Model Quality Objectives for embedded software development with MATLAB and Simulink

Jérôme Bouquet, Stéphane Faure, Florent Fève, Matthieu Foucault, Ursula Garcia, François Guérin, Thierry Hubert, Florian Levy, Stéphane Louvet, Patrick Munier, et al.

Model Quality Objectives for Embedded Software Development with MATLAB and Simulink

Jérôme Bouquet (Renault), Stéphane Faure (Valeo), Florent Fève (Valeo), Matthieu Foucault (PSA Peugeot Citroën), Ursula Garcia (Robert Bosch), François Guérin (MathWorks), Thierry Hubert (PSA Peugeot Citroën), Florian Levy (Renault), Stéphane Louvet (Robert Bosch), Patrick Munier (MathWorks), Pierre-Nicolas Paton (Delphi Technologies), Alain Spiewek (Delphi Technologies)

Abstract: This paper presents standard quality objectives for models developed with Simulink® at different phases of the software development lifecycle. This standard, named Model Quality Objectives (MQO), has been defined by a group of leading actors from the automotive industry and MathWorks, the company that develops the MATLAB®, Simulink, and Polyspace® products. The purpose of this standard is to clarify and ease the collaboration between OEM and suppliers when sharing Simulink models in the context of embedded software development to drive the production of higher quality and integrity software.

This paper first defines a software development approach based on four types of design models used at four different phases of the software development lifecycle. Then, a specific quality objective, named MQO, is proposed for each type of model. Each objective is defined as a set of quality characteristics with some measurable criteria named Model Quality Requirement (MQR). Some additional guidelines are provided to manage the planning and quality assessment activities related to MQO and MQR. This paper concludes with some expected impact on the adoption of MQO by the automotive industry.

1 Background and Motivation

Design models developed with the Simulink software are widely used in the industry to accelerate the development of embedded software. Those models enable engineers to accomplish various engineering tasks such as frequency-domain analysis, desktop simulation, formally-based verification, and automatic code generation. This development process is known as Model-Based Design.

Design models can be developed at a very early stage to validate requirements and quickly explore design solutions. Such models can also be incrementally refined until they reach a level of maturity that is sufficient to generate code that complies with international software safety standards. To incrementally increase the maturity of the design models, different engineering disciplines need to be involved such as system engineering, control engineering and software engineering. Collaborating with the same language, tools, and models is a great way to improve communication between engineers and reduce the project cost and development time. However, with different disciplines using design models at different project phases, confusion may arise about the contribution of models and what they represent.

An incorrect interpretation of what the models represent can lead to an incorrect use of those models and ultimately impact the quality of the software produced.

OEM and tier-one suppliers that participate in the definition of MQO have shared many concrete use cases when underspecified models or models with insufficient maturity have been prematurely promoted as “ready for coding”.

Consequently, higher development effort than planned, bugs, and difficult conversations related to responsibilities would then take place. In order to avoid this situation, this document proposes to clarify the role of design models for the development of embedded software and standardize measurable criteria to verify their quality.

This approach has been inspired by the Software Quality Objectives (SQO) [1] defined by a group of automotive actors and MathWorks in 2010, at a time when most exchanges between car manufacturers and suppliers were based on textual specification and manual code. This approach also aims to go one step further in the formalization of model sharing, as defined by Bosch [2] in 2014, and in the implementation of techniques and measures proposed by software safety standards such as ISO26262-6. [3]

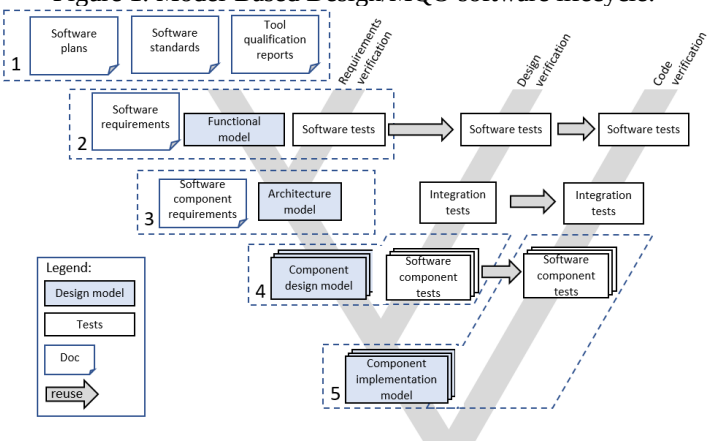
2 Purpose and Scope

The purpose of this paper is a) to define the main use cases of design models for software development and b) to define a standard and generic approach to assess the quality of models depending on their use cases.

