



Contract number: ITEA2 – 10039



Safe Automotive soFtware architEcture (SAFE)

ITEA Roadmap application domains:

Major: Services, Systems & Software Creation

Minor: Society

ITEA Roadmap technology categories:

Major: Systems Engineering & Software Engineering

Minor 1: Engineering Process Support

WP 5, WT 5.6

Deliverable D.5.6.c:

First prototype "Safety Code Generation" - Report

Due date of deliverable: 28/02/2014

Actual submission date: 28/02/2014

Start date of the project: 01/07/2011

Duration: 36 months

Project coordinator name: Stefan Voget

Organization name of lead contractor for this deliverable: ZF Friedrichshafen AG

Editor: Jürgen Lucas

Contributors: Helmut Miller, Christoph Ainhauser, Kazi-Khaled Al-Zahid, Raphael Trindade

Version	Date	Reason
0.1	2012-10-26	Initialization of document
0.2	2012-11-06	Adapted document structure according to feedback from P. Cuenot
0.9	2012-12-19	First Version for D5.6a (Ready for Review)
0.9.1	2012-12-22	Incorporated changes driven by review
1.0	2013-01-08	Preparation of official deliverable D5.6a
1.1	2013-12-09	First Version for D5.6b (Ready for Review)
1.2	2014-02-18	Added some statements about qualitative evaluation
1.3	2014-02-24	Added figures, added appendix
2.0	2014-03-03	Preparation of official deliverable D5.6b
2.1	2014-10-10	First Version for D5.6c
2.2	2014-10-28	Integration of Evaluation Results
2.3	2014-10-30	First Version for D5.6c (Ready for Review)
2.4	2014-11-18	First integration of Review Comments
2.5	2014-11-24	Pre-final Version
3.0	2014-11-26	Official Version for Deliverable D5.6c

1 Table of contents

1	Table of contents	3
2	Executive Summary	5
3	Evaluator “Safety Code Generation”	6
3.1	General description.....	6
3.1.1	<i>Torque Vectoring Use Case</i>	6
3.1.2	<i>Technical safety concept (Overview)</i>	7
3.1.3	<i>Demonstrator Environment</i>	9
3.2	Safety Measure “Gradient Checker”.....	10
3.2.1	<i>Motivation and Argumentation</i>	10
3.2.2	<i>Implementation</i>	11
3.3	Safety Measure “Program Flow Control”	22
3.3.1	<i>Motivation and Argumentation</i>	22
3.3.2	<i>Implementation of Program Flow Control</i>	23
3.4	Dependencies.....	24
3.5	Final Implementation State of the Evaluator	24
4	Evaluation Results	25
4.1	Qualitative Evaluation	25
4.1.1	<i>Setup of Safety Case</i>	25
4.2	Quantitative Evaluation	26
4.2.1	<i>Integration effort vs. implementation effort</i>	26
4.2.2	<i>Memory Consumption</i>	27
4.2.3	<i>Runtime Effects</i>	27
4.2.4	<i>Safety Case Effort</i>	27
4.3	Fulfillment of WP 3 requirements	27
4.4	Final quantification of Safety Code Generation as evaluated in WT5.6:	30
5	Conclusion.....	31
6	References	32
7	Acknowledgments	33
8	Common Metrics for evaluation.....	34
9	Appendix	35
9.1	Tool prototype “control flow monitor”	35
9.1.1	<i>Adapted SW architecture</i>	36
9.1.2	<i>Watchdog configuration</i>	37
9.1.3	<i>Task body</i>	38
9.1.4	<i>Source code for watchdog service</i>	38

2 Executive Summary

The objective of WP5 (see SAFE FPP [5]) is a) to refine requirements for, b) provide feedback on and c) evaluate methods and tools developed in WP3 and WP4 as well as methodologies and application rules defined in WP6 in context of realistic industrial case studies. Best practices established during the evaluation will be documented.

This document describes the prototype of the evaluation scenario implemented in WT5.6. It introduces the demonstrator platform giving an overview of the architecture and of the technical safety concept. The experience with code generation for SW Safety Components (SSC) is collected; the integration into the demonstrator platform is described and evaluated.

In addition, it contains a reference to the addressed methods and tools developed in WP3 by evaluating how they fulfill their associated requirements defined WP2 (as far as in the scope of the WP5.6).

3 Evaluator "Safety Code Generation"

3.1 General description

The evaluation environment is the torque vectoring system (QMVH) implemented first for BMW X6.

3.1.1 Torque Vectoring Use Case

The basic function of QMVH system is the improvement of agility and stability in a SUV application. This is achieved by an active torque distribution on the rear axle by using 2 clutches (one on each side).



FUNCTION:

- Adaptive torque distribution to rear wheels
- Improve stability and agility
- BMW X6

HAZARD:

- Unintended torque rise can lead to uncontrollable instability.

SAFETY GOAL:

- Unintended torque rise of more than 400Nm must be avoided

Figure 1 - Effect of QMVH

The gearbox itself consists of one classic differential and two actuators realizing the torque transmission function. If a clutch is engaged, torque is transmitted from one wheel to the opposite, thus leading to a gear torque that may be used to facilitate steering or add stability in case of over steering. The clutches are operated by electro motors with a ball ramp mechanism.

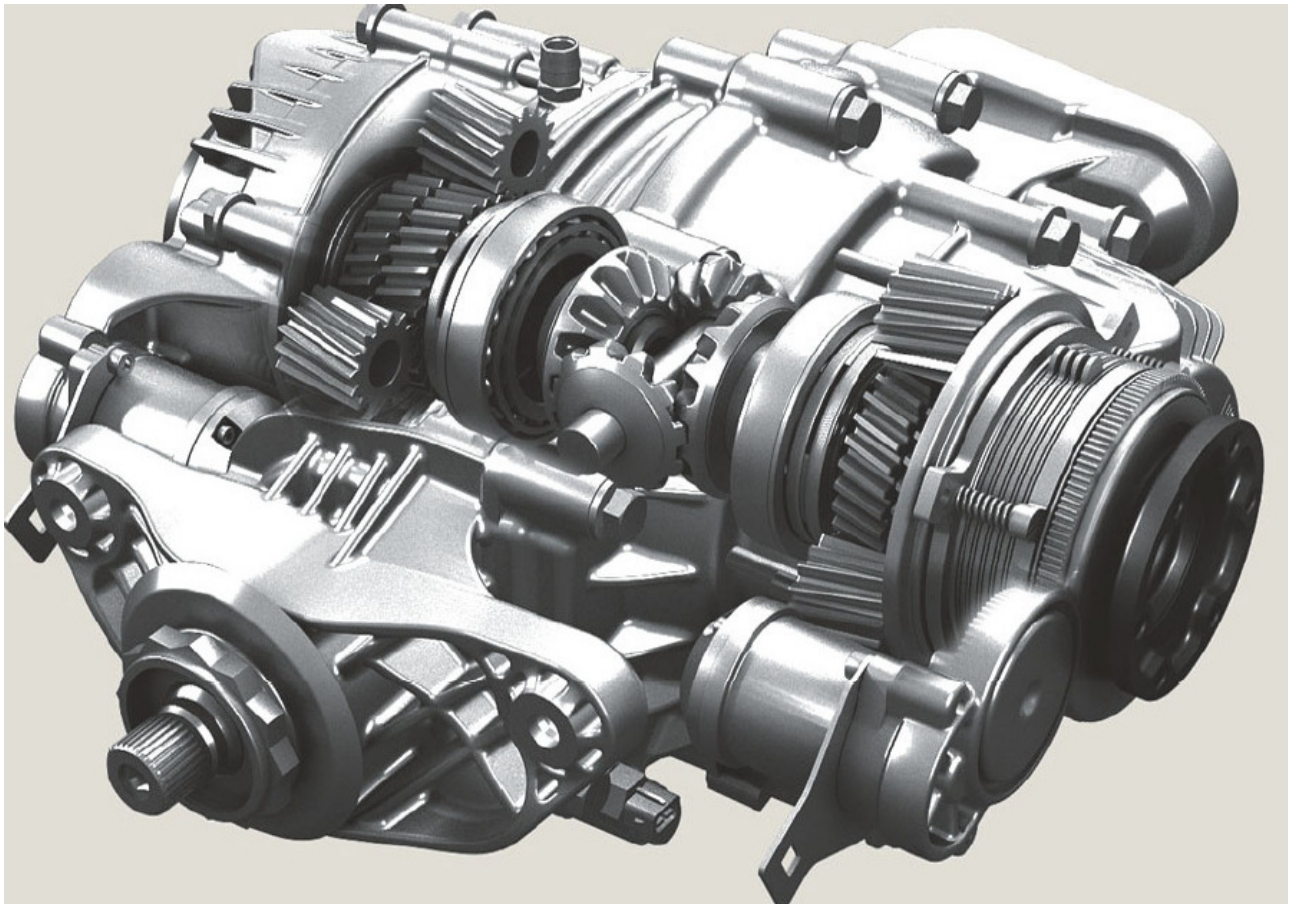


Figure 2 - Gearbox

As to its steering effect, the system is safety relevant. One safety goal is the detection and handling of an unintended torque step of more than 400Nm at a wheel.

Safe state is identified as “open differential” (no e-motor torque is applied).

The QMVH system has been introduced to series production in 2008 and was developed according to IEC61508. The Hazard & Risk Analysis (HARA) based on IEC61508 yielded a classification of SIL-3.

Base development of the system (2006 – 2009) has not been performed in compliance with AUTOSAR.

In a product update (2011-2013), the HW has been partially redesigned in order to cope with AUTOSAR and FlexRay requirements, but most parts of the existing safety concept have not been changed, as they are not influenced by the changes. Only minimal adaptations in the function of the SW have been introduced.

As to the introduction of ISO 26262, the system has been reclassified as ASIL D.

The requirements on the performance of the system (response time, accuracy) are high, leading to the need to compensate various effects in the actuator (e.g. the change of friction in the clutch in case of a temperature increase, mechanical extensions depending on oil temperature, ...). A wrong step in a temperature may lead to a torque step that conflicts with the safety goal mentioned above.

A rough overview of the safety measures is given in the next section.

