Modeling and code generation for safety critical systems

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

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

***Abstract*— The design and implementation of embedded safety (multicore) systems is highly challenging without good tool and methodology support, especially for small and medium sized development teams. Safety artifacts like hazard and risks analysis, specifications of safety functions and architectural solutions typically are realized as individual documents without a common repository, making the maintenance hard and error prone. Moreover, reference architectures are rare and design tools typically only cover parts of the design process.**

Together with the FZI Research Center for Information Technology and the company HighTec, an innovative modeling and code generation tool for safety related systems has been realized based on the open source Eclipse Modeling Framework (EMF). In this paper, it is demonstrated how the tool unifies architectural and safety aspects and how an entire multicore runtime environment can be generated using EMF. The developed design patterns are described and it is demonstrated how a safety architecture can be realized using a multicore safety controller.

***Keywords*— EMF; Modeling; Code Generation; Architectures; Multicore; Runtime environment; Functional Safety;**

I. Introduction

Compared to industrial applications, the development of an embedded safety system is much more complex.

- Other than automation systems, for which standard components like Safety PLCs and well known design patterns are available, embedded systems provide a much higher degree of freedom during the design phase.
- Embedded systems usually are built up using a variety of different components from different vendors. Modern safety multicore controllers, real time operating systems (RTOS) and software libraries come with a huge number of safety features, which need to be configured properly to ensure safe operation.
- Many embedded applications are developed as mixed criticality systems, requiring proper separation of criticality domains. At the same time, these domains often need to exchange data, requiring safe communication channels.

Multicore microcontrollers increase the complexity even more. Shared resources such as memory and peripherals not only require consistent timing to avoid races, but they also need to be protected against unintended access.

When developing a safety application, it is not enough to “believe” that the system is safe. Safety standards like the IEC61508, ISO26262 or ISO13849 require a proof of the safety concept – the safety case. Although details may differ in the standards, the overall structure of such a safety case or safety proof is comparable, consisting of the following main elements:

- A hazard and risk analysis identifying safety functions and resulting in a required safety integrity level (SILreq).
- The use of selected processes and methods to avoid systematic errors.
- The application of architectural patterns to reduce the impact of possible faults.
- A proof of sufficient qualification methods (reviews and tests).

The focus of this paper are the contractual aspects of the safety case, i.e. the hazard and risk analysis and the provision of architectural safety patterns for embedded (multicore) controllers.

II. Searching the Safety Modelling Standard

In order to develop a safety system, all standards require the utilization of models as well as the establishment of traceability between safety requirements, architectural elements and qualification artifacts like test cases and reviews.

Which modelling notation is best suited? An obvious choice, due to its wide spread, probably would be the Unified Modelling Language (UML). The focus of this notation is the description of static structures as well as dynamic behavior of software systems. The class diagram is great to illustrate class hierarchies, while the state diagram and activity diagram can be used to describe the behavior of algorithms. Unfortunately, the UML also comes with some weaknesses:

- The diagrams serve as illustrations with no binding meta-model behind them, which is a prerequisite for proper code generation.
- The UML does not provide suitable diagrams for the visualization of data flows, which have proven helpful to describe safety functions.
- The description of explicit safety artefacts is not supported.

Another standard that has been evaluated is the Systems Modeling Language (SysML). The development of a safety architecture is a typical system-engineering task, not limited to software. Especially the block diagram provided by SysML is an interesting candidate for the description of signal flows on various levels (functional, hardware, software) but lacks features describing the runtime behavior of an application.

Although the AUTOSAR Virtual Function Bus / RTE originally has been developed for the automotive industry, the general concept can be extended to any domain. In addition to the signal flow, concepts like RTE events describe trigger actions for runnables. The notation is comparable to the SysML block diagram and is applied as proprietary standard in widely used tools such as MATLAB.

