embeddedworld2021 Exhibition&Conference ...it's a smarter world DIGITAL

A showcase for model based code generation for multicore safety systems

Prof. Dr.-Ing. Peter Fromm
Department of Electrical Engineering
Darmstadt University of Applied Sciences
Darmstadt, Germany
peter.fromm@h-da.de

Thomas Barth
Department of Electrical Engineering
University of Applied Sciences
Darmstadt, Germany
thomas.barth@h-da.de

***Abstract*— Most safety standards recommend the application of formal or semi-formal modelling standards for the architectural phase. Although many modelling standards do exist, ranging from the widely used UML/SYSML to domain specific notations like AUTOSAR or tool specific solutions like Matlab/Simulink, the systematic use still lacks in many projects, especially in combination with code generation.**

- **Existing modelling notations often do not address safety relevant resource modelling (memory, CPU, peripherals).**
- **Code generation is limited to only parts of the system (e.g. classes, individual algorithms or RTE's).**
- **Especially domain specific solutions often require specific RTOS, MCALS or toolchains, which might not be the perfect choice for a specific project.**

In this paper, we will present the results of the ZIM project "Safe and Secure". Based on the SYSML block diagram, a signal flow oriented modelling notation has been developed. We will demonstrate how to model a safety control application and how we can generate a complete project framework consisting of communication objects, triggers, tasks, linker description files and MPU configurations. All artefacts can be generated for different controllers (e.g. Multicore AURIX and ARM Cortex), operating systems (e.g. PXROS, FREERTOS MPU, AUTOSAR) and programming languages (C/C++) using a flexible template based approach.

Keywords — Safety Standards, Modelling, Multicore, Code Generation, Runtime Environments

I. INTRODUCTION

Modelling software architectures is state of the art in engineering projects since the millennium and a mandatory requirement in most safety standards. The automotive standard 26262 part 6 e.g. requires

"a) to develop a software architectural design that satisfies the software safety requirements and the other software requirements;

b) to verify that the software architectural design is suitable to satisfy the software safety requirements with the required ASIL; and

c) to support the implementation and verification of the software. " [1]

Table 2 of the same standard recommends the following methods for the creation of the software design.

Table 2 — Notations for software architectural design

	Notations	ASIL A	ASIL B	ASIL C	ASIL D
1a	Natural language[a]	++	++	++	++
1b	Informal notations	++	++	+	+
1c	Semi-formal notations[b]	+	+	++	++
1d	Formal notations	+	+	+	+

[a] Natural language can complement the use of notations for example where some topics are more readily expressed in natural language or providing explanation and rationale for decisions captured in the notation.

[b] Semi-formal notations can include pseudocode or modelling with UML®, SysML®, Simulink® or Stateflow®.

NOTE UML®, SysML®, Simulink® and Stateflow® are examples of suitable products available commercially. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of these products.

Figure 1 – Recommended notations for software design [1]

Interesting enough, semi-formal methods like the UML and SysML in combination with natural language are considered good enough even for the highest ASIL level in the ISO26262, the IES 61508 requires formal methods for SIL 4.

"Formal methods provide a means of developing a description of a system at some stage in its specification or design. The resulting description takes a mathematical form and can be subjected to mathematical analysis to detect various classes of inconsistency or incorrectness. Moreover, the description can in some cases be analyzed by a machine with a rigor similar to the syntax checking of a source program by a compiler, or animated to display various aspects of the behavior of the system described." [2]

The above description already describes one fundamental problem of formal methods – they are complex in handling and

due to the amount of details which must be added to the model in order to fulfil the requirements for the code generation not very efficient. Even worse – due to the complexity and high level of detail, additional and hard to find bugs may be introduced into the system.

Table A.1 – Software safety requirements specification

(See 7.2)

Technique/Measure *		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1a	Semi-formal methods	Table B.7	R	R	HR	HR
1b	Formal methods	B.2.2, C.2.4	---	R	R	HR
2	Forward traceability between the system safety requirements and the software safety requirements	C.2.11	R	R	HR	HR
3	Backward traceability between the safety requirements and the perceived safety needs	C.2.11	R	R	HR	HR
4	Computer-aided specification tools to support appropriate techniques/measures above	B.2.4	R	R	HR	HR
NOTE 1 The software safety requirements specification will always require a description of the problem in natural language and any necessary mathematical notation that reflects the application.						
NOTE 2 The table reflects additional requirements for specifying the software safety requirements clearly and precisely.						
NOTE 3 See Table C.1.						
NOTE 4 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.						
* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. It is intended the only one of the alternate or equivalent techniques/measures should be satisfied. The choice of alternative technique should be justified in accordance with the properties, given in Annex C, desirable in the particular application.						

