ISO 26262 STANDARD: APPLICATION ON THE PSA BODY CONTROL MODULE (BCM)

Alin MIHALACHE  
SECTOR  
12, avenue du Québec  
BP 636 Villebon-sur-Yvette  
F91965 Courtabœuf 1 Cedex  
Tél. : 01 69 59 27 27  
Fax : 01 69 59 27 28  
Email : alin.mihalache@sector-group.eu

Fabrice BEDOUCHA  
PSA PEUGEOT CITROEN  
Route de Gisy  
78943 Vélizy-Villacoublay Cedex  
Tél : 01 57 59 62 23  
Fax : 01 57 59 92 06  
Email : fabrice.bedoucha@mpsa.com

Résumé

La norme ISO 26262 est applicable dans l’industrie automobile depuis novembre 2011. Dans cet article nous décrivons la méthodologie mise en place chez PSA Peugeot Citroën appliquée sur le logiciel embarqué, nous nous intéressons donc plus particulièrement à la partie 6 de la norme ISO 26262 : développement du produit au niveau du logiciel.

Pour illustrer notre méthodologie, nous avons choisi le calculateur qui gère les feux avant : Boitier de Servitude Intelligent.

Nous présenterons également les difficultés rencontrées et les améliorations envisagées dans le futur.

Summary

The safety standard ISO 26262 has been applied in automotive industry since November 2011. In this article we describe the methodology put in place in PSA Peugeot Citroën to apply ISO 26262 on embedded software so we mainly focus on ISO 26262 – Part 6: Product development at the software level.

To illustrate our methodology we have chosen an Electronic Central Unit that manages the headlights: Body Control Module.

We will also present the difficulties encountered and the improvements forecasted in the future.

Introduction

ISO 26262 is derived from IEC 61508 like other standards for aeronautics, railroad, and nuclear industry. However unlike these domains, safety standard is rather new in automotive industry. The embedded systems are more and more complex and safety risks are high especially with Electronic Central Units that can react and replace the driver: decelerates when the vehicle comes closer to the front vehicle, switches on or switches off lights when luminosity is lower or higher than a threshold, activates wipers when it rains…

Of course, PSA Peugeot Citroën like many actors in automotive industry, has not waited for ISO 26262 to invest in safety. Work on safety was already a standard but every car manufacturers had its own rules. ISO 26262 gives a common vocabulary, requirements and guidelines to follow. For the software, ISO 26262 is complete to describe activities expected through the V-cycle and according to the ASIL (Automotive Safety Integrated Level).

After a presentation of the Body Control Module responsible for managing headlights, in a second part we present our processes and software activities of the software safety team. In a third part, we describe the software safety mechanisms used by the software safety team. Fourthly, we propose a methodology to implement ISO 26262 standard application on the embedded software. Finally, we illustrate the improvement of the methodology, before concluding and discussing further research.

BCM in PSA vehicle architecture and internal software architecture

BCM is the central ECU in cockpit (see the figure 1). Its role is to manage different functionalities such as wipers, doors, windows, lights, air conditioning, alarm… In PSA vehicle architecture, this ECU is also the gateway between all networks.
Concerning software architecture, BCM is compliant with AUTOSAR ICC3 (see figure 2).

In PSA only applicative layer, i.e. software components, is developed internally whereas basic software is developed by a supplier. So in this article we will only discuss about software safety management in the applicative layer, software safety at basic software level is out of the scope of this article. However, software safety is of course also managed at basic software level. For example critical data are duplicated in EEPROM and protected by CRC to avoid accidental loss, to detect non wanted writing, to manage EEPROM wear or to detect power off during EEPROM writing.

As an example to illustrate our process in part 5, we will study « Unexpected extinguished of the low beam light when driving in dangerous situations » hazardous event.

Processes and software safety team activities

As input, we do not work directly with vehicle safety goals. At system level (Brindejonc et al., 2010), a system safety team (see table 1 – Annex) is responsible to provide us a set of derived safety goals and hazardous events to be managed in BCM at software level.

Then the aim of the processes that are put in place is to be compliant with ISO 26262 and guaranty reproducibility along software versions and projects.