III. Domain Specific Modelling Language

As no modelling standard satisfied all requirements [1], it was decided to propose an own safety domain specific language (DSL). The key goal is a complete but still easy to use DSL, fulfilling the following requirements

- It shall be possible to store all elements of a safety case in a common repository, i.e. requirements, architecture (functional, hardware, software) and reference to qualification artifacts (Test cases, etc.).
- Based on the model, it shall be possible to generate well readable and easily qualifiable source code, focusing on system configuration (RTOS, memory etc.), data flow and runnable activation.
- Furthermore, it shall be possible to automatically validate the model, supporting safety assessments.

IV. Tool/Workflow

As part of the publicly funded ZIM project “Zukunftstechnologie Multicore”, the Darmstadt University of Applied Sciences together with the FZI Research Center for Information Technology and the company HighTec, developed an Eclipse based toolchain [2]. The developed tool (see Fig. 1) integrates different views (requirements, functional, hardware and software architecture) into a single model, supporting direct linkage among the artifacts. The resulting model then is processed to generate code files and safety documentation for an entire multicore runtime environment.

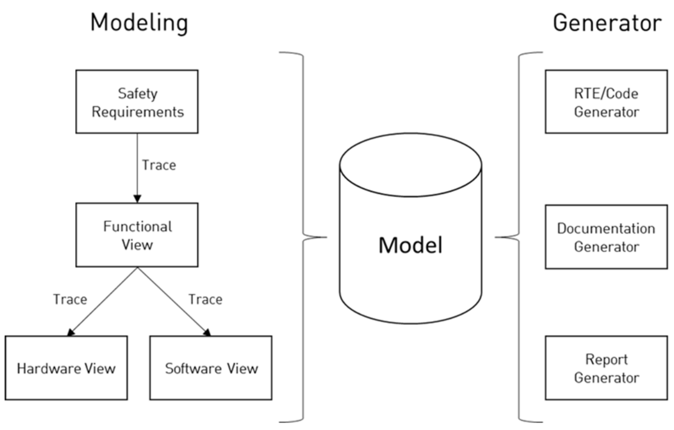


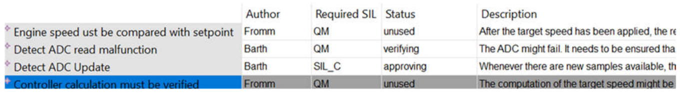
Fig. 1 tool structure

Since all views are based on the same model, different workflows are being supported – including traditional V-model approaches as well as agile development.

In the following sub-chapters, an electronic throttle control shall serve as an example to introduce the different views: A dual sensor throttle pedal acts as input to an ECU, which will set the target speed of an electric motor relative to the pedal position. The motor shall have a safe state in which its power supply is cut off by a relay.

A. Safety Requirements

The tool supports the documentation of hazard and risk scenarios as well as the deviation of safety goals and safety requirements in a tabular format (see Fig. 2). Depending on the applied safety standard, the required SIL or ASIL level can be added. Requirements can be linked to other entities within the model, supporting traceability. Safety information is also generated into the code files as in-line documentation to support the system understanding of developers.



	Author	Required SIL	Status	Description
Engine speed ust be compared with setpoint	Fromm	QM	unused	After the target speed has been applied, the re
Detect ADC read malfunction	Barth	QM	verifying	The ADC might fail. It needs to be ensured tha
Detect ADC Update	Barth	SIL_C	approving	Whenever there are new samples available, th
Controller calculation must be verified	Fromm	QM	unused	The computation of the target speed might be

Fig. 2 Safety Requirements

B. *Functional view*

The functional view (see Fig. 4) describes a high-level architecture of the system, focusing on the functional signal-flow and neglecting details about the hard- and software. There are only three types of entities: input, service and output, where each entity contains information about the required and achieved SIL. Each entity can have multiple ports and ports can be connected to each other.