Figure 2 – Recommended methods IEC 61508 [3]

Semi-formal methods like the UML or SysML are way easier to be used and the developer has sufficient freedom to choose the appropriate abstraction level. But especially the behavioral diagrams lack the ability for a system level code generation, especially considering the specific requirements for multithreading or even multicore architecture.

Considering these constraints, one goal of the publicly funded and recently successfully finished ZIM project "Safe and Secure" has been the design of a domain specific notation for safety control applications. The modelling notation thereby shall fulfil the following key requirements:

R1	The modelling notation shall be based on existing, well-known standards and shall provide good usability.
R2	Based on the model, efficient C/C++ code shall be generated, which can easily be understood and qualified. The code shall run on different controllers and RTOS.
R3	The model shall support to (semi) automatically assess the system safety integrity (e.g. freedom from interference, event chains,…).

Table 1 – System requirements

II. FINDING THE BEST MODELLING NOTATION

Most modelling notations like the UML and SysML provide different views for describing the static and dynamic behavior of software systems.

Static diagrams like the well-known class diagram can easily be translated into code and code can be translated back into a diagram format. When it comes to the behavioral diagrams, things are not that easy. The activity diagram provides a good overview e.g. for algorithm design, but the "gab" between design and code is too large to allow good code generation. The state diagram is better suited, but limited to state like structure. The sequence diagram can be used to describe sequential call chains but fails when it comes to multithreading or asynchronous implementations. New diagrams like the UML 2.0 timing diagram introduce a first view on the temporal behavior, but is rather hard to understand on focusses more on a component view rather than a system perspective.

Furthermore, the UML puts a strong emphasis on structural artifacts like classes and methods, whereas communication patterns showing the data flows (e.g. using global data pools or messaging services provided by operating systems) are missing.

Especially for embedded control oriented systems, which typically consist of sensors, logic units and actor interfaces, a data flow oriented perspective has proven to be an extremely helpful modelling notation, not only for providing a good overview of the system behavior, but also to model safety functions. Concepts like AUTOSAR therefore extended the idea of the Real Time Object Oriented Modeling (ROOM) [4] concept introduced at the end of the last millennium especially for embedded systems. The ROOM and AUTOSAR notation put a stronger focus on the data flow by introducing ports, data interfaces, signals and triggers. One limitation of ROOM however has been the strong focus on state machine, which has been the main pattern for the code generation inside functions.

AUTOSAR has overcome this constraint by reducing the code generation to a system perspective: Instead of aiming to generate a complete software system, only software code files, function bodies and global data objects representing the signal flow and a runtime environment based on the AUTOSAR OS calling the functions are generated. The code inside the functions is written using the best possible approach for the individual function, e.g. state machines, control logic or filters using MATLAB Simulink or manual coding – the only requirement is to use the AUTOSAR API for accessing RTE data objects, also referred to as the virtual function bus.

The general concept of AUTOSAR is definitely a good choice for designing embedded architectures, also beyond the scope of automotive application. AUTOSAR and AUTOSAR OS however never have been designed for safety or multicore applications. Although features for these domains have been added over the last releases, the historical background of the solution remains visible. Further disadvantages especially for non-automotive small and medium sized companies are the complex and expensive toolchains, the tight coupling with the AUTOSAR OS and the limitation to automotive microcontrollers.

To overcome these limitations, the good ideas of ROOM and AUTOSAR have been combined with

- a simple to use modelling notation, based on the SysML block diagram
- extended properties for safety applications like configurable memory regions
- a flexible code generator based on templates, supporting different RTOS and controller architectures

- and last but not least a model analyzer focusing on safety features

For modelling the system, different views are being provided.

- Requirement view, supporting hazard and risk analysis, definition of hazardous events and safety goals / safety requirements. The SIL / ASIL required is defined on this level.
- Functional view, providing a high level architecture based on the requirements. Functional units later-on are mapped to hardware and software components, providing the SIL/ASIL achieved.
- Hardware view, describing the hardware components like sensors, ECU and actors of the system as well as their interfaces and internal components such as peripherals or memory.
- Software Overview, providing a high-level overview of all software components and their interfaces
- Software detailed view, which describes for every software component the runnables, their triggers and the runnable data interfaces.

