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

WP3

Deliverable D3.6.b: Safety Code Generator Specification

Due date of deliverable: 31/12/2013

Actual submission date: 20/12/2013

Start date of the project: 01/07/2011 **Duration:** 36 months

Project coordinator name: Stefan Voget

Organization name of lead contractor for this deliverable: BMW Car IT GmbH

Editor: Raphael Trindade

Contributors: Christoph Ainhauser, Raphael Trindade, Vladimir Rupanov

Reviewer: Arthur Gauthier – Dassault Systèmes

Revision chart and history log

Version	Date	Reason
0.1	05.10.2013	Initial draft
0.2	05.11.2013	Added sections related to CHROMOSOME contribution
0.3	15.11.2013	Review of document
0.4	21.11.2013	Iteration over reviewer feedback
0.5	26.11.2013	Final version draft
0.6	06.12.2013	Review of document
0.7	13.12.2013	Final version

1 Table of contents

2 List of figures

3 Executive Summary

This deliverable describes in detail the requirements regarding the generation of code for software safety mechanisms. There are two main targets for code generators within WT3.6: AUTOSAR [1] and CHROMOSOME [2]. For each of these targets there are elements which are realized using C code and elements which are realized using specific formats related to each technology (e.g. ARXML for AUTOSAR). The details about code and additional information to be generated by code generators implemented in the context of the SAFE project are specified for central meta-model elements.

The main goal of this document is to provide a solid base of information regarding the mapping of meta-model elements to elements pertaining to the target technology supported by specific generator implementations. For given elements examples of possible realizations (code and target technology information) are provided. These examples shall only be used as guidance and do not strictly specify how given meta-model elements are to be generated.

Code generators implemented according to SAFE must comply with the semantic specification of safety mechanisms given in this document. However, if the implementation regards the detailed mechanism generation description in sections 7 and 8, code generators can deviate from the standard structure defined in this document provided a rational for it.

4 Introduction

The development of systems for the automotive domain functionality is either realized as software elements, hardware elements or both. The development of safety critical systems for the automotive domain implies additional requirements for the realization of functions which in turn can influence how software and hardware elements are developed. In this document the development of software based functionality in compliance to the ISO 26262 is addressed regarding the ISO 26262 part 6 requirements allocated to the product development phase at the software level.

According to the ISO 26262, during the software development phase one of the first tasks to be executed is the definition of software safety requirements (SSR). These requirements shall be derived from the technical safety concept, more concretely technical safety requirements used as input for this phase. Therefore before automatically generating software safety mechanisms it is important to be able to specify such SSRs.

In WT3.6 software safety requirements have been assumed to be the starting point for the implementation of automatic generators. These requirements usually express patterns regarding the safety measures to be applied on a given software system in order to realize the specified technical safety concept. The approach taken by WT 3.6 is to require generators to be developed based on a formal specification of specialized software safety requirements. In the case of SAFE this formalism is the SAFE meta-model.

The SSRs provide the necessary information for the generation of software safety mechanisms. Within WT 3.6 the realization of software safety mechanisms (SSM), namely their implementation as architectural elements or C code, is seen as the fulfillment of SSRs.

By processing the specified SSRs it shall be possible to automatically generate SSMs and to generate additional information regarding the traceability link between generated SSMs and the originating SSRs. This link back to the specification of safety related elements allows the traceability requirement to be fulfilled and a complete chain linking implementation to specification to be achieved. The SAFE meta-model provides a construct for managing the artifacts obtained via generative approaches for SSR realization.

4.1 Scope of WT 3.6

In the context of work package 3 – *Model based development for functional safety*, work task 3.6 is responsible for the identification of architectural and software patterns for functional safety measures and furthermore for the evaluation of how suitable generative approaches can be for the automatic realization/implementation of such architectural and software based measures. The results obtained within the work task are condensed in this deliverable and encompass:

- The identified software and architectural patterns commonly used in the realization of technical safety concepts
- The specification of the information required for the realization/implementation of evaluated patterns
- The requirements on tools realizing generative approaches for the implementation of such patterns.

4.2 Structure of this document

In the next sections the details regarding the meta-model elements of the software safety requirement specification language and the required contract between meta-model and safety code generators are defined. In section 5 the approach for modeling SSRs and generating software safety mechanisms proposed by WT 3.6 is described in depth and when possible examples regarding the realization of generators is given. Section 6 provides a detailed view on SSR and SSM semantics, on the required information for the generation of SSMs and the on mapping of SSRs to SSMs. Sections 7 and 8 provide a detailed specification for a subset of the mechanisms described in this document. Section 9 provides an overview on how the generated artifacts can be validated and tested. Finally Section 10 presents the achieved goals of WT3.6 regarding safety code generation and software safety requirements.

5 Proposed approach to model software safety requirements and generate software safety mechanisms

In this section the details of the approach proposed by WT3.6 for the specification of software safety requirements and the generation of software safety mechanisms are described. First the modeling of software safety requirements approach used in WT3.6 is described. Moreover, the approach taken for the realization of generators based on the modeling formalism for SSRs is presented. Finally examples for the specification and generation regarding the two main target technologies considered within WT3.6 (AUTOSAR and CHROMOSOME) are given.

5.1 Modeling

The safety requirements specific to software elements of an item are commonly refined up to the point where concrete implementations of functionality fulfilling these requirements can be provided by software engineers. For example, the functional safety concept required by the ISO 26262 is done taking into account elements of the preliminary architecture of the item. As the development moves forward more concrete concepts are defined and finally requirements are specified for concrete item configurations (hardware and/or software). In the case of WT 3.6 these are software safety requirements and they are related to the concrete software architecture and software elements providing functionalities to the item.

Ultimately the modeling formalism used to specify SSR is the SAFE meta-model. The SAFE meta-model is to be considered the basis for the exchange format between the different tools used during the safety lifecycle of a product. However the SAFE meta-model does not provide syntactic sugars and does not worry about ease of use on the part of the user (engineer) modeling SSRs. For this reason it might be interesting to allow SSRs to be modeled using a formalism which better suits specific situations.

Usually software safety requirements will be expressed or defined for concrete system software architectures. Therefore it is very beneficial if the specification of such SSRs is able to be integrated to the formalism used to define the system software architecture. For example, if an UML composite structure diagram is being used to model the item architecture ports and interfaces the formalism for specifying SSRs could be an UML profile. Afterwards the formalism used to model SSRs can be automatically transformed into a SAFE model.

The recommendation from SAFE regarding the modeling of software safety requirements is that modeling should happen with the support of a modeling tool which provides a more specialized language (e.g. textual language) which directly relates to the modeling context (e.g. requirements on items designed using CHROMOSOME). The models created with such a language should be transformed into the SAFE exchange format and this could in turn be used by safety code generators. A more detailed example of how such an approach would look like is given in Section 5.3.

5.2 Generation

In order to generate software safety mechanisms generators implemented based on SAFE shall support the processing of SAFE conformant input, namely, software safety requirement specifications in the SAFE exchange format.

5.2.1 Workflow

The SAFE meta-model provides so far three possibilities for code generators to process specifications:

1. Process specifications in an agnostic way where the software safety requirements are simply SAFE SSRs and are not related to any target technology. In this case the generator must be able to interpret generic mechanisms. The consequence of this approach is that a lot of information related to the target technology platform might be missing. One possible solution would be to add the missing information as configuration information for given SSR specifications. For example, an agnostic gradient checker could be configured with information regarding AUTOSAR and automatic code generation would then be possible.

2. Process specifications in which SSRs are specialized into more concrete SSRs (e.g. a built-in self-test requirement). In this case it is clear for the generator which kind of software safety mechanism has to be generated for the SSR.

3. Process SSR specifications which define a concrete SSR for a given target technology (e.g. an AUTOSAR alive monitor). Hence, the generator is provided with information necessary for generating and integrating the generated artifacts into preexisting models and code.

Regarding the generation of artifacts the generators implemented according to this specification can either take as basis for the generation the SAFE meta-model or any intermediary representation derived from the SAFE meta-model. It is common sense in the model driver development community to base code generators on intermediate representations this is also the recommendation of WT 3.6. The code generators shall allow the users to configure output targets for the different types of artifacts being generated this provides the flexibility of integrating generated artifacts to preexisting artifacts (e.g. project folders).

5.2.2 Generated artifacts

Which artifacts are generated based on the software safety requirements specification using SAFE depend on the what kind of requirement is specified, what target technology is used and what artifacts already exist. There are four main types of artifact which are usually generated, these are: code, models, test data (unit tests, interface tests, etc.) and traceability information. These artefacts are addressed in next sections.