3 Software Development with Design Models

This document defines a development approach based on four types of design models supporting the left-hand side of the V-cycle.

Figure 1. Model-Based Design/MQO software lifecycle.



The Model-Based Design/MQO software development lifecycle involves five specific phases marked as 1 to 5 in Figure 1. Sections 3.1 to 3.5 will provide greater details on the phases.

Figure 2 shows how the Model-Based Design/MQO software development lifecycle maps to other software development lifecycles from the industry. The phases supported by design models are highlighted with a dark background, and Model-Based Design is referred to as MBD.

Figure 2. Model-Based Design / MQO software phases versus other industry standards [3], [4], [5], [6].

	MBD/MQO	ISO26262	Automotive SPICE	DO-331	EN50128
1	Software planning	Initiation of product development at the software level	Initiation of product development at the software level	Software planning	Software planning
2	Software requirements	Specification of software safety requirements	Software requirements analysis	Software requirements	Software requirements
3	Software architectural design	Software architectural design	Software architectural design	Software design (architecture and Low level requirements)	Software architecture and design
4	Software component design and testing	Software unit design and implementation	Software detailed design and unit construction		Software component design
5	Software component implementation and testing	Software unit testing	Software unit verification	Software coding	Software component implementation and testing
	Software integration	Software integration and testing	Software integration and integration test	Software integration	Software integration
	Software testing	Verification of software safety requirements	Software qualification test	Software testing	Software validation

3.1 Software Planning Phase

This section defines the planning activities that must be carried out to prepare the use of design models. This is recommended for the use of functional models and mandatory for the use of architecture, component design, and component implementation models. Most of these concepts are already imposed by safety standards such as DO-331 [5].

Scope definition: All design models may not be applicable to all projects. For instance, the scope of Model-Based Design can be reduced to the development of a single software component or only used to support the software architectural design specification. The project shall define the software development phases that will be supported by design models. Each design model shall be managed independently as a work product of the software development phase it belongs to.

Tools definition: The tools that support the development and verification of design models shall be identified and classified at the beginning of the project. Those tools shall be qualified, if required by the project.

Standards definition: The modeling standard used to support the development of design models shall be defined prior to entering the software architecture phase. The coding standard used to support the development of design models shall be defined prior to entering the software component implementation phase, or ideally, prior to entering the software component design phase.

MQR identification and allocation: The MQR shall be identified and agreed to by the project stakeholders at the beginning of the project. Some MQR shall be adapted to the project requirements (e.g. model and code coverage criteria). Each MQR shall be allocated to a project stakeholder.

Strategy to achieve MQR: Once the MQR has been defined for the project, a strategy shall be defined to achieve the objective. Such a strategy can include intermediate steps corresponding to project milestones, specific training, or a tools migration process. For instance, it is recommended to gradually increase the coverage criteria and not wait for the final version of the software to perform most of the test development effort.

MQR conformance demonstration: The conformance with the project MQR shall be planned and demonstrated at the end of the project. The verification of each MQR shall lead to the production of a report produced by the project stakeholder responsible of the MQR. Sufficient justifications must be provided when MQR are not met (e.g. missing coverage should be justified). The person in charge of assessing the compliance shall have the necessary skills to understand the justifications.

3.2 Software Requirements Phase

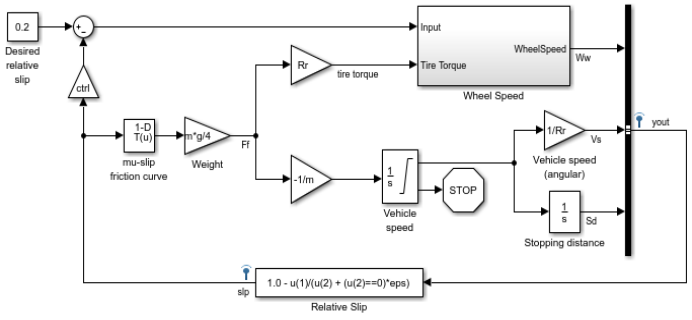
This section focuses on the functional model developed during the software requirement phase. The role of the functional model is to clarify and refine complex dynamic behaviors that need to be translated into software requirements.