The following diagram shows a functional model of a safety engine controller. The safety requirements, which have been identified, are mapped to functional unit for traceability.

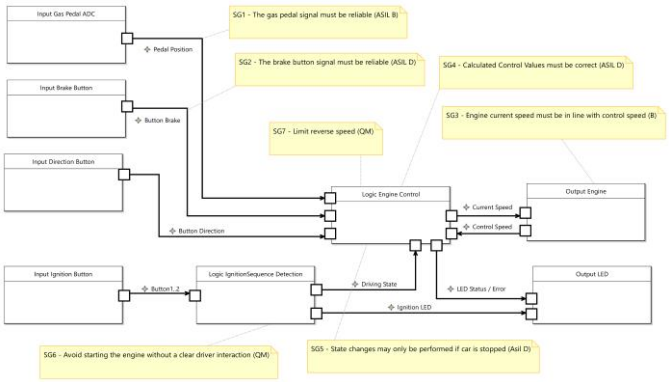

Figure 3 – Functional Model

The next diagram illustrates the hardware structure of the system. The input pedal (green) as well as the ECU (blue) have been implemented in a redundant fashion in order to have a sufficient diagnostic coverage. An external watchdog as well as a safety switch off logic has been added for a fail-safe error state.

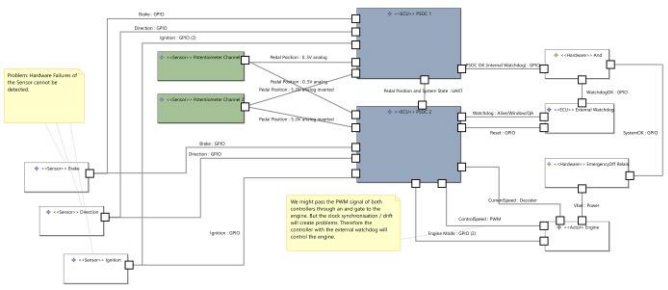

Figure 4 – Hardware View

For every controller, the software architecture is described as a high-level signal flow diagram. The diagram contains the software components and the signals exchanged between them.

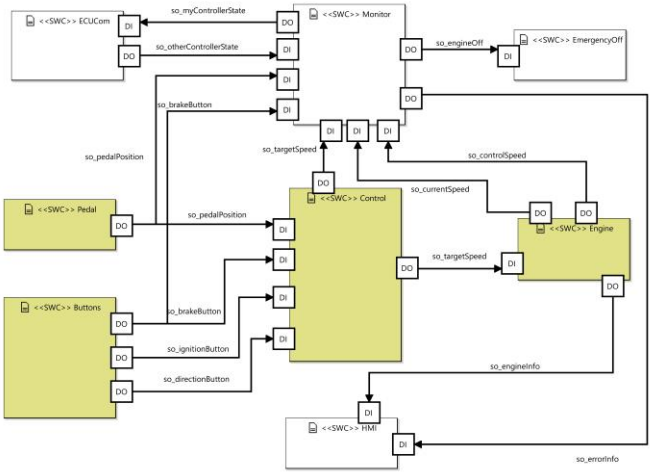

Figure 5 – Software Overview

For every software component, a detailed view is provided, which describes the runnables, the runnable triggers as well as the data ports of the runnables. The detailed design of the pedal component illustrates the runnable PEDAL_readCheckADC_run, which will read in both channels of the pedal sensor, checks the data for validity and will store the correct data in the signal so_pedalPosition. This signal now can be used by other runnables for further processing. In case of invalid data, the error handler runnable of the pedal component will be called.

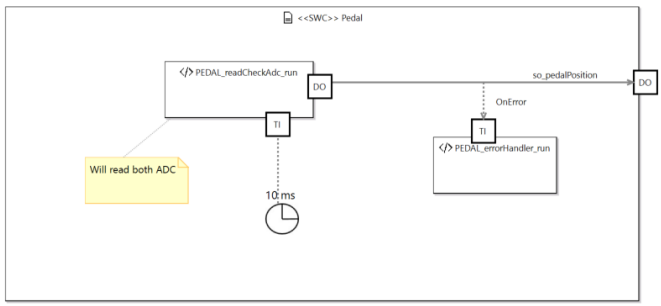

Figure 6 – Software Detailed View

Although the diagrams have a certain complexity, the overall concept is easily understandable. On every level, the SysML block diagram concept is applied. The blocks and connections have different semantics, but the general idea is the same for every view: Functional blocks transfer data to other blocks, typically following the Input-Logic-Output pattern.