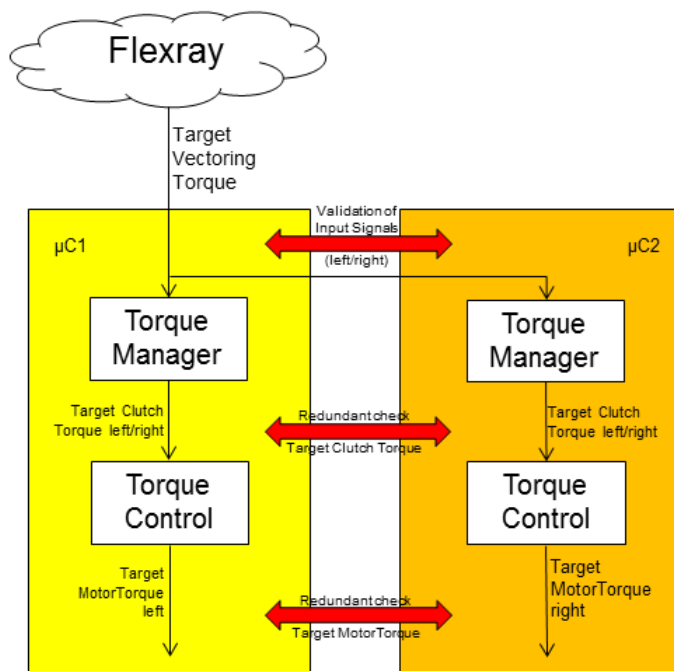
3.1.2 Technical safety concept (Overview)

So, as to the safety requirements identified in HARA, the technical safety concept has (besides other means) introduced in the SW:

- E2E-protection of communication on CAN/FlexRay
- Checkers for all input data that may influence the E-Motor torque that is applied to a clutch

- Dual controller system the redundant execution of critical portions of the code
- Binary comparison of results
- Redundant evaluation of environment data (sensors and CAN/FlexRay)
- Tolerant comparison of input data
- Program flow control
- Plausibility checks on input data and calculated results
- Actuator monitoring (diverse programming)
- RAM checks, ROM checks
- WD-Tests
- Independent check of operating modes
- Independent check of fault reactions

When a safety relevant error has been detected, an error handling procedure is triggered, leading to a switch off (fail safe).



SAFETY MECHANISMS:

- Redundant data acquisition (CAN, UART, ADC)
- Redundant computation (safety relevant parts)
- Data plausibility checking
 - a) range
 - b) change rate
 - c) consistency
- Safe States (FTI 100ms):
 - a) Zero Gear Moment
 - b) Shutdown

Figure 3 - Abstract Technical Safety Concept

Identification of validation items

As to the classification of the system, a set of safety mechanisms have been integrated into the SW (see previous section).

In WT5.6 only a subset of these safety mechanisms will be used to validate the code generation concepts defined in WT3.6. As the SW architecture is not adhering to AUTOSAR (e.g. no RTE is present), it is necessary to make some adaptation to the SW architecture (and also to the SW realization) to allow the SAFE concepts to get in good contact with the existing SW. In order to keep the effort for this preparation as low as possible, the mechanisms that are subject to replacement by means of code generation are identified with respect to these criteria: can be a) easily isolated from the rest of the system and b) replaced by components generated by means of code generation as identified in WT3.6.

As a result of this analysis, the following use cases for analysis of code generation for *software safety components* (SSC) have been identified:

- *Gradient checkers* required for temperatures (both measured and calculated) according to the technical principles (a temperature may not increase with more than X °C per 10ms).
- *Program flow control* checking that SW components are called in the correct order

3.1.3 Demonstrator Environment

This section will describe the testing and demonstration environment. The testing and demonstration environment is built up for validation of the SW measures to show the effects in a way that is easily understandable in a presentation to a wider audience.

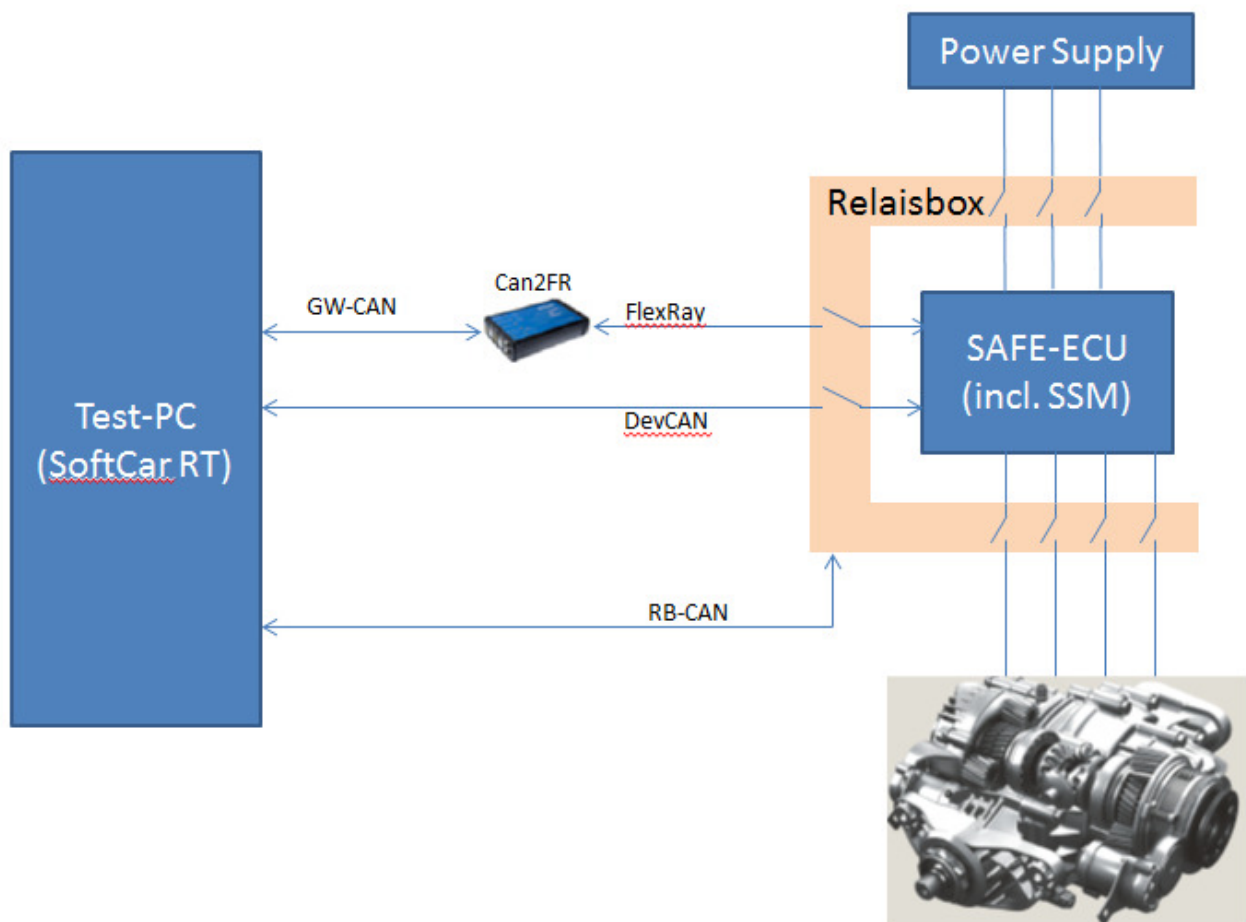


Figure 4 - Overview of Evaluation Environment

The Evaluation Environment consists of the following main components:

- Power Supply
- Simulation PC
- PCI-Expansions-Box
- 2 IXXAT-Karten
- 1 FlexRay-Gateway (TTTech)
- ZGW (Central Diagnostic Gateway)
- Relais-Box (HW Fault Introduction)

- ECU
- Gearbox (TV-HAG)
- Cabling

SW Fault Injection:

SW Fault Injection is used to enter faults inside the ECU, where it is not possible to introduce faults by HW means. This is used, for instance, to emulate an internal signal being corrupted by a SW-error or RAM fault.

The SW Fault Injection uses an additional CAN message. A set of error hooks in the code is defined and a set of error models is defined (stuck at, gradient, jump, ...). Via the additional CAN message, the error hook and the error model is selected.

3.2 Safety Measure "Gradient Checker"

3.2.1 Motivation and Argumentation

3.2.1.1 Development approach before SAFE

During product development in 2008, the gradient checkers have been developed manually using C programming language. As to the need to check two temperatures simultaneously, an instance concept has been introduced. So, data and function are separated using data containers with access via pointers.

One important component in the existing gradient checkers is a filter mechanism. A wrong gradient will trigger an error only if it is present consecutively for $> x$ ms.

The behavior of the output signal in case of a detected gradient error is specified in a way that the value is ramping to the new input value with the gradient that is the specified as maximum gradient.

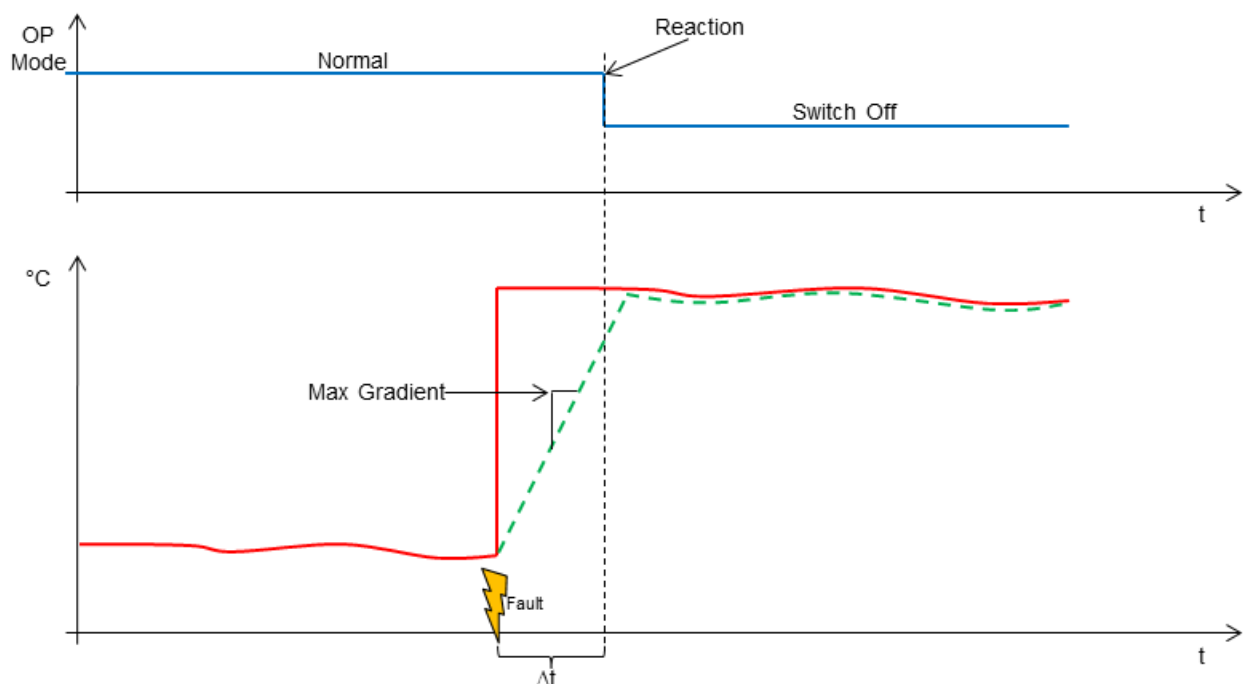


Figure 5 – Gradient Checker: Limitation of gradient, fault handling

The red line describes the faulty raw signal, the green line describes the gradient limited (safe) signal. The blue line shows the error reaction.

