Automatic Safety mechanisms implementation in Software Model-Based Development

Florent Fève

Automatic Safety mechanisms implementation in Software Model-Based Development

ERTSS 2014 : (thema : Model-based system engineering, Safety)

Author : Florent Feve
VALEO Laiernstrasse 12, 74321 Bietigheim-Bissingen, Deutschland; e-mail: florent.feve@valeo.com
Keywords : Model Based Development, Safety, Error detection, Fault injection, MIL, SIL

1. ABSTRACT

Model Based Development (MBD) is now a common approach for the automotive industry. Using modeling tools to simulate the behavior of a system before developing the corresponding product(s) through automatic code generation has proven its efficiency. The **Road vehicles — Functional safety — ISO 26262 standard (Part6)** [2] identifies MBD as a recommended approach especially for software architecture design with semi-formal notation and software verification with back-to-back testing through Model in the Loop (MIL) , Software in the Loop (SIL) and Processor in the Loop (PIL).
Regarding error detection the standard recommends a certain number of monitoring methods such as “Range checks of input and output data”, “Plausibility check”, “Control flow monitoring”, but does not give any concrete recommendation for software implementation of those methods and therefore how to test through fault injection. In the MBD approach, since code is generated automatically, safety mechanisms must be introduced at model level.

This paper explains the automated process that has been developed to implement safety mechanisms for error detection at model level. Concrete example is described showing the advantage of MBD for safety mechanisms implementation as well as for testing. The safety mechanisms tests are done through fault injection at both model and code level. It illustrates how, from a graphical model implementing functional requirements, dedicated safety mechanisms are automatically generated in the model on selected elements.
In addition, test cases are automatically generated with complete validation environment to allow full test of the safety requirements. For functional requirement verification, the MIL, SIL, PIL and HIL regular process is applied in block box. For safety mechanism verification, the same process is applied but as white box (fault injection). Results reports for model simulation and code as well as coverage reports for model and code are automatically generated.

Benefits of this innovating process are to:

- Minimize the risks for the implementation of monitoring methods since safety mechanisms are automatically introduced in the model through a plug-in.
- Improve consistency through code generation but also through automatic documentation generation from the model.
- Support full testing and verification process (test cases creation, execution and reports generation).
- Push standardization across projects for safety mechanisms implementation.

2. INTRODUCTION

Model Based Development Process including automatic code generation is used as standard in VALEO for 10 years. It has now reached a mature level (even though it continuously improves). In recent years new challenges appeared with safety requirements defined in the Road vehicles — Functional safety — ISO 26262 standard (Part6). The Part 6 of the standard, released end of 2011, specifies Software related Safety requirements. For each main topics (SW architecture design, unit design, implementation, testing, integration ,…) the standard gives some tables in which are listed different methods to address the topic. From the process point of view, many of the methods are already used in “non-safety related“ developments especially for Design approach and Verification and Validation strategy.

Regarding **Mechanisms for error detection at the software architectural level** (**Table 4 (Part 6))** the ISO 26262 standard defines generic mechanisms to be applied for error detection and handling without proposing detailed implementation. The systematic approach described in this paper gives an example of one of the concrete and automated solutions for covering Control flow monitoring method.

3. MODEL- BASED SOFWARE DEVELOPMENT PROCESS

3.1 Overview

This chapter will give a brief description of MBD process. The aim is to describe only the part of the process which is involved in the automatic safety mechanism.

Model Based Development Process including automatic code generation is used as standard in VALEO for 10 years. For reactive system, which will be the use case for MBD safety mechanism described in this paper, Statemate tool is used. The scope of the model is the functional requirements which will be implemented as part of the application layer of the embedded software. From Requirements analysis, a functional model is first designed in order to validate the functional requirements. In parallel Requirement based testing is initiated. Validation team analysis requirements and build test cases that will be used in MIL (Model In the Loop), SIL (Software In the Loop) and HIL (Hardware In the Loop). Those test cases are generated as test scripts that will be automatically applied as inputs to the model for MIL testing. MIL reports cover both results comparison (expected results vs simulation results) and model coverage. Later on for HIL testing with the final ECU the exact same test cases will be performed.

