embeddedworld2020
Exhibition&Conference
... it's a smarter world

Planning Software Architecture and Modeling Patterns for ISO 26262 Compliance

Tjorben Groß[1], Jason Moore[2], John Lee[2]
1: The MathWorks GmbH, Aachen, Germany
2: The MathWorks Inc, Natick (MA), USA

***Abstract*—The ISO 26262 standard for functional safety provides high-level guidance on the development of automotive electronics and electrical systems, including embedded software. At MathWorks we have observed that the room for interpretation of the standard can be both useful and burdensome. In this work we will contribute to closing this gap. We will provide guidance for Model-Based Design development projects, particularly how to address challenges such as freedom from interference, testability at different software architecture levels, and reusability. These best practices will be addressed by means of modeling patterns and will lay the foundation for more streamlined software verification.**

***Keywords*—ISO 26262; functional safety; automotive; modeling; architecture; Model-Based Design;**

I. INTRODUCTION

This paper highlights best practices around developing ISO 26262 compliant software. These will be illustrated with Simulink models for architecture and design. Note that we will focus on leveraging concepts of Model-based Design rather than introducing Model-based Design itself.

For software development in the context of ISO 26262:2018-6 [1] OEMs and suppliers need to be able to provide documentation that the system has been developed, verified and tested according to state of the art methodologies. It is crucial to think through possible design choices for the algorithm's architecture as early as possible because these choices can have a substantial impact on the efficiency of the development organization, reusability of the software, and testability of the software.

MathWorks provides guidance documents, as part of an IEC certification kit, which can be used to demonstrate compliance with the portions of the ISO 26262 standard that are applicable to Model-Based Design. The kit provides a high-level workflow that can be referenced, tailored, and extended as needed, as well as support for tool qualification [2], thus addressing ISO 26262:2018-8 [3] and helping to reduce efforts for tool qualification efforts significantly.

By adopting an appropriate modeling style and architecture it is possible to greatly reduce the work needed to meet key aspects of the standard such as freedom from interference. MathWorks Consulting Services has developed best practices during consulting engagements with various automotive companies. Throughout this paper, we provide suggestions on modeling practices that can be used to segment algorithms to reduce verification and deployment efforts when adhering to ISO 26262-6 and using Model-Based Design. These best practices can be classified into the following categories that match with the next three sections: model architecture in section 2, signal routing and definition in section 3, and code generation configuration in section 4. Finally, section 5 provides a summary and future directions.

II. MODEL ARCHITECTURE

One of the first decisions during algorithm development in Simulink involves the general model architecture. Decisions at this stage affect software testability, software reusability, unit and integration testing methods, ease of software integration, and software segmentation for freedom from interference.

The best practices center around areas that make ISO 26262 compliance easier while increasing efficiency in the validation, verification, and documentation phases.

A. Using Model Reference for Unit-Level Models

One of the main focal points in part 6 of ISO 26262 is the workflow for developing, validating, and verifying software units. Software units are intended to be the smallest testable parts of an application that must be individually tested for proper operation. To maintain traceability, requirements will be mapped to and/or within these individual software units, and requirements-based test cases will be built to extensively test the software unit. Therefore, the modeling construct used for unit development needs to consider various aspects, including testability, code generation, complexity, testing workflow, traceability, and documentation. These aspects point to reliance on model references as the primary modeling pattern for unit

development. Model references have multiple characteristics that are conducive to the desired outcome including:

- Code generation – There is a one-to-one mapping from model reference to source files generated.
- Reusability – Model references can be used in multiple places throughout an integration model.
- Testability – Model references are ideal for test harness construction.
- Team collaboration – Model references enable parallel development across developers and simplify the overall configuration management and version control processes.

B. Strategy for Grouping Units into Features

When integrating units and grouping them to features, one can choose from various types of modeling constructs to add model hierarchy such as virtual subsystems, atomic subsystems, model references, and library blocks.

