ASSURER LA SDF D’UN LOGICIEL EXISTANT AYANT DE NOUVELLES FONCTIONNALITES SPECIFIEES EN MODEL-BASED DESIGN

ENSURE SAFETY ON THE EXISTING SOFTWARE WITH NEW FUNCTIONALITIES SPECIFIED IN MODEL-BASED DESIGN

Alin MIHALACHE, Fabrice BEDOUCHA, Yann Mikaël FOLL, Matthieu FOUCAULT
PSA PEUGEOT CITROEN
Route de Gisy
78943 Vélizy-Villacoublay Cedex
Tél : 01 57 59 09 18
Fax : 01 57 59 92 06
Email : alin.mihalache@mpsa.com, fabrice.bedoucha@mpsa.com, yannmickael.foll@mpsa.com, matthieu.foucault@mpsa.com

Résumé

Le but de cet article est d’expliquer les méthodologies mises en place afin d’assurer la sûreté de fonctionnement d’un logiciel existant écrit manuellement et qui intègre de nouvelles fonctionnalités en Model Based Design – code généré.

Pour le code écrit manuellement, une méthodologie est déjà en place et elle est conforme à l’ISO 26262. Par contre, cette méthodologie ne peut pas être reprise en l’état pour traiter la sûreté de fonctionnement logiciel pour les fonctionnalités spécifiées en Model Based Design. En particulier, le code généré ne peut être traité comme du code écrit manuellement.

Cet article traite de la méthodologie employée pour traiter la sûreté de fonctionnement logiciel d’un logiciel spécifié en Model Based Design et conformément à l’ISO 26262. Il traite également des différences et similitudes pour traiter la sûreté de fonctionnement d’un logiciel écrit manuellement.

Summary

The purpose of this article is to explain the methodologies to ensure the safety of existing handwritten code and incorporates new features in Model Based Design - generated code.

For the handwritten code, a methodology is already in place and is compliant with ISO 26262. But this methodology cannot be used without changes to treat software safety for features specified in Model Based Design. In particular, the generated code cannot be treated as handwritten code.

This article discusses the methodology used to treat the software safety specified in Model Based Design and compliant ISO 26262. It also discusses the differences and similarities to software safety process for handwritten code.

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 the 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.

Maximum ASIL level that BCM has to manage is B. For example hazardous events managed by BCM are:
- Loss of wipers (ASIL A);
- Stop indicator light erroneous at off (ASIL A);
- Unexpected extinguished of the low beam light (ASIL B);
- Insufficiently load shedding (ASIL B).

Most of the software is still written manually in C language. However, more and more complex algorithms are introduced into BCM software, these algorithms are usually in Model Based Design with Matlab/Simulink®.

For example, new ADAS (Advanced Driver Assistance System) functions, like LKA (Line Keeping Assist, function used to keep the car in its lane if the driver takes the wrong decision to change the lane whereas there is another car in his blind spot for example), are modeled with Matlab/Simulink® then code is generated from model.

At the time this article is written, around 15% of the BCM software is generated whereas 85% is still written manually; it corresponds in fact to the legacy code which does not evolved a lot. We estimate that in few years, more than 80% of the software will be generated as more and more complex functions are integrated into BCM.

Modeling with an appropriate tool has many advantages compared with standard specification and handwritten code. Just to cite the most important ones for us:
- Time from model to code and validation is reduced. Once model is finished and reviewed, code generation takes a little time. The main work for BCM is to make sure that conversion from float number from simulation to integer from microcontroller calculation part does not modify much the algorithm and errors are negligible.
- Generated code is more bug free than handwritten code. Of course the probability of having zero bug due to the code generation is not equal to zero, however the probability is lower to introduce bugs compared with handwritten code. Moreover ISO 26262 requires qualification of the tools like Matlab/Simulink®, this topic is not in the scope of this article.
- Specification is more understandable when it is modeled than when it is written in a document or deduced directly by reading the code in reversed engineering process. As a consequence, maintenance is much easier.
Of course having introduced Model Based Design, we still need to have a robust process to be compliant with ISO 26262 recommendations.

As an example in “Software safety mechanism” section, we will study “untimely activation of Hill Assist Descent Control (HADC) function by a push” hazardous event.

**Processes and software safety team activities**

Few changes have been made in our safety process to take into account Model Based Design.

With or without modeling, as input, we do not work directly with vehicle safety goals. At system level (Brindejonc et al., 2010), a system safety team 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 usually based on Fault Tree Analysis (FTA) (Villemeur, 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. Whereas the architecture document is the input to perform safety analysis for software without MBD, the input for MBD is the model itself.

For MBD functionalities, most of the safety mechanisms imply modification of the model, which will be described in details next part of this article.

The tasks to modify model, to write detailed design 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.