5.2.2.1 Code

Whenever *code* is to be generated, requirements regarding the use of the generated code shall be taken into account (e.g. ASIL level). Code generators must, for instance, state according to which standard is the code generated. For example a code generator producing C *code* would state it generates MISRA C [3] compliant code if that is the case. This allows tool users to provide this information whenever proof of compliance is necessary. Furthermore naming conventions for generated code shall be defined. Depending on the situation the conventions can come from the target technology being adopted.

Moreover, the recommendations of the ISO 26262 regarding code shall also be followed, namely: low code complexity, strong typing, naming conventions, hierarchical structuring, cohesion and coupling, etc. For detailed information please refer to the ISO 26262 part 6 – Product development at the software level [4].

5.2.2.2 Model

Besides generating code, depending on the target technology models also have to be generated. The generated models must also be compliant to the recommendations made by the ISO 26262 regarding software development. Furthermore, it might be the case that preexisting artifacts have to be modified or adapted. In this case the original artifact shall not be modified rather a copy shall be made and afterwards adapted to include the generated information. The generated artifacts must be differentiated from preexisting ones even in the case of adaptation. One of the possibilities is to achieve this through the traceability structure provided by the SAFE meta-model. It would also be possible to generate new and adapted artifacts into a new project which consists of an adapted copy of the original project. The concrete integration of generated artifacts can be defined by each safety code generation implementation. The main requirement is that tool users must be able to differentiate generated and adapted artifacts from preexisting ones.

5.2.2.3 Test data

Additionally to code and model data, for given scenarios it is also possible generate test data. The necessary information for this step can be obtained from the specified SSR and from additional generator configuration information. For instance, given an SSR specifying a gradient check of a given interface a unit test for the generated code can be generated. The parameters for the test are the ones specified in the SSR.

5.2.2.4 Traceability Information

Regardless what kind of technology being used to realize SSR traceability information about what was generated and where it was generated has to be provided by code generators. This information has to be persisted within the SAFE meta-model and the mechanism to realize this is provided via *Satisfy* feature from EAST-ADL [6] depicted in Figure 1. The traceability information encompasses code, model elements, models, configuration files, metrics, tests and any other information/artifact which somehow influences the software system.

The traceability information is responsible for linking the generated artifacts to their specification. Furthermore it is of great importance to be able to identify and trace generated artifacts. The identification allows engineers to determine which additional elements belong to the design after generation has taken place. Moreover such traceability is also important for the generation of evidence of compliance for certification purposes.

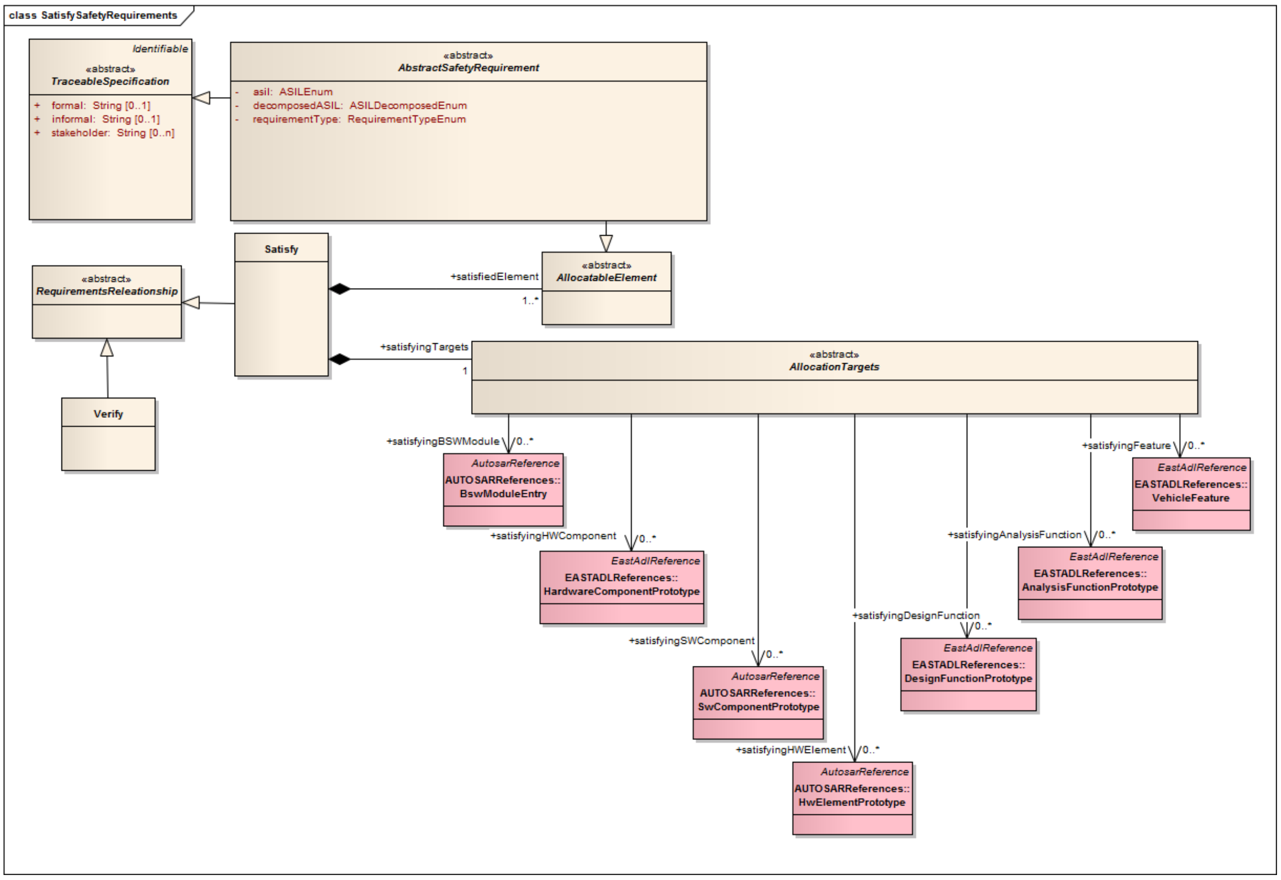

class SatisfySafetyRequirements
Identifiable
«abstract»
TraceableSpecification
+ formal: String [0..1]
+ informal: String [0..1]
+ stakeholder: String [0..n]
«abstract»
AbstractSafetyRequirement
- asil: ASILEnum
- decomposedASIL: ASILDecomposedEnum
- requirementType: RequirementTypeEnum
Satisfy
«abstract»
RequirementsReleationship
Verify
+satisfiedElement
1..*
«abstract»
AllocatableElement
+satisfyingTargets
1
«abstract»
AllocationTargets
+satisfyingBSWModule
0..*
AutosarReference
AUTOSARReferences::BswModuleEntry
+satisfyingHWComponent
0..*
EastAdlReference
EASTADLReferences::HardwareComponentPrototype
+satisfyingSWComponent
0..*
AutosarReference
AUTOSARReferences::SwComponentPrototype
+satisfyingHWElement
0..*
AutosarReference
AUTOSARReferences::HwElementPrototype
+satisfyingDesignFunction
0..*
EastAdlReference
EASTADLReferences::DesignFunctionPrototype
+satisfyingAnalysisFunction
0..*
EastAdlReference
EASTADLReferences::AnalysisFunctionPrototype
+satisfyingFeature
0..*
EastAdlReference
EASTADLReferences::VehicleFeature

Figure 1 – SAFE Meta-model *Satisfy* mechanism for tracing realization to requirements

5.3 Target Platforms

The specification of software safety requirements and the (semi-)automatic generation of software and model elements depend on the adopted target platform. In the next sections details regarding the modeling of SSR and generation of software related artifacts for different target platforms are given.