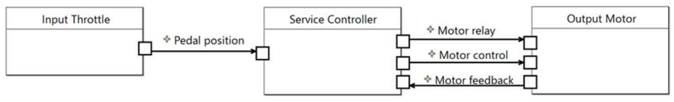


Fig. 4 Functional view

C. *Hardware view*

Safety systems require assessment of all physical components involved in critical signal-paths. Similar to the functional view, the hardware view (see Fig. 5) is a high-level representation focusing on the physical structure of the system. Possible entities are sensors, actors, communication channels and controllers. Again, all entities contain information about the required and achieved SIL and can be connected to each other. While the graphical representation is rather minimalistic in order to increase readability, the underlying model is more granular also supporting modeling of multicore ECUs, where safety related features such as lockstep can be documented.

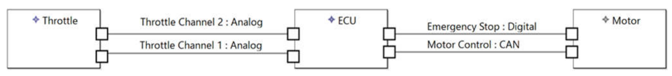


Fig. 5 Hardware view

D. *Software view*

The software view (see Fig. 6) describes the signal- and control-flow on a single controller, providing two levels of detail. On the first level, software components (SWC) representing functional units are visualized. Software components are connected to each other using ports. Data-ports transport payload while trigger-ports control the activation of runnables and therefore the runtime behavior and control-flow. The connection between ports is called signal where a signal has attributes such as name, datatype and many more.

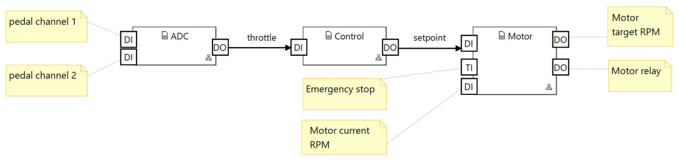


Fig. 6 High-level software view

The second level (see Fig. 3) is a detailed view, showing the internal details of a software component. A software component represents a collection of runnables. Runnables are real execution units (i.e. code functions), which have data- and trigger-ports. Data ports describe the input and output parameters of a runnable, whereas the trigger ports control its activation. Triggers can be periodic events, system events such as task starts or signal events such as "new data available" or "error detected". The trigger port defines the context and system state in which the runnable is executed. Moreover, attributes such as guard functions or priority support flexible configuration of the intended control flow.

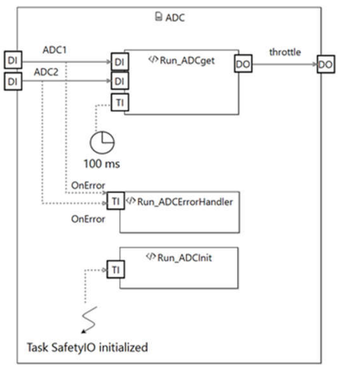


Fig. 3 Detailed software view

V. RTE

The tool contains a code and document generator, which creates an entire C++ Runtime Environment (RTE). Files that may be modified by the user contain "user-sections" in which the data is preserved if the system is re-generated, easing file handling for systems under development. The code generation itself can be configured using the model while advanced users can modify the underlying templates e.g. to support other programming languages or to customize the RTE. The interface of the RTE towards the ecosystem is very lightweight. Besides a global timestamp source, the RTE requires RTOS tasks that can be paused from within the task and activated from another task.

The RTE consists out of two main parts. The signal-layer contains classes representing the payload, while the activation engine invokes runnables and controls the runtime behavior. A typical event driven signal-flow is shown in Fig. 7: The activation engine activates a runnable "A" cyclically (1). Runnable "A" writes to a signal (2), which causes a notification to the activation engine (3). The activation engine now activates runnable "B" (4), which reads from the signal.

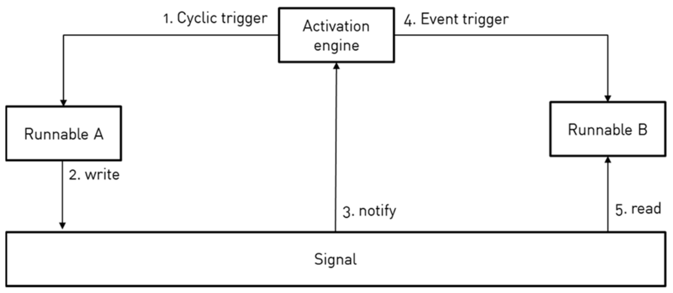


Fig. 7 RTE example control-flow

The code files implementing the signal-layer as well as the activation engine are fully generated. For the runnables, an executable outline containing the input and output parameters based on the designed signals is generated. The functional code is added by the user or a tool like MATLAB can be used.