In most cases, the functional model and the software requirements are concurrently developed by the person in charge of the software requirements. This functional model engineer supports the stabilization of the software requirements (the “what”) while identifying good design solutions (the “how”) that could be further elaborated during the design and implementation phases. The functional model is often referred to as an executable specification because it provides a functional behavior that satisfies the functional requirements. However, the functional model does not replace the software functional requirements. The functional model contributes to the validation activities of the software requirements.

The functional model focuses on the correctness of algorithms and equations. It does not have to consider design constraints related to embedded software development. However, when developing the functional model, it should anticipate the main characteristics of the hardware platform and their impact on the software requirements.

The functional model may not be needed if the software functional requirements are simple to implement, nor does it have to be representative of all the software functional requirements. Figure 3 shows an example of a functional model using continuous time and is limited to a small function of a larger software.

Figure 3. Example of a functional model (anti-lock braking system).



3.3 Software Architectural Design Phase

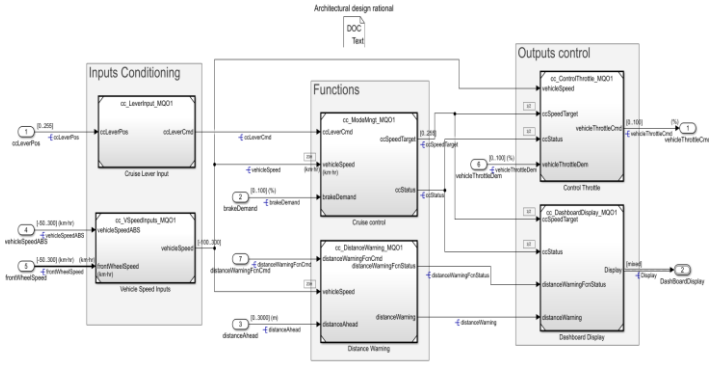
This section focuses on the architecture model developed during the software architectural design phase. The role of the architecture model is to contribute to the specification of the software architectural design.

Graphical notation is naturally well-suited to defining an organization of components, representing interfaces and connections, and specifying component scheduling. For a complex architecture, it is not conceivable to develop such a diagram without a proper modeling language and a computer-aided design tool such as Simulink.

The architecture model fully specifies the static software architectural design (e.g. component models, interfaces) and provides links/references to the component design models that will be built or are already built. The architecture design model is associated with a data dictionary that defines the data and interfaces of the software and its components.

The architecture model directly contributes to the design activities and is therefore subject to conformance with industry quality standards, safety standards, and/or architecture standards (e.g. traceability to requirements, compatibility with architecture standard).

Figure 4. Example of architecture model.



3.4 Software Component Design and Testing Phase

This section focuses on the component design model developed during the software component design and testing phase. The role of the component design model is to provide a complete specification of the software component design and support its verification with dynamic and static analysis.

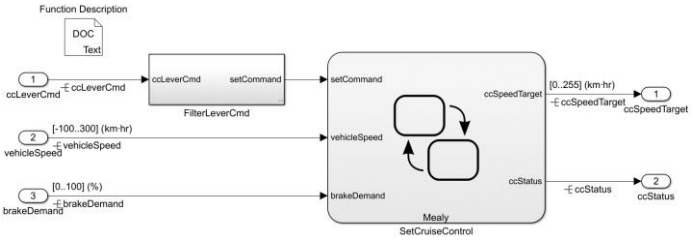
The use of a high-level modeling and programming language enables better management of the complexity of algorithms and reduces the probability of design errors. The support of simulation and static analysis contributes to elimination of design errors.

The component design model fully specifies the algorithms and equations that will be part of the embedded software and excludes any elements used for debugging or prototyping such as measurement points or override mechanisms. Each component design model is associated with a data dictionary that defines its interface, parameters, and monitored signals.

The component model directly contributes to the development activities and is therefore subject to conformance with industry quality standards, safety standards, and/or design standards (e.g. conformance to modeling standard, traceability to requirements).

Figure 5 shows an example of a component design model with fully defined interfaces and sub-functions implemented with state machines.

Figure 5. Example of component design model.