Additional properties required for the code generation like memory sections, task assignments and similar are added in an configuration editor for the individual objects.

III. SOFTWARE ARCHITECTURE

The software structure follows the well-established concept of the AUTOSAR virtual function bus but provides some important simplifications as well as extensions, focusing on generating safe code.

The most important structural elements are software components (SWC) and runnables (RUN). Software components contain runnables and are generated as C or C++ source and header files. The runnables are (initially empty) functions inside these files.

The data flow is realized using so-called signals. Signals are implemented as data structures, containing the application specific payload as well as metadata like the age and validity of the data. Furthermore, signals can be linked to hardware drivers (synchronous and asynchronous) in order to receive or send data from or to peripherals. User defined scaler functions translate the signal application datatype to the corresponding peripheral datatype and vice versa. User specific validator function verify the validity of the data and provide the result to the application.

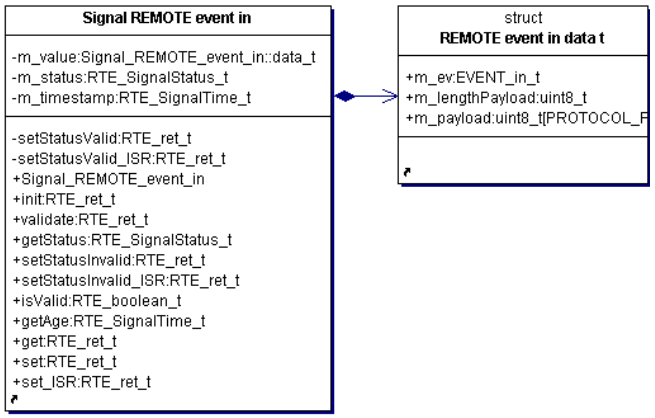

Figure 7 – Common API of a signal

Signals come in 2 different flavors, global signals (comparable to the AUTOSAR concept) and local signals.

Local Signals are defined within a single task context. This brings a couple of significant advantages for safety applications: Local signals can be stored on the task stack, which makes the MPU protection very straight format. Furthermore, these signals by definition are thread safe, i.e. mutexes and other blocking protections are not required.

If local signals are required by more than one task, deep copies of the signal are exchanged between the tasks/cores if a change occurs, using e.g. OS message services. As the intertask communication mechanisms of the RTE ensures a sequential execution of signal modifications, the risk of data races is significantly reduces compared to global data.

In case signals are frequently being accessed from different tasks and performance is a critical parameter, global signals can be used. These signals are stored as global data in the RAM. In order to protect them against illegal access, signals can be grouped in so-called signal pools, which are linked to specific memory ranges and can be protected by the MPU. This allows a protection of the data (e.g. only one task has write access, all other tasks have read access), but it is the system architect's responsibility to avoid race conditions, while the signals provide an interface to implement user defined locking mechanisms.

From the perspective of the application developer, local and global signals share the same API, i.e. local signals can be changed into global ones and vice versa without affecting the application runnables.

Runnables are mapped to individual RTOS tasks / cores and are triggered by activation events, which are available across core boundaries in case of multicore systems. Activation events can either be system state change events (e.g. ev_OnBoot), cyclic events (e.g. ev_10ms) or data update events (e.g. ev_CAN_onData). For handling error cases, error states and data error events are provided.

IV. CODE PATTERN AND CODE GENERATION

Based on the designed model, the framework code for the software components and runnables as well as all signals and the activation engine is being generated.

The Activation Engine is responsible for processing all events, transferring the data and calling the runnables. A strong focus has been placed on creating well readable, high-performant code. The following snippet shows the generated code for a runnable frame and illustrates some of the concepts used for the code generation:

- Note the highly verbose comments, which are created out of the model describing the execution context. Any change in the model is automatically reflected in the description.
- The signals which may be used inside the runnable are provided as parameters, ensuring that model and code remain in sync, easing code/design analysis and verification.
- The user code is placed inside a user code section ensuring that the code is not modified when updating the model.

```
/* --- Details for runnable "CANOPEN_rx_run" ---
 *  Description: dispatches CAN frames from hardware into the RTE.
 *  SIL required: QM; SIL achieved: QM; Justification:
 *  System States: bootup ,  normal
*  Signal association
 *     InPorts:
 *        Trigger IN
 *           Trigger Name "TI_10ms"
 *           Task: "Safety"
 *           System Events
 *              Cyclic Triggers:
 *              "CAN_dispatch_clock" Cycle Time: 10; Offset 0
 *     OutPorts:
 *        Data OUT
 *           DataPort Name "DO_FAULHABER_state_FL_in"
 *            - Associated Signal "FAULHABER_state_FL_in":
 *               Description:   front left engine state
 *               SignalPool:   gds_FAULHABER_state_FL_in "SAFETY_SIGNALS"
 *               Signal-Type:  FAULHABER_state_t (atomic: false; Primitive: false)
 */
void CANOPEN_rx_run(RTE_Triggers_t triggerPort,
Signal_FAULHABER_state_FL_in* const p_sig_OUT_FAULHABER_state_FL_in)  {
//Start of user code implementation CANOPEN_rx_run
//End of user code implementation CANOPEN_rx_run
}
```

Figure 8 – Code snippet for a runnable

The code is OS and controller independent, allowing an easy adaption of the generated activation engine to different ecosystems. Until now, the following operating systems and controllers have been successfully interfaced to the activation engine:

- Infineon AURIX multicore with PXROS-HR (C++)
- STM F7 single core ARM Cortex-M7 with Freertos-MPU (C++)
- Cypress PSOC 5 single core ARM Cortex-M3 with Erika / Autosar OS (C)

V. System Verification

As the model contains very rich data describing the structure and behavior of the system, current research activities focus on the implementation of a static design analysis, supporting the argument for a safety case. The following scenarios are currently under investigation

(A)SIL required versus (A)SIL achieved

As the model provided a seamless traceability from the hazard and risk analysis down to code and hardware components, the (A)SIL required can be compared against the (A)SIL achieved. This is particularly interesting in case the (A)SIL required for certain safety functions is being changed during the project life cycle or in case safety functions are being reused in a different context.

Freedom from interference

Freedom from interference typically comes in three flavors:

- Memory Usage
- CPU Usage
- Peripheral Usage

Based on the memory configuration present in the model, it is technically possible to check if code representing different safety functions or QM functionality share data in unprotected memory regions. As the access to the data is entirely handled by the activation engine (by providing references to required signals only) and the MPU configuration as well as the linker file are generated out of the model, a comparable deep analysis is possible.

Concerning CPU Usage, the cycle times and offsets can be analyzed to identify high load conditions on a core. [5] presents an additional approach to use a dynamic traceability concept for control flow monitoring and load analysis based on this runtime environment.

By tracing the drivers connected to the signals, conflicting usage of peripherals can be found.

Running on safe hardware

Often multicore controllers provide slightly different CPU configurations, e.g. on the AURIX, not every core is equipped with a lockstep core. By analyzing the mapping of runnables to individual cores, it can be verified that safety critical runnables are executed on safe hardware only.

VI. Results

The concept presented in this paper has been implemented using the Eclipse EMF framework using the extension packages Sirius and Acceleo.

The modelling notation has been proven to be well understandable not only for senior software engineers, but also for functional safety managers and junior developers.

In order to evaluate the concept, the code for an autonomously driving (model)car has been created (around 300.000 LOC total). The total effort as compared to a manual implementation has been approximately reduced by a factor of 3, mainly because complex messaging concepts and driver interfaces are now handled by the generator and the application developer can focus on the application code alone.

More time has been spent on creating, discussing (reviewing) and optimizing the design.

The created code structure and comments describing the execution context of the runnables have proven to be extremely helpful for understanding and optimizing the system behavior.

Debugging efforts have been drastically reduced, most problems were caused by bugs/limitations in the low level drivers, which were not generated by the tool. The stability of the system is significantly higher as compared to the manual coding of the previous system.

References

[1] ISO 26262, „Part 6: Product development at the software level,“ ISO, 2018.

[2] IEC 61598, „Part 7 - Overview of techniques and measures,“ IEC, 2000.

[3] IEC 61508, „Part 3 - Software Requirements,“ IEC, 2010.

[4] Wikipedia, „Real Time Object Oriented Modeling,“ 2019. [Online]. Available: https://de.wikipedia.org/wiki/Real_Time_Object_Oriented_Modeling. [Zugriff am 30 12 2020].

[5] B. F. Radtke, „Control-Flow Evaluation on Multicore Controllers using Real-Time Trace Information,“ in *embedded world Conference*, Nürnberg, 2021.