Safety in Automotive Software: an Overview of Current Practices

Paolo Panaroni[1], Giovanni Sartori[1], Fabrizio Fabbrini[2], Mario Fusani[2], Giuseppe Lami[2]
[1] *Intecs SpA, Italy*
{paolo.panaroni, giovanni.sartori}@intecs.it
[2] *Istituto di Scienza e Tecnologie dell'Informazione "A.Faedo" – Pisa, Italy*
{fabrizio.fabbrini, mario.fusani, giuseppe.lami} @isti.cnr.it

Abstract

Electronic control units and on-board networks for automotive applications cover a big variety of functions that in many cases are responsible of safety-critical behavior of the vehicle. Safety needs and goals claim that the software involved in such functions be designed by adopting opportune methods and practices. The paper presents a survey on most of these practices in the light of emerging standards.

1. Introduction

Modern vehicles are definitely "software-intensive" systems (someone says "computers with wheels"). Software is now implementing and/or controlling a growing number of traditional functions as well as new innovative functions, made possible only by software. Software is also taking charge of functions traditionally controlled by the driver.
It is not surprising that a growing number of these functions are "safety related" at various level of risk depending on the possible hazards they are related to.
A safety standardization process specific for automotive is well in progress known as WD 26262 [1], derived from the established industry-generic standard IEC61508 [3]. Part 6 of WD 26262 addresses specifically software development. Software safety requirements and constraints are also being defined within AUTOSAR [5]. These two initiatives just mention, but do not provide specific details about, methods and techniques to address software safety. As in other standards derived from IEC61508 (such as EN5012x for railways applications), most of these techniques are separately listed in example annexes. This is right for a standard to do, because technology evolves at a higher pace than the standard does, so that some techniques usually get obsolete in a few years and are anyway left for interpretation to the know-how and experience of software producers. It must be observed that the purpose of those standards is not to describe safety practices and techniques, but to discipline and manage their adoption in the framework of a project of a safe software-intensive system.
This paper intends to help users fill the gap left by the (necessary) incompleteness of the standards themselves. It provides an overview of these practical techniques capitalizing from years of experiences of software development and software consultancy on the automotive domain.
The paper is structured as follows: in Section 2 problems and characteristics of safety-critical software in automotive are presented; in Section 3 the most common practices adopted in practice to address safety-critical software in automotive are introduced; in Section 4 the interactions between software and hardware are considered; in Section 5 a safety lifecycle view is sketched (reflecting the emerging automotive standards), where the software practices mentioned in Section 3 are framed and integrated, together with criteria for their selection, and finally, in Section 6, some conclusions are provided.

2. Safeware in Automotive

Software implementing safety related functions needs special attention: somebody calls it "safeware" [8] to distinguish it from ordinary software.

Conceptually software per-se does not directly harm anybody and one may be tempted to consider safety only at system level, but software has the "power" to command (through actuators) system elements (e.g. brakes, steer) and doing so it may give commands potentially leading to accidents, thus killing people. As such, software may become a major source of hazards of the overall system.

Safety should not be grossly confused with reliability: a car that does not start is safe but unreliable [8]. Reliability just focuses on low probability of failure occurrence while safety places the focus on the consequences of failures and their severity. For the sake of safety it is often accepted to put the system in a safe state, thus sacrificing reliability in favor of safety.

0730-3157/08 $25.00 © 2008 IEEE
DOI 10.1109/COMPSAC.2008.139

IEEE computer society

It is often the case that a safety protection mechanism may trigger safety actions degrading proper functionality but assuring continued safety. Nuclear reactors are the most prominent example of this effect: most of their shutdowns are caused by safety protection measures that, at the detection of any minimum anomaly, stop the system. Safety is assured by affecting availability.

Automotive safeware is growing and becoming pervasive. A list (though not exhaustive) of well-known software-intensive safety-critical devices is provided below:

- ABS (Anti-lock Braking Systems)
- ACC (Adaptive Cruise Control)
- ESP (Electronic Stability Program)
- EBA (Emergency Brake Assistant)
- BBW (Brake By Wire)
- SBW (Steer By Wire)
- Airbags, light control, dashboard
- Tire pressure, lane departure warning

The question is: how the software part of these devices can actually harm people? The software is definitively a potential source of risk according to the chain of causes/consequences shown in figure 1.