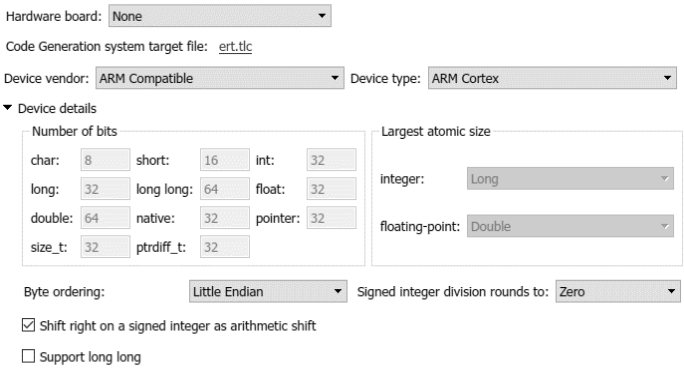
3.5 Software Component Implementation and Testing Phase

This section focuses on the component implementation model developed during the software component design and testing phase. The role of the component implementation model is to enable the generation of production code for a specific embedded target and basic software.

The component implementation model fully specifies the software component implementation. Implementation details are added to the data dictionary to refine how to represent parameters and signals in the target memory. Code configuration options and customization are defined to integrate the generated code with specific basic software functions, so they match the target characteristics (e.g. byte ordering) and satisfy the component code memory footprint and execution performance requirements allocated to the software component.

The generated code of the component implementation model directly contributes to the development activities and is therefore subject to conformance with the industry quality standard, safety standard, and/or coding standard (e.g. MISRA C®). Each component implementation model is associated with a data dictionary that defines its interface parameters and monitored signals.

Figure 6. Example of code generation configuration for the component implementation model.



3.6 Relationship Between Design Models

Each design model shall be independently managed as a work product of the software development phase in which it belongs. At the same time, design models can share design information and shall be consistent. For instance, the component design model in Figure 5 shares its interface definition with the architecture model of Figure 4. Whenever consistency is required, reuse is encouraged.

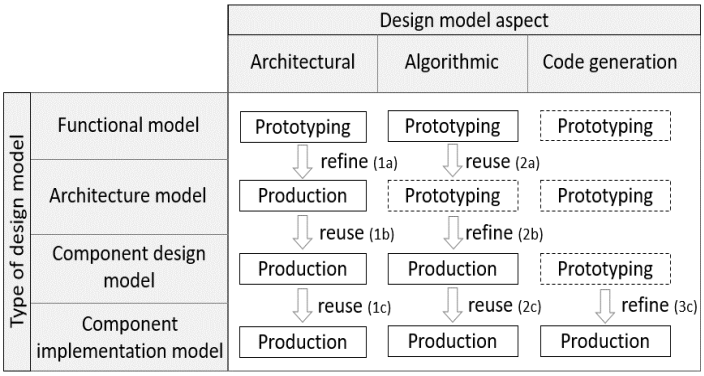
Figure 7 indicates which aspects can be reused between design models ("reuse" arrow). It also provides guidance on which aspects of design models can be partially reused to accelerate development ("refine" arrow). The arrows on Figure 7 can apply to the following modeling aspects of design models:

- Architectural aspect: interface, scheduling, partitioning, intercomponent control and data flow, etc.
- Algorithmic aspect: mathematical calculation, component control and data flow, state machine, truth table, etc.
- Code generation aspect: memory management, data access, function prototype, code optimization, etc.

The design models differ from each other by the level of maturity and importance of the different modeling aspects described above. Figure 7 indicates the levels of maturity and importance based on the following definitions and representations:

- Maturity level: high (Production) / low (Prototyping)
- Importance level: mandatory (solid line) / recommended (dotted line)

Figure 7. Design model relationships and contribution to prototyping and production development.



The functional model shall have structured algorithms that can contribute to the validation of the software requirements with modeling and simulation. A model's code generation configuration for rapid-prototyping can be useful to validate the software requirements with a real-time environment. The development focus shall be on the software requirement (not represented on the figure). The entire model shall be considered a prototype.

The architecture model shall define the component interface and scheduling of the software architectural design. The architectural design aspect of the functional model can serve as a baseline to initiate the development of the software architecture for production (1a). The prototype algorithms of the functional model can populate the architecture model to enable early dynamic verification of the model in simulation to evaluate the impact of the architecture on the functional behavior (2a).

A prototype code generation configuration representative of the software architecture standard (e.g. AUTOSAR) can be created to achieve early verification of the impact of the functional behavior in real time and its integration with software and hardware (e.g. AUTOSAR RTE).

The component design model shall fully define the software component design with its structure, scheduling, and algorithms. The interface of the model shall be consistent with, and can be reused from, the architecture model (1b). The prototype algorithms developed for the functional model can serve as a baseline to define the production algorithms (2b). A prototype code generation configuration can be used for early verification of the non-trivial impacts of the design model on the generated code (e.g. compliance with the coding standard, level of code coverage versus model coverage, code expansion).