3.2.1.2 New approach

The new approach takes the technical safety concept as given and focuses on the implementation of identified and required safety mechanisms by means of code generation. Based on an abstract model of the evaluator SW, the necessary software safety component (here: gradient checker) is modeled and integrated into the existing SW while replacing the original, manually coded gradient checker component.

The generated artifacts have software interfaces according to the AUTOSAR Virtual Function Bus (VFB) principle **Fehler! Verweisquelle konnte nicht gefunden werden.** Thus, the pre-existing software architecture has been restructured to communicate with the generated artifacts via the AUTOSAR RTE **Fehler! Verweisquelle konnte nicht gefunden werden.**

The newly introduced software elements realize section 6.4.3 (gradient checker) as well as section 6.4.6 (filter mechanism) in D3.6 [6].

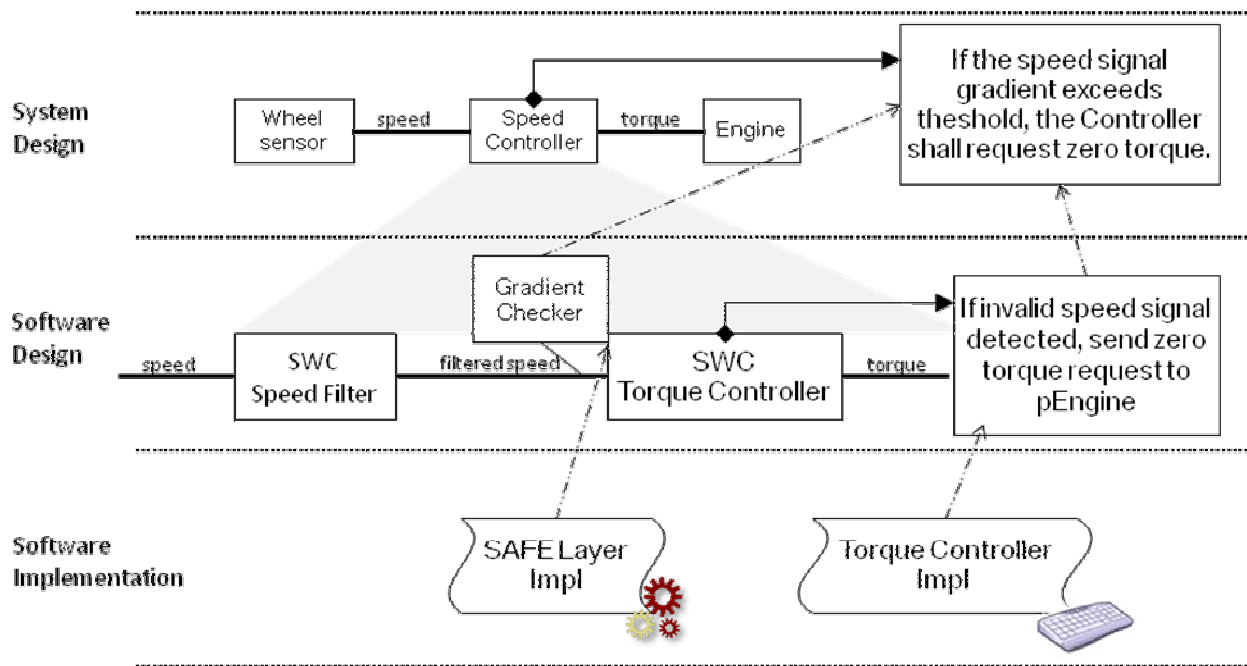


Figure 6 - Generation workflow for gradient checker

3.2.1.3 Benefits / drawbacks of new approach compared to current approach

Given the necessary infrastructure, the interaction with the generated software safety components is easy. This holds for the specification phase as well as for code integration. No major drawbacks are identified. Details of the evaluation are given in the evaluation section 4.

3.2.2 Implementation

As the existing software is a pre AUTOSAR SW (no SWC's, no RTE), the work had to be structured as follows:

- Preparing of the SW environment: Modeling of the existing SW in SWC's, introduction of a prototype RTE. This is necessary to allow the usage of new components generated with the means of SAFE (See section 3.2.2.1)
- Specification of the Gradient Checker with the means of SAFE and replacement of the old checker component (See section 3.2.2.2)

As a conclusion, in section 3.2.2.3 key work products can be found representing both approaches. The key work products that are generated using the SAFE approach (with automated code generation) can be easily compared with the corresponding work products of classic development approach (manual design and coding).

3.2.2.1 Preparation of the SW environment

Based on the analysis of the existing SW system, an abstract model of the SW system has been developed. This model adheres to the principles of AUTOSAR conventions and is based on the SAFE meta model. Within this model, the main components of the system are modeled:

- OperationManager: it has the task to manage detected malfunctions of the system, degrade the actuators if needed and perform finally a restart of the ECU.
- RestECU: this component is an abstraction of the concrete SW architecture dealing with the access to the sensors and actuators. One of its tasks is to provide the information needed by the safety mechanisms. Temperature Models and sensors are modeled in this component.
- TorquePosCalculation: functional component performing the actual torque distribution on the rear axle.

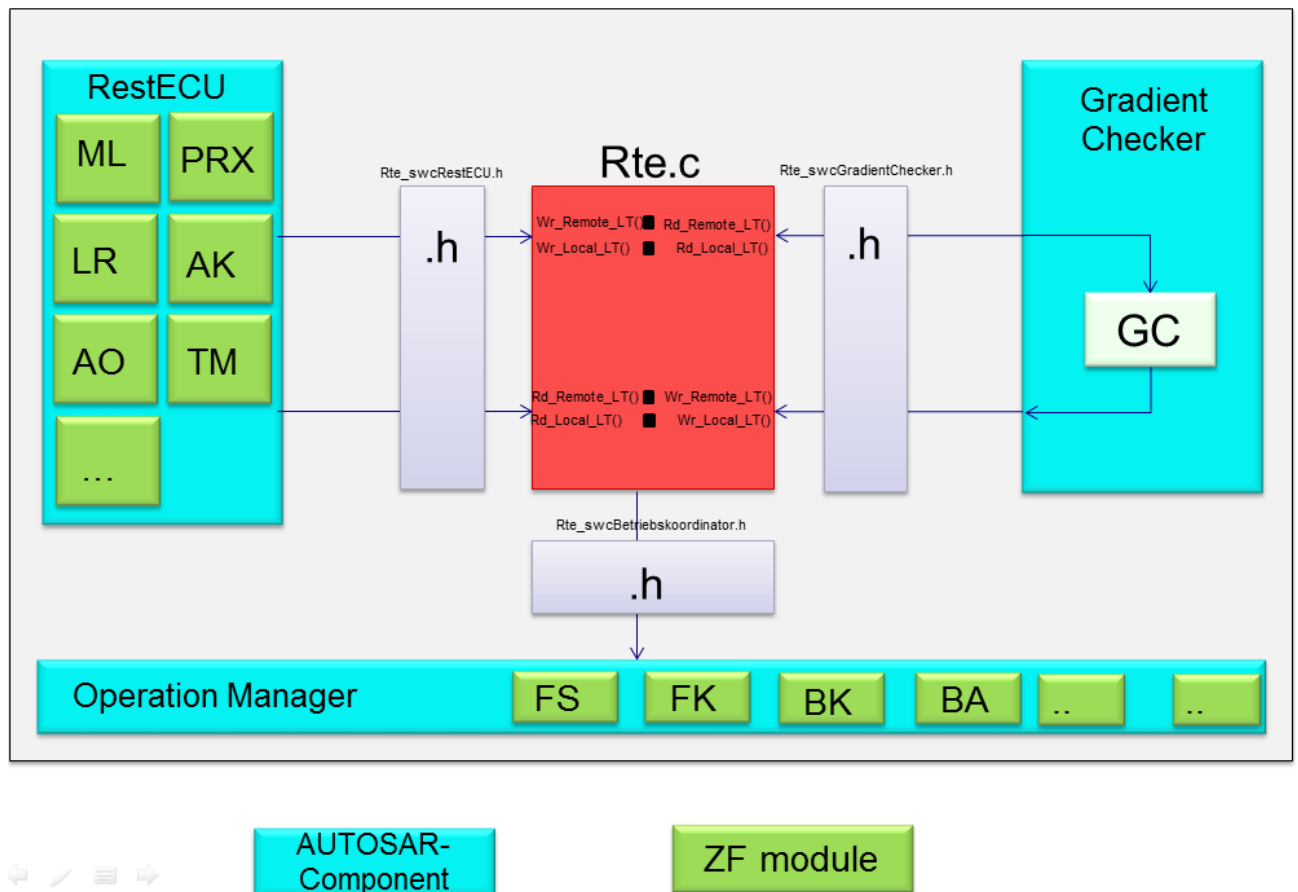


Figure 7 - SWC's and Source Modules

The list reflects the mandatory components deployed on one concrete ECU. Driven by the safety concept, the system architecture consists of two instances of the same ECU. Each ECU has the task to manage one single actuator and monitor possible malfunction on the redundant ECU driven by random HW failures.

3.2.2.2 Replacement of the existing Gradient Checker Component

The WT3.6 work result under consideration is the gradient checker. Implementation was performed as follows:

- (1) Definition of SW structure and interfaces within the existing SW.
- (2) Definition and implementation of an RTE realizing the communication between ZF application SW modules and the planned gradient checker module.
- (3) Interfaces are specified in the style of AUTOSAR.
- (4) Based on the definition of a Gradient-Checker using the SAFE exchange format, a C-Code module is generated according to the specified interfaces.

- (5) The threshold values (gradient boundaries and filter values) are taken from the existing implementation and used as parameters in the SAFE model of the gradient checker.
- (6) The old filter and gradient checker code has been removed.