5.3.1 AUTOSAR

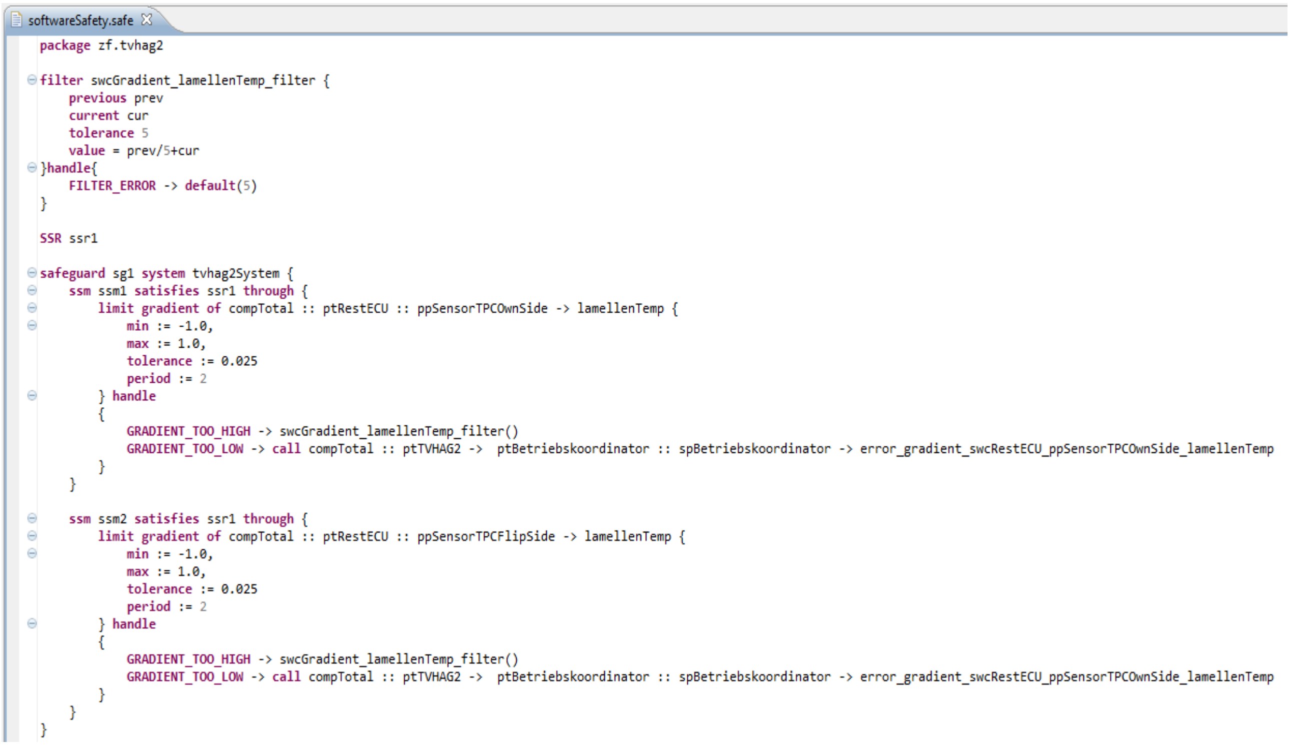
In this subsection an overview regarding modeling SSRs and generating SSMs for AUTOSAR is given. The goal is not to describe in detail how an AUTOSAR generator shall work, but rather to provide general information regarding the generation of software safety mechanisms which have AUTOSAR as target technology.

The goal of modeling software safety requirements for AUTOSAR architectures has been defined in SAFE using a loosely coupled approached. The modeling of SSRs shall not require any changes in the AUTOSAR meta-model. This is achieved by providing a safety view on top of preexisting AUTOSAR artifacts namely either the VFB [7] architecture or a concrete system model.

As previously state, a language allowing the modeling of SSR is preferred. For example, in order to integrate the safety view into AUTOSAR a domain specific language (DSL) could be developed. This language would allow users to directly reference existing AUTOSAR elements and to define software safety requirements for these elements.

The software safety requirements are defined according to the SAFE meta-model. However the original format for SSR specifications can vary from tool to tool. Figure 2 presents an example of how such a DSL (based on ARText [8] for illustration) could be implemented.

The SSR specification is decouple from the AUTOSAR system architecture specification. The referencing to existing AUTOSAR elements is allowed via the meta-model. Figure 3 presents the exemplary AUTOSAR model referenced in Figure 2.

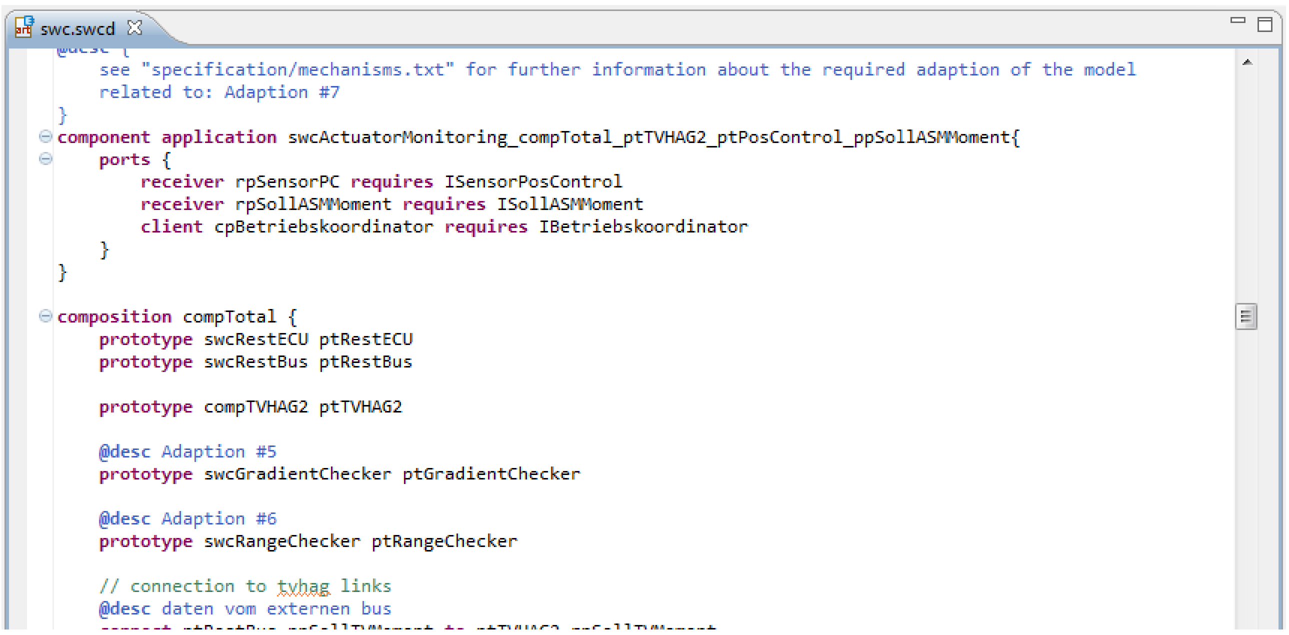


```
package zf.tvhag2

filter swcGradient_lamellenTemp_filter {
    previous prev
    current cur
    tolerance 5
    value = prev/5+cur
}handle{
    FILTER_ERROR -> default(5)
}

SSR ssr1

safeguard sg1 system tvhag2System {
    ssm ssm1 satisfies ssr1 through {
        limit gradient of compTotal :: ptRestECU :: ppSensorTPCOwnSide -> lamellenTemp {
            min := -1.0,
            max := 1.0,
            tolerance := 0.025
            period := 2
        } handle
        {
            GRADIENT_TOO_HIGH -> swcGradient_lamellenTemp_filter()
            GRADIENT_TOO_LOW -> call compTotal :: ptTVHAG2 ->  ptBetriebskoordinator :: spBetriebskoordinator -> error_gradient_swcRestECU_ppSensorTPCOwnSide_lamellenTemp
        }
    }

    ssm ssm2 satisfies ssr1 through {
        limit gradient of compTotal :: ptRestECU :: ppSensorTPCFlipSide -> lamellenTemp {
            min := -1.0,
            max := 1.0,
            tolerance := 0.025
            period := 2
        } handle
        {
            GRADIENT_TOO_HIGH -> swcGradient_lamellenTemp_filter()
            GRADIENT_TOO_LOW -> call compTotal :: ptTVHAG2 ->  ptBetriebskoordinator :: spBetriebskoordinator -> error_gradient_swcRestECU_ppSensorTPCOwnSide_lamellenTemp
        }
    }
}
```

Figure 2 – Gradient check specified using an SSR DSL based on ARText

In the case of AUTOSAR, a safety code generator could transform each group of software safety mechanisms of the same type into an AUTOSAR software component. The realization of the internal behavior of this software component is generated according to information obtained from the preexisting AUTOSAR system model. The deployment of software components to ECU instances is used to define where the source files implementing the internal behavior of the component shall be generated.

Figure 4 shows the AUTOSAR software component generated for the gradient check mechanism specified for monitoring the thin plate temperature of the demonstrator used to validate the development within SAFE [10], Figure 3.