The main activities for the software safety team (Mihalache et Bedoucha, 2013) is, for each hazardous event, to perform safety analysis at software architecture level, usually based on Fault Tree Analysis (FTA) (Villmeur, 1988) with hazardous event at the top of the tree, and action plan elaboration according to safety analysis. Software safety action plan is a list of software safety requirements.

The tasks to write detailed conception based on software architecture and software safety action plan, to write code and to test it (for example checking coding rules compliancy and performing unit tests) are performed by developers (see table 1 – Annex).

Software safety team intervenes also after implementation to review parts of code related to safety, especially the good implementation of the action plan, and to review tests that are run.
All along this process which is compliant with CMMI level 2, traceability is maintained between architecture requirements, safety requirements in action plans, conception documents, code and tests. Traceability matrixes are maintained all along the process to check the coverage of all requirements and to be able to detect a change. All items in this process are identified and managed in configuration.

Quality assurance is involved all along the process to guaranty that it is strictly followed and to guaranty the strict application of the formalism.

**Software safety mechanism**

Our first strategy was to put all the applicative software in the same task, i.e. all applicative runnables are called in the same task. In this way we avoid protection of shared data during switch of tasks. The runnable consist of a set of instructions that can be determined to terminate within a finite time. Classically, it is a C-function.

In order to work with valid data, at the start of each runnable all input data are locally saved in critical section. During its execution, a runnable works only with local data. And at the end of each runnable, data are still saved in memory in critical section. Structure of the software does not allow to avoid corruption of data due to errors in other software parts. At validation stage most of these errors are detected and this can be acceptable for parts of the software that are not related to safety. However for parts of software that are related to safety, this is not enough.

As we do not use MMU (Memory Management Unit), we want to avoid developing all the software as ASIL B in order to be compliant with freedom from interference. At the start of the study software safety engineer elaborate a FTA to detect critical data, i.e. data that if they are modified by error can trigger the hazardous event. These data are very important and must be protected. For example (see the figure 3) if software component A is safety related and manage critical data, another software component not related to safety can modify by mistake (due to an out of bound array, a pointer badly initialized...) one of these critical data and trigger a hazardous event (Mihalache et Bedoucha, 2013).

![Figure 3. Data corruption in memory](image-url)

The main part of our strategy is to detect these critical data and to mirror them. To detect unwanted writing of critical data, each time we want to write a critical data, we write its mirrored value in another part of the memory. Each time we want to read a critical data, we compare it with its mirror to detect an unwanted writing (see the figure 4 and figure 5). In our software, as soon as a data is tagged as critical, this functionality is automatically put in place (by the use of a macro).

![Figure 4. If a mechanism for error detection is not use, a corruption of critical data is not detected and can trigger hazardous event](image-url)

![Figure 5. Mechanism for error detection of critical data is the redundancy by mirroring, a corruption is detected and limp-home mode (or safety state) is activated](image-url)
Another strategy is to detect critical functions, i.e. functions that if they are called by error can trigger a hazardous event. Critical functions are also detected by FTA and for each critical function a specific 32-bit parameter must be sent with the function.

At the start of a critical function, the parameter is verified, as if it were the key to unlock it. If the parameter is wrong it means that it is an unwanted call and limp-home mode is activated and if the parameter is the good one, the critical function is executed.

**ISO 26262 standard application on the embedded software**

The purpose of this chapter is to describe the PSA methodology on the embedded software. An example on the automotive lighting function is described, in order to illustrate how the principles of the ISO 26262 are applied by the software safety team (see table 1 – Annex).

1. **Automotive lighting function**

The automotive lighting function of a vehicle consists of lighting and signalling devices mounted or integrated to the front, sides, rear and in some cases the top of the vehicle. The purpose of this system is to provide lighting for the driver to operate the vehicle safely in the dark, to increase the visibility of the vehicle, and to display information about the vehicle's presence, position, size, direction, and driver's intentions regarding direction and speed.

In the case of the PSA Electronic and Electrical Architecture, control lighting system is based on several ECU car systems. The control algorithms are principally made by the BCM.

The low beam light functionality of the lighting system function is described in the figure 6. For this functionality the Electronic and Electrical Architecture must contain the Body Control Module (BCM) which established a gateway between two CAN networks and LIN network, the Display (DISP), Under hood Control Module (UCM) and Top Column Module with 3 or 4 positions (TCM) and, optionally, the Rain/Light Module (RLM).