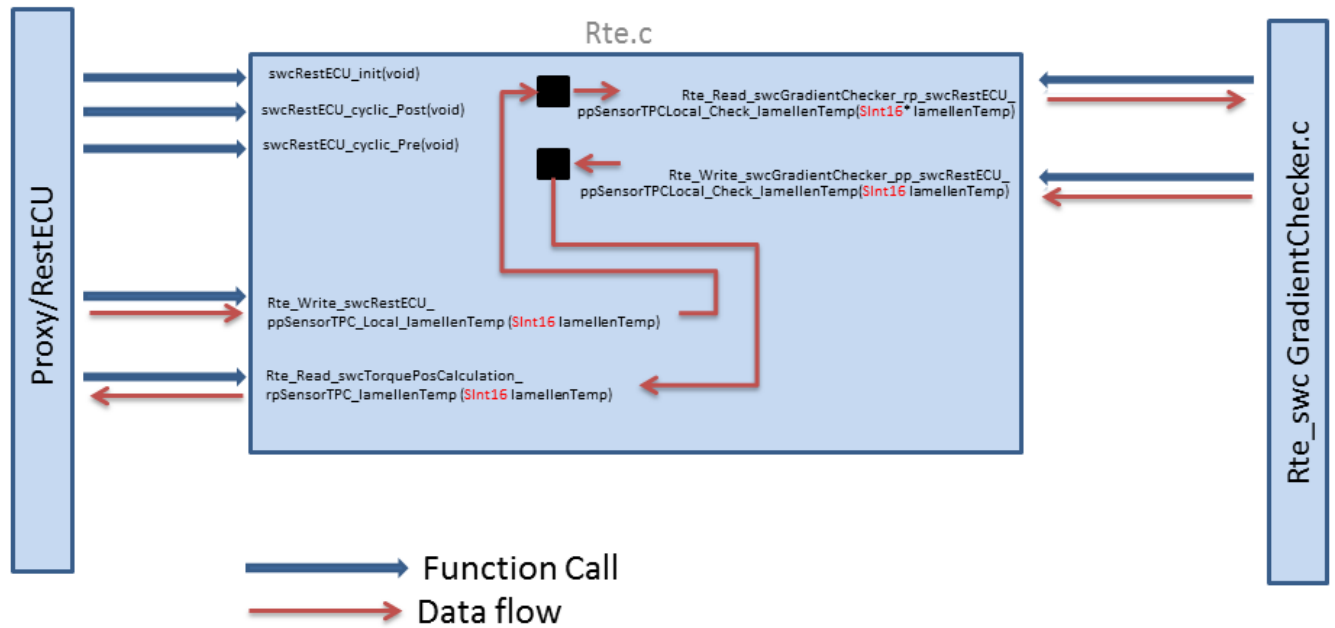


Figure 8 - RTE with Data and Control Flow

3.2.2.3 Development of Gradient Checker: Classic Approach vs. SAFE Approach

For the purpose of comparison, this section gives an overview of the different work products as far as they are influenced by the code generation approach. First, the different work products are sketched, then in Figure 9, Figure 10 and Figure 11 the work products of the classic development approach are presented. Further, from Figure 12 to Figure 17, and the corresponding work products of the SAFE development approach are presented.

	Classic Development Approach (Manual Implementation)	SAFE Development Approach (Automated Generation)
Hazard and Risk Analysis	<ul style="list-style-type: none"> • ... • A Torque Step > 400Nm can lead to a undesired driving behaviour. • ... 	
Safety-Requirements	<ul style="list-style-type: none"> • ... • A temperature step of > x °C must be detected and controlled • ... 	
Technical Safety Concept	<ul style="list-style-type: none"> • ... • The gradient in temperature value shall be limited to the physical possible value (Delta(Oiltemp)/10ms < x °C). • If the maximum gradient is exceeded for more than y ms, an error shall be raised • This error shall trigger a switchoff reaction. • ... 	
Design Result	Usually some semi formal notation including graphic elements (See Figure 9)	Specification of the Software Safety Requirement in ARTEXT (See Figure 12)
Implementation Result	Manually developed Source Code in C (See Figure 10 and Figure 11)	Automatically generated Source Code in C (See from Figure 13 to Figure 17)

Test	Automated Test shall make sure that (with each release) the reaction is performed. As the temperature signal under consideration is an internal calculated signal, a potential misbehaviour of this signal shall be elicited via fault introduction into the software. No difference between both approaches.
Validation	The result shall be validated in a driving event, with fault injection facility. No difference between both approaches.

3.2.2.3.1 Development of Gradient Checker: Classic Approach

First, the Safe Engineer defines the signals that need to be subjected to gradient checking. The parameters and the output signals are specified in conventional, not strictly formal "language" resulting in a design as can be seen in Figure 9.

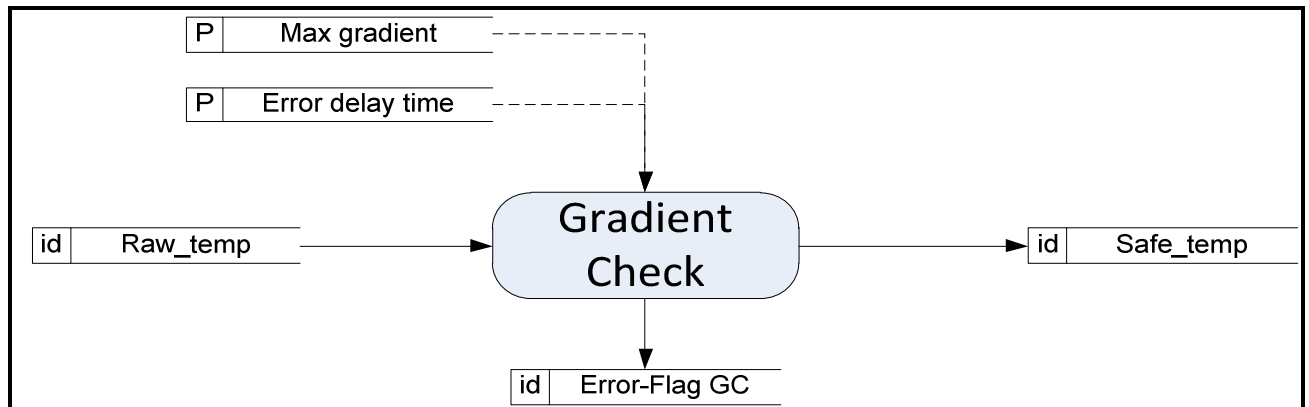


Figure 9 – Design in Classic Development Approach

The SW developer then takes this specification and elaborates the necessary code.

First, a structure is defined containing all data needed per gradient checker.

```
typedef struct
{
    ui8  werte_unguelteig;
    ui8  bit_maske;           // Fault Flag
    ui8  fehler_zaehler;     // Error Counter
    si16 voriger_wert;       // Last Value
    ui16 max_gradient;       // Maximum Gradient
    ui8  entprellung;        // Debounce time
} gradient_struct;
```

Figure 10 – Definition of Gradient Checker Structure (Classic Development Approach)

Second, the SW developer implements the required behaviour in a suitable routine, which is called cyclic every 10ms:

```

static si16 prx_limit_gradient(si16 wert, gradient_struct *parameter )
{
    si16 return_wert = wert;

    if ( parameter->werte_ungueltig == (ui8) 0 )
    {
        ui8 gradienten_fehler = (ui8) 0;
        const ui8 gradient_state = GetVal_prx_gradient_state();
        const si16 differenz = wert - parameter->voriger_wert;
        const si16 differenz_absolut=((differenz < (si16) 0)?(0-differenz):differenz);

        if ( parameter->max_gradient < (ui16) differenz_absolut )
        {
            /* Threshold value exceeded*/
            parameter->fehler_zaebler++;
            if ( parameter->fehler_zaebler > parameter->entprellung )
            {
                gradienten_fehler = (ui8) 1;
                /* avoid overflow */
                parameter->fehler_zaebler--;
            }

            /* Limit to maximum allowed gradient */
            if ( differenz >= 0 )
            { return_wert = parameter->voriger_wert + (si16)parameter->max_gradient; }
            else
            { return_wert = parameter->voriger_wert - (si16)parameter->max_gradient; }
        }
        else
        {
            /* Gradient OK, decrement error counter */
            if ( parameter->fehler_zaebler > (ui8) 0 )
            {
                parameter->fehler_zaebler--;
            }
        }

        /* Handling of Error Flag */
        if ( gradienten_fehler )
        { /* Set Error flag */
            SetVal_prx_gradient_state( side, gradient_state | parameter->bit_maske );
        }
        else
        { /* Reset Error flag */
            SetVal_prx_gradient_state(side, gradient_state & (ui8)~(parameter->bit_maske));
        }
    }
    /* Save value for next cycle */
    parameter->voriger_wert = return_wert;

    ...

    return return_wert;
}

```

Figure 11 – Runnable (Cyclic Routine) (Classic Development Approach)

Main sections of the cyclic routine are:

Check whether gradient is ok

Error debouncing, setting of error flag

Limitation of output value to maximum allowed gradient

Start of error propagation

3.2.2.3.2 Development of Gradient Checker: SAFE Approach

Remark: Long identifiers, that tend to wrap around into the next line, are concatenated using “\”.