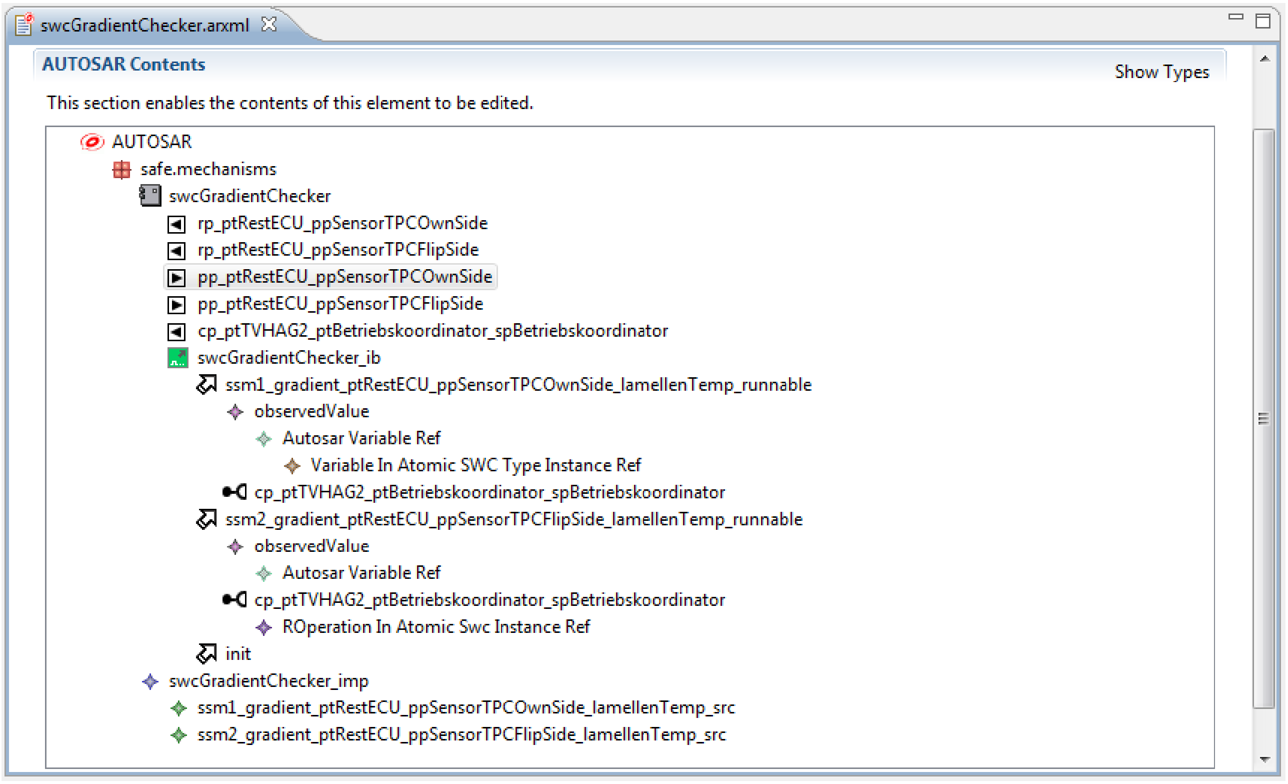

swc.swcd

```
@desc {
    see "specification/mechanisms.txt" for further information about the required adaption of the model
    related to: Adaption #7
}
component application swcActuatorMonitoring_compTotal_ptTVHAG2_ptPosControl_ppSollASMMoment{
    ports {
        receiver rpSensorPC requires ISensorPosControl
        receiver rpSollASMMoment requires ISollASMMoment
        client cpBetriebskoordinator requires IBetriebskoordinator
    }
}

composition compTotal {
    prototype swcRestECU ptRestECU
    prototype swcRestBus ptRestBus

    prototype compTVHAG2 ptTVHAG2

    @desc Adaption #5
    prototype swcGradientChecker ptGradientChecker

    @desc Adaption #6
    prototype swcRangeChecker ptRangeChecker

    // connection to tvhag links
    @desc daten vom externen bus
```

Figure 3 – Excerpt of an AUTOSAR software component model using ARText

The generated component has then to be instantiated and integrated into the preexisting system architecture and the previous connections between components has to be adapted to accommodate the newly introduced component. It is the job of safety code generators to realize the traceability between the original and adapted version of the model. For example, the integration could be done in a copy of the original model and produced as output out of the generation process. Figure 5 presents a simple example of how the generated gradient check component is integrated into a preexisting AUTOSAR software architecture.

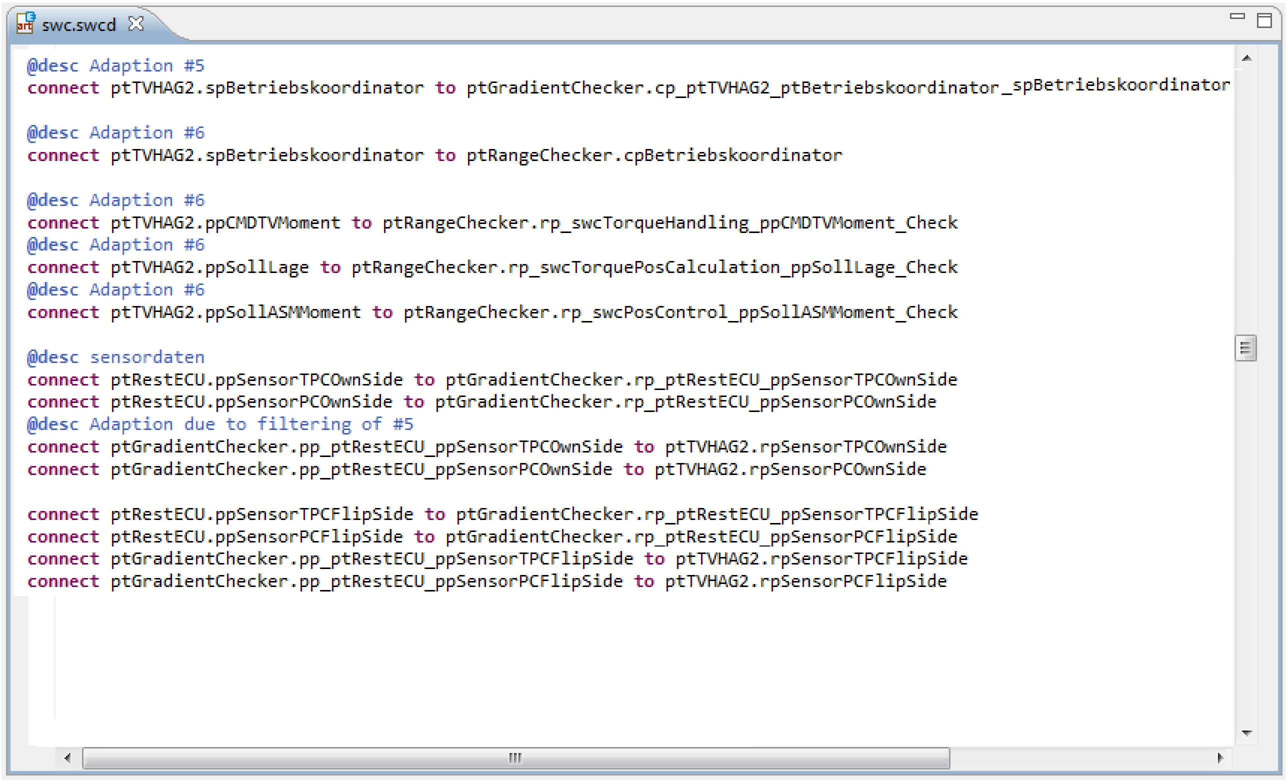
For SSR specified at the system model level of AUTOSAR the target ECU to which software components are mapped is known. In this case the generator uses this information to generate the implementation of the software safety mechanisms (e.g. gradient check) in the corresponding target locations where the software for each specific ECU lies.