Software safety team intervenes 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, design 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 defined software process.

**Software safety mechanism**

1. **Error detection and error handling**

As we cannot afford to develop all our software in ASIL B, constraints of freedom from interferences must be respected between different ASIL and QM parts. ASIL relevant parts must be protected against errors as data corruption and input inconsistency.

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 manages 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, 2014).

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).
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 while if the parameter is the good one, the critical function is executed.

2 Integration of SW safety mechanism

Two main approaches are used to integrate safety mechanism:

The first approach (see the figure 6) consists in analyzing the whole treatment chain using FTA. Critical path is identified starting by critical output, and, step by step, critical interfaces and module are identified. This method is quite consuming: each critical module detected must be developed according to the ASIL level of the hazardous event. Similarly, critical function and data must be protected by safety mechanism as described above.

The second approach (see the figure 7) consists in monitoring treatment chain. For that purpose, error detection can cover consistency of inputs and outputs, availability and consistence of inputs. Error detection and error handling are integrated on a specific supervision module which is able to bypass functional modules when needed. This latter method allows limiting ASILB constraints on the supervision module itself instead of developing all modules in ASIL B.
Figure 6. Initial approach: integration of safety mechanisms on the entire processing chain. Module 1,3, 4 are identified as contributor to the hazardous event. On the contrary, Module 2 does not connect to any critical interfaces and can be developed as QM.

Figure 7. Supervision approach: a safety module monitors execution of software component by monitoring critical inputs and outputs. All functional modules are developed as QM.

3 Supervision applied to MBD
As MBD is new to our software, we have seized the opportunity to integrate safety, using supervision approach. One major benefit is to reduce impact on complex code without sacrificing safety. Moreover, for the most critical function, usual monitoring can be upgraded to functional redundancy. At last, MBD development is very convenient to this modular approach.

Most safety impacts are integrated into dedicated safety pre-monitoring and post-monitoring blocks (see the figure 8). A pre-monitoring block is inserted before execution of functional blocks. Its aim is to control availability and consistency of inputs. In case of failure, this block can impose default value on input(s) and/or signal error to functional block. Use of this block is recommended to avoid disturbance in functional chain of treatment or avoid non recoverable error (when soliciting services of basic software for instance). The main block is the post monitoring block inserted after execution of functional blocks. Its aim is to perform consistency check between inputs and output. In case of failure, this block can impose safe state on output and/or signal error to functional block. This latter block can also perform in parallel simplified functional treatment and compare its result with functional outputs.

4 Application for activation of critical a critical PUSH
Supervision for MBD has been applied to HADC. This function is responsible for limiting the descent speed by activating the braking system without requiring any further driver action. The Hazardous event is the untimely activation of HADC function by a push. This push is filtered by BCM software.

The figure 8 illustrates the integration of the supervision blocks. Filtering of The push is treated by two periodical runnables, one for decoding CAN signal (RUNA_PHADC) the other to filter, on several cycles the state of the push (RUNA_HADC). This latter runnable (RUNA_HADC) produces the critical output “HADC_ACTIVATION”.The two functional blocks are monitored by the safety blocks:

- Pre-monitoring blocks checks availability and coherence of inputs
  - CAN signal: SIGNAL_HADC_PUSH
  - CAN Life phase: PhaseVieCAN
  - Non filtered Push State: HADC_PUSH
- Post-monitoring blocks check consistency between critical functional output FUNCTIONAL_HADC_ACTIVATION, input Safety_HADC_PUSH and internal data RISING_PUSH

If any failure is detected, the safety blocks can impose safe state on critical data. That is why critical inputs and outputs have been duplicated through the safety blocks (for example: PhaseVieCAN is not directly used by the functional runnable but instead, safety data SAFETY_PhaseVieCAN is used).
This is not enough in case of unrecoverable failure. For instance, if any doubt is raised on calculation of the output “HADC_ACTIVATION” or consistence of CAN signal, this is the whole filtering chain that must be reinitialized, not only the output itself. This “functional reset” is performed through data “SAFETY_POST_ERROR” or data “SAFETY_PRE_ERROR”: Effect on functional is to reset treatment chain including filtering so that at the next execution step the system shall work correctly and on a predictable and secure way.

Figure 8. Integration of post an pre-monitoring box to secure activation of critical output

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.

1 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 the figure 9).

