes for failures/accidents, 2) to prevent their propagation and 3) to keep remaining failures/risks on a level defined as tolerable for the system in its intended environment and usage.

Safety and reliability are related but not equivalent and serve different purposes. In reliability the main concern is the realization and verification of the specified functionality. *Use case scenarios* are studied to find how to break the chain of events from a *fault* over to an *error* (erroneous state) to a *failure* on the system level (ref. 4). Focus is on desired features not fulfilled.
In safety the attention is on undesired features that should not be fulfilled. Possible *anti-scenarios* are identified and analyzed with the intent to disable the chain of events from a *hazard* which –together with other conditions– eventually will lead to some *unsafe state* resulting in an *accident.*

For sofware the question whether a malfunction in a component will remain silent or trigger either a failure or an accident chain will depend on actual circumstances during the operation.

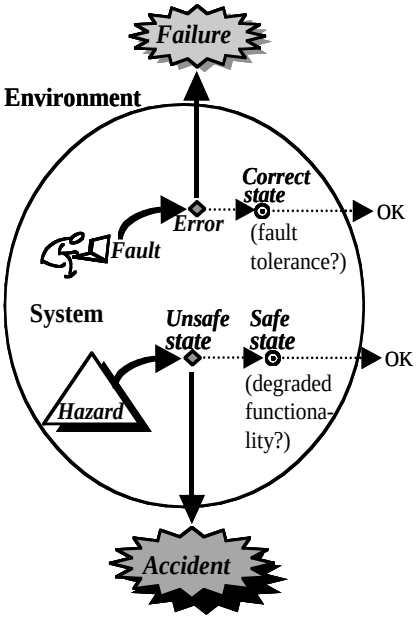

Figure 2 – Failure and Accident Scenarios

Some examples of reuse

Case 1: An illustrative example is Ariane 4, a system with more than 10 years of successful operation, where a modification changed the software behaviour jeopardizing the entire mission of the new version, Ariane 5. The immediate cause of the accident was a self destruction triggered by a safety function due to unreasonable flight data (a reaction according to the specifications, ref. 5). A trace back in the chain of events from the accident showed that diagnostics had been interpreted as flight data and was produced due to the inoperability of both the master and the stand-by systems. This was an effect of an unhandled overflow situation in the software: the modified system was equipped with a stronger engine resulting in a different trajectory causing an overflow (higher horizontal velocity values in early flight mode than those possible and expected for the original system). The contributing factors in the chain of events could be attributed to deficiencies in the interface specifications (expected trajectory data only implicitly given as conditions in the code, incomplete documentation of the interaction between the platform and the flight control). Furthermore, the design contained unhandled exceptions, a mode-specific function for count-down operational also in flight mode (where its result was of no interest) and a fault tolerance ineffective for software errors. Apart from these defects in the software product some oversights in the production process were found afterwards; no safety analysis on the effect of introduced changes were made; the verification procedures were insufficient.
Even if the root cause could be attributed to the specifications it is interesting to note that if only one of these contributing factors had not been present, a break in the chain of events would have been possible. This case stresses the importance of software quality factors and general safety principles –especially during reuse in safety-critical systems. It may be possible to plug in and run a software component in a different context, even if

some of its prerequisites are not met. These conditions may be evident for the original system (and therefore not explicitly documented), but in a new system (even a closely related one), where some of these assumptions not are valid, the execution may take other paths than those possible for the original system.

Case 2: The second example of erroneous reuse is a system for automatic detection and destruction of attacking aircraft (ref. 6). The system was specified to be operational for 14 hours, but was used several days (100 hours) without the stipulated restart and for tracking targets with a velocity several factors higher than the intended target. The code was not built to compensate for rounding errors accumulated over a longer period of time. The effect on time and – as a consequence – distance calculations after 4 days of continuous operation was fatal; a second echo from a target (assumed to be within the intended velocity range) was interpreted as a new target. No automatic launch and interception was issued resulting in heavy casualties.
In this case, the accident had its origin in deviations from the specified usage domain (different target velocities) and from the operational instructions (no restart).