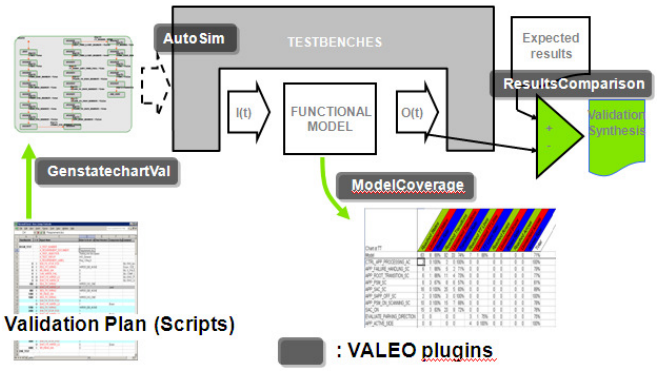


Figure 1 — MIL validation process

The process is fully automated with a set of plugins allowing translating test scripts into Statemate Statecharts that will be added to the simulation environment as test drivers for simulation.

Once the model has formally been validated, the implementation phase starts with dedicated design rules checked automatically by an in-house plug-in. Code is generated to be executed first on a host PC. SIL testing is then performed by complete reuse of the MIL environment. Same as for MIL testing, SIL reports cover both results comparison (expected results vs code execution results) and code coverage. Code coverage is performed using a third party tool which is part of the full automation. No new scripting or configuration is needed. This part of the process is especially useful in case of Safety relevant Sw development as we will see later on.

Full traceability from requirements to code is covered though dedicated attributes for all elements of the model which are generated as comments in the source code by code generator configuration.

4. MBD PROCESS AND SAFETY REQUIREMENTS

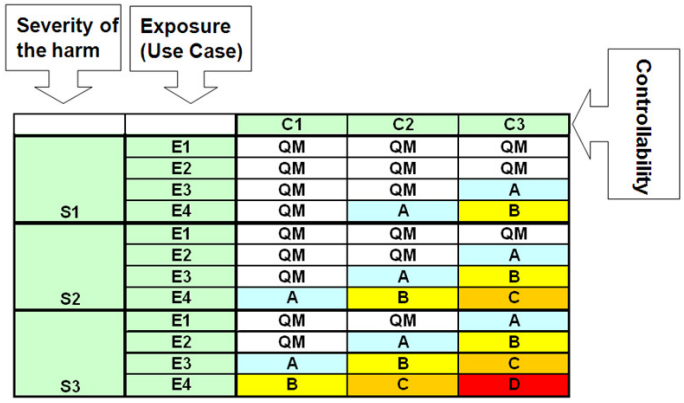
4.1 Road vehicles – Functional safety -

The ISO 26262 is a set of documents defining the standard for Functional safety for Road vehicles. It defines an automotive specific risk-based approach for determining risk classes: ASIL (Automotive Safety Integrity Level). Four classes (A, B, C, D) are used to specify the necessary safety requirements for achieving an acceptable residual risk. “D” represents the highest risk.

4.1.1 ASIL determination

ASIL determination is based on the combination of 3 factors: Severity of potential harm (S), Probability of exposure (E) and Controllability / chances to avoid harm (C).

- Severity of potential harm has 4 classes: S0 (No injuries) to S3 (Life-threatening injuries or fatal injuries).
- Probability of exposure has 5 classes: E0 (incredible) to E4 (High probability). Only E1 (Very low probability) to E4 have affect on ASIL determination.
- Controllability/chances to avoid harm has 4 classes: C0 (Controllable in general) to C3 (Difficult to control or uncontrollable). Only C1 (Simply controllable) to C3 have affect on ASIL determination.



		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

QM stands for “Quality Management” and means that there is no ISO 26262 requirement to apply.

Figure 2 — ASIL determination

The ASIL determination is based on the Hazard Analysis and Risk Assessment and leads to the Safety goal concept. Next step of the Functional Safety process is to establish the Functional Safety requirements which allow to build the Technical Safety

concept out of which Safety requirements can be allocated, among others, to software. Therefore ASIL determination for SW activity is established. It will be the base for Safety methods choice for Software.