For ISO 26262, there is flexibility on how these unit models can be grouped under their respective ASIL. The recommendation from the previous section added the constraint that model reference should be used for the unit-level algorithm. However, for grouping of units, either subsystems or model reference can be used. Some of the tradeoffs to consider are the number of model references that need to be managed and how firm a modeling boundary is required at a feature level. The following items need to be considered when determining a feature model segmentation strategy:

- Generated code file and functionality grouping
- Parallel feature development by multiple developers
- The overhead associated with model reference files
- The ease of visibility for feature grouping in the design

At this intermediate level of the model between unit and the top level, the choice of modeling constructs is up to the organization's modeling guidelines and preferences.

C. Spliting ASIL and QM Levels at the Top Level of the Model

One key concept in ISO 26262 centers around freedom from interference. ISO 26262 has five distinct safety levels (Quality Management (QM), and ASIL A–D) that can be used to classify system- and software-level functionality based on the functional safety aspects of the system. Electrical and electronic systems that are following ISO 26262 may have components at different ASILs. For example, a component that reports out non-critical diagnostic data may be classified as a QM component, whereas a component that could impact the vehicle's ability to brake, may be classified to a higher-level ASIL component due to the high degree of hazard/injury risk if a failure occurs.

A system with multiple ASIL components will benefit from an architecture that efficiently segments these algorithms into separate containers. The benefit will be seen for two reasons: 1. Each ASIL can have different development, validation, and verification requirements. 2. Separating and segmenting ASILs enables freedom from interference.

Since the various ASILs will be split, we recommend choosing a modeling construct to aid in that segmentation. Using model references ensures that when the algorithm is deployed, there will be a firm boundary at each components' border. Therefore, we would split the top level of the system-level model into multiple model references with each model reference representing a separate component with different ASIL. Note that in this case, due to code generation configuration settings (see the Code Generation Configuration section for detail), the system-level model is for simulation only, and code generation can only be done separately based on ASIL. Generating code at the unit level and then integrating it is another option. However, generating code at as high a level as possible reduces the amount of overhead during code integration.

By using a model reference for each ASIL or even more generically, whenever freedom from interference is needed, separate functions and source files will be generated for each model reference. By following this and the code generation configuration best practice, each segmented partition will have its own source files, shared utilities, and data definitions, which will make it easier to achieve freedom from interference between the different sections of the algorithm. Figure 1 demonstrates how a model could hierarchically be split based on ASIL.

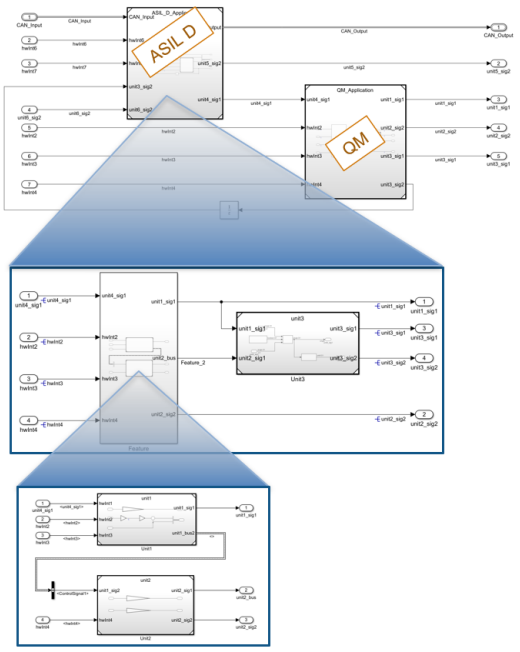

Fig 1: Model hierarchy based on ASIL.

D. Eliminating Algorithmic Content at the Integration Level

ISO 26262 has the notion of multiple testing levels in the representative architecture, including unit level, integration level, and system level. Typically, a software unit must go through various levels of testing rigor based on the targeted ASIL. For example, ASIL D may require full modified

condition and decision coverage (MCDC), whereas for ASIL A or B, condition and decision coverage may be acceptable. Because of this, units are the only place where algorithmic functionality should be implemented. If algorithmic content occurs at the integration or system level of the model, it can be more difficult to achieve full coverage of the design. For this reason, it is recommended to ensure that no algorithm content occurs outside of unit-level models (see Figure 2).