These two cases show that a component with a safe behaviour in the intended environment and usage may at reuse, in a different situation, or in a modified system cause a failure leading to an accident. For COTS products with limited access to internal details, the possibilities to analyze the effect of changes and the resulting behaviour will be even more curtailed.

Software deficiencies

Software faults and hazards can often be traced to the different instances representing the software during its life-cycle. Analysis of failures, incidents and accidents attributed to software have also shown that the majority emanated from the requirement specifications (refs. 3, 7-8). Nevertheless it is important to realize that other types of deficiencies than those in the software product may influence the outcome. If these could be avoided a propagation of software failures into accidents may be prevented.

The following is a categorization of software deficiencies, which can be found either in the

a) Specification of system, interfaces and operational conditions
b) Design and implementation
c) Production process
d) Production tools

or are due to erroneous usage

e) Reuse under changed conditions
*f) Usage domain different from that specified for the syste*m
g) Procedures in violation of instructions and specifications

In the reuse examples above the defects in the first case were of types a,b,c and e, while those in the second one were of types f and g.

Software reliability versus failure

Reliability is an **ability** to perform a requested function or service satisfactory for a prescribed time under specified environmental conditions. The property is often evalutated in terms of its opposite (the inability to the above, i.e. *failure*). For hardware the **probability** of a failure free performance is used as a measure.

Deficiencies in software of type a-d above are logical faults introduced in the production phase. To quantify the failure intensity of a software part and to assess that a specified value has been met is more complicated than for hardware with random failures (one reason why software reliability often is expressed in terms of **likelihood** rather than probabilities).

Software reliability models: A number of software reliability models exist for prediction versus estimation of failure rates (refs. 9-10).
A *prediction model* is based on characteristics of an existing or a similar program (developed in a similar environment by a similar process). The latter requires some caution, since projections even from a very closely related system might fail (cf the successful history record of Ariane 4).
An *estimation model* uses statistical techniques to draw conclusions from failures observed from a large number of tests or real use cases. Among the assumptions are that data is taken from a static context –no changes are allowed to the production process, the user profile, the system parts or the environment. Some caution in selecting an appropriate model is necessary. Those assuming that the failure rate is directly related to the number of faults in the program should be avoided. For systematic faults a small share of these may be responsible for a majority of the failures. The opposite may also be true: the failure rate may be low –despite a large number of

remaining faults (ref. 11). Another important aspect is that statistic failure analysis on software does not give an estimation of the probability that a random fault will occur, but rather that a number of random test cases within the user profile will encounter a systematic fault or error. This is why changes in the environment or the usage profile may be fatal even if the system or software is intact: the execution may branch off into other paths triggering previously unknown bugs.

Testing effort: Another complication is the amount of tests required. To demonstrate a failure rate less than 10^{-4} per unit (usage or time) with 95% confidence will require more than 30 000 tests –an upper limit of what might be feasible (ref. 12). Every factor 10^{-1} of the above failure rate would require 10 times more tests.
Difficulties in achieving reliable, quantitative estimates of software failure rates (especially at higher criticalities) means that the effort has to be directed towards qualitative methods –not only to assess the resulting failure probability once the product is built– but more importantly to include properties and procedures during the design that can prevent deficiencies to propagate into failures or accidents. In the Ariane 5 case better quality in the production process and the product itself could have made this achievable.

Safety architectures

How can the overall safety properties be incorporated into a safety-critical system? Among the most important activities in any construction is the establishment of a well-defined architecture. For a "system-of-systems"-structure this will rather take the form of a hierarchy of architectures, with the principles for interaction between included systems and their environment defined on each level. For a safety-critical part the architecture has to be complemented with a safety view satisfying the overall safety approach, e.g. strategies for how (on which level and by which means) different types of safety threats should be met. In particular for the software system the software components interacting with safety-critical equipment and environment have to be identified together with features responsible for preservation of the overall system safety. The principles for risk reduction are applied in order to 1) *eliminate* threats from the environment and other parts of the system, 2) to *monitor and control* safety-critical equipment to keep remaining threats on a tolerable level and 3) to provide *protection* preventing and mitigating any damage that a remaining threat may imply.