The ISO 26262 Part 6 defines how ASIL level will influence SW development in terms of Process, Design and Verification strategy.

Next chapter will give examples on how the ASIL level will influence MBD process, and what could be the added value of using MBD approach for safety relevant requirements to be implemented at Software level. This approach comes in addition of other approaches for Safety mechanisms implementation used for Software modules that are developed manually (no automatic code generation).

4.1.2 Use of MBD for safety related Software

Part 8 of ISO 26262 [1] : "**Supporting processes**" gives a set of tables to indicate which kind of methods are recommended for each part of the process. For each method a quotation is given depending on ASIL level.

"++": The method is highly recommended for this ASIL
"+": The method is recommended for this ASIL.
"o": The method has no recommendation for or against its usage for this ASIL.

In the document we can find that Model Based Development is either recommended or highly recommended for safety related product development depending on the ASIL level. For example, specification methods are defined in the table "Specification of safety requirements" [1]. Three general methods for requirements specification are listed:

- Informal notations (1a)
- Semi-formal notations (1b)
- Formal notations (1c)

Tools used for MBD are semi-formal notations. This method is recommended ("+") for ASIL A and B and highly recommended ("++") for ASIL C and D.

Another example is for verification methods which are listed in the table « Specification Verification of safety requirements in part 8 » [1]. Four methods are listed:

- Verification by walk-through (1a)
- Verification by inspection (1b)
- Semi-formal verification (1c)
- Formal verification (1d)

For method 1c, there is the following note : "can be supported by executable models" which is indeed in-line with the MBD process. Actually MBD is also an added value for method 1b in the way that large part of the inspection can be automated.

Method 1b is recommended ("+") for ASIL A and highly recommended ("++") for ASIL B, C and D.
Method 1c is recommended ("+") for ASIL A and B and highly recommended ("++") for ASIL C and D.

In addition Part 6 of ISO 26262 [2] mentions the following:

- Table 6 (PART 6)— **Methods for the verification of the software architectural design**, method **1C : Simulation of dynamic parts of the design** which "requires the usage of executable models for the dynamic parts of the software architecture".

- Table 10 (PART 6)— **Methods for software unit testing**, method 1e **: Back-to-back comparison test between model and code, if applicable** which "requires a model that can simulate the functionality of the software units. Here, the model and code are stimulated in the same way and results compared with each other".

Both methods are either recommended or highly recommended for safety related product development depending on the ASIL level.

Globally, Model based development process is a recommended approach for Safety related product development in terms of Requirement formalization, architecture, testing and validation. VALEO MBD process is already in line with most of the recommendation of the ISO 26262 Standard.

The main adaptation needed comes from the safety mechanisms to be implemented at software level. In the MBD approach, since code is generated automatically from the model, safety mechanisms must be introduced at model level.

4.2 Safety mechanisms for error detection

This steps of the process comes after the Safety analyze, that is to say, the Software functions/modules to be protected by safety mechanisms has been already identified. Critical Path Analysis can be applied to identify all modules for which safety mechanisms should be implemented.

[2] Table 4 (PART 6)— **Mechanisms for error detection at the software architectural level**, indicates methods to be applied but does not specify any concrete implementation mechanism. Six methods/measures are listed :

- Range checks of input and output data (1a)
- Plausibility check (1b)
- Detection of data errors (1c)

- External monitoring facility (1d)
- Control flow monitoring (1e)
- Divers software design (1f)

Method 1b is recommended (“+“) for ASIL A, B and C and highly recommended (“++”) for ASIL D.
Method 1e is recommended (“+“) for ASIL B and highly recommended (“++”) for ASIL C and D.

The ISO 26262 Part 6 lists methods without giving definitions. “Plausibility check” and “Control flow monitoring” for example are not clearly defined. Examples of those methods can be found in some publications but it seems that different interpretations can be found. This paper describes one safety mechanism implemented for error detection which we have classified as “Control flow monitoring” even though some would see it has “Plausibility check”. The implementation is done automatically in the model through a plug-in. It should be performed when functionality is implemented using state machines.