The component implementation model shall define both the software component design and implementation. The structure, scheduling, and algorithms shall be reused from the software component design model (1c, 2c). The way algorithms are implemented can be adapted to address non-functional requirements (e.g. optimization, safety). The code generation configuration shall be used for production code generation and shall then be compatible with the software coding standard and the target hardware.

4 Design Model Quality

4.1 Overview

As design models are critical for software development using Model-Based Design, their quality must be carefully assessed. Design models can automatically transform into other design artifacts such as documentation, source code, or executables. Therefore, the quality objectives defined on the design models shall impact the models themselves as well as their derived products. A specific quality objective is defined for each type of design model to account for their specific role.

Table 1. Model Quality Objectives of design models.

Software development phase	Type of design model	Model Quality Objective
Software requirements phase	Functional model	MQO-1
Software architectural design phase	Architecture model	MQO-2
Software component design and testing phase	Component design model	MQO-3
Software component implementation and testing phase	Component implementation model	MQO-4

Table 2 provides the list of Model Quality Requirements applicable to achieve the quality objective of each type of design model.

Table 2. Overview of Model Quality Requirements of MQOs.

MQR ID	MQR Title	MQO-1	MQO-2	MQO-3	MQO-4
MQR-01	Model layout	M	M	M	M
MQR-02	Model comments	M	M	M	M
MQR-03	Model links to requirements	M	M	M	M
MQR-04	Model testing against requirements	M	R	M	M
MQR-05	Model compliance with modeling standard		M	M	M
MQR-06	Model data		M	M	M
MQR-07	Model size			M	M
MQR-08	Model complexity			M	M
MQR-09	Model coverage			M	M
MQR-10	Model robustness			M	M
MQR-11	Generated code testing against requirements			R	M
MQR-12	Generated code compliance with coding standard			R	M
MQR-13	Generated code coverage			R	M
MQR-14	Generated code robustness			R	M
MQR-15	Generated code execution time				M
MQR-16	Generated code memory footprint				M

M: Mandatory
R: Recommended for early verification

Note: An additional MQR to verify the generated source code against the model can be required in the context of DO-331.

4.2 Model Quality Requirements
This section provides further details on the MQR introduced in Table 2.

MQR-01	Model layout
Description	The model shall define Simulink and Stateflow® diagrams that are completely visible on A4 paper size.
Recommendation level	MQO-1: Mandatory; MQO-2: Mandatory; MQO-3: Mandatory; MQO-4: Mandatory
Notes	Fit to view with a zoom ratio smaller than 80% is harder to read on screen and likely not to be readable on A4 paper size. The model zoom ratio is visible at the center of the model status bar below the diagram.
References / Examples of techniques	- Simulink subsystems - Stateflow sub-charts - Simulink bus
Rationale	Printing a Simulink model can be necessary to archive or share models as documents. A model diagram that can be completely displayed on screen improves readability and eases model review. Reducing the size of the diagrams forces the model developer to better organize large model and data into hierarchical structures of buses and model references or subsystems.

MQR-02	Model comments
Description	The model comments shall provide a description of the model itself and the following types of elements: - Simulink subsystem - Simulink function and S-function mask - Stateflow chart, sub-chart, truth table, state transition table, and flowchart - Simulink and MATLAB function blocks and sub-functions
Recommendation level	MQO-1: Mandatory; MQO-2: Mandatory; MQO-3: Mandatory; MQO-4: Mandatory
Notes	A comment can include a mix of text, equations, diagrams, and pictures. A comment can be embedded in the model or a link can be established from the model to a separate and accessible document. The quality of the comments is not in the scope of this requirement and shall be assessed by peers during the model review.
References / Examples of techniques	- Insertion of blocks for documentation - Description in Simulink subsystems masks - Stateflow diagrams annotations - Comments in Simulink and MATLAB function codes

Rationale	Like code, a model without comments is harder to understand by peers. Lack of description can negatively impact the efficiency of the peer review activity and maintenance activities.