Basic software safety requirements: The safety-critical software is responsible for the second and third task above, leading to requirements on *safety* for control features and on *reliability* for safety features (e.g. supervision or protection). In addition, interference from features of lower criticality with those of higher criticality must be prevented. This leads to requirements on *independence* for critical software w.r.t. these other parts.
Independence: There are several ways to achieve independence between parts or features of different criticalities. Most straightforward is *physical separation* (e.g. system functions of different criticalities placed in separate processors). Among the benefits are a simplified fault tolerance (faults in one processor can more easily be prevented to propagate to another processor). For some applications weight, volume and an inappropriate separation of functions that dynamically are closely coupled might be a problem (ref. 13). Moreover, the possibilites to meet hardware failures by a dynamic reconfiguration of the software on the remaining resources will not be available. These are some reasons for a growing interest in techniques for *logical separation* (a subject within the research on Integrated Modular Avionics). Incapsulation, safety kernels, -ports and -firewalls are among the techniques suggested (refs. 14-16). The logical separation implies requirements also on *temporal separation* allowing critical software to access shared resources without interference from parts of lower criticality. Thus requirements on independence of critical parts or features within the application software will have a far-reaching effect down to the embedded software and the underlying operating and run-time systems. These requirements have to be considered early –already during top design– and will affect the architecture.

The basic safety requirements will be applicable to features in the left part of the simplified architecture below.

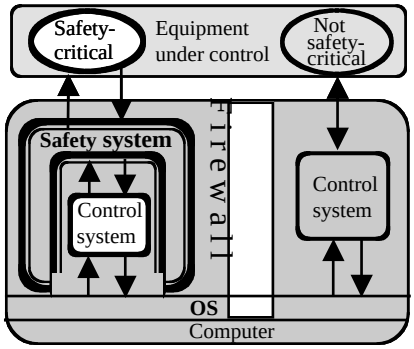

Figure 3 – Safety Architecture

Software Safety for the Swedish Defence

Early system safety activities in the Swedish Defence were carried out separately for the different Armed Forces to focus hazardous material in the systems. The system safety work has since then evolved from accident-preventing safety procedures (improved after retrospective analysis of occurred mishaps) towards a joint and proactive system safety approach as defined by the System Safety Manual (refs. 2, 17). The manual defines the overall system safety requirements and procedures (safety organization, responsibilities, interactions and methodology) in a terminology close to MIL-STD- 882C (ref. 18).
The increased use of computers in safety-critical applications and shift from more hardware-orientated, operator-driven realizations, to software implementations have created a demand for a system safety approach addressing software. A new software safety manual has been developed in response to this need (ref. 1).

Software Safety Manual:
The handbook provides directives and instructions for the Swedish Defence at procurement of safety-critical software systems. It is based on the above system safety manual and international standards on software life-cycle processes, quality management and quality assurance (refs. 2, 19-20).

Document structure: Included requirements are structured after the stakeholders in an acquisition process (Customer, Purchaser, Contractor), requested qualifications for the personnel, the processes, the production environment and the different instances representing a software product.

Criticality classes: The requirements in the handbook are numbered and classified after criticality (High, Medium, Low). The higher the criticality the more rigorous the requirement addressing a specific software aspect. Variant formulations are therefore given (e.g. a requested test coverage may be w.r.t. source code or object code). A necessary condition in allowing requirements of different criticality in the same system is that parts of higher criticality are unaffected by those of lower criticality (see “Safety architectures”).