This section enables the contents of this element to be edited.

```
AUTOSAR
  safe.mechanisms
    swcGradientChecker
      rp_ptRestECU_ppSensorTPCOwnSide
      rp_ptRestECU_ppSensorTPCFlipSide
      pp_ptRestECU_ppSensorTPCOwnSide
      pp_ptRestECU_ppSensorTPCFlipSide
      cp_ptTVHAG2_ptBetriebskoordinator_spBetriebskoordinator
      swcGradientChecker_ib
        ssm1_gradient_ptRestECU_ppSensorTPCOwnSide_lamellenTemp_runnable
          observedValue
            Autosar Variable Ref
              Variable In Atomic SWC Type Instance Ref
          cp_ptTVHAG2_ptBetriebskoordinator_spBetriebskoordinator
        ssm2_gradient_ptRestECU_ppSensorTPCFlipSide_lamellenTemp_runnable
          observedValue
            Autosar Variable Ref
          cp_ptTVHAG2_ptBetriebskoordinator_spBetriebskoordinator
            ROperation In Atomic Swc Instance Ref
        init
    swcGradientChecker_imp
      ssm1_gradient_ptRestECU_ppSensorTPCOwnSide_lamellenTemp_src
      ssm2_gradient_ptRestECU_ppSensorTPCFlipSide_lamellenTemp_src
```

Figure 4 – Gradient check generated as AUTOSAR component



```
@desc Adaption #5
connect ptTVHAG2.spBetriebskoordinator to ptGradientChecker.cp_ptTVHAG2_ptBetriebskoordinator_spBetriebskoordinator

@desc Adaption #6
connect ptTVHAG2.spBetriebskoordinator to ptRangeChecker.cpBetriebskoordinator

@desc Adaption #6
connect ptTVHAG2.ppCMDTVMoment to ptRangeChecker.rp_swcTorqueHandling_ppCMDTVMoment_Check
@desc Adaption #6
connect ptTVHAG2.ppSollLage to ptRangeChecker.rp_swcTorquePosCalculation_ppSollLage_Check
@desc Adaption #6
connect ptTVHAG2.ppSollASMMoment to ptRangeChecker.rp_swcPosControl_ppSollASMMoment_Check

@desc sensordaten
connect ptRestECU.ppSensorTPCOwnSide to ptGradientChecker.rp_ptRestECU_ppSensorTPCOwnSide
connect ptRestECU.ppSensorPCOwnSide to ptGradientChecker.rp_ptRestECU_ppSensorPCOwnSide
@desc Adaption due to filtering of #5
connect ptGradientChecker.pp_ptRestECU_ppSensorTPCOwnSide to ptTVHAG2.rpSensorTPCOwnSide
connect ptGradientChecker.pp_ptRestECU_ppSensorPCOwnSide to ptTVHAG2.rpSensorPCOwnSide

connect ptRestECU.ppSensorTPCFlipSide to ptGradientChecker.rp_ptRestECU_ppSensorTPCFlipSide
connect ptRestECU.ppSensorPCFlipSide to ptGradientChecker.rp_ptRestECU_ppSensorPCFlipSide
connect ptGradientChecker.pp_ptRestECU_ppSensorTPCFlipSide to ptTVHAG2.rpSensorTPCFlipSide
connect ptGradientChecker.pp_ptRestECU_ppSensorPCFlipSide to ptTVHAG2.rpSensorPCFlipSide
```

Figure 5 – Reorganization of connections for accommodating the generated gradient check mechanism

5.3.2 CHROMOSOME

In this subsection an overview regarding modeling SSRs and the generation of SSMs for CHROMOSOME is given. The aim of this subsection is to define the general outline of code generation scenarios for CHROMOSOME, where SAFE SSRs are implemented by CHROMOSOME SSMs.

As previously stated, it is recommended to model SSRs in a loosely coupled manner. It is expected that a language implementing SAFE meta-model is used to specify SSRs, so the modeling of SSRs does not require modification of CHROMOSOME meta-model. A preexisting CHROMOSOME system model containing all artifacts excluding safety view is defined in detail in CHROMOSOME domain-specific language. A SAFE model extends existing CHROMOSOME artifacts with safety requirements.

In SAFE model SSRs have to be defined to make further code generation possible. The original format for SSR specifications can vary depending on the tool used. We expect that the main input method for SSRs is a full or partial implementation of SAFE meta-model, in form of textual language, graphical tool, or just a file in SAFE model interchange format. Independent of the choice the language should allow users to reference external model elements and specify SSRs for those elements in the way SAFE meta-model allows such referencing through modeling elements defined in the CommonStructure::References::CHROMOSOMEReferences package of the SAFE meta-model.

CHROMOSOME relies strongly (just like AUTOSAR) on tool-supported model-based code generation. Safety mechanisms specification is decoupled from the existing CHROMOSOME models. Code generator therefore should modify the original CHROMOSOME model. Traceability between the original and adapted versions of the model should be provided to enable repeated generation. SAFE SSR model shall also be modified by the generator to include references to newly generated elements and to allow requirement traceability within SAFE model. Such modifications can be implemented by, for example, producing copies of existing models, or by annotating the generated and modified elements to distinguish from the original and by keeping history of model changes.

Generation of safety mechanisms code for CHROMOSOME targets results in instantiation of new component elements and corresponding modification of data path elements (topics) within the CHROMOSOME model. Code generator should locate the elements referenced by the SAFE SSR specifications and create new relevant elements in the CHROMOSOME model. The generated components have to be configured to produce implementations corresponding to the input SSR model. For some SSRs (like, for example, Health Monitor) iteration through other SSR specifications may be necessary to generate the SSM configuration.

Transformation of multiple input models (predefined CHROMOSOME model and SAFE SSRs) can be implemented using one of the numerous model transformation frameworks. Almost every modeling framework or tool today is accompanied by at least one model transformation framework.

It is the choice of code generator developer, whether C code will be directly generated for every generated component instance. An alternative approach is to provide a library of predefined generic SSM implementations along with the code generator and only generate configuration data for new component instances. The former approach results in generation of source code that is optimal for a specific instance of SSM, but increases the maintenance effort for the generator developer due to the need to support both the existing components and code generation templates.

The final generation of code for the CHROMOSOME runtime and projects allowing build of binary images shall be performed by transforming the adapted CHROMOSOME model with a technology-specific configuration tool. An example generation workflow and additional details regarding CHROMOSOME can be found in Appendix B – CHROMOSOME.

6 Meta-model based generator specification

In this section the elements of the SAFE meta-model related to software safety requirement specifications and therefore of interest for safety code generators are described in detail. The work done within SAFE regarding the modeling of software safety requirements has been mainly divided in two parts: generic and concrete. On one hand the generic part of the SAFE meta-model a base for the concept of software safety requirement (SSR) specification is provided. This enables users to specify abstract requirements related to SAFE and a given target technology. On the other hand the concrete part of the SAFE meta-model provides detailed software safety requirement structures where the concrete relations of such requirements are defined (e.g. which concrete elements from external meta-models are necessary for a given SSR specification). In the following subsections the meta-model elements for SSR specification are described and the implications of such elements on safety code generators are detailed.