In the SAFE approach, the safety engineer first specifies the necessary software safety requirements in the tool prototype based on ARText (see below). In our example, it specifies the expected gradient properties, the error filtering (to mask the error for a limited time interval) and the final error reaction (call the OperationManager).

```

filter lamellenTempFilter {
    previous prev
    current cur
    value = cur
}

SSR ssr1

safeguard sg1 system tvhag2System {
    ssm ssm2 satisfies ssr1 through {
        limit gradient of compTotal :: ptRestECU :: ppSensorTPCFlipSide -> lamellenTemp {
            min := -20.0,
            max := 20.0,
            tolerance := 0.0
            period := 10
        } handle
        {
            GRADIENT_TOO_HIGH -> lamellenTempFilter()
            GRADIENT_N_TIMES_TOO_HIGH(5) -> call compTotal :: ptTVHAG2 ->
                ptBetriebskoordinator :: spBetriebskoordinator ->
                error_gradient_swRestECU_ppSensorTPCOwnSide_lamellenTemp
            GRADIENT_TOO_LOW -> call compTotal :: ptTVHAG2 ->
                ptBetriebskoordinator :: spBetriebskoordinator ->
                error_gradient_swRestECU_ppSensorTPCOwnSide_lamellenTemp
            .
            .
        }
    }
}

```

Figure 12 – Safety Requirement Specification (SAFE Development Approach)

Based on the semi-formal SSR specification, the safety code generator produces required C code pieces as shown in the figures below (struct for filters and gradient properties, filter routine, gradient check routine,

```

typedef struct filter_state_t {
    SInt16 tolerance;
    SInt16 previousValue;
    SInt16 currentValue;
}filter_state;

#define FILTER_EXPRESSION(prev,cur) prev / 5 + cur

```

Figure 13 – Definition of Filter Structure and Filter Expression (SAFE Development Approach)

```
static filter_state filter;

void swcGradientChecker_check_rp_ptRestECU_ppSensorTPCOwnSide_\\
    lamellenTemp_lamellenTempFilter_init () {
    filter.tolerance = FILTER_TOLERANCE;
    filter.previousValue = FILTER_PREVIOUS_VALUE;
    filter.currentValue = FILTER_CURRENT_VALUE;
}

SInt16 swcGradientChecker_check_rp_ptRestECU_ppSensorTPCOwnSide_\\
    lamellenTemp_lamellenTempFilter (SInt16 value) {
    SInt16 _filterResult =FILTER_EXPRESSION(filter.previousValue,filter.currentValue);
    return _filterResult;
}
```

Figure 14 – Filter Routines (SAFE Development Approach)

```

typedef struct
  swcGradientChecker_check_rp_ptRestECU_ppSensorTPCOwnSide_lamellenTemp_gd_state_t {
    SInt16 last_value;
    SInt16 min_value;
    SInt16 max_value;
    int tolerance;
    int time_delta;
    int gradient;
    int error;
}swcGradientChecker_check_rp_ptRestECU_ppSensorTPCOwnSide_lamellenTemp_gd_state;

```

Figure 15 – Definition of GradientChecker Structure (SAFE Development Approach)

```

boolean b_swcGradientChecker_check_rp_ptRestECU_ppSensorTPCOwnSide_lamellenTemp_gcError
    = FALSE;

static swcGradientChecker_check_rp_ptRestECU_ppSensorTPCOwnSide_lamellenTemp_gd_state state;

int swcGradientChecker_check_rp_ptRestECU_ppSensorTPCOwnSide_lamellenTemp_\\
    calculate_gradient (SInt16 current_value,int time_delta) {
    state.gradient = (current_value-state.last_value)*time_delta;
    return state.gradient;
}

int swcGradientChecker_check_rp_ptRestECU_ppSensorTPCOwnSide_lamellenTemp_check_gradient
    (SInt16 current_value) {
    int ret = TRUE;
    int res = swcGradientChecker_check_rp_ptRestECU_ppSensorTPCOwnSide_\\
        lamellenTemp_calculate_gradient(current_value,state.time_delta);
    if ((res * 1000) > (state.max_value * 1000)+state.tolerance) {
        ret = FALSE;
        state.error = ERROR_GRADIENT_TOO_HIGH;
    }
    else if ((res * 1000) < (state.min_value * 1000)-state.tolerance) {
        ret = FALSE;
        state.error = ERROR_GRADIENT_TOO_LOW;
    }
    return ret;
}

/** init runnable */
void swcGradientChecker_check_rp_ptRestECU_ppSensorTPCOwnSide_lamellenTemp_init () {
    state.max_value = MAX_VALUE;
    state.min_value = MIN_VALUE;
    state.tolerance = (TOLERANCE * 1000);
    state.time_delta = 1000/TIME_DELTA;
}

```

Figure 16 – GradientChecker: Service Routines (SAFE Development Approach)

```

/** runnable for autosar */
void swcGradientChecker_check_rp_ptRestECU_ppSensorTPCOwnSide_lamellenTemp () {
    //read current value
    SInt16 currentValue;
    int res;
    Rte_Read_swcGradientChecker_rp_ptRestECU_ppSensorTPCOwnSide_lamellenTemp\
        (&currentValue);
    res = swcGradientChecker_check_rp_ptRestECU_ppSensorTPCOwnSide_\
        lamellenTemp_check_gradient(currentValue);
    if (res == FALSE){
        //set error
        b_swcGradientChecker_check_rp_ptRestECU_ppSensorTPCOwnSide_\
            lamellenTemp_gcError = TRUE;
        if (state.error & ERROR_GRADIENT_TOO_HIGH) {
            SInt16 _filterReturnValue = swcGradientChecker_check_rp_\
                ptRestECU_ppSensorTPCOwnSide_\
                lamellenTemp_lamellenTempFilter
                (currentValue);

            Rte_Write_swcGradientChecker_pp_ptRestECU_\
                ppSensorTPCOwnSide_lamellenTemp(_filterReturnValue);
        }

        if (state.error & ERROR_GRADIENT_TOO_LOW) {
            Rte_Call_swcGradientChecker_cp_ptTVHAG2_\
                ptBetriebskoordinator_spBetriebskoordinator_\
                error_gradient_swcRestECU_ppSensorTPCOwnSide_lamellenTemp();
        }
    }
    else {
        if (b_swcGradientChecker_check_rp_ptRestECU_\
            ppSensorTPCOwnSide_lamellenTemp_gcError) {
            //reset all handling mechanisms used
            b_swcGradientChecker_check_rp_ptRestECU_\
                ppSensorTPCOwnSide_lamellenTemp_gcError = FALSE;
        }
        Rte_Write_swcGradientChecker_pp_ptRestECU_ppSensorTPCOwnSide_\
            lamellenTemp(currentValue);
    }
    state.last_value = currentValue;
}

```

Figure 17 – GradientChecker: Runnable (Cyclic Routine) (SAFE Development Approach)

3.3 Safety Measure “Program Flow Control”

3.3.1 Motivation and Argumentation

3.3.1.1 Development approach before SAFE

In the original product development, the program flow control has been developed manually using C programming language. A set of checkpoints has been defined. The checkpoints are distributed in the code. When a checkpoint is passed, a value is transmitted to a program flow control checker, which calculates a checksum for each 10ms cycle. At the end of a cycle, the actual value is compared to a predefined target value. In case of a deviation, an error handling procedure is triggered, leading to a switch off (fail safe).

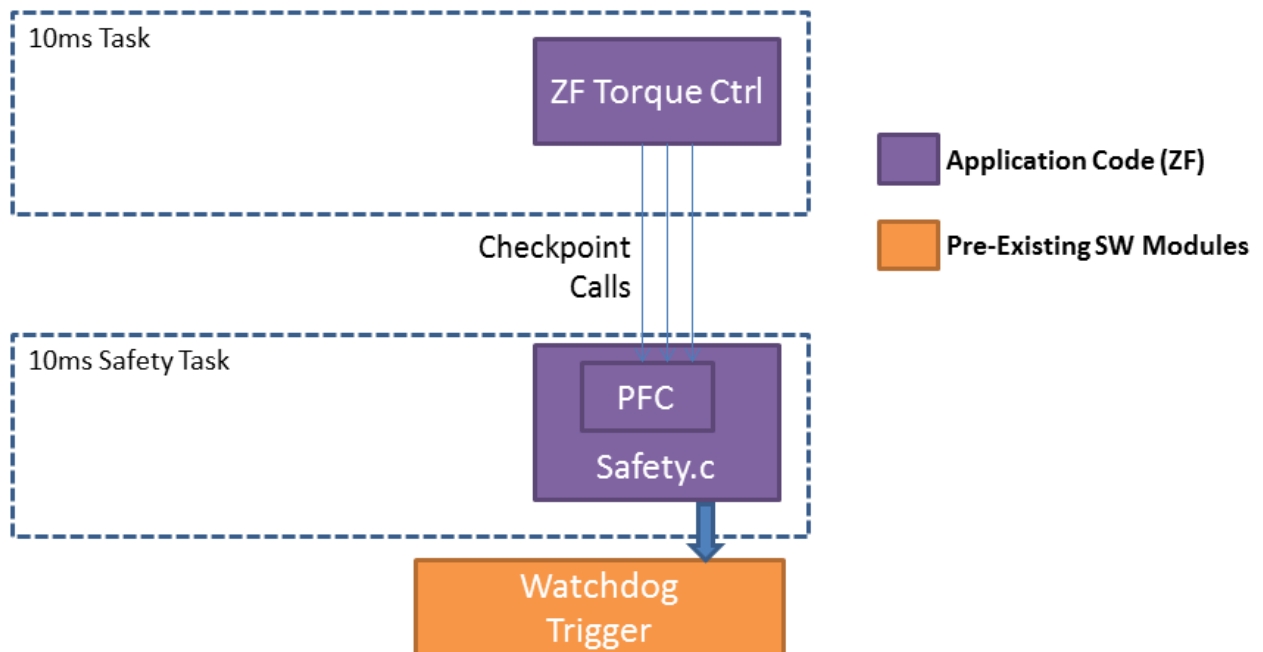


Figure 18 - Architecture of existing PFC

The checker function is implemented in a dedicated safety task that is (by means of task scheduling) activated at the end of a 10ms cycle. In case of a deviation, a WD-Reset is triggered.

3.3.1.2 New Approach

The new approach takes the technical safety concept as given and focuses on the implementation of identified and required safety mechanisms by means of code generation. Based on an abstract model of the evaluator SW, the necessary software safety component (here: PFC checker component) is modeled and integrated into the existing SW while replacing the existing PFC checker component. In AUTOSAR, the control flow is monitored by the watchdog manager service. Thus, the code generation approach derives appropriate configuration for the watchdog manager based on the control flow requirements as specified according to the SAFE meta-model. The AUTOSAR watchdog manager can be accessed via standardized AUTOSAR interfaces by the application software. Thus, the pre-existing software architecture has been restructured to use the AUTOSAR RTE API of the watchdog manager.

The newly introduced software elements realize section 7.1 (Control-Flow Monitor) as well as section 6.4.6 (Filter) in D3.6 [6].