MQR-03	Model links to requirements
Description	The model elements that specify algorithms and calculations shall trace to the model higher level requirements. The design model elements that specify interface shall trace to the software interface requirements or software component interface requirements.
Recommendation level	MQO-1: Mandatory; MQO-2: Mandatory; MQO-3: Mandatory; MQO-4: Mandatory
Notes	A model element is implicitly traced to a model higher level requirements if one of its parents is traced (e.g. its parent subsystem). The model shall trace to the right level of requirements: - Functional model and architecture model shall trace to software requirements - Component design model and component implementation model shall trace to software component requirements The correctness of the links to model higher level requirements is not in the scope of this requirement and shall be assessed by peers during the model review. When model references are used inside component design and implementation models, each referenced model shall trace to its own model higher level requirements.
References / Examples of techniques	- Bidirectional links between model and requirement tool
Rationale	Traceability to requirements eases static model verification against requirements. It facilitates: - Requirement coverage analysis - Impact analysis on design following changes on requirements - Identification of unintended or useless design to be present in the model

MQR-04	Model testing against requirements
Description	The model shall produce the expected outputs when exercised by tests derived from and traced to the model higher level requirements.
Recommendation level	MQO-1: Mandatory; MQO-2: Recommended; MQO-3: Mandatory; MQO-4: Mandatory
Notes	The model tests shall be derived from and traced to all model higher level requirements which verification strategy is testing. Each test shall have a defined procedure, stimuli, and expected outputs. The model test environment shall not impact the behavior of the model under test. The correctness of the tests and links to model higher level requirements are not in the scope of this requirement and shall be assessed by peers during the tests review.
References / Examples of techniques	- Stimuli and expected outputs time series - Test sequences and test oracles - Automation of test procedure, execution, and reporting
Rationale	The simulation of the design model enables the discovery of design errors at design time.It can also contribute to refining model higher level requirements or correcting and validating requirement-based tests.

MQR-05	Model compliance with modeling standard
Description	The model shall be compliant with the modeling standard.
Recommendation level	MQO-1: ; MQO-2: Mandatory; MQO-3: Mandatory; MQO-4: Mandatory
Notes	The modeling standard shall be defined during the project software planning phase and shall be compatible with the software safety standard, software design standard, coding standard, and targeted hardware (e.g. floating-point support). Model compilation warnings and errors reported by Simulink diagnostics are considered modeling standard violations. The modeling standard could be adapted to software architectural design modeling and software component design modeling.
References / Examples of	- MathWorks modeling guidelines for high-integrity systems (Include compatibility with MISRA C® compliance)

techniques	- MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow
Rationale	The model standard can enforce best practices and define a subset of the modeling language that limits the possibility of incorrect use of the language.

MQR-06	Model data
Description	The model I/O signals, calibrations, and observable signals shall be fully defined with the following properties: • Name • Description • Design min/max • Initial value (output only) • Data type (e.g. base type, fixed-point type, enumerated type, structured type) • Size • Physical unit • Safety integrity level • Memory storage
Recommendation level	MQO-1: —; MQO-2: Mandatory; MQO-3: Mandatory; MQO-4: Mandatory
Notes	The compute method is necessary for data coming from external software, driver, or communication network. An initial value or safe value can be added for output and safety critical data. Memory storage only needs to be defined in the component implementation model. Display format for measured signal and calibration for floating point is recommended.
Examples of techniques	- Simulink data objects - Simulink data dictionary
Rationale	Model data are part of the design and need to be fully defined. For instance, incorrect or unknown data integrity level or data design min/max can impact the model and software reliability and robustness.

MQR-07	Model size
Description	The model shall have less than 500 elements including: - The number of Simulink blocks - The number of MATLAB executable lines of codes - The number of Stateflow transition, states, and connections - The number of truth tables decision
Recommendation level	MQO-1: —; MQO-2: —; MQO-3: Mandatory; MQO-4: Mandatory
Notes	The model reference block only counts as one element. The company standard utility function (e.g. Simulink library block, MATLAB function file) only counts as one element. Please refer to MathWorks guidance on large-scale modeling in Simulink documentation.
References / Examples of techniques	
Rationale	Very large models are more difficult to merge and are more likely to be modified by several users at the same time. Smaller models are more likely to be reusable and easily configurable. Generated code of very large models cannot be incrementally tested.

MQR-08	Model complexity
Description	The model and its subsystems, Stateflow charts, and MATLAB functions shall have a local cyclomatic complexity lower or equal to "30".
Recommendation level	MQO-1: —; MQO-2: —; MQO-3: Mandatory; MQO-4: Mandatory
Notes	Local complexity is the cyclomatic complexity for objects at their hierarchical level. Aggregated cyclomatic complexity is the cyclomatic complexity of an object and its descendants.