6.1 Software Safety Requirement Specification in SAFE

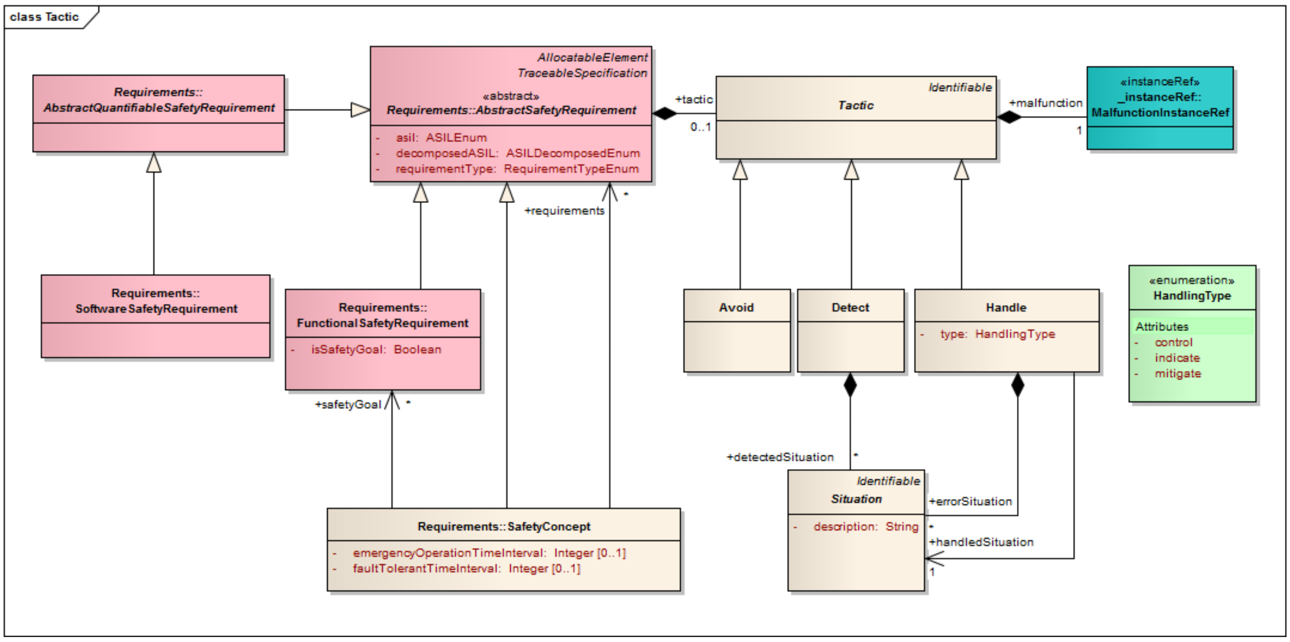


Figure 6 – Structure for software safety requirements specification

The central concept in the meta-model is the abstract structure for specifying software safety requirements. This structure is depicted in Figure 6. The meta-model enables models to specify a implementation safety extension composed by a set of software safety requirements (SSR) and code generation configuration information as shown in Figure 7. The purpose of software safety requirements is defined using a tactic mechanism. The *Tactic* defines how a given malfunction (error) is treated by the requirement. There are three possible tactics: avoid, detect and handle the malfunction. The tactics have been identified during the exploration phase realized within WT3.6, documented in Appendix A – Classification of Software Safety Mechanisms. The SSRs are specified within specific SAFE “Safety Extensions” in order to provide a context for the requirements.

In the case of specifications relating to AUTOSAR there is an abstract meta-class *AutosarSafetyExtension* which is to be used as base for all possible AUTOSAR related safety extensions. For CHROMOSOME, the ChromosomeSafetyExtension shall be used. The different possibilities of extensions are shown in Figure 7. A software safety requirement is a refinement of given technical safety requirements (TSR) and each SSR can trace back to the originating TSR via the RequirementsLink element which allows tracing requirements as covered, refined or decomposed.

The specification of SSRs is still abstract and, depending on the generative approach (i.e. agnostic SSR specification or targeted SSR specification), further configuration of SSRs is necessary. For this reason the *CodeGeneratorConfiguration* meta-class has been introduced. This element allows configuration parameters to be specified for SSRs. Such parameters are to be interpreted by safety code generators in order to obtain further information regarding the SSR specification. This configuration structure can also be used for storing generator specific information on a SAFE model.

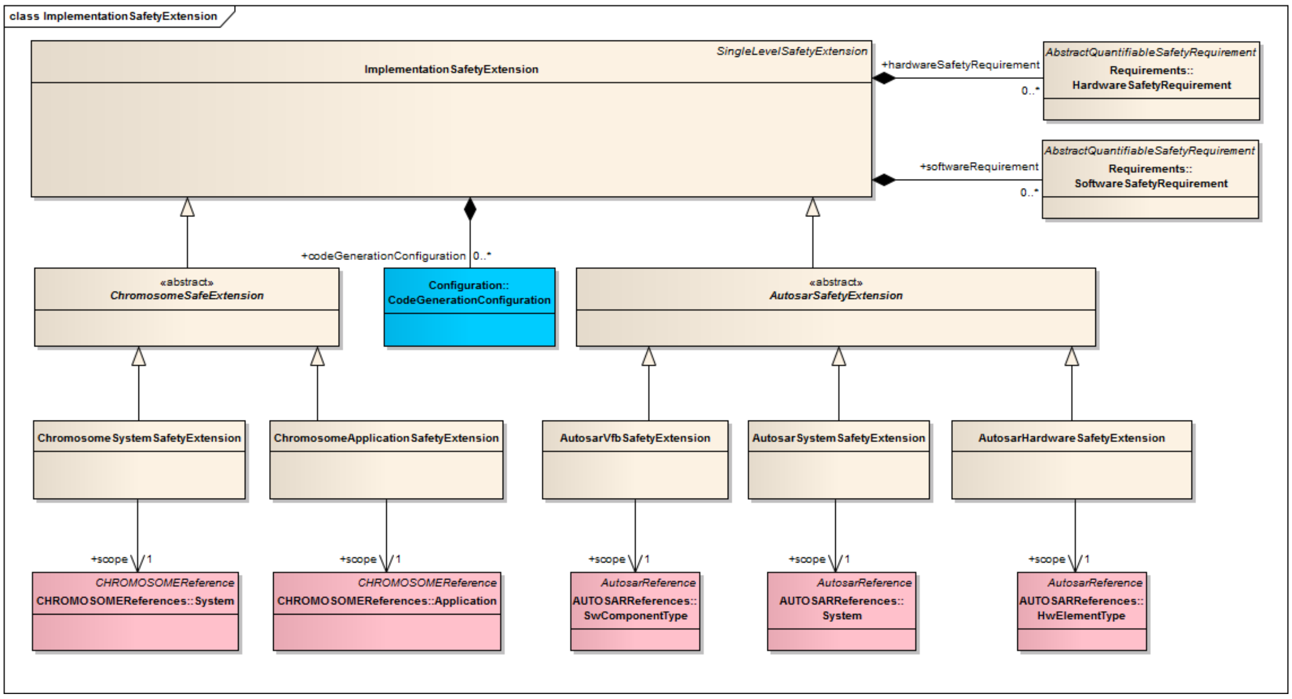

class ImplementationSafetyExtension
SingleLevelSafetyExtension
ImplementationSafetyExtension
+hardwareSafetyRequirement
0..*
AbstractQuantifiableSafetyRequirement
Requirements::
HardwareSafetyRequirement
+softwareRequirement
0..*
AbstractQuantifiableSafetyRequirement
Requirements::
SoftwareSafetyRequirement
+codeGenerationConfiguration 0..*
«abstract»
ChromosomeSafeExtension
Configuration::
CodeGenerationConfiguration
«abstract»
AutosarSafetyExtension
ChromosomeSystemSafetyExtension
ChromosomeApplicationSafetyExtension
AutosarVfbSafetyExtension
AutosarSystemSafetyExtension
AutosarHardwareSafetyExtension
+scope 1
CHROMOSOMEReference
CHROMOSOMEReferences::System
CHROMOSOMEReference
CHROMOSOMEReferences::Application
AutosarReference
AUTOSARReferences::
SwComponentType
AutosarReference
AUTOSARReferences::
System
AutosarReference
AUTOSARReferences::
HwElementType

Figure 7 – Implementation Safety Extension

6.2 Error avoidance, detection and handling

The meta-model depicted in Figure 6, makes possible to specify software safety requirements related to error avoidance, error detection and error handling using the tactics relation. The behavior of the system in case of error can be modularly defined using the three kinds of SSR tactics. In this way, the engineer might define requirements which cover the avoidance of errors, such as a barrier requirement. Requirements that cover the detection of errors, and might in turn have another SSR defined as an error reaction. Requirements covering error handling, which are related to error detection requirements and might also specify a reaction in case new errors happen during the handling process (e.g. a filtering mechanism with a threshold of 5). Examples of possible reactions for detected errors are: filtering, notification, reset, memory partition reset and default value. More details about the structuring of tactics can be seen in Figure 6.

Code generators supporting the specification of handling mechanisms shall generate the necessary interfaces to access the resources specified by the engineer for handling the error (error reaction). For instance, in case a reset handling mechanism is specified as a reaction to a given error in an AUTOSAR system, the code generator shall generate the necessary requests to the BSW components of AUTOSAR in order to execute the reset procedure. Moreover, for mechanisms whose handling is realized through communication (e.g. notifying another component) the code generator must generate the correct connections between the related components and if necessary adapt the target/source component with newly required interfaces for realizing the communication.

6.3 General requirements on safety code generators

In this section general remarks for the implementation of safety code generators are given. These remarks provide an overview on the common issues to be handled while generating code for software safety requirements specified using the SAFE exchange format.

6.3.1 Scheduling of generated executable entities

During the generation of safety code the lines of code realizing the functionality specified through SSRs will in most cases represent an executable entity. An executable entity can be either a function or a set of functions and both share the characteristic that a given points in time (schedule) the entity has to be executed by the underlying technology platform. In order to guarantee that the specified software safety requirements are working correctly the executable entities generated for the SSR have to be scheduled.

Generators have therefore to take into consideration what options the target technology platform offers for scheduling executable entities. Furthermore, the required information for scheduling can be required by generators and stored as configuration information of the given SSR as described in Section 6.3.4.

For instance, while generating a gradient check implementation the safety code generator has to take into account how the executable entity for the gradient check will be scheduled and what has been specified within the SSR. The reason why the scheduling plays an important role is that the time delta specified for the gradient check might have to be refined into smaller slots in order to correctly verify if the gradient is varying within acceptable ranges.