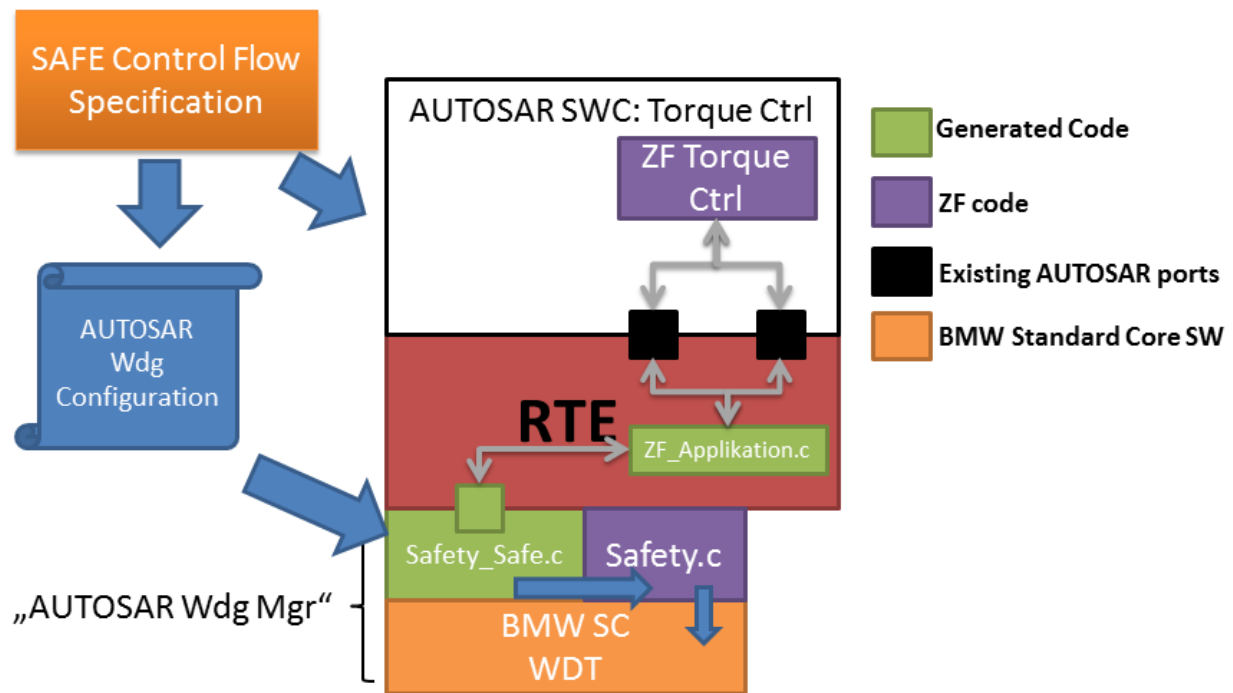


Figure 19 - Architecture of SAFE PFC

The ZF Torque Control components are taken as present, the task schedule (ZF_Applikation.c) is moved to the RTE and generated using the SAFE Control Flow Specification. In this task body, the PFC checkpoint calls are performed, by calling the Client-Server interface of the watchdog service before and after applications RunnableEntities executions. In the evaluation scenario, the newly generated software component "Safety_Safe.c" provides the implementation of the watchdog service. On the one side, this component provides an AUTOSAR compliant API, on the other side it uses existing mechanisms of the ECUs HW/SW stack to implement the expected functionality. In this component, the checker routine is responsible for the detection of deviations from a proper control flow. An asynchronous, periodic main function (in 10ms safety task) of the Safety_Safe.c component triggers a watchdog reset if there is an error observed, by using a new port to the already existing component "Safety.c". This is necessary to keep the interfaces to pre-existing SW modules unchanged.

3.3.1.3 Benefits / drawbacks of new approach compared to current approach

In this use case, the preparatory work to restructure the pre-existing SW according to AUTOSAR interfaces has been substantial. Analysis and implementation of the interfaces were estimated to 20h. But, given the necessary infrastructure, the interaction with the generated software safety components is easy. This holds for the specification phase as well as for code integration. So far, no drawbacks are identified. Details of the evaluation are given in the evaluation section 4.

3.3.2 Implementation of Program Flow Control

Based on a formal specification of the expected execution order of executable entities, the code generator generates the following elements required for the evaluation scenario:

- AUTOSAR watchdog configuration, including the expected checkpoints and transitions to conduct the logical supervision
- Task body, to trigger a) the executable entities in the correct order and b) the watchdog manager with the respective checkpoints (executable entity started/stopped)
- Adapted software architecture, including the required AUTOSAR interfaces to access the watchdog manager via the RTE
- Rudimentary implementation of the AUTOSAR watchdog service (called Safety_Safe.c), as the used basic software stack does not provide an AUTOSAR compliant watchdog implementation.

For more information about the implementation, please refer to the user manual included in the appendix of this document (see 9.1).

3.4 Dependencies

WT 5.6 depends on WT3.6. The artifacts and concepts that are produced in WT3.6 are evaluated in this WT5.6.

3.5 Final Implementation State of the Evaluator

The work task has been organized into 3 steps:

1. Integration Prototype: limited functionality, SW fault injection only. Set up of demonstration environment. (M12 – M18)
2. Extended Prototype: broader functionality, first evaluation of results (M19-M30)
3. Final evaluation of results (M31 – M36)

Main activities in Step 1 and 2 are:

No	Subtask	Status (fulfillment)
1	Generate an understanding of the AUTOSAR style of modeling SW systems at the engineering team at ZF.	100%
2	Generate an understanding of the existing safety concepts and establishment of a conceptual model of the system (FAA) at the research team at Car IT. Result of this activity is a list of building blocks giving an abstract view of the system that makes it accessible for modeling according to SAFE Meta Model.	100%
3	Identification of possible safety mechanisms to be replaced by code generated SW components	100%
4	Specification of communication pattern to integrate the generated SW components into the existing SW running, establishment of contracts and interface specifications.	100%
5	Relevant properties of the system and its subcomponents, i.e. safety requirements, functional components, and applied safety mechanisms are modeled according to the SAFE system meta model.	100%
6	The model is used as input to generate assets necessary to integrate the functional subcomponents.	80%
7	The existing SW will not be rearranged to meet the structure of the model. However the components that are to be replaced by the generated SW components have to be isolated and taken out in order to offer enough resources for the new components.	80%
8	Implementation of communication layer according to the specification identified above.	100%
9	Finally, the system is reintegrated using those generated assets. Test on PC and ECU-HW	100%
10	Identification of resource impacts of the code generated SW components.	100%

Further, the infrastructure necessary for cooperation has been prepared:

- Establishment of a build environment enabling the cooperation between the partners
- Setup of validation environment at BMW Car IT
- Setup of demonstration and test environment at ZF

4 Evaluation Results

In this chapter, the evaluation results of the individual WP3/4/6 deliverables are described.

4.1 Qualitative Evaluation

This section covers the evaluation of the code generation concept with respect to the methodologies already defined in AUTOSAR and the new concepts found in SAFE Project. Furthermore, the more practical aspects are considered when such components are integrated into the surrounding code context.

Evaluation criteria	Qualitative statement	Rationale
Correct and comprehensible documentation	good	User manual for the tool prototype compliant with the SAFE method defined in WT3.6 has been attached to this appendix.
Compliant with SAFE meta-model	perfect	Tool prototype is fully aligned with the SAFE method defined in WT3.6, incl. SAFE meta-model.
Correct implementation of SAFE methods	good	Tool prototype implements a selected list of the SAFE methods defined in WT3.6 (range checker, gradient checker, control-flow monitoring, filtering).
Stability and robustness against incorrect input	not applicable	The tooling is a prototype and thus not developed to be robust against incorrect usage.
Correct and seamless interoperability with other SAFE work products	good	Tool prototype is prepared to link the specification of software safety requirements with technical safety requirements. Generated artifacts of the tooling are completely in line with the AUTOSAR software architecture.
Reasonable support for manual or semi-automated activities	perfect	The main driver of the safety code generation concept is to liberate the safety engineer from manual and error prone activities, by automatically generating safety relevant assets like code, adapted software architecture or argumentation.
Training level and expertise required for usage	not applicable	As the developed tool is only in a prototypic state and solely implemented for the purpose of the evaluation scenario, a qualitative statement is not possible.
Tailoring capabilities	sufficient	The output of the code generator can be configured depending on the required strategy (e.g. how to handle detected errors). Further improvement could be realized (e.g. allowing the safety engineer to judge whether a safety mechanism shall be isolated in a separate software component or be integrated together with other mechanisms into on "safety component").
Ease of Integration	good	The output of the code generator can be easily integrated into the existing code environment via well known interface structures (.h-files)
Implementation complexity	good	The complexity of gradient checker is now hidden to the SW engineer.

4.1.1 Setup of Safety Case

For each safety relevant project, a proper argument must be given that the product is safe enough. This argument must be elaborated during development and must cover (among other aspects) all the different measures that are defined in

functional / technical safety concept. In our case, it must be found an argument why the generated components are operating safely and fulfill the requirements of the technical safety concept.

For this argument, two approaches are possible

Approach 1: Validation of the generated result

- The generated code is reviewed.
- Testing on Code Level
- Check of Code coverage according to ISO26262 Requirements

Approach 2: Validation on Model level

- Validation of Code Generator
- The model (input to the generator) is reviewed
- Testing on Model level

Within WT5.6, approach 1 is evaluated.

4.2 Quantitative Evaluation

In this section, the measurable effects coming with the integration of code generation are evaluated. Focus is set on the critical components in embedded automotive SW development: memory consumption and runtime (processor load). Further, the effort needed to integrate the generated components into the evaluation environment is roughly evaluated and compared to the effort that was necessary in a classic pre-code-generation development approach.

This evaluation tries to elaborate the consumption only for the new concept, not compromised with the effort necessary to make the integration of the generated components possible. This implies that the interface layer emulating an RTE in the evaluation environment is not taken into account.

4.2.1 Integration effort vs. implementation effort

The integration of the generated component was an “ordinary” SW engineering task. According to the different process steps, the effort can be characterized as follows:

Process Step	Remark	Classic Development Approach	SAFE Development Approach
Requirement Analysis	Technical Safety Concept	No difference observed. (Not covered in WT5.6)	
Specification	of SW interfaces, allocation, scheduling, ... Definition of the technical details for Gradient Checker	2h	Effort increased due to frontloading: specification in semi-formal language 4h
Design		2h	2h
Implementation / Integration		20h	4h
Review		4h	1h
Test	Module test based on actual threshold values, must be performed on target ECU-.	No deviation observed.	

4.2.2 Memory Consumption

Evaluation of Memory consumption:

	Classic Approach	SAFE-Approach	Remark
RAM – consumption (static variables)	8	21	SAFE Approach is without optimization
Stack	21	10	

4.2.3 Runtime Effects

It has been found that it is not possible to compare the two approaches thoroughly. The integration of the generated Gradient Checker has made it necessary to restructure the infrastructure (Introduction of RTE-based communication and SW-Components). But it can be stated that no negative effect has been observed.

4.2.4 Safety Case Effort

As defined above, validation method 1 (validation of the generated source code) is chosen. The review effort of the generated code seems to be higher, as the code is more difficult to read and understand than the manually implemented code.

The main problem for a reviewer without AUTOSAR experience is the length of the identifiers (of variables, functions and macros). Identifiers are up to 90 characters long with the significant difference sometimes well hidden in the center of the string. Redundancy in naming is high, but is not facilitating the understanding.