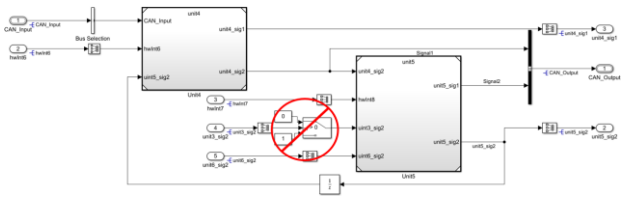

Fig 2: Avoiding algorithm content outside of the unit.

E. Using Model Metrics to Monitor Unit Complexity

Many organizations realize late in the development cycle that their algorithms will be difficult to validate to the level of coverage that ISO 26262 recommends. This typically stems from the lack of architectural consideration at design time and management of unit-level size and complexity during development. One methodology that can alleviate this issue is to set unit size metric thresholds for the entire development organization. These thresholds can include maximum number of inputs and outputs per unit, reusable libraries, cyclomatic complexity or number of elements. To manage the unit's complexity, these metrics should be reviewed during the model review process of the development cycle. This will enable visibility of the complexity and scope of how the algorithm has been designed during the development process. A model Metrics Dashboard, as included in Simulink Check can make this process easier. This dashboard provides metrics for total block count, MATLAB lines of code (LOC), Stateflow LOC, model complexity, number of blocks, and other metrics to the modelers and model reviewers. The dashboard supports the design workflow by continuously monitoring models to ensure that they are not being divided into units that are too small (increasing management complexity) or too large (unable to be easily tested and difficult to reuse).

Threshold values are often an area of debate. This is because the model complexity usually cannot be measured from a single aspect of the model. It is often necessary to analyze multiple metrics to make meaningful decisions. For example, one Stateflow chart can have very high cyclomatic complexity while its block count is only one. The recently published paper Model Quality Objectives for Embedded Software Development with MATLAB and Simulink [2], from MathWorks and a group of automotive companies (Delphi Technologies, Bosch, PSA, Renault, and Valeo) provides guidance on metrics and thresholds. For example, model cyclomatic complexity should be 30 or lower, and the number of model elements should be less than 500. Specific threshold values may be subject of discussions among OEM and suppliers.

III. Interface Definition and Data Exchange

Part 6 of ISO 26262:2018 has multiple considerations that address interface complexity and data exchange between units and components. For example:

- Table 3 - 1b: Restricted size and complexity of software components
- Table 3 - 1c: Restricted size of interfaces
- Table 7 - 1g: Data flow analysis
- Table 7 - 1k: Interface tests

To fulfil these requirements and to also address freedom from interference, it is important to determine architectural strategies for how data will be exchanged between units, components, and different ASILs. This section provides four best practices aimed at managing interface complexity and data exchange.

A. Grouping Bus Signals by ASIL, Feature, and Rate

ISO 26262 Part 6 Table 7 - 1g recommends that development teams perform data flow analysis. Data flow analysis is necessary to understand how signals are passed through a software algorithm and down to the unit level. This type of analysis can spot areas where signal requirements are contradictive or where various signals should not be directly used based on characteristics of the provider. To perform this analysis at the model level, a bus hierarchy strategy must be developed to make it easier to understand where the signal is coming from and what the characteristics of the signal are. Not specifying how bus signals should be grouped hierarchically, can result in the following issues:

- Inefficient bus segmentation
- Inconsistent bus grouping from developer to developer
- Modeling difficulty when splitting and recreating buses
- Inefficient code generation

Just as model architecture requires a top-down design approach, the same is true for bus hierarchy. To better manage the ASIL, signals should be grouped based on task rate and ASIL in the bus hierarchy (see Figure 3). By grouping these signals based on ASIL, it will be easier to determine the provider of a signal and determine if the signal is being used in a unit of a higher ASIL.

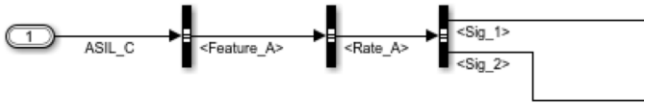

Fig 3: Grouping signals in a bus hierarchy.