6.3.2 Interface generation for software components

For accessing information coming from outside the generated software elements (components) safety code generators have to generate the related interfaces. Common patterns such as separation of concerns and coupling shall be taken into account in order to generate interfaces which allow the generated artifacts to be seamlessly integrated with preexisting artifacts.

In the case where the concrete software architecture defined using a given target technology platform provides concepts for the interconnection of software components, the safety code generators shall generate the software component interfaces of generated software component artifacts according to the concepts defined by the target technology platform.

For example in the case of AUTOSAR the safety code generator has to generate the component ports required and provided by the generated software components and additionally the variable access elements within the executable entities contained in these software components.

6.3.3 Re-routing of connectors if inter-component communication is addressed

On given target technology platforms, the communication between software components is done via the definition of connectors. Safety code generators shall be able to handle the necessary modifications regarding connectors in order to integrate the generated artifacts into the preexisting ones. It has also to be noted that different technologies might define different types of connectors and safety code generators must be able to understand the implications of generating each type of connector and the possible side-effects caused by the integration of generated components.

6.3.4 Configuration information

As previously describe in Section 5.2 the SAFE meta-model enables the modeling of configuration information regarding SSR. This information can be used by generators to define parameters related to the generation which do not influence the SSR specified using the SAFE exchange format.

Whenever a safety code generator requires additional information related to an SSR this information shall reflected back into the SSR model. If the generation procedure is applied more than once to the same model the configuration information provided by the user is in this case already present could be used for further analysis.

6.3.5 Annotations

Whenever possible, safety code generators shall make use of annotations to document the rational regarding the implementation of SSR and to provide the user with useful information about the generation. These annotations shall be realized according to the possibilities provided by the target technology platform and could be used to fulfill traceability requirements (e.g. function annotations trace back to SAFE model elements).

6.4 Software Safety Requirements Specified within the SAFE Meta-model

Besides the generic meta-model structure previously described, the meta-model contains also a specialized set of software safety requirements which were chosen for the purpose of validating the concepts developed within SAFE. The specialized SSR elements are limited in number since it is not the purpose of WT 3.6 to propose a meta-model for every possible software safety requirement which can be transformed into a software safety mechanism.

In the following subsections the specialized SSR elements are described together with the relevant information for safety code generation. Where applicable examples of how code generation could work are given.

6.4.1 Aliveness Monitor

Description: the aliveness monitor depicted in Figure 8, also known as heart-beat monitor, supervises the execution of executable entities through the use of checkpoints [11]. Given that an error has occurred it can notify the specified handling element. The possible errors for this specification are: too late, meaning the checkpoint was achieved after the expected point in time; too early, meaning the checkpoint was achieved before expected; too often, meaning the checkpoint was arrived too many times regardless of how early or late the executable entity has reached checkpoints.

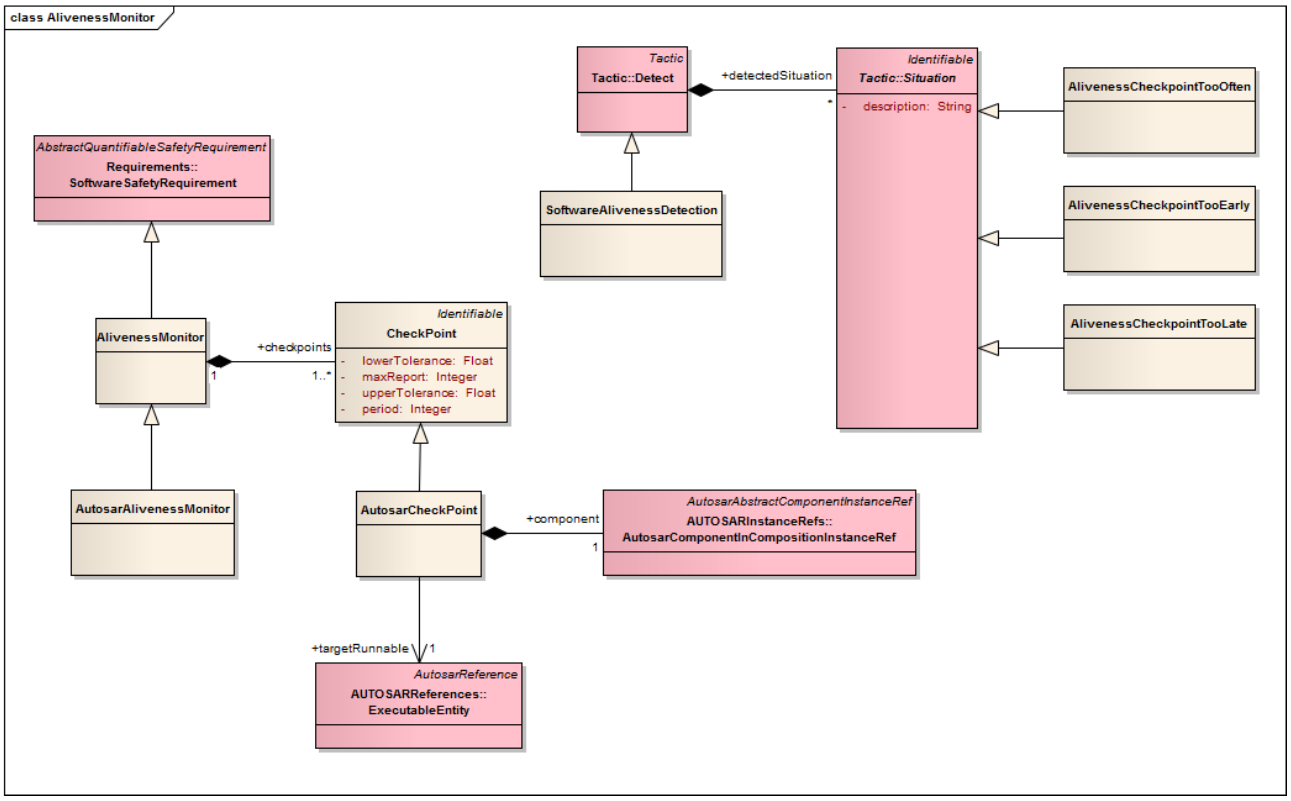


Figure 8 – Aliveness monitor meta-model

Semantics: the specification of an aliveness monitor requirement implies that generators shall transform the aliveness monitor SSR into a set of checkpoint definitions configured with the parameters defined in the SSR specification. The generated mechanism shall have the necessary interfaces for communicating with the system, which can be specified as a reaction to aliveness monitor errors. This is especially relevant for the communication of errors to other components of the system. Furthermore the details related to scheduling of the generated software components must be also generated and integrated to the target technology platform.

6.4.2 Context Range Check

Description: the context range check shown in Figure 9 detects errors related to the range of values provided to it. The context feature allows the specification of different ranges given different contexts (modes of operation). Furthermore, it is also possible to define what kind of reactions shall be performed when range errors occur.