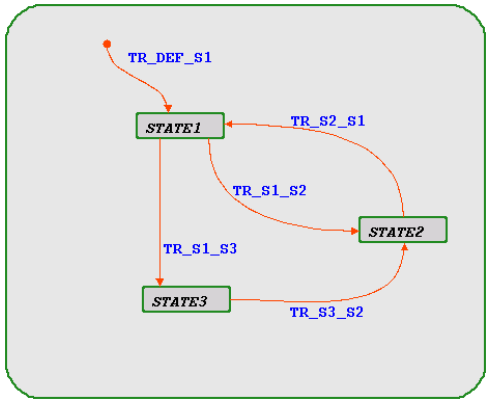
The workflow is the following:

- functional model is designed without safety mechanisms.
- functional test scripts are created based on Requirements analyze (classical MIL and SIL process)
- once safety analyze has identified which part of the design should be protected (this can be done through Critical Path Analisis), automatic implementation is generated through a plug-in.
- Test scripts are also generated automatically by the plug-in in order to be able to produce Model and code coverage reports for all safety related code. This is done by automatic modification of existing test scripts introducing Faults to trigger the ”safety code “ (white box testing).

4.3 Control flow monitoring :

4.3.1 Rational

The aim of the safety mechanism is to detect, in a state machine, during SW execution, any transition between one state to the other which is not in line with the design. From formally designed Statecharts from functional requirements, a table is build which lists all basic states and indicates all allowed former state. The table is a 2 dimensional array. The size is determined with the number of basic states of the state machine, including subcharts. Bit- array is used to represent the valid transitions to reach a state. Each bit represents a state. Value “1” indicates a valid transition. In the basic example below, 5 transitions are allowed : TR_DEF_S1, TR_S2_S1, TR_S1_S2, TR_S1_S3 and TR_S3_S2.



This leads to the following table :

Dst \ Src	STATE3	STATE2	STATE1
STATE1	0	1	0
STATE2	1	0	1
STATE3	0	0	1

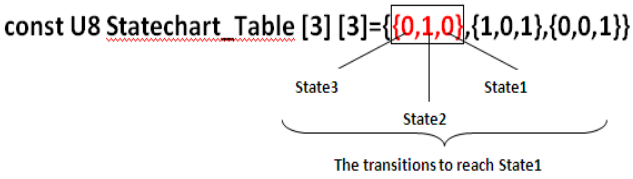
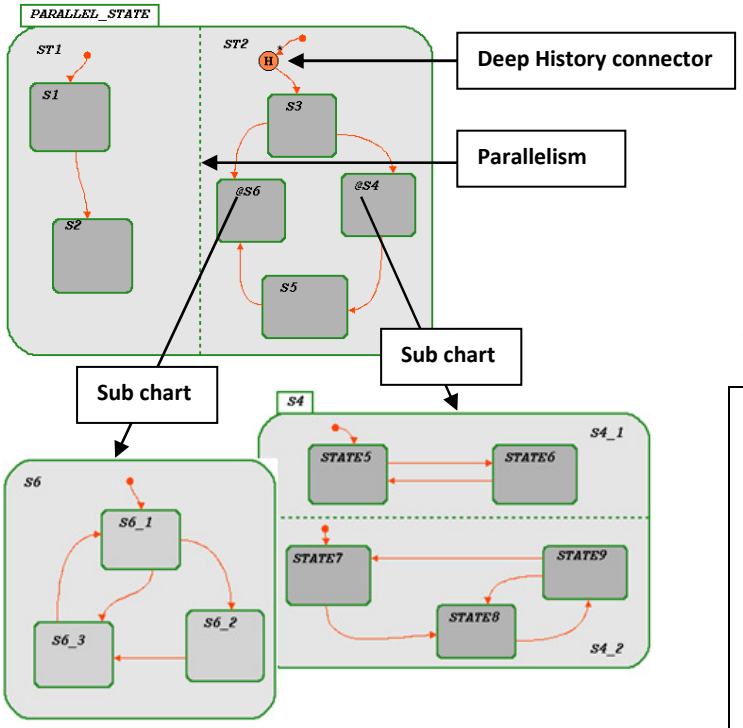


Figure 3 — Transition Statechart table: example 1

The algorithm for this table is pretty simple, but becomes more complicated when considering all possibility of Statechart modeling with sub charts, parallelism, and History connector. History connector is a notation indicating that when leaving a chart or an encapsulated state, the last validate state will be entered when the chart is active again (instead of the default state). Parallel charts leads to consider more than 1 default state (as many as there are parallel state machines in a Statechart), History connector leads to consider all transitions leading to the chart as possible transition to each elementary state.