**Figure 6. Lighting system function : Software architecture of BCM based on AUTOSAR and the others ECU on CAN et LIN networks**

Many software architecture configurations are possible for the low beam light functionality in the PSA Electronic and Electrical Architecture. For example, we illustrate one possible configuration by the following flowchart diagram:

---

Communication 4D-5 Page 4 sur 10
For this configuration (see figure 7) we are able to switch on the low beam light if electrical network power is not degraded and:

- TCM is in second position;
- Or, DISP is set at automatic low beam light and RLM command is switched on.

2. Safety goal of the automotive lighting function

Functional safety is concerned with the absence of unreasonable risk to individuals caused by potential malfunctions in E/E systems. Functional safety focuses primarily on risk arising from random hardware faults as well as the systematic faults in system design, hardware or software development. During the system design phase, the technical safety requirements are refined and allocated to hardware and software. This is the procedure to derive the software safety requirements.

For the automotive lighting function the most important hazardous event is « unexpected extinguished of the low beam light when driving in dangerous situations ». The safety goal with an ASIL B level is « a sufficient lighting of the road in front of the vehicle shall be maintained when requested by the driver » and the limp-home or safety state associated is « in case of detected failure that could lead to the violation of the safety goal, the low beam light shall be forced ». The entire ECU of the car (TCM, DISP, RLM, BCM, UCM) inherit an ASIL B because they can directly contribute to the loss of light.

In this case, the software safety requirement allocated to BCM has an ASIL B associated to the risk « unexpected extinguished of the low beam light when driving in dangerous situations ». The implementation on this safety goal in the BCM software using the ISO26262 standard is presented in the next part.

3. Initiation of product development at the software level

In this sub-phase, the scope is to plan and initiate the PSA functional safety activities for the following sub phases of the software development (see figure 8).

The following PSA functional safety activities are initiated:
The software safety requirements are assigned to the software components whose error can lead to the hazardous event. The specification of the software safety requirements is derived from the technical safety concept and the system design specification. This activity is made by a PSA system safety team. They take into account and make the ECUs allocation of the system hazardous events by a system FMEA (Brindejonc et al., 2010), (Villmeur, 1998).

The software safety requirements are assigned to the software components whose error can lead to the hazardous event. The software safety requirements are based on the system safety requirements. For example, the system safety requirement “The ECU output X must not unexpected drive the output X when the speed data is non-zero”. In this phase the hardware-software interface specification shall be detailed and shall describe each safety-related dependency between hardware and software.

The verification of the software safety requirements is done by the PSA system safety team.

5. Software architectural design

The objective of this sub-phase is to develop and to verify a software architectural design that takes into account the software safety requirements. The software architectural design is described by using the informal notations and more and more semi-formal notation (Model Based Design for example).

The characteristics like verifiability, configurability, feasibility, testability are adequate for the BCM software architectural design. The software architectural design is a good hierarchical structure of software components due to AUTOSAR, but restricted size of software component or interfaces are not considered because we use small C functions. The software components are designed in order to fulfill car functionality (like lighting function, electric parking brake, easy move...); ensuring that a high cohesion within each software component.

In this phase the hardware-software interface specification shall be detailed and shall describe each safety-related dependency between hardware and software.

The other functions that are not related to safety requirements shall be also described. For each hazardous event, the software safety engineer addresses, by a software FTA (see for example the figure 9, next chapter), each software-based function (for us is the C language function) whose failure could lead to a violation of a technical safety requirement allocated to software. It is difficult to make the ASIL decomposition to a software safety requirement because the ISO 26262-9, Clause 5 asks to justify the independence between the functions (it must prove a AND Gate into the software FTA).

The verification of the software safety requirements is done by the PSA system safety team.

Software FTA example

The safety analysis is covered by software FTA on the automotive lighting function a software FTA (see the figure 9) treats the hazardous event « unexpected extinguished of the low beam light when driving in dangerous situations». This analysis allows to identify the safety-related parts of the software: functions, data ...

This software FTA is based principally on the application software. The undeveloped events marked by a lozenge are:

- Error function call is due to the scheduling problems or corruption function pointer;
- Unexpected transition of EtatResElec is a hazardous event treated by electrical network management function;
- Tools errors are related to the code generation, the compilation chain, etc.;
Failure of a basic software are many: like sending a wrong message on a network, unexpected reset of the ECU, blocking of
the application software, inability to control the direct outputs, etc. More than 60 generic hazardous events are defined and
implemented in order to guarantee a B level on BCM basic software.