Comparable frameworks often produce rather complex and unreadable code. All code files generated by this solution have a strong focus on easy to understand structures (no abstract tables or function pointers) and readable code along with model specific in-code documentation. Due to its simplicity, the RTE eases debugging, testing and verification.

VI. SIGNAL-LAYER

The signal layer is a wrapper for the data flow handled by the RTE. A signal contains data as well as meta-information and is transported between runnables and/or hardware (see Fig. 9). Typical signal-flows are runnable to runnable, runnable to hardware and hardware to runnable.

Each signal defined in the model is generated as a dedicated C++ class and individual documentation. The user optionally can provide signal specific scaler methods to translate e.g. driver data types into physical application data types. Moreover, the user can define an optional verification method to check if a new value is plausible before it is assigned to the signal. Signals can be associated with driver functions reading and writing to the hardware if a signal is refreshed, written or if new asynchronous data is available.

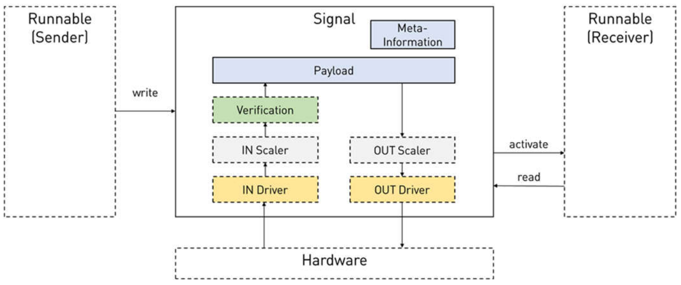


Fig. 9 Signal structure

VII. ACTIVATION ENGINE

The activation engine controls the runtime behavior of the system (see Fig. 8). For each RTOS task, a C++ class is generated which activates the runnables based on the triggers defined in the model. The interface is very lightweight with only two methods to be used by the hosting task. In addition to the task activation engines, there is one central activation engine task responsible for inter-task communication. The global time-base of the activation engine along with the utilization of global time-stamps within the signal layer ensures consistent timing on a controller wide scale.

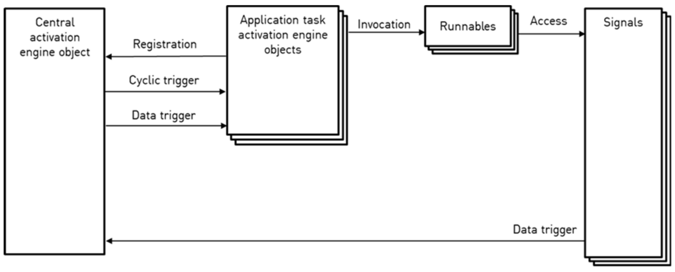


Fig. 8 Activation engine overview

Runnables are invoked by cyclic- or data-events sent by the central activation engine. The signals associated with a runnable are passed as parameters and can be accessed within the runnable, providing a high degree of modularity and encapsulation. Snippet 1 shows an implementation of a runnable as well as the (reduced) in-code documentation generated. If runnables have been linked to signal events (OnData, OnError) in the model, signals will send a trigger to the central activation engine, which activates the runnable that subscribed the signal, supporting event driven architectures.