The initial version of the plug-in supported only simple Statecharts. It now supports all possibility of modeling, See example Figure 4.



Src / Dst	S6_3	S6_2	S6_1	STATE9	STATE8	STATE7	STATE6	STATE5	S4	S5	S3	S2	S1	PARALLEL_STATE
PARALLEL_STATE	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S2	0	0	0	0	0	0	0	0	0	0	0	0	1	0
S3	0	0	0	0	0	0	0	0	0	0	1	0	0	0
S5	0	0	0	0	0	0	0	0	1	1	0	0	0	0
S4	0	0	0	0	0	0	0	0	1	0	1	0	0	0
STATE5	0	0	0	0	0	0	1	1	0	0	0	0	0	0
STATE6	0	0	0	0	0	0	1	1	0	0	0	0	0	0
STATE7	0	0	0	1	0	1	0	0	0	0	0	0	0	0
STATE8	0	0	0	1	1	1	0	0	0	0	0	0	0	0
STATE9	0	0	0	1	1	0	0	0	0	0	0	0	0	0
S6_1	1	0	1	0	0	0	0	0	0	1	1	0	0	0
S6_2	0	1	1	0	0	0	0	0	0	0	0	0	0	0
S6_3	1	1	1	0	0	0	0	0	0	0	0	0	0	0

Figure 4 — Transition Statechart table: example 2

Each time a state is entered, the above table is analyzed to detect any invalid former state by comparing the former state value to the "allowed former state" defined in the table for each state. Current state is then stored to be used for next state entered. If an invalid former state is detected an error function is called.

The plug-in analyzes the Statecharts in which safety mechanism needs to be introduced. to be secured, creates the necessary table and adds in each state an "entering static reaction" to perform the check each time the state is entered. Following items are automatically created in the model :

- enumeration listing all basic states (includes subcharts) and Parallel states. Number of enumeration values equal number of states.
- Statechart table listing valid transitions for each state
- subroutine for error detection (to be customized for each project)
- "entering static reaction" (actions performed when entering a state) for each state to detect non allowed transition.

This instrumentation of the model is possible due to a set of API provided with Statemate which allows any user to develop plugins able to parse the model and retrieve any elements by using a set of query functions but also to create any kind of model element. This approach of using the so called dataport APIs was already used in the Standard MIL process to translate test scripts into Satecharts (see Figure 1 plug-in GenStatechartVal).

Therefore each elementary state will be instrumented with following pseudo code:

S2

```
/*CTFL start*/
entering/
$OLD_TEMP=CTFL_TOPCHART_OLD_STATE(1);
$NEW_STATE=TOPCHART_S2;
$CTFL_MASK=CTFL_GET_MASK($OLD_TEMP);
$CTFL_VALUE=CTFL_TOPCHART_TABLE($NEW_STATE) and ($CTFL_MASK);
if ($CTFL_VALUE==0) then

CTFL_LO_ERROR_TOPCHART(1)=CTFL_STM_ERROR($NEW_STATE,$OLD_TEMP);
end if;
CTFL_TOPCHART_OLD_STATE(1)=$NEW_STATE;
/*CTFL end*/
```

Figure 5 — Pseudo code instrumentation if elementary states

"entering" is a keyword for specifying actions to be performed only when the state is entered. The former state (hold in the array CTRL_TOPCHART_OLD_SATE in the example) and the current one are checked against the Statechart table. If a mismatch is detected an error handler is triggered (CTFL_STM_ERROR) passing as parameter the current state value and the former one.

For Default state (initial state when a Statechart becomes active) specific pseudo code is generated in order to distinguish the entering in the state due to the first activation of the Statechart with further entry in the state.

4.3.2 Plug-in workflow

The plug-in displays the whole tree structure of the model allowing choosing which part of the model should be instrumented with Safety mechanisms.