The principal mechanism for error detection used in the BCM software is the redundancy by mirroring for the safety data. Also,
but not systematically, the range and plausibility checks on input and output data are implemented.

The mechanisms for error handling, after detection, are based on a safety state. Sometimes, a reset is performed as static
recovery mechanism.

The methods used for the verification of the software architectural design are walk-through and inspection (with checklist) of the
design made by software safety engineers, software architects, integrators and quality engineers (see table 1 – Annex).

![Software FTA for the low beam light functionality](image)

Figure 9. Software FTA for the low beam light functionality

6. Software unit design and implementation

The notations used to describe the software unit design are the natural language and the informal notation (like pseudo code or
flowchart diagrams). Those notations describe the functional behaviour and the internal design to the level of detail necessary
for their implementation.

Design principles for software unit design and implementation at the source code level are applied in order to respect the design
and coding guidelines and naming conventions. For example, here are some design principles for software unit design used:
one entry and one exit point in subprograms and functions; no dynamic objects or variables, or else online test during their
creation; initialisation of variables; no multiple use of variable names; avoid global variables or else justify their usage; limited
use of pointers; no implicit type conversions; no hidden data flow or control flow; no unconditional jumps; no recursions, etc.

The methods used for the verification of the unit design and implementation are walk-through and inspection (with checklist) of
the design made by software safety engineers, software architects, developers and quality engineers (see table 1 – Annex). The
data flow analysis is partially covered by software FTA.

The design principles are included in the MISRA rules that are checked with the support of a MISRA checker tool, by static code
analysis verification. Also, the semantic code analysis is done by compiler and specific tools.
7. Software unit testing
The methods used for software unit testing like requirement-based test, interface test are based on requirements of software architectural design and not on the requirements of software unit design because they are close to the code. The fault injection test is only used for the safety functions. The resource usage test is not done, but there is a regularly monitored resource exam when compiling the complete software.

The methods for deriving test cases for software unit like analysis of requirements, generation and analysis of equivalence classes, analysis of boundary values or error guessing testing are not applied systematically, but they are used indirectly for the function coverage.

The structural coverage metrics at the software unit level are 100% for statement coverage and branch coverage; 80% - ASIL A and 90% - ASIL B for MC/DC (Modified Condition/Decision Coverage) which is very high compared with what is usually done.

8. Software integration and testing
The software integration uses the defined steps for integrating the supplier and PSA basic software and application software. The result is the integrated software that is tested by integrators.

The methods uses for the software integration testing are requirements-based test and interface test.

The methods for deriving test cases for software integration testing are principally the analysis of requirements and the error guessing following experience.

Structural coverage metrics at the software architectural level are 100% for function coverage and call coverage.

9. Verification of software safety requirements
The verification that the embedded software fulfils of the software safety requirements is done by the validation team.

Methodology improvement

The purpose of this chapter is to present current and future ideas in order to extend and improve our methodology on the embedded software.

1. Use of MMU/MPU
The first improvement is the use of MMU. This will allow to avoid redundancy by mirroring critical data in RAM. Indeed, safety modules can be sealed in one of several partitions and non-safety module will not be able to modify critical data of a safety module without being detected by MPU. Next generation of BCM will use this mechanism.

2. Maximum duration of execution
As critical and non-critical runnables are called in the same task, a non-critical runnable can take too much time for its execution and delaying execution of a critical runnable. To avoid this situation, each runnable can have a maximum execution time. In case of this time is exceeded, runnable is aborted and an error of execution is recorded for debug.

3. Check of runnable scheduling at runtime
Some critical runnables may be executed in a specific order. The aim of this improvement is to check at runtime that runnables are effectively executed in the expected order. For example, in some systems a polynomial is calculated at the start of each runnable to check that the order is respected. However, this solution consumes CPU and is usually limited to few runnables.

4. Multi-core
Further improvements have been studied. One of the most promising is the use of multi-core. We can imagine for example a core dedicated to safety modules while the other cores contain the rest of the application. However synchronization problems must be resolved.