Requirements: Four types of safety requirements are identified for software
- Basic requirements
- General safety requirements
- Domain-specific safety requirements
- System-specific safety requirements

The document includes basic requirements and general safety requirements. Part of the project work is to select general safety requirements relevant for the actual system. The outcome of different safety analysis on the system is then used to identify new safety requirements specific for the system in its intended environment and usage. These are refined as the inner structure of the critical parts evolves.

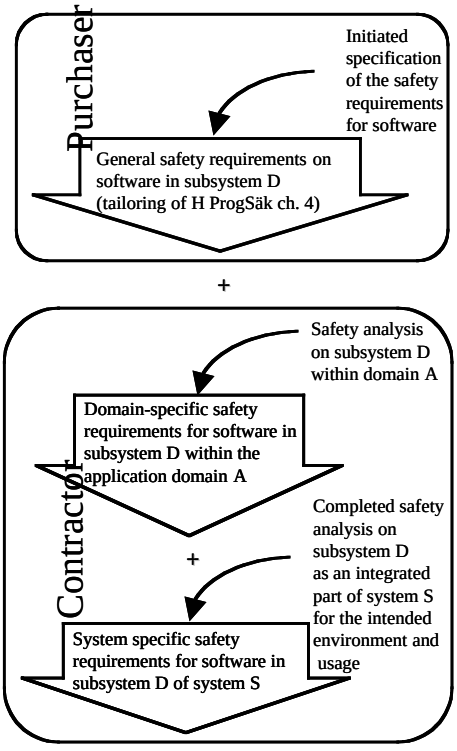

Figure 4 – Safety requirements refinement

General safety requirements: The high-level safety principles applied to software result in general safety requirements. A fundamental safety property with a profound impact all the way down in the software structure is *predictability* –a necessity in performing safety analysis (to find out before operation the system behaviour in different situations from given conditions). Predictability for a programming language means, that ambiguous language constructs and features have to be avoided: functions with side effects, with results dependent on the evaluation order of the compiler or dependent on the parameter transfer mechanism (by value, by reference). Dynamic constructs, where it is impossible to decide whether limited resources (e.g. memory, heap, processor time) will suffice during the entire execution period (e.g. dynamic allocation, garbage collection, recursion, exception handling) can not be used without restrictions –especially for continuous real-time systems (less so for on-demand systems).
One way to promote this type of safety aspects is to use a language allowing automatic blocking of certain constructs together with a lean run-time system avoiding unpredictable language elements. A good combination here is Ada with the Ravenscar tasking profile (refs. 16, 21-23), the latter also providing support for logical and temporal separation (see above). Further restrictions on the program can be enforced by use of the SPARK subset, where additional information is given as annotations (in form of Ada comment) enabling given pre- and post-conditions to be checked by formal tools (ref. 24).

Informative parts: Most part of the software safety manual is informative. Each section containing requirements start with an introductory text clarifying the context. Footnotes are used for short comments and examples. A denotation list is provided showing variations in meaning between different publications. Software engineering issues are highlighted (e.g. safety versus new acquisition or development models). Tailoring instructions, checklists and short summaries of related publications are given.

Current status: The manual has since 1997 evolved over a number of working issues on review and trial both at FMV and in the industry. A Swedish version was completed 2000 to be approved in 2001. An English version will appear later the same year.