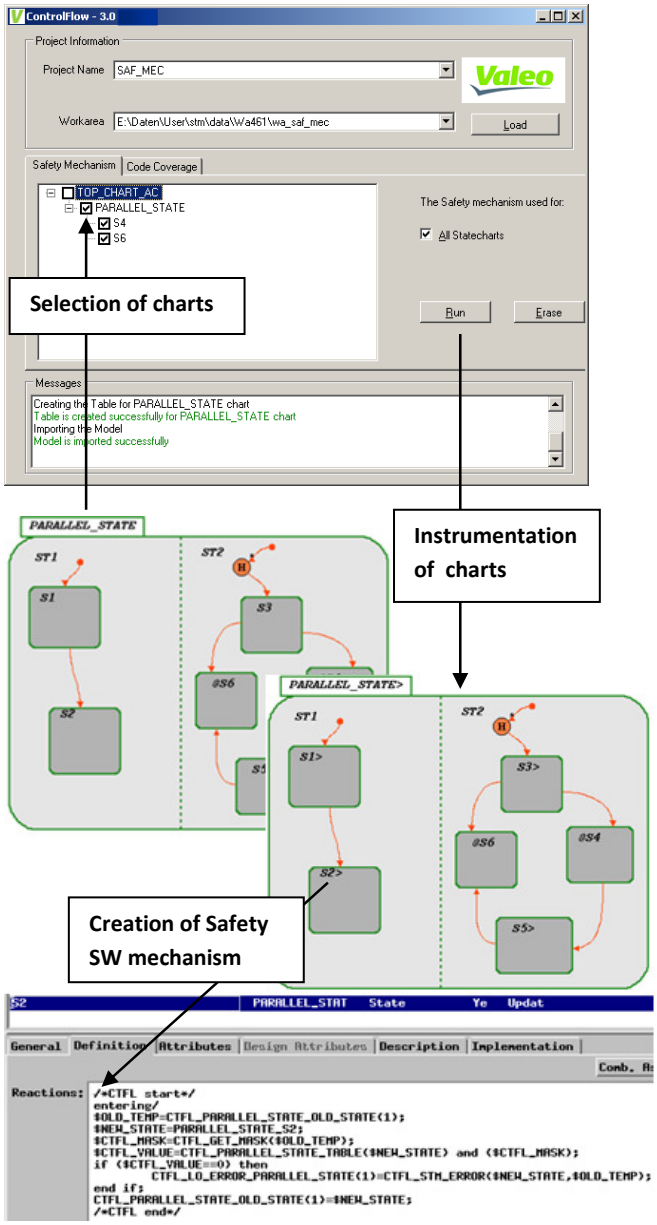


Figure 6 — Model instrumentation workflow

In the above example the whole model is selected. Statechart PARALLEL_STATE_SC has 2 subcharts S4 and S6. In addition it has some parallel charts ST1 and ST2 (identified we the green dotted line) and a History connector. This use case is taken into account by the plug-in algorithm. Parallel states impact the number of Statecharts table, history connector impacts the values of the Statecharts table. The plug-in will create Statechart tables (one per parallel chart) and static reaction for States S1, S2, S3, S5 (+ all basic states of the subcharts).

Static reaction of elementary states now contains pseudo code (later on translated in C language as part of the graphical model). A generic function is created. This function is called during simulation and execution in case of fault error detection. Each project using this plug-in will customize the function to define the failure reaction depending of the safety analyze.

4.4 Verification and Testing:

Regarding Validation and Verification strategy, the ISO 26262 has a dedicated chapter “Software unit testing” in which we can find that [1] Software unit testing can be executed in different environments with MIL, SIL, PIL and HIL. It is also mentioned that in the case of model-based development, software unit testing may be moved to the model level using analogous structural coverage metrics for models.

The following tables can be found:

Table 12 (Part 6)— **Structural coverage metrics at the software unit level**, mentions Branch coverage and MC/DC coverage as recommended or highly recommended methods.

Table 10 (Part 6)— **Methods for software unit testing**, method 1c : Fault injection test

We have seen how a safety mechanism is implemented automatically in a model. To be compliant with the standard the instrumented code needs to be covered in MC/DC mode.