B. *Passing Only Necessary Signals to Units*

Table 3 and Table 7 of ISO 26262 Part 6 have important suggestions on unit-level interfaces and data exchange. These two tables mention that the size and complexity of software components and interfaces should be reduced where possible. Also, during the verification process, users should perform interface tests. If unused variables are passed down to a unit-level model, additional testing will be necessary to ensure that these signals do not have an impact on the unit. To alleviate this concern, reducing the inputs that are passed to the unit level can help. Splitting bus signals prior to entering a unit-level model facilitates this (see Figure 4).

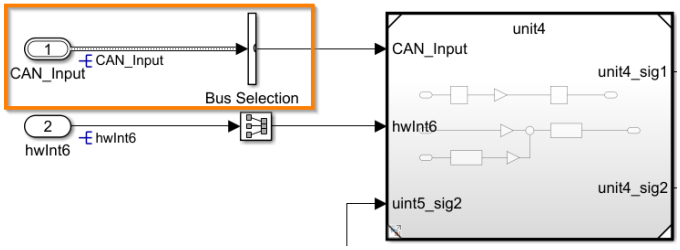

Fig 4: Splitting bus signals entering a unit-level model.

C. *Placement of Signal and Parameter Objects*

The high-level use case for signal and parameter objects is to define the interface between the model and base software. In such a use case, parameter objects are typically used for specifying calibration values and placed inside model blocks such as Gain and Lookup Table blocks. The usage of signal objects is more complex, but it usually is associated either with internal signals to support calibration activities or with root input and output ports. It is important to note that the interface to the model reference block does not contain signal objects. This is because signals at the boundary of the unit level are considered internal interface signals. This also assumes that the code is being generated at the highest ASIL partition as mentioned in section II.C. For the internal interface signals, it is best to let the code generator, e.g. Embedded Coder define those signals based on its internal optimization algorithms. Data type information, however, does need to be explicitly defined as part of the Port block configuration at model reference boundaries, as shown in Figure 5.

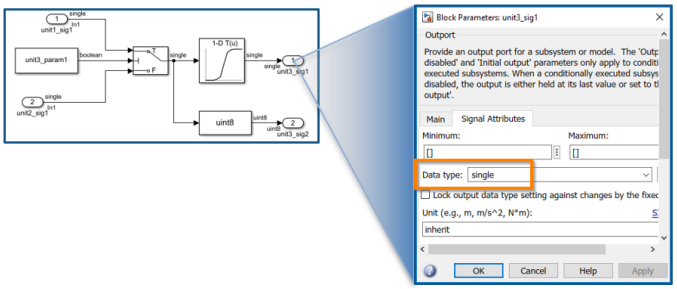

Fig 5: Defining data type for a port.

This best practice restricts the placement of interface signal objects to the code generation model boundaries. It also ensures that when code is generated for each ASIL, the root-level output ports will point to the corresponding interface function in the other ASIL section.

The storage class used for the signal and parameter objects can be set based on the software architecture and coding practices. The exception will be the protection needed for data exchange between ASIL models or partitioning as required by freedom from interference.

D. *Data Exchanged Protection Between ASILs*

One consideration when code for each ASIL is generated separately is that a protection method is needed to exchange data between each ASIL. Multiple strategies exist using storage classes on the ASIL sections' root-level input and output ports. One prevalent method is to use a storage class that has get and set access functions for the data. By using get and set access functions, it is possible to add more protection to these interfaces so that only appropriate software components are accessing data (see Figure 6 and Figure 7).

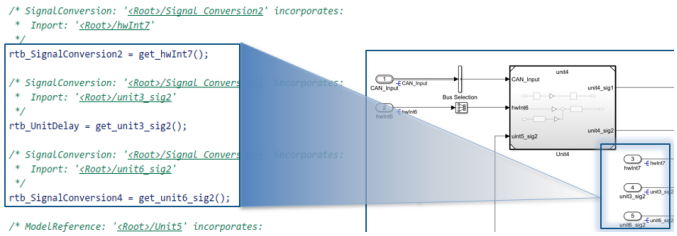

Fig 6: GetSet storage classes used on the root-level input ports and the corresponding generated code.