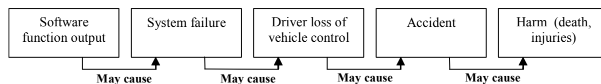

Fig. 1 – Chain of causes/consequences for software

If a software function output may "by merest chance" (through the cause/consequence chain) cause a harm, that software function is said to be safety relevant (safety critical or safeware).

3. Practices for handling safeware

Safeware requires a combination of:

1) An extremely rigorous development and verification/validation process.
2) A number of safety measures within the product.

Several safety related process standards have been developed and are currently in use. Among them the most notable are:

- IEC 61508 Part 3 (though generic and not specific to automotive) [3]
- MISRA Integrity Guidelines (less known and adopted) [4]
- MISRA C:2004 (well established and well supported by tools) [15]
- MIL STD 882D [9], DO 178B [10]

The ISO launched in late 2005 a project (called 26262 – Functional Safety in Automotive Electronics) aiming at becoming the reference standard for safeware in automotive, still in progress. A number of studies and research projects have laid down foundations on automotive software safety and among them DRIVE Safely [11], PASSPORT [12] and EASIS [13] should be mentioned.

In the following, the principal complementary strategies to approach safety in automotive software are shortly discussed.

3.1 Fault Prevention/Avoidance

A vast range of approaches are put in place to prevent faults from being inserted in the software.

- Mature and rigorous software process (e.g. SPICE [2], Automotive SPICE [7], CMMI [6]), Safe CMMI [14]
- A disciplined software life cycle (e.g. a V life cycle)
- Failure analysis (e.g. FMEA, but rarely FTA)
- Semi-formal methods (e.g. UML, Simulink)
- Formal methods (mathematical models and state models representing software properties as theorems)
- MISRA C, Certified compilers, kernels, libraries
- Dual/Pair programming
- Metrics conformances (e.g. nesting depth, cyclomatic complexity, sometime also at Model level)
- Schedulability Analysis (WCETA Worst Case Execution Time Analysis)
- Memory Usage Analysis (WCMUA Worst Case Memory Usage Analysis)
- Safety oriented organizations (independent safety team, safety specific competence, safety planning, safety verification and assessment)

3.2 Fault Removal

A wide range of techniques are put in place to detect (and then remove) faults during development. Among them:

- Peer Reviews/Code inspections
- Static Analysis Tools (on code, on models)
- Unit test (100% MD/MC, branch or statement coverage)
- Boundary value and Equivalence class Partitioning tests
- SW Integration test (100% call-pair coverage)
- HW/SW Integration test
- Validation test (100% requirement coverage)
- Defects seeding/Fault injection (rarely, to assess removal effectiveness)
- Systematic regression testing

- Model checking (automatically demonstrating properties of a formal model that describes the software behavior)

Tools like Polyspace (www.MathWorks.com) perform an extensive analysis capable to intercept common run time failures. Run time failures may be as high as 30%-40% of total observed failures.

3.3 Fault Detection /Recognition

Fault detection is one of the most challenging tasks: how can we set up mechanisms capable to detect a software failure? Software may fail in a number of different modes; one of the most exhaustive lists of failures modes is the following:

- Omission (output is not provided)
- Commission (output is provided when not required)
- Late
- Early
- Wrong Value coarse (detectable/not plausible)
- Wrong Value subtle (undetectable/plausible)
- Stuck Value (does not change over time)
- Run time error (e.g. divide by zero)

Depending on the particular failure modes a possible set of different protection measures may be identified. The most used strategies are:

- Watchdog: no response/wrong timing is usually simply detected by a watchdog. If after a given deadline the software has not responded one assumes that it is giving no response or taking too much time (wrong timing). Watchdogs are usually implemented by hardware. Sometimes a software watchdog can be implemented by controlling maximum loop iterations, so that in the code there is never an "open-ended loop".
- Plausibility check: wrong value failure is much more intriguing: how can we say that this value is wrong? The strategies adopted include:
  - Plausible range (e.g. a speed cannot be 500 Km/h)
  - Plausible trend (a speed cannot be 120 Km/h if one second ago was 30 Km/h)
  - Double check function (a speed value shall be coherent with wheels rotation values)
  - Diversified function (a fully alternative function applied to the same input values gives a different value)
  - Assertion failed during computation
  - Run time errors checks (e.g. divide by zero, array index out of bounds, etc. during computation)
- Assertions: it covers all types of "sanity checks" interspersed in the code such as plausibility check of input parameters, checks during computations, etc. Safety code is often augmented with extra code just to verify that some properties hold before a computation (pre-conditions) or hold after a computation (post-conditions). Proper usage of assertions may detect a number of irregular, anomalous computations raising a flag for an internal failure.