By definition the code introduced for safety mechanism cannot be reached by regular test cases generated from the model interfaces. Those tests will need to be generated as white box through fault injection by modifying internal state variables in order to generate an error. The aim of those test scripts is to cover the structural tests of the implemented safety mechanism.

The solution chosen for the automation of the fault injection through test scripts was to reuse the functional test cases and introduce fault to cover fault error detection functions. As mentioned in chapter 3, our regular process for black box testing uses test scripts which describes some test sequences with the expected results. In general the signals used are the interfaces of the module. During MIL testing, inputs and outputs are stored by the simulation tool. The format of this file contains Timing reference, name of variables and new value. List of variables to be recorded is defined by the test cases. Statemate will record during simulation every signals in the list each time a signal changes. In example of recorded output file Figure 7, we can find regular output signals (EWO_xxx).

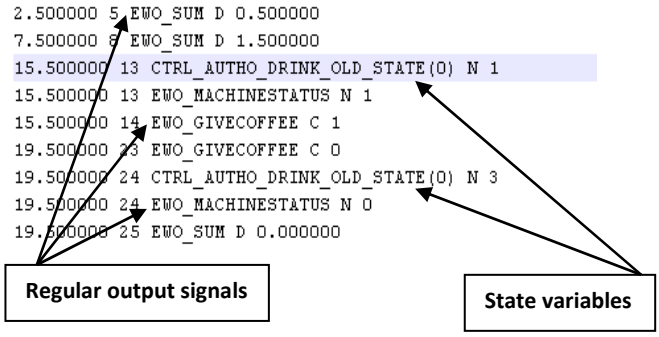


Figure 7 — Simulation output files format

In regular functional MIL testing, this output file allows comparing the results of the simulation with the expected results defined in the script. Model coverage and, after performing SIL, code coverage reports are produced in order to check coverage. Therefore the verification that all states and all transitions are reached by functional test cases is available. The 100% coverage for states and transitions is a mandatory condition in order to generate test scripts for Safety mechanisms as we will see further on. For conditions, only Decision coverage (DC) is needed, no need of MC/DC coverage. Indeed what is needed is to have at least one test scenario that will lead to cover a transition, no matter if we cover all the possibility to trigger the transition. For example if the label of a transition is [Condition_A] OR [Condition_B], a scenario in which Condition_A is true is enough.

The test cases that will be generated for the Safety mechanism structural coverage are not generated from scratch but from existing tests scripts developed based on functional requirements. Necessary step is to run the functional tests with adding in the list of “output” signals to be stored the state variables that were introduced by the plug-in. This will lead to have output files containing references of state variables (see Figure 7 CTRL_AUTH_DRINK_OLD_STATE state variables).

The test scripts are, as well has the output recorded files during simulation, inputs for the plug-in. Both files are parsed to detect which input change in the script leads to a new state. Due to the presence of State variables in the “outputs” file it is possible to detect in the output recorded file each reference to the entering in a state with its timing reference. This timing reference will be used to identify in the input test script where to modify. The scripts are then duplicated and modified so that state variables could be corrupted in order to generate safety error.