The following PSA functional safety activities are initiated:
- Definition of the safety plan and take into account the tailoring of the lifecycle for product development at the software level.
- Use the configurable software - applied the annex C of the ISO26262-6 standard (ISO 26262, 2011).
- Specific for handwritten coding:
  - Selection of the C language programming for the BCM software.
  - Definition of the design and coding guidelines for the programming with C language. The MISRA C rules was selected and completed with PSA own design rules.
  - Maximum low complexity for a C function is defined at 20.
  - Use of style guides and naming conventions is verified with MISRA checker.
  - Some defensive implementation techniques are used.
  - Definition and use of established design principles
- Specific for Model-Based Design:
  - Selection of the tools Matlab/Simulink® for model-based development in order to generate code for the BCM software.
  - Definition of the Modelling design and style guidelines for the model-based development The MISRA AC SLSF rules was selected and completed with PSA own design rules.
  - MISRA model-based design checker.
2 Specification of software safety requirements

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), (Villemeur, 1988) or a Technical Safety Concept (TSC).

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 unexpectedly open the relay when driving” is applied to the software “The software must not unexpectedly 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 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 (Mihalache et Bedoucha, 2014), 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.

3 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 for MBD.

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.

However, there is no limit to the number of interfaces because they come from system requirements.

Also, even if a PSA formal rule is not defined in the case of constraint time of execution sequences, we check systematically the runnable execution time and if necessary we take action in order to reduce the temporal constraints risks.

Today there are no quantitative interrupts rule imposed by PSA, but it is the responsibility of the supplier to restrict or explain the use of interrupts and this point is verified at the software reviews.

The BCM software architectural design describes the static design aspects and the dynamic aspect design (like sequence diagram or finite state machine). In the case of model-based design, modelling the software structure represents a major activity...
of the architect team. One of the main constraints of the model is to generate a code consistent with BCM resources (RAM, processor, flash ...).

In the case of reuse, the software architectural design is adapted and qualified in accordance with ISO26262-6 standard. (BCM software recycling a small number of old software architectural design).

The safety analysis of a dependent failure is made in order to implement the software safety requirements. It relied on freedom from interference or sufficient independence between software components. The software partitioning is not used to implement freedom from interference between software components what is used is redundancy by mirroring for the critical data.

The methods used for the verification of the unit design and implementation are walk-through and inspection with checklist (for both model-based or not) of the design made by software safety engineers and software architects.

The design principles included in MISRA model-based design are checked with the support of a MISRA checker tool: Model Advisor Checks of Matlab/Simulink®.

4 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.

These principles are applied also for MBD.

The methods used for the verification of the unit design and implementation (including MBD) are walk-through and inspection with checklist of the design made by software safety engineers, software architects, developers and quality engineers. The data flow analysis is partially covered by software FTA. In the case of MBD the software unit design and implementation are verified at the model level.

The design principles included in the MISRA C rules 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.

5 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.

6 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 used 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.

7 Verification of software safety requirements

The verification that the embedded software fulfills of the software safety requirements is done by the validation team.

Methodology improvement

1 Documentation and traceability

As modeling does not exclude the need of textual information to justify the software architecture and the software design, we plan to generate a document from the model. A tool will extract all the textual descriptions (that the developer writes in the model) and the screenshot of the associated Matlab/Simulink® boards to generate the document. This document will also be used for the traceability job: it will be compared to the several specification documents to generate a traceability report. We actually try to find another way to ensure the traceability straight from the modeling tool.

2 Test and test coverage

During the integration and unit testing phase we wish to perform back-to-back comparisons of implementations code and models. We want also to develop the analysis of structural coverage at the model level, using analogous structural coverage
metrics for models. Getting tests with a 100% coverage on our software components and/or our models costs lots of time, that's why we try to find the better way to get this result with automatic tools.

3 Validation
Currently, we work on the validation plan exchange flow; the goal is to complete the construction of all the paths between all the validation platforms. A test vector shall be efficiently translated to be used on MIL/SIL/HIL/vehicle. By this way any use-case created on any validation platform will be easily transformed to be consumed by any other validation platform.

Conclusion
The goal of this paper was to present the PSA approach to design embedded software safety, structured around the ISO 26262 standard and adapted to our needs. Our V cycle process is compliant with the ISO 26262 standard.

An example of our safety approach applied to a specified functionality in MBD was described in this article.
We also presented the differences and similarities between generated code - handwritten code for the safety point of view and the difficulties encountered during the implementation of this methodology.
Furthermore, this methodology is in continuous improvements which are identified and soon be applied in our process.

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: Electronic control unit
EEPROM: Electrically-Erasable Programmable Read-Only Memory
FTA: Fault Tree Analysis
HADC: Hill Assist Descent Control
HIL: Hardware in Loop
ICC3: Implementation Conformance Class 3
MBD: Model Based Design
MC/DC: Modified Condition/Decision Coverage
MISR: Motor Industry Software Reliability Association
MIL: Model in Loop
MMU: Memory Management Unit
SIL: Software in Loop
TSC: Technical Safety Concept

References

(Brindejonc et al., 2010), V. Brindejonc, G. Marcucilli, S. Petit: "Définition 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