3.4 Fault Isolation/Containment

Some portions of the software are more critical than others and shall be protected against failures propagation. Executable code (instructions) may be stored in read-only memory and cannot be contaminated by failures but read/write data are extremely vulnerable.

Critical data can be protected by a number of mechanisms. The most prominent is a checksum/CRC associated with critical variables (individual variables, a full table or an entire group of variables).

After writing the variable its checksum is computed and stored aside. Before using the variable its checksum is tested for consistency. This allows detecting corruption but does not allow any recovery. Where those variables are really critical they may be stored in pair (each with its own CRC), so that it is possible to detect the corrupted variable and chose the non corrupted one. It is wise to store the pair into areas far away from each other so as to reduce the chance of being both corrupted.

This mechanism may envelop a full set of "critical variables" so that these variables - though living in the same memory space with less critical variables - may be considered "segregated" from the others. While this does not fully support the segregation concept set forth by standards such as IEC 61508 [3] or DO178B [10] it may well provide a protection shield.

The same applies to all messages exchanged through communication channels, where sequence order checks and timing controls can be added too.

Other techniques include special "patterns" (e.g. F0F0F0) of values placed at the end of arrays so that a potential overrun is detected (this problem is very serious and frequent using the C Programming Language). Some place special patterns also on top of the stack to detect stack overflow.

3.5 Fault Recovery/Handling

Well, we have found a failure. So what? What can we do if we detect a failure? The most used strategies are:

- Fail Silent: in most situation when a wrong output is detected it is better to give no output all (i.e. remain silent) rather than giving a wrong output. In case of a detected failure, stopping safety related controls is often the safest course of action.
- Fail degraded/Fail reduced: in a number of case a wrong/missing output is replaced with something not exact but plausible (e.g. by extrapolation, or trend analysis). The output is not good but it is an acceptable approximation.
- Fail safe: in this case the wrong/missing output is replaced with some value that, though not valid, it is safe in the sense that does not generate consequences potentially leading to accidents.
- Fail (full) Operational: this is the optimal (but very expensive) strategy where it is possible to fully recover from the wrong/missing output by replacing it with a correct value.

Of course, in all the above cases the failures are communicated or stored somewhere for supporting the on board diagnosis and future maintenance (e.g. a DTC Diagnostic Trouble Code).
Where possible, it is a used strategy just to reset the system, after failure detection. A fresh re-start is a powerful mechanism to restore the software fully operational.

4. Hardware/Software Interactions

In many cases software can effectively detect hardware failures. The most notable example is memory checks (writing and re-reading back some defined patterns). These memory checks can be conducted once at power-up or even periodically (one memory area at a time, when the CPU load is low).
Microprocessor instructions are also checked by executing pre-defined sequences of code and checking the actual resulting value against a predefined stored expected output.
Nevertheless such a strategy can really improve its effectiveness by using a second micro controller that acts such a monitor of the main controller by repeating calculations. Program monitoring is a common adopted practice in the automotive field, that allows also the logical or the temporal control flow of the program.

Checks can be done on input coming from sensors (e.g. plausibility, redundancy of sensors); it may also be done on actuators using feedback sensors.
Software may compensate for sensor "noise", e.g. by averaging over a set of values or by discarding anomalous values in an input sequence.
Software may mistrust the start-up process and check its own integrity with a checksum associated to the code and to data, thus answering such questions such as:

- Have I been loaded correctly?
- Am I the correct version?
- Is my own version consistent with the hardware version?

5. Using safety techniques in a project

A question arises even after the preceding short survey: When and which ones from the presented techniques are cost-effectively used in a project? Should a project manager plan on using all or as many as possible of them? Cost issues and difficulty in finding specialized human and infrastructure resources would advise to adopt a careful selection. On the other hand, safety needs would require extreme confidence that the system performs its service in any possible conditions without causing serious accidents.

Is there any quantitative expression of such safety needs, and is there any mapping between these expressions and the mentioned techniques?

Recent and current standardization work is likely to provide an answer and a workable solution to such trade-off, and we can expect this to happen very soon in the automotive application domain.

5.1 Safety lifecycle

Safety lifecycle issues were first addressed in the MIL family standards, then in the general IEC 61508 and derivates, and probably will be in the long expected, and feared, ISO 26262.

The driving concept is system lifecycle.