Figure 8 describes this structural coverage test script generation workflow

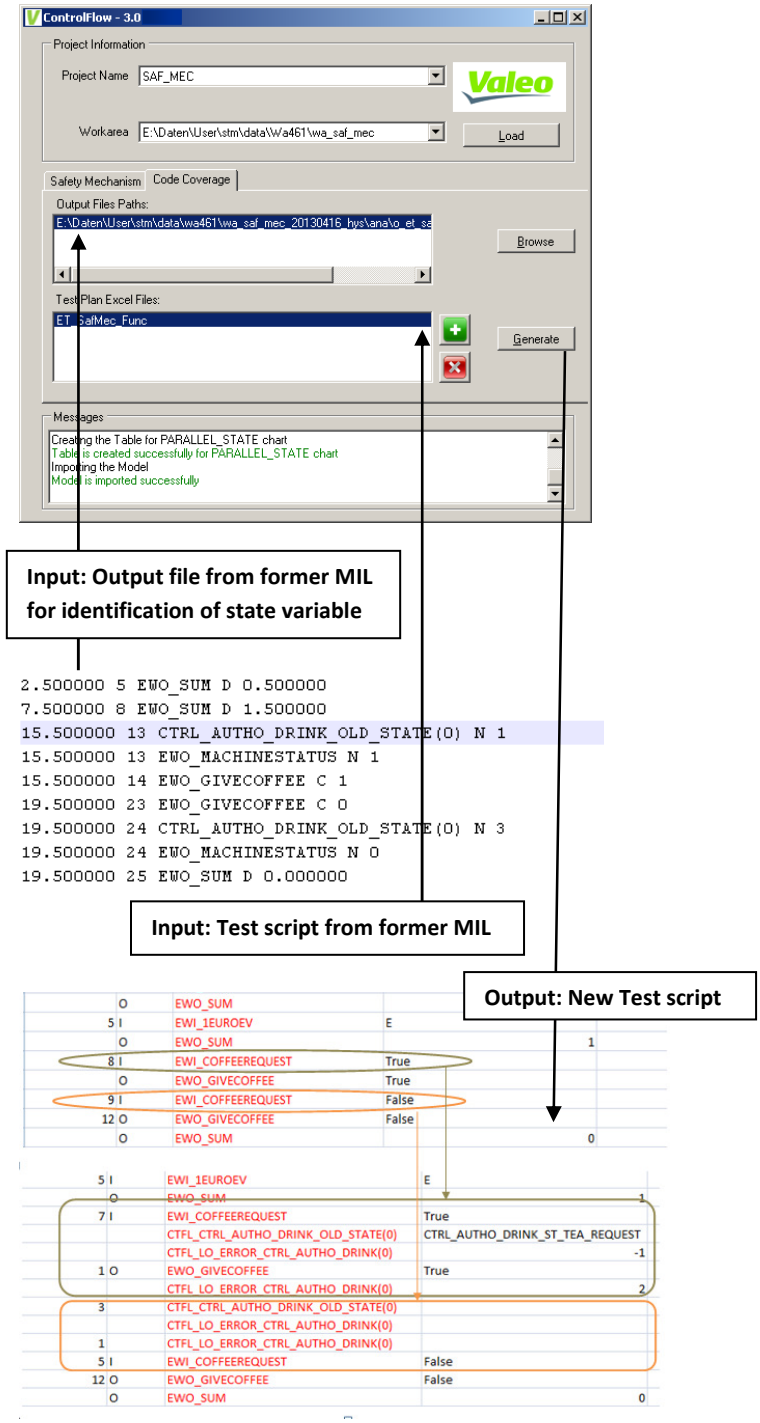


Figure 8 — workflow for Structural coverage script generation

During the generation of the structural coverage test scripts, in addition of the fault injection, the expected results are defined (triggering of error handler). The scripts are now ready to be performed in MIL and SIL.

5. CONCLUSION

This example of automated Safety mechanism generation is only one part of safety techniques to be designed and implemented at Software level but it has the great advantage to be fully automated not only for implementation but also for test cases. This innovating automated process is a first step on how to fill the gap between MBD regular approach and the Safety related requirements defined in the ISO 26262. It minimizes the risks for the implementation of monitoring methods since the safety mechanisms are automatically introduced in the model through a plug-in. In case of functional change request, model is modified and safety mechanism is regenerated with no effort. The process also supports full automated testing and verification workflow (test cases creation, execution and reports generation). Finally it pushes standardization across projects for safety mechanisms implementation. Projects only need to determine the actual error handler implementation for error detection.

6. GLOSSARY

MBD Model Based Development

MIL Model In the Loop

SIL Software In the Loop

PIL Processor In the Loop

HIL Hardware In the Loop

ASIL Automotive Safety Integrity Level

MC/DC Modified Condition / Decision Coverage

7. REFERENCES

[1] Road vehicles — Functional safety — Part 8: Supporting processes

[2] Road vehicles — Functional safety — Part 6: Product development: software level

[3] MISRA AC AGC Guidelines for the application of MISRA-C:2004 in the context of automatic code generation, ISBN 978-1-906400-02-6, MIRA, November 2007.