5. Software architectural design
To improve software architectural design, we plan to extend the use of semi-formal notation (Model Based Design, UML). Also we will consider to restrict the size of software component or interfaces and to limit the number of these interfaces.

6. Software unit design and implementation
Soon, we will use dedicated tool to verify the software unit design and implementation. This tool is under development and is an internal tool which will allow analysis of the data flow.

7. Software unit testing
Our Working Group on the unit tests make a considerable effort to derive the test cases for software unit like analysis of requirements, generation and analysis of equivalence classes, analysis of boundary values or error guessing testing. We plan to train people to do this activity, training material is under development.

8. Software integration and testing
The same Working Group plans to integrate in our process the methods for the software integration testing. We will use internal tools for testing developed by PSA.
Conclusion

The goal of this paper was to present an approach to design embedded software safety, structured around the ISO 26262 standard and adapted to our needs. The application field of this approach is the management of low beam light in BCM.

FTA is our main design methodology for software safety. It allows us to identify the safety data and functions. A corruption in software safety data can be detected by the mechanism for error detection. This mechanism is the redundancy by mirroring. Once a corruption detected, the limp-home mode is activated in order to avoid the hazardous event.

The safety methodology assumes that the MISRA rules are checked and coding guidelines are applied.

Our V cycle process is compliant with the ISO 26262 standard. Furthermore, this methodology is in continuous improvements which are identified and soon be applied in our process.

Annex

<table>
<thead>
<tr>
<th>V cycle phase</th>
<th>Main Contributor</th>
<th>Other Contributors</th>
<th>Reviewers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification</td>
<td>System architect</td>
<td>System safety engineer</td>
<td>System architect (peer review) System architect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>System safety and software engineer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Validation team</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Quality engineer</td>
</tr>
<tr>
<td>Architecture</td>
<td>Software architect</td>
<td>Software safety engineer</td>
<td>Software architect (peer review) Software safety engineer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Developers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Integrators</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Quality engineer</td>
</tr>
<tr>
<td>Design and implementation</td>
<td>Developers</td>
<td>Quality engineer (coding rules)</td>
<td>Developers (peer review)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Software architect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Software safety engineer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Quality engineer</td>
</tr>
<tr>
<td>Unit Testing</td>
<td>Developers</td>
<td>Quality engineer (coverage checking)</td>
<td>Developers (peer review)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Software safety engineer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Quality engineer</td>
</tr>
<tr>
<td>Integration and testing</td>
<td>Integrators</td>
<td>Quality engineer (coverage checking)</td>
<td>Integrators (peer review)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Developers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Software architect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Software safety engineer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Quality engineer</td>
</tr>
<tr>
<td>Verification</td>
<td>Validation team</td>
<td>Quality engineer (coverage checking)</td>
<td>Validation team (peer review)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>System architect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>System safety engineer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Quality engineer</td>
</tr>
</tbody>
</table>

Table 1. Teams distribution on the V cycle phases

Acknowledgement

The authors acknowledge the contribution of their colleagues to this work.

Glossary

- AUTOSAR: Automotive Open System Architecture
- BCM: Body Control Module of PSA Electronic and Electrical Architecture
- CMMI: Capability Maturity Model Integration
- CRC: Cyclic Redundancy Check
- ECU: Engine control unit
- EEPROM: Electrically-Erasable Programmable Read-Only Memory
- FTA: Fault Tree Analysis
- ICC3: Implementation Conformance Class 3
- MC/DC: Modified Condition/Decision Coverage
- MISRA: Motor Industry Software Reliability Association
- MMU: Memory Management Unit
- MPU: Memory Protection Unit
References


(Brindejonc et al., 2010), V. Brindejonc, G. Marcuccilli, S. Petit: "Démarche AMDEC système dans le cadre de l’ISO 26262", Lambda Mu 17, La Rochelle, France, 2010.

(ISO 26262, 2011), ISO 26262 Road vehicles — Functional safety

Part 1: Vocabulary
Part 2: Management of functional safety
Part 3: Concept phase
Part 4: Product development: system level
Part 5: Product development: hardware level
Part 6: Product development: software level
Part 7: Production and operation
Part 8: Supporting processes
Part 9: ASIL-oriented and safety-oriented analyses
Part 10: Guideline on ISO 26262