	The threshold of 30 for local cyclomatic complexity is a recommendation and can be adapted on a project basis. The number 30 for cyclomatic complexity has been derived from the HIS (Hersteller Initiative Software) code metric and adapted to Model-Based Design.
References / Examples of techniques	Cyclomatic complexity is a measure of the structural complexity of a model. It approximates the McCabe complexity measure for code generated from the model. The McCabe complexity measure is slightly higher on the generated code due to error checks that the model coverage analysis does not consider. To compute the cyclomatic complexity of an object, such as a block, chart, or state, model coverage uses the following formula: $c = \sum_{1}^{N} (o_n - 1)$ *N* is the number of decision points that the object represents and o_n is the number of outcomes for the *n*th decision point. The tool adds one to the complexity number for atomic subsystems and Stateflow charts.
Rationale	Cyclomatic complexity is a leading testability metric. Test harness can be created for simulation at model, subsystem, chart, and MATLAB function level.

MQR-09	Model coverage
Description	The model structure shall be fully covered by the test suite that is derived from and traced to the model higher level requirements.
Recommendation level	MQO-1: — MQO-2: — MQO-3: Mandatory MQO-4: Mandatory
Notes	The structural coverage criteria chosen shall be at least conformant to the structural coverage criteria imposed by the software safety integrity level.
References / Examples of techniques	Types of coverage analysis available on Simulink model: - Execution Coverage (EC) - Decision Coverage (DC) - Condition Coverage (CC) - Modified Condition/Decision Coverage (MCDC) EC, DC, CC, MCDC, saturation on integer overflow coverage, and relational boundary coverage can be used to measure the model structural coverage.
Rationale	Model coverage enables to identify untested design, untestable design, or unintended design.

MQR-10	Model robustness
Description	The model shall be robust in normal and abnormal operating conditions.
Recommendation level	MQO-1: — MQO-2: — MQO-3: Mandatory MQO-4: Mandatory
Notes	In normal operating condition, inputs and tunable parameters values are within their design ranges. In abnormal operating condition, inputs, and tunable parameters values are outside their design ranges. Robustness shall prevent errors such as: - Divisions by zero - Integer overflows - Out of design range - Out of bound array The level of robustness shall be compliant with the software safety integrity level.
References / Examples of techniques	- Test generation based on relational boundary coverage - Formally-based verification technique with abstract interpretation - Defensive programming
Rationale	Model robustness verification prevents edge case or incorrect use of model, which can cause unexpected results or simulation errors.

MQR-11	Generated code testing against requirements
Description	The model generated code shall produce the expected outputs when exercised by tests derived from and traced to the model higher level requirements
Recommendation level	MQO-1: — MQO-2: — MQO-3: Recommended MQO-4: Mandatory

Notes	For MQO-03, tests can be run in software-in-the-loop. For MQO-04, tests shall be run in processor-in-the-loop. A representative hardware or an emulator can be used in place of the actual processor.
References / Examples of Techniques	- Test reuse from component design model testing - Test generation for back-to-back testing
Rationale	Code testing is required to verify the output of the code generator and compiler or cross-compiler, linker, load, and flash utilities. For MQO-3, code testing in software-in-the-loop increases confidence in the code generator.

MQR-12	Generated code standard compliance
Description	The generated code shall be compliant with the coding standard.
Recommendation level	MQO-1: — MQO-2: — MQO-3: Recommended MQO-4: Mandatory
Notes	The coding standard shall be defined during the project software planning phase and shall be compatible with the software safety standard, software architecture standard, and targeted hardware (e.g. floating-point support). The modeling standard shall anticipate the compliance with the coding standard. The project coding standard can be tailored for generated code.
References / Examples of techniques	- MISRA C 2012 for safety - CERT C for cyber security
Rationale	Coding standard verification is required to verify the output of the code generator.

MQR-13	Generated code coverage
Description	The model generated code structure shall be fully covered by all the tests that are derived from and traced to the model higher level requirements.
Recommendation level	MQO-1: — MQO-2: — MQO-3: Recommended MQO-4: Mandatory
Notes	The structural coverage criteria shall be at least conformant to the structure coverage criteria imposed by the software safety integrity level. The model tests shall be reused to cover the structure of the generated code. The code coverage can be different than the model coverage depending on the blocks used (e.g. look-up table interpolation algorithm) or code generation optimization options (e.g. for loop unrolling).
References / Examples of techniques	Types of coverage analysis available on the generated code: - Statement Coverage for Code Coverage - Condition Coverage for Code Coverage - Decision Coverage for Code Coverage - Modified Condition/Decision Coverage (MCDC) for Code Coverage
Rationale	Code coverage is required in addition to model coverage to verify that the code generator do not add unintended functionalities.