Reading must be very carefully, which is tedious and time consuming and can easily lead to errors.

AUTOSAR experience reduces this problem, but does not avoid it fully.

The overall size of the generated code is 2 – 3 times the size of the manually implemented Code.

Testing and Code Coverage Check is done as usual on application level (re-use of existing fault injection possibilities)

The effort for generating test cases may be smaller as the values from the formal specification may be used to generate test cases.

4.3 Fulfillment of WP 3 requirements

This section summarizes the fulfillment of requirements from work tasks of work packages 3.

Evaluated Requirement	Qualitative Statement	Rationale
WT36_REQ_1	Not evaluated	Not in the scope of the evaluation.
WT36_REQ_2	Not evaluated	Not in the scope of the evaluation
WT36_REQ_3	Complete	The meta-model provides a gradient check element and this has been used to model the gradient check software safety requirement of the evaluation scenario.
WT36_REQ_4	Complete	The meta-model provides the attributes and these have been used to define properties of the gradient check requirement.
WT36_REQ_5	Complete	The meta-model provides means to specify the periodicity of the gradient check and this has been used to define this attribute in the evaluation scenario.
WT36_REQ_6	Complete	The meta-model provides this element and it has been used to model the range check software safety requirement of the evaluation

		scenario.
WT36_REQ_7	Complete	The meta-model provides this element and it has been used to model the necessary information for the range check.
WT36_REQ_8	Complete	The meta-model provides this element and it has been used to define the necessary information for the range check.
WT36_REQ_9	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_10	Complete	The meta-model provides the necessary attributes and the range check ranges have been defined using these attributes.
WT36_REQ_11	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_12	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_13	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_14	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_15	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_16	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_17	Complete	The requirement is being evaluated in the context of a control-flow monitor. A control-flow monitor has been modeled using the available meta-model elements.
WT36_REQ_18	Complete	The necessary checkpoints have been defined using the available meta-model elements.
WT36_REQ_19	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_20	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_21	Complete	The attributes regarding tolerances for arriving at checkpoints have been defined using the meta-model elements and are used for monitoring by the control-flow monitor.
WT36_REQ_22	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_23	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_24	Complete	The requirements modeled for the evaluation scenario can be traced to the realization of the requirement.
WT36_REQ_25	Complete	Gradient checker and range check code has been generated automatically and has been integrated into the validation environment of the evaluation scenario.
WT36_REQ_26	Complete	(see WT36_REQ_25)
WT36_REQ_27	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_28	Complete	Attributes from the gradient check and range check requirements, e.g., periodicity, are modelled using value-pair elements of the meta-model.
WT36_REQ_29	Complete	(see WT36_REQ_28)
WT36_REQ_30	Complete	(see WT36_REQ_24)
WT36_REQ_31	Complete	(see WT36_REQ_24)
WT36_REQ_32	Complete	(see WT36_REQ_24)
WT36_REQ_33	Complete	(see WT36_REQ_24)
WT36_REQ_34	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_35	Complete	The software safety requirements can define error handling requirements and the relation between detection scenarios and

		handling scenarios.
WT36_REQ_36	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_37	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_38	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_39	Complete	The software safety requirements can define which errors are produced by the detection procedure.
WT36_REQ_40	Complete	The software safety requirements can define which reactions shall be executed in case errors are detected.
WT36_REQ_41	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_42	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_43	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_44	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_45	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_46	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_47	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_48	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_49	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_50	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_51	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_52	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_53	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_54	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_55	Complete	(see WT36_REQ_40)
WT36_REQ_56	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_57	Not evaluated	The requirement is not in the scope of the evaluation scenario.
WT36_REQ_58	Not evaluated	The requirement is not in the scope of the evaluation scenario.

4.4 Final quantification of Safety Code Generation as evaluated in WT5.6:

Performance:

- Level: 3
- Rationale: A significant improvement has been achieved, but not all criteria are met

Interest:

- Level: 3
- Rationale: Software Safety Measures can be generated automatically. Using this approach, the overall complexity of safety engineering measures is not reduced (specification effort is similar or even slightly increased). But the approach significantly reduces the complexity of the implementation. Further, the tracing of measures is supported.

		Performance				
		1	2	3	4	5
Interest	4	4	8	12	16	20
	3	3	6	9	12	15
	2	2	4	6	8	10
	1	1	2	3	4	5
	0	0	0	0	0	0

5 Conclusion

The results of WT 5.6 indicate that the effort in implementation is reduced by the introduction of code generation, so the effect is positive. It is also possible to extend the approach to legacy systems that have solved safety requirements without MBSE, AUTOSAR, and code generation. However, the mere generation of software safety mechanisms is not the main effort driver in the development of a safety relevant SW system. The automation of such higher-level activities has been investigated in a thought-experiment described in [21] where not only mechanisms could be generated but also suggested to safety engineers in order to realize system safety based on the system architecture. Such an approach can be investigated in future research projects.

The generated components can be integrated into the pre-existing source code. The effort for the preparation of the environment depends on the complexity of the measures under consideration, Single check routines are easy to replace.

If AUTOSAR is already in place, the infrastructure will allow an easier integration of the generated code, as the structure (SWCs, RTE) is present. This situation was not validated in WT5.6.

In general, approaches involving the use of new tools or technologies - such as the semi-formal specification language used during this evaluation - require learning efforts from engineers. This might slow down development throughput during this learning phase. However, given that the approach can be applied to legacy and new projects, the effort can be distributed over the lifecycle of safety activities for all projects to which the approach is applied. Furthermore, once the safety requirements for safety mechanisms are formally defined, this information can be used to support other engineering steps such as validation activities and test-case generation.

6 References

- [1] SAFE Requirements
https://safe.offis.de/svn/svndav/40_Deliverables/SAFE_D2.1.a/SAFE_D2.1.a.pdf
- [2] SAFE Report of WT3.6
https://safe.offis.de/svn/svndav/33_WP3_Model_Based_Development/WT3_6_Safety_Code_Generation/Deliverables/SAFE_WT3.6_Report.doc
- [3] Classification of safety mechanisms
https://safe.offis.de/svn/svndav/33_WP3_Model_Based_Development/WT3_6_Safety_Code_Generation/Documents/safetymechanismclustering/safetymechanismclustering.eap
- [4] SAFE Risk List
https://safe.offis.de/svn/svndav/10_Project_Management/SAFE_Plus-Minus-Risks.xlsx
- [5] SAFE FPP
https://safe.offis.de/svn/svndav/10_Project_Management/FPP/!Actual_Official_Version/SAFE_FPP.docx
- [6] SAFE Deliverable D3.6.b: Safety Code Generator Specification
https://safe.offis.de/svn/svndav/33_WP3_Model_Based_Development/WT3_6_Safety_Code_Generation/Deliverables/D3.6.b/SAFE_D3.6.b.doc
- [7] SAFE_D2.1.a-ISO-Part_2.pdf (Management of functional safety)
- [8] SAFE_D2.1.a-ISO-Part_3.pdf (Concept Phase)
- [9] SAFE_D2.1.a-ISO-Part_4.pdf (Product development at the system level)
- [10] SAFE_D2.1.a-ISO-Part_5.pdf (Product development at the hardware level)
- [11] SAFE_D2.1.a-ISO-Part_6.pdf (Product development at the software level)
- [12] SAFE_D2.1.a-ISO-Part_7.pdf (Production and operation)
- [13] SAFE_D2.1.a-ISO-Part_8.pdf (Supporting Processes)
- [14] SAFE_D2.1.a-ISO-Part_9.pdf (Automotive Safety Integrity Level (ASIL)-oriented safety-oriented analysis)
- [15] ISO/FDIS 26262 parts 2-9: 2011.
- [16] ARTop www.artop.org
- [17] AUTOSAR Specification of Timing Extensions
http://www.autosar.org/download/R4.0/AUTOSAR_TPS_TimingExtensions.pdf
- [18] ARText www.artop.org/artext
- [19] AUTOSAR – Specification of RTE http://www.autosar.org/download/R4.0/AUTOSAR_SWS_RTE.pdf
- [20] AUTOSAR – Virtual Function Bus http://www.autosar.org/download/R4.0/AUTOSAR_EXP_VFB.pdf
- [21] Trindade, R., Bulwahn, L. and Ainhauser, C. (2014). Automatically Generated Safety Mechanisms from Semi-Formal Software Safety Requirements. *Computer Safety, Reliability, and Security*, pp.278-293.

7 Acknowledgments

This document is based on the SAFE project in the framework of the ITEA2, EUREKA cluster program Σ! 3674. The work has been funded by the German Ministry for Education and Research (BMBF) under the funding ID 01IS11019, and by the French Ministry of the Economy and Finance (DGCIS). The responsibility for the content rests with the authors.

8 Common Metrics for evaluation

For each work product, a metric *performance* will be setup rating how well the expectations given in the work product description have been met.

Level 5: beyond expectations described in the Full Project Proposal and evaluation criteria

Level 4: expectation from Full Project Proposal and good level evaluation criteria met

Level 3: expectations not fully met or some evaluation criteria not reached sufficient level but significant improvement achieved

Level 2: no significant improvement achieved or some evaluation criteria are rated incomplete

Level 1: negative impact (performance degraded) and all evaluation criteria are incomplete

This evaluation will be crossed with a metric *industrial interest* qualifying the relevance of the method (or tool or methodology, respectively) covered by the corresponding evaluation scenario.

Level 4: Interesting for evaluation scenario and ready for application in the field

Level 3: Interesting for evaluation scenario but needs to be slightly matured for application in the field

Level 2: Interesting for evaluation scenario but needs to be significantly matured for application in the field

Level 1: Not of interest for the specific evaluation scenario but interesting anyway for application in the field (not considered further for project evaluation – no detailed evaluation result available)

Level 0: Out of scope of evaluation scenario, not of interest for application in the field.

Thus, a graphical representation can be provided for each evaluated work product which gives an interpretation of the industrial potential of the latter.