While the software is likely to be considered the most critical component of a safety-critical system, safety issues in the product lifecycle are first addressed at system level.

Without getting into further details, it is sufficient to recall that safety analyses (such as preliminary hazard analysis, risk determination) are conducted at the

earliest phases of system lifecycle to determine system safety requirements and, also using techniques such as FMECA and especially FTA, software safety requirements. Safety analyses are repeated through the whole lifecycle to check that the overall risk of the evolving product continuously meets its safety goals (that is, it stays in an acceptable area).

The safety requirements (determined and refined after the analyses) express: 1) how much the risk associated to the system must be reduced to be kept within an acceptable level, and 2) the confidence that the system will perform, in any possible conditions, within such "safe area".

5.2 Safety Integrity and Safety Integrity Levels

The combination of the aspects 1) and 2) above is known as the Safety Integrity of the system, which can also be expressed as the "probability that a safety-related system satisfactorily performs the required safety functions" [3].

One of the most interesting achievements of the mentioned IEC61508 and derivate standards is a particular "quantification" of the Safety Integrity, which would otherwise be a complex function of many different arguments, into 4 increasing steps, or Safety Integrity Levels (SILs, ASILs in automotive), whose range is {1, 2, 3, 4}. The higher the Safety Integrity Level, the more severe are: the need of risk reduction to an acceptable level; the need of confidence that this reduction is really implemented; the measures to be adopted in the project (that include management and engineering practices).

There are rules to determine, from the preliminary safety analyses, the SIL of system, functions and components. The software gets its SIL as well, basically inheriting it from the systems it affects.

And here we get the mapping we were looking for between the system safety needs and the software practices presented in Section 3. It is a conventional one, mediated through the SIL, but it represents a defined, repeatable way of adoption. The mentioned standards, and likely those still to be published, prescribe (without describing their contents, as we pointed out) a defined set of practices and techniques, with options and other adoption rules, for each SIL. For instance, formal methods are highly recommended from SIL 3 up, and so is diverse programming.

Due to the very nature of standards, we cannot expect that the new automotive regulations provide much information about the rationale for assigning particular techniques to a defined SIL (in the IEC 61508 application to railways systems, the options, that allow some interpretation on the user side, are scarcely commented).

Nevertheless, the SIL-based approach has the advantage of facilitating the project decisions, the comparison among different software and systems as well as the performing of project safety assessments.

6. Conclusions

The exponential increase of electronics systems in automobiles occurred in the last decade has determined the involvement of software also in safety aspects of vehicles.
The paper has presented a survey of the main strategies and techniques today in use in automotive to face software-related safety problems, together with adoption criteria prescribed by emerging standards.

7. References

[1] ISO/CD PAS 26262: Road Vehicles – Functional Safety – Part 1..9 (ISO TC22/SC3, 2008-02-29).

[2] ISO/IEC 15504 International Standard "Information Technology – Software Process Assessment: Part 1–Part 5" 2006.

[3] Functional Safety of Electrical/Electronic/ Programmable Electronic Safety-Related Systems. IEC 61508, Int'l Electrotechnical Commission, 1999.

[4] Motor Industry Software Reliability Association, MISRA-C:1998 Guidelines for the Use of the C Language in Vehicle Based Software, April 1998, ISBN 0-9524156-2-3.

[5] www.autosar.org

[6] Chrissis MB, Konrad M, Shrum S. CMMI Guidelines for Process Integration and Product Improvement. Addison-Wesley, 2004.

[7] http://www.automotivespice.com

[8] Leveson N. Safeware: System Safety and Computers, Addison-Wesley, 1995

[9] DoD, MIL-STD-882D – System Safety Program Requirements. US Department of Defence. Defence Standard. February 2000.

[10] DO-178B, Software Considerations in Airborne Systems and Equipments Certification, RTCA Inc. document, Dec. 1992.

[11] www.drivesafeproject.org

[12] Hobley K.M., Jesty, P.H. Integrity levels in Automotive Telematic Systems. In Integrity of Automotive Electronic Systems, IEE Colloquium on. March 1993. Pages 3/1-3/3.

[13] www.easis.org

[14] +SAFE, V1.2 A Safety Extension to CMMI-DEV, V1.2 - Technical Note, CMU/SEI-2007-TN-006

[15] Motor Industry Software Reliability Association, MISRA-C:2004 - Guidelines for the use of the C language in critical systems, October 2004, ISBN-0-9524156-4