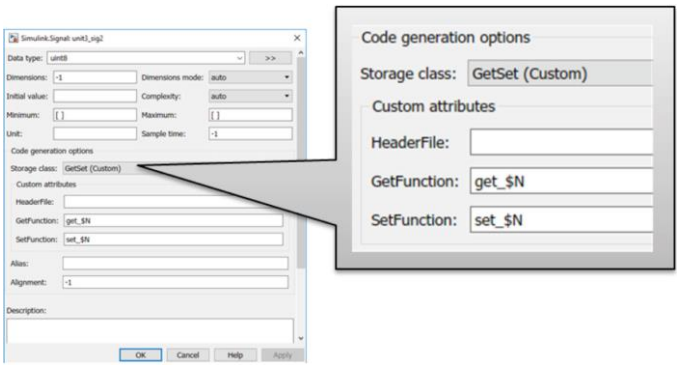

Fig 7: Storage class configuration ensuring that the Get and Set APIs match between ASILs.

It is important to note that the GetSet storage class objects only provide an entry point for the interface protection. The actual implementation of the protection is typically done through low-

level software layers implemented by hand coding, which can be easily integrated using the GetSet storage class.

IV. Code Generation Configuration

Once the model has been developed according to the best practices listed above, there are still code generation configuration settings that need special attention to achieve objectives set forth by ISO 26262. This section provides best practices for code generation configuration setting with respect to code placement and separation of shared utility files. Again, the configuration listed below assumes the previous best practices have been followed.

A. Code Placement Strategy

Freedom from interference is necessary to ensure that if there is an issue with one section of the system at an ASIL, it will not impact functionality at a separate ASIL. For example, if a QM or ASIL A component has an issue or a failure occurs, the design should segment this functionality away from functionality that is ASIL D to ensure that the ASIL D functionality can continue operating. In embedded systems, one type of fault that is a concern is if portions of the application have access to sections of memory or functions that they should not have access to. One way to address this concern is to separate the functionality into distinct memory sections or cores of a microprocessor. This can be done by telling the compiler into which section each variable, function, or file should be placed. Figure 8 demonstrates how the application could be separated.

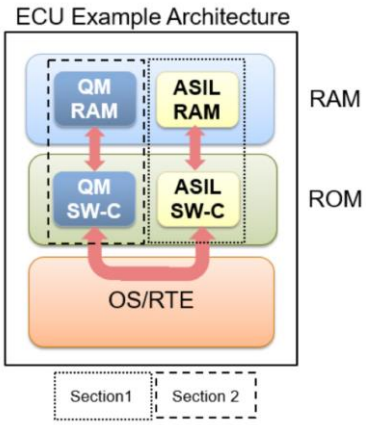

Fig 8: Segmenting the system architecture to ensure freedom from interference.

Another technique is to split various ASIL and QM levels into separate memory sections. Splitting the memory sections alleviates some concerns with various ASILs unintentionally interacting with each other, such as a function within the QM software component (SW-C) writing to a protected ASIL RAM location. To configure this, memory sections can be selected in the configuration options as shown in Figure 9. This method does not have to be used, but it does simplify the overall workflow.

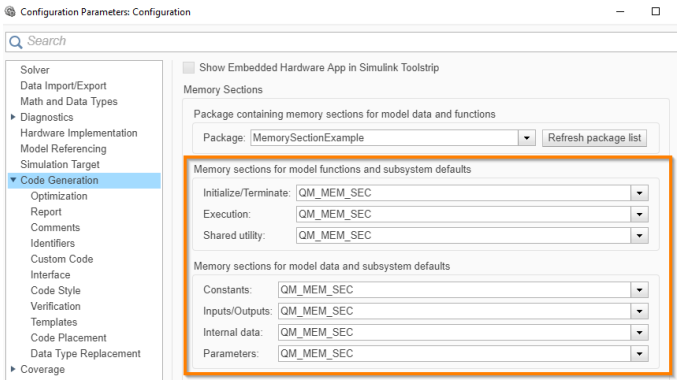

Fig 9. Configuring parameter settings for memory sections.