References

1. I.L. Bratteby-Ribbing, Handbok för Programvara i säkerhetskritiska tillämpningar (“Handbook for Software in Safety-critical Applications”), H ProgSäk, Armed Forces book and form store, M7762-000531, www.fmv.se, 2001.
2. H SystSäkE, System Safety Manual, Armed Forces book and form store, M7740-784861, 1996.
3. N.G. Leveson, Safeware: System Safety and Computers, Addison-Wesley Publishing Company Inc., ISBN 0-201-11972-2, 1995.
4. J.C. Laprie et. al., Dependability: Basic Concepts and Terminology, Springer Verlag, ISBN 3-211-82296-8, 1992.
5. J.L. Lions, Ariane 5, Flight 501 Failure, Report by the Inquiry Board, www.esrin.esa.it/htdocs/tidc/press/Press96/ariane5rep.html, 1996.
6. P.G. Neumann, The Risks Digest, Forum on Risks to the Public in Computers and Related System, ACM Committee on Computers and Public Policy, Vol. 10-15, 18, 20, http://catless.ncl.ac.uk/Risks/.
7. J.M. Voas et.al., Software Assessment: Reliability, Safety, Testability, Wiley Interscience, ISBN 0-471-01009, 1995
8. R.R Lutz, Analyzing software requirement errors in safety-critical, embedded systems, IEEE Software Requirements Conference, 1992.
9. J.D. Musa, Software Reliability Engineering, McGraw-Hill, ISBN 0-07-913271-5, 1998.
10. Recommended Practice for Software Reliability, ANSI/AIAA R-013-1992.
11. A Critique of Software Defect Prediction Models, IEEE Transactions on Software Engineering, Vol. 25, No. 3, jun 1999.
12. John Rusby, Formal Methods, CSR/ENCRESS Annual Conference, 1995.
13. John Rusby, Partitioning in Avionics Architecures: Requirements, Mechanisms, and Assurance, NASA/CR-1999-209347, 1999.
14. G.T. Watt, Firewalls in Safety-Critical Systems, Proc. of the 18th Intern System Safety Conf, pp. 22-29, 2000.
15. B.L. Di Vito, A Model of Cooperative Noninterference for Integrated Modular Avionics, 7th Intern. Working Conf on Dependable Computing for Critical Applications, pp. 251-268, 1999.
16. B.J. Dobbing, Building Partitioned Architectures Based on the Ravenscar Profile, Ada Letters, Vol XX, No. 4, Dec 2000.
17. G Myhrman, R Ekholm, Striving for Safety within the Swedish Armed Forces, Proceedings of the 16th International System Safety Conference, 1998.

18. System Safety Program Requirements, MIL-STD-882C, 1993.
19. Information technology – Software life cycle processes, ISO/IEC 12207, 1995.
20. ISO 9000-3:1997, Quality management and quality assurance standards -- Part 3: Guidelines for the application of ISO 9001:1994 to the development, supply, installation and maintenance of computer software, 1997.
21. B.J. Dobbing, A. Burns, The Ravenscar Tasking Profile for High Integrity Real-Time Programs, Ada-Europe´98, Lecture Notes in Computer Science 1411, Springer Verlag, 1998.
22. K. Lundqvist, L. Asplund, A Ravenscar-Compliant Run-Time Kernel for Safety-Critical Systems, to appear in Real-Time System, Kluwer Academic Publishers Group.
23. Ada95 Reference manual, International Standard ANSI/ISO/IEC-8652:1995, 1995.
24. J.G.P. Barnes, High Integrity Ada – The SPARK approach, Addison-Wesley, 1997.

Biography

Inga-Lill Bratteby-Ribbing, Strategic Specialist, Defence Material Administration, P.O. Box 228, SE-75104 Uppsala, Sweden, telephone – (+46) 18 12 02 63, facsimile - (+46) 18 12 02 72, email: ilbra@fmv.se

Inga-Lill Bratteby-Ribbing is Strategic Specialist in Software Safety at the Swedish Defence Materiel Administration (FMV) with 35 years experience of software development and techniques for safety-critical systems at the former Swedish Defence Research Establishment (now FOI), the defence industry (former Celsius Tech Systems) and since 1995 at FMV. She has been active on the boards of Ada-Europe and Ada in Sweden, in study and working groups within IEEE and ISO/IEC JTC1/Sc22. She has presented articles and introductions in software safety within different Swedish organisations, such as the Swedish Encress Club and SESAM (the Defence Sector Software Engineering and Ada User Group). At FMV she has worked as project leader and technical expert in software safety within different projects. One of these has produced a handbook for software in safety-critical applications (H ProgSäk).