Runnables can have a return value. If a runnable returns a value other than “OK”, an error handler is invoked, supporting centralized but flexible error handling. The RTE itself handles internal errors with user defined error handler functions.

```
/* --- Details for runnable "Run_control" ---
 *  Description:
 *      wird aufgerufen, wenn throttle einen neuen Wert bekommen hat. […]
 *  SIL: QM
 *  System states:
 *      normal
 *  Signal association
 *       InPorts:
 *           Trigger
 *             Trigger source throttle.onData
 *             Task: "Control"
 […]
 */
RTE_ret_t Run_control(Signal_throttle& sig_IN_throttle, Signal_setpoint* const
p_sig_OUT_setpoint){

    //runnable return
    RTE_ret_t ret=RTE_RET_OK;

    //local IN signal value buffers
    Signal_throttle::data_t throttle_data;

//Start of user code implementation Run_control

    //get throttle value
    if(RTE_RET_OK==sig_IN_throttle.get(&throttle_data)){

        //calculate new setpoint
        setpoint_data_t new_setpoint=(throttle_data/100.0)*SETPOIN_T_MAX;

        //set setpoint
        p_sig_OUT_setpoint->set(new_setpoint);

    }else{

        //throttle is invalid, invalidate the setpoint
        p_sig_OUT_setpoint->setStatusInvalid();
        ret = RTE_RET_SIGNAL_INVALID;
    }

//End of user code
    return ret;
}
```

Snippet 1 Runnable

Consistency in the time domain on multicore systems is difficult to achieve, as there is truly concurrent execution of code as well as RTOS multi-threading. Consequently, the use of shared global data may lead to highly critical race conditions and should be avoided. Each activation engine object stores a local copy of all signals used in the context of this object. Local storage on the task stack has the advantage that this memory is comparable easy and reliable to protect. Since the signal objects are allocated statically, the maximal stack consumption is predictable and only varies with the implemented code in the runnables. Whenever a signal is updated, local copies of this signal are synchronized throughout the system. This strong decoupling causes a certain overhead but its clear structure, simplicity, testability and predictability outweighs the disadvantages especially in safety critical systems.

The central activation engine supports a number of safety relevant features. An alive monitoring mechanism, working along with physical watchdog timers and checking the availability of all RTOS tasks and activation engine objects has been implemented. In future research, it will be investigated if the centralized activation of runnables can be used to implement more advanced features such as control flow monitoring. Moreover, the decoupling of the central activation engine and the task activation engines supports the dynamic reallocation of tasks even to other cores, opening the door for dynamic but predictable reconfiguration of multicore controllers in the event of a failure, increasing the availability of a system and therefore its safety.

VIII. Conclusion and outlook

The developed tool already has been proven itself in an academic environment. Due to its strict structure, potential architectural design flaws become visible in early design stages. While porting legacy systems to the new architecture, latent bugs became obvious and the overall system complexity was reduced. In comparison to manual approaches, the tool has shown a tremendous simplification of the overall implementation process. However, the usage of the tool requires a certain degree of training and experience.

The clear and intuitive structures along with the good readability of the generated code allows developers to adapt to the new runtime environment relatively fast. Moreover, the system is comparable easy to debug, test and verify. Changes in the model only have very local effects on the generated code, allowing changes easily to be merged and enabling incremental/iterative development.

The lightweight interface of the RTE towards the ecosystem allows the usage on a variety of controllers and operating systems.

The strict memory partition by the RTE ensures freedom from interference between safety components of different criticality. Together with the modeling of safety requirements, graphical views, generated documentation, clear structure and good debug and testability, the resulting system is comparable easy to be qualified.

Abbreviations

ASIL	automotive safety integrity level
AUTOSAR	automotive open system architecture
DSL	Domain specific language
EMF	eclipse modeling framework
FZI	Forschungszentrum Informatik
MPU	memory protection unit
PLC	programmable logic controller
RTE	runtime environment
RTOS	real-time operating system
SIL	safety integrity level
SWC	Software component
SysML	systems modeling language
UML	unified modeling language
ZIM	Zentrales Innovationsprogramm Mittelstand

References

[1] P. Liggesmeyer and M. Trapp, “Trends in Embedded Software Engineering,” IEEE Softw., 26. Jg., Nr. 3, S. 19–25, 2009.

[2] T. Barth et al. “Modeling and Assessment of Safety Critical Systems” in *Embedded World Conference 2019*