		Performance				
		1	2	3	4	5
Interest	4	4	8	12	16	20
	3	3	6	9	12	15
	2	2	4	6	8	10
	1	1	2	3	4	5
	0	0	0	0	0	0

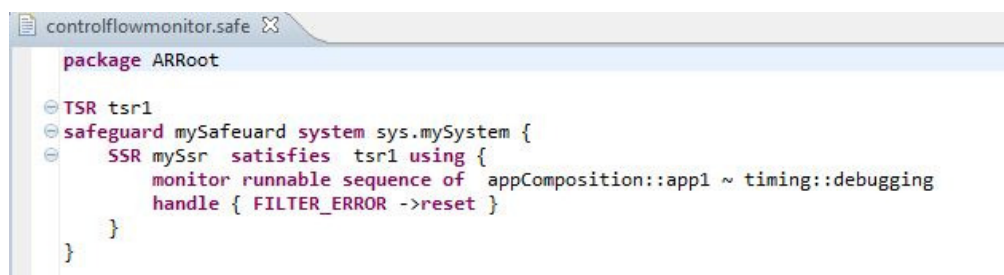
9 Appendix

9.1 Tool prototype “control flow monitor”

This section provides a basic user manual for the tool prototype used to generate the software part for the Program Control Flow as described in section 3.3.2

In order to deal with safety requirements to detect and handle improper control flow in the safety relevant application code, the tool prototype documented here allows the specification of software safety requirements for control flow monitoring and generates required artifacts to automatically satisfy the specified requirements.

The software safety requirements (SSR) ensuring the correct behavior of the safety relevant application software can be specified equivalent to the means of the SAFE meta-model. An exemplary code snippet to capture such a SSR by the safety engineer is shown in Figure 20:



```

package ARRoot
{
  TSR tsr1
  safeguard mySafeguard system sys.mySystem {
    SSR mySsr satisfies tsr1 using {
      monitor runnable sequence of appComposition::app1 ~ timing::debugging
      handle { FILTER_ERROR ->reset }
    }
  }
}

```

Figure 20 - Control flow specification based on RunnableEntity execution chaining

The definition of such a requirement consists of three main elements: an application SW architecture using the rudimentary tree editor in Artop [16], a textual modeling language and a generation engine. A snippet of example SW architecture is shown in Figure 21[17].

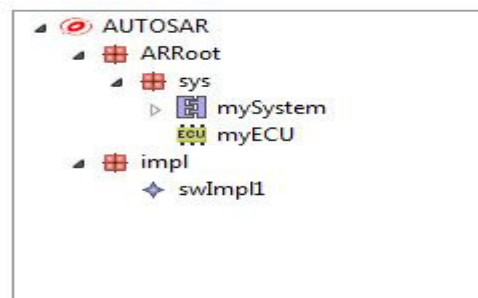


Figure 21 - Software modeled in Artop

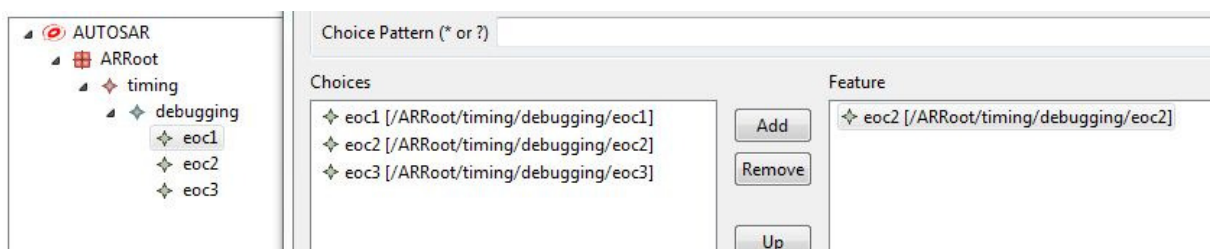


Figure 22 - RunnableEntity execution order modeled in Artop

Figure 20 shows safety requirement for control flow monitoring using textual language. A software safety requirement statement has the syntax: **ssr** <name> **satisfies** <link to technical safety requirement (tsr)> **using** { <expectation > **handle** <reaction>}. The <expectation> describes the behavior of control flow monitoring of a given element. Inside <expectation> AUTOSAR package “timing” holds the ExecutableEntity execution order constraint “debugging” that specifies an expected sequence of ExecutableEntities. Figure 22 shows, how the referenced AUTOSAR execution order constraint is defined via the means of Artop [16]. The <reaction> handles the error in case of failure. Please refer to the documentation of the AUTOSAR Timing Extensions [17] for more information how to specify execution order constraints.

According to this information, the tool prototype is able to perform code generation required for the purpose of the TV-HAG2 evaluation scenario. During code generation tool prototype generates different things such as adapted SW architecture, watchdog configuration, task body and necessary code to reach BSW (in this case [watchdog manager](#)) from the SWC to query if a certain checkpoint is already reached. Through this code the SWC itself can trigger some mechanism such as debounce or reset for the TV-HAG2 system.

```

component application appControlflow {
  ports {
    sender providerPort provides controlflowSRIF
    receiver requiredPort requires controlflowSRIF
    server clientserverProvidedPort provides controlflowCSIF
  }
}
composition appComposition {
  prototype appControlflow app1
  prototype appControlflow app2
  connect app1.providerPort to app2.requiredPort as conn1
  connect app2.providerPort to app1.requiredPort as conn2
}

internalBehavior app_ib for appControlflow {
  dataTypeMappings {
    myDataTypeMappingSet
  }
  runnable init [0.0] {
    symbol "controlflow_init"
    timingEvent 10.0
  }

  runnable readRbl [0.0] {
    symbol "swcControlFlowMonitor_read_runnable"
    timingEvent 10.0
  }
  runnable writeRbl [0.0] {
    symbol "swcControlFlowMonitor_write_runnable"
    dataSendPoint providerPort.srDataElement
    timingEvent 10.0
  }
}

```

Figure 23 - Autosar SWC described in ARText [18]

Once the requirement for generation is ready the following generated artifacts will follow using the tool platform.

9.1.1 Adapted SW architecture

During this phase Service Software Component is generated. The component provides the required API to initialize the watchdog manager and to invoke the watchdog manager with reached checkpoint. Based on this information, the RTE generator is able to generate the RTE API for the watchdog manager and based on the assembly connectors wiring the port prototypes of the [watchdog manager](#) (in this case, port “providerPortFor_appComposition_app1”), the RTE does the wiring between application software and watchdog manager.

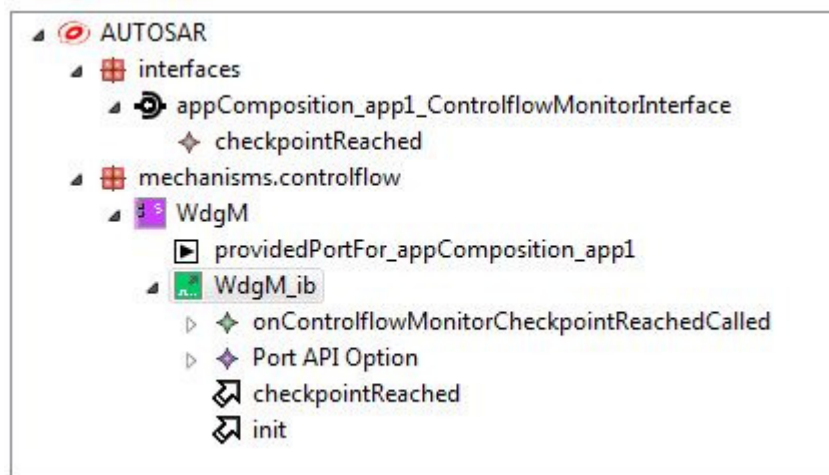


Figure 24 - Adapted SW architecture for control flow monitoring

9.1.2 Watchdog configuration

It generates a [watchdog manager](#) configuration where the right checkpoint definition is contained automatically, based on the control flow software safety requirement. A list of transition is also generated based on the execution order of the executable entities related with the SSR. A code snippet for [watchdog manager](#) configuration is shown in Figure 25.

```

module WdgM appComposition_app1_UsingRunnableSequence {
  WdgMConfigSet appComposition_app1_wdgMConfigSet {
    WdgMMode appComposition_app1_SE {
      WdgMModeId = 1
    }
    WdgMInitialMode = appComposition_app1_SE
  }
  WdgMGeneral appComposition_app1_wdgMGeneral {
    WdgMSupervisedEntity appComposition_app1_SupervisedEntity {
      WdgMSupervisedEntityId = 0
      WdgMCheckpoint init_Start {
        WdgMCheckpointId = 1
      }
      WdgMCheckpoint init_Stop {
        WdgMCheckpointId = 2
      }
      WdgMCheckpoint readRbl_Start {
        WdgMCheckpointId = 3
      }
      WdgMCheckpoint readRbl_Stop {
        WdgMCheckpointId = 4
      }
      WdgMCheckpoint writeRbl_Start {
        WdgMCheckpointId = 5
      }
      WdgMCheckpoint writeRbl_Stop {
        WdgMCheckpointId = 6
      }
      WdgMInternalCheckpointInitialRef = init_Start
      WdgMInternalCheckpointFinalRef = writeRbl_Stop
      WdgMInternalTransition init_to_readRbl {
        WdgMInternalTransitionSourceRef = init_Stop
        WdgMInternalTransitionDestRef = readRbl_Start
      }
      WdgMInternalTransition readRbl_to_writeRbl {
        WdgMInternalTransitionSourceRef = readRbl_Stop
      }
    }
  }
}

```

Figure 25 - Generated watchdog configuration using SAFE tool for Control flow monitoring

9.1.3 Task body

It is a contract between the application software and the basic software running inside the ECU that is triggered by the OS. An example Task body is shown in Figure 26.

This is an optional generator output for the purpose of the TV HAG2 evaluation scenario. To free the application software from watchdog manager calls (i.e. checkpointReached calls), the task body invokes the watchdog manager before starting the RunnableEntities and after its termination.

```
* Name      : appComposition_app1_Taskbody.c
#include "AppComposition_App1_Taskbody.h"
#include "AppComposition_App1.h"

void ZF_SafetyControlflow_For_appComposition_app1() {
    WdgM_CheckpointReached(init_Start);
    controlflow_init();
    WdgM_CheckpointReached(init_Stop);
    WdgM_CheckpointReached(readRbl_Start);
    swcControlFlowMonitor_read_runnable();
    WdgM_CheckpointReached(readRbl_Stop);
    WdgM_CheckpointReached(writeRbl_Start);
    swcControlFlowMonitor_write_runnable();
    WdgM_CheckpointReached(writeRbl_Stop);
}
```

Figure 26 – Task body for Control flow monitoring

9.1.4 Source code for watchdog service

As the TV-HAG2 evaluation scenario does not provide a compliant AUTOSAR watchdog manager, the tool prototype generates a simple AUTOSAR watchdog manager implementation which a) complies with the AUTOSAR standard interface of the watchdog manager and b) provides the rudimentary features required by the evaluation scenario. It does not claim to be a full-featured AUTOSAR watchdog manager implementation.