MQR-14	Generated code robustness
Description	The model generated code shall be robust in normal and abnormal operating conditions.
Recommendation level	MQO-1: — MQO-2: — MQO-3: Recommended MQO-4: Mandatory
Notes	In normal operating condition, inputs and tunable parameter values are within their design ranges. In abnormal operating condition, inputs and tunable parameter values are outside their design ranges. Robustness shall prevent errors such as: - Divisions by zero - Integer overflows - Out of design range - Out of bound array The level of robustness shall be compliant with the software safety integrity level.
References / Examples of techniques	- Test generation based on relational boundary coverage - Formally-based verification technique with abstract interpretation - Defensive programming
Rationale	Code robustness verification is required to verify the output of the code generator

MQR-15	Generated code execution time
Description	The model generated code running on the production target shall be instrumented to measure and verify the execution time.
Recommendation level	MQO-1: — | MQO-2: — | MQO-3: — | MQO-4: Mandatory
Notes	Worst case execution time shall be specified during software architectural design phase. The execution time shall include the generated code and its calling functions (e.g. basic software services). The production target can be an emulator or a representative hardware. The model tests can be reused on the generated code running on the production target (aka processor-in-the-loop) and the expected outputs shall still be obtained.
References / Examples of techniques	- Profiling in processor-in-the-loop from Simulink
Rationale	The component software execution time shall be measured prior the component integration to verify compatibility with architecture requirements, avoid shortage of hardware resource, and enable reuse of component on different architecture.

MQR-16	Generated code memory footprint
Description	The model generated code memory footprint shall be measured and verified.
Recommendation level	MQO-1: — | MQO-2: — | MQO-3: — | MQO-4: Mandatory
Notes	Memory footprint, such as RAM, ROM, and stack, shall be specified during software architectural design phase. The memory footprint shall include the generated code and its calling functions.
References / Examples of techniques	- Stack estimation tool
Rationale	The component software memory footprint shall be measured prior the component integration to verify compatibility with architecture requirements, avoid shortage of hardware resource, and enable reuse of component on different architecture.

5 Conclusion

This paper clarifies how Simulink design models contribute to accelerate development and verification activities from software requirements specification to software implementation. Four types of design models with specific purposes have been introduced, each with a specific quality objective to control their proper usage. Each quality objective is a set of measurable metrics with quantified satisfaction criteria in order to facilitate and standardize model quality assessment.

The organizations that apply the concepts presented in this paper should experience the following benefits:

a) Shared understanding of Model-Based Design within the organization
b) Application of a quality model adapted to Model-Based Design projects and compatible with industry software quality and safety standards
c) Assessment of model quality at different phases of projects

The organizations that also collaborate with partners to execute Model-Based Design projects should experience the following benefits when applying the concepts presented in this paper:

a) Clear split of responsibility between parties at the beginning of projects
b) Common understanding of model quality
c) Common expectation on model quality when sharing models

6 References

[1] Patrick Briand (Valeo), Martin Brochet (MathWorks), Thierry Cambois (PSA Peugeot Citroën), Emmanuel Coutenceau (Valeo), Olivier Guetta (Renault SAS), Daniel Mainberte (PSA Peugeot Citroën), Frederic Mondot (Renault SAS), Patrick Munier (MathWorks), Loic Noury (MathWorks), Philippe Spozio (Renault SAS), Frederic Retailleau (Delphi Diesel System), Software Quality Objectives for Source Code, ERTS 2010-Conference, 2010.

[2] S. Louvet, Robert Bosch (France) SAS, Dr. U. Niebling, Dr. M. Tanimou, Robert Bosch GmbH Model Sharing to leverage customer cooperation in the ECU software development; Toulouse, ERTS 2014-Conference, 2014.

[3] ISO - International Organization for Standardization. ISO 26262 Road vehicles Functional Safety Part 1-10, 2011.

[4] RTCA/Eurocae, Software Considerations in Airborne Systems and Equipment Certification, RTCA DO-331 / Eurocae ED-218, December 13, 2011.

[5] Automotive SPICE Process Assessment 3.0 / Reference Model from VDA QMC Working Group 13 / Automotive SIG

[6] Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems, EN 50128:2011