B. Different Name Tokens for Shared Utilities

Embedded Coder generates common utility files known as shared utilities. These files are basic functions that are used across the code generation model. This method presents an issue in a safety-related application because there will not be any distinctions between the sources of the shared utility files. To compile code into a signal application with the segmentation concept, shared utilities must be generated and allocated based on their ASIL. This can be done the through the shared utilities identifier format option in the code generation configuration (see Figure 10).

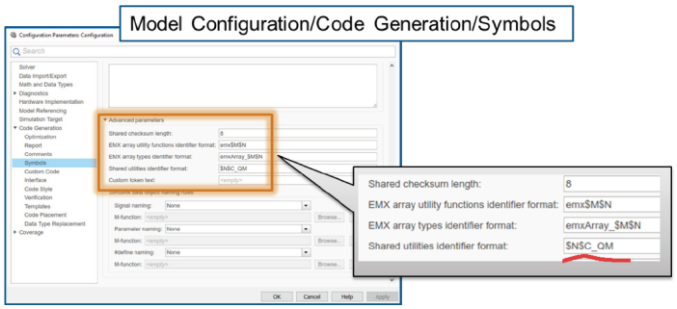

Fig 10: Configuring shared utility settings.

With the above settings, one can create the generated shared utility with a unique identifier that is a function of the ASIL. For example, the shared utility for the Lookup Table block when configured based on the above QM suffix is shown in Figure 11.

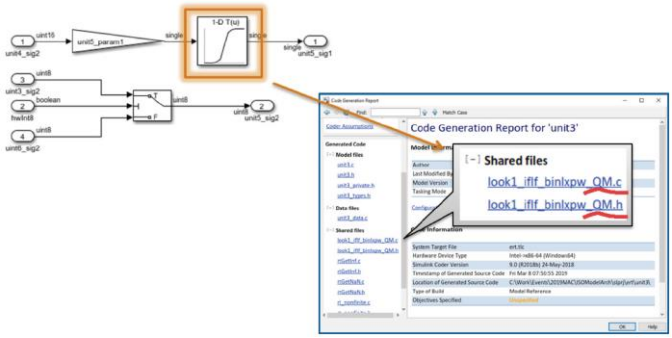

Fig 11: Shared utility configured for QM.

V. SUMMARY AND FUTURE WORK

The findings presented in this paper are best practices created through multiple MathWorks consulting engagements. These best practices are proven enablers to adoption of ISO 26262. However, following these best practices does not guarantee ISO 26262 compliance because they address a subset of all ISO 26262 requirements, and each application has unique needs.

Future work is underway in applying the above best practices for AUTOSAR standards [5,6]. Fortunately, AUTOSAR facilitates concepts presented in this work so that many or most of the topics discussed in the previous sections can be mapped directly onto AUTOSAR concepts. For addressing the system-level requirements of ISO 26262, System Composer [7] can be leveraged for modeling and simulation and to help users meeting requirements of ISO 26262:2018-4 [8] and ISO 26262:2018-9 [9].

REFERENCES

[1] ISO: ISO 26262-6, Road vehicles — Functional safety — Part 6: Product development at the software level, December 2018.

[2] Conrad, M., Sandmann, G., Munier, P.: Software Tool Qualification According to ISO 26262. SAE 2011 World Congress, Detroit, MI, USA, April 2011. SAE Techn. Paper #2011-01-1005

[3] ISO: ISO 26262-8, Road vehicles — Functional safety — Part 8: Supporting processes, December 2018.

[4] Jérôme Bouquet, Stéphane Faure, Florent Fève, Matthieu Foucault, Ursula Garcia, et al.. Model Quality Objectives for embedded software development with MATLAB and Simulink. 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Jan 2018, Toulouse, France.

[5] AUTOSAR, www.autosar.org. Automotive Open System Architecture, 2019.

[6] AUTOSAR, www.autosar.org. AUTOSAR Adaptive Platform for Connected and Autonomous Vehicles, 2019.

[7] System Composer, www.mathworks.com/products/system-composer.html, 2019

[8] ISO: ISO 26262-4, Road vehicles — Functional safety — Part 4: Product development at the system level, December 2018.

[9] ISO: ISO 26262-9, Road vehicles — Functional safety — Part 9: Automotive safety integrity level (ASIL)-oriented and safety-oriented analyses, December 2018.