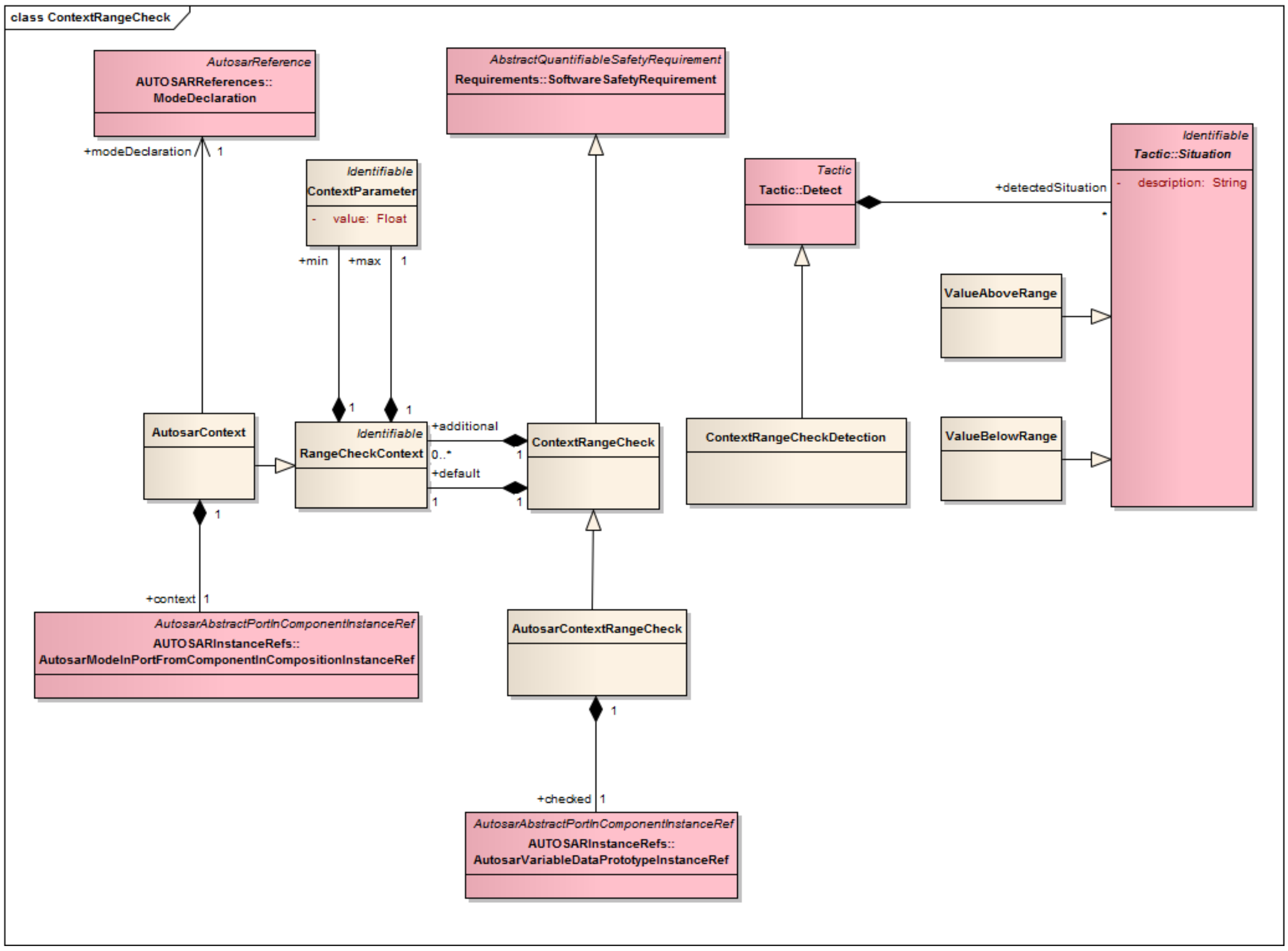

class ContextRangeCheck
AutosarReference
AUTOSARReferences::ModeDeclaration
AbstractQuantifiableSafetyRequirement
Requirements::SoftwareSafetyRequirement
Identifiable
Tactic::Situation
- description: String
Identifiable
ContextParameter
- value: Float
Tactic
Tactic::Detect
+detectedSituation
+modeDeclaration
+min
+max
ValueAboveRange
ValueBelowRange
AutosarContext
Identifiable
RangeCheckContext
+additional
+default
ContextRangeCheck
ContextRangeCheckDetection
+context
AutosarAbstractPortInComponentInstanceRef
AUTOSARInstanceRefs::AutosarModeInPortFromComponentInCompositionInstanceRef
AutosarContextRangeCheck
+checked
AutosarAbstractPortInComponentInstanceRef
AUTOSARInstanceRefs::AutosarVariableDataPrototypeInstanceRef

Figure 9 – Context range check meta-model

Semantics: the specification of a context range check requires safety code generators to generate the context structures which are able to hold context information (e.g. maximum range allowed) furthermore the generated code shall make use of context information provided by the system in order to define what are the valid ranges for operation while in the current context. The context information shall be obtained through specific interfaces generated by the safety code generator for the concrete range check mechanism. Furthermore generators must generate interfaces for error communication for each of the specified handling SSR. For example, when a filter SSR is specified as the reaction for the detection of values above the maximal specified range the safety code generator must generate the proper interface for communicating this error and triggering the corresponding handling mechanism.

6.4.3 Gradient Check

Description: the gradient check depicted in Figure 10 monitors the temporal behavior of a given value and detects when an invalid variation of values happen. This is done through the specification of maximal and minimal gradient variations and a time delta which is used for the computation of the gradient. The detection of a gradient error is communicated or handled by defining a SSR having a handling tactic and using the two possible notifications defined in the meta-model (gradient too high or too low).

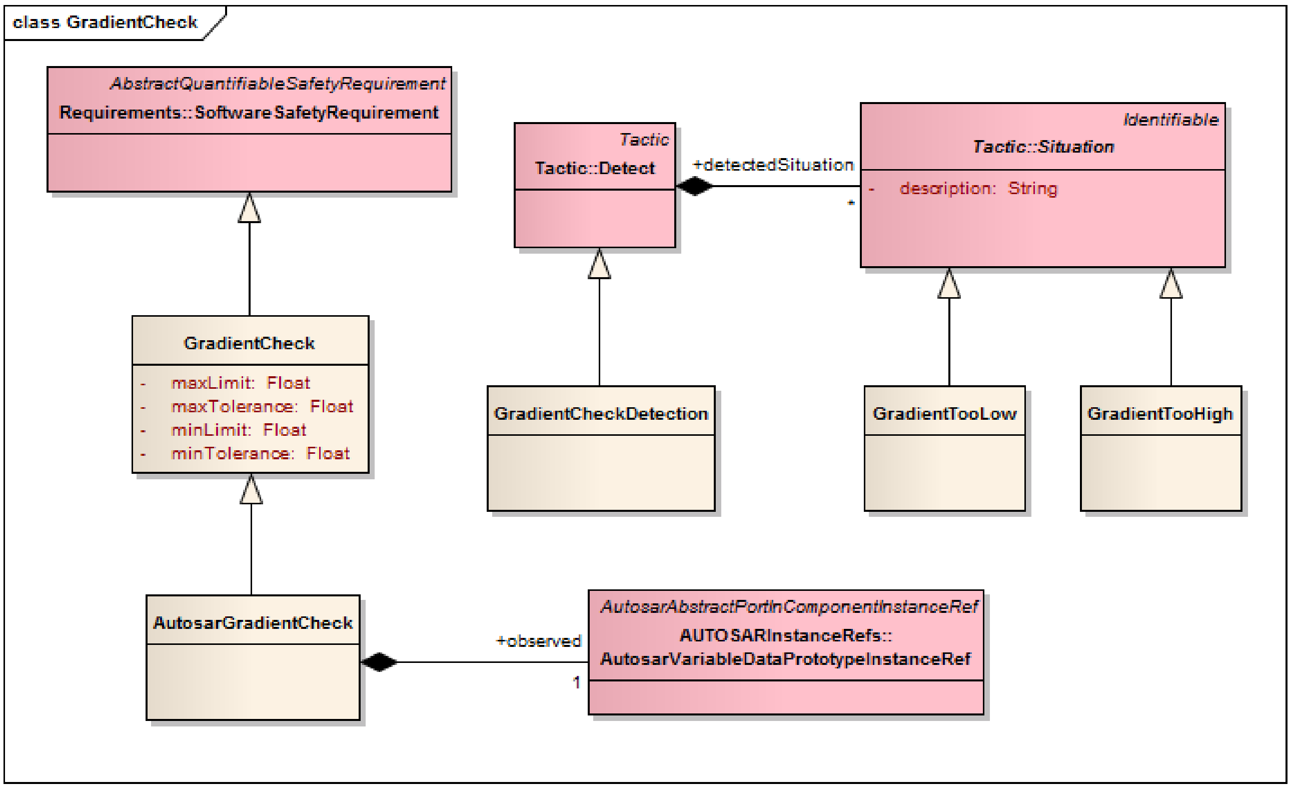


Figure 10 – Gradient check meta-model

Semantics: the generation of code for a gradient check mechanism has to take primarily into account that a gradient is a stateful property. This means that the gradient can only be computed by using previously measured / observed values. Therefore the generated gradient check mechanism shall provide a structure where the parameters specified in the SSR are store and additional structures for storing the previously obtained value. Furthermore the interface for obtaining the value whose gradient is observed has to be generated together with the required error communication interfaces. Depending on the generator realization the error handler specified via SSR can be generated within the gradient check mechanism (e.g. filtering as a handling mechanism). In the case the handler provides a corrected value the related interface shall be also generated.

6.4.4 Comparison

Description: the comparison SSR depicted in Figure 11 takes in as input two values and executes the specified operation over the two inputs. The use of a comparison is especially important when specifying requirements related to redundant or diverse reading/processing of values and/or computation of values, etc. The two possible errors defined for this SSR are the Boolean evaluation of the operation. Depending on the result further reactions can be specified. For example, given the result of a comparison is false an error handler SSR for providing a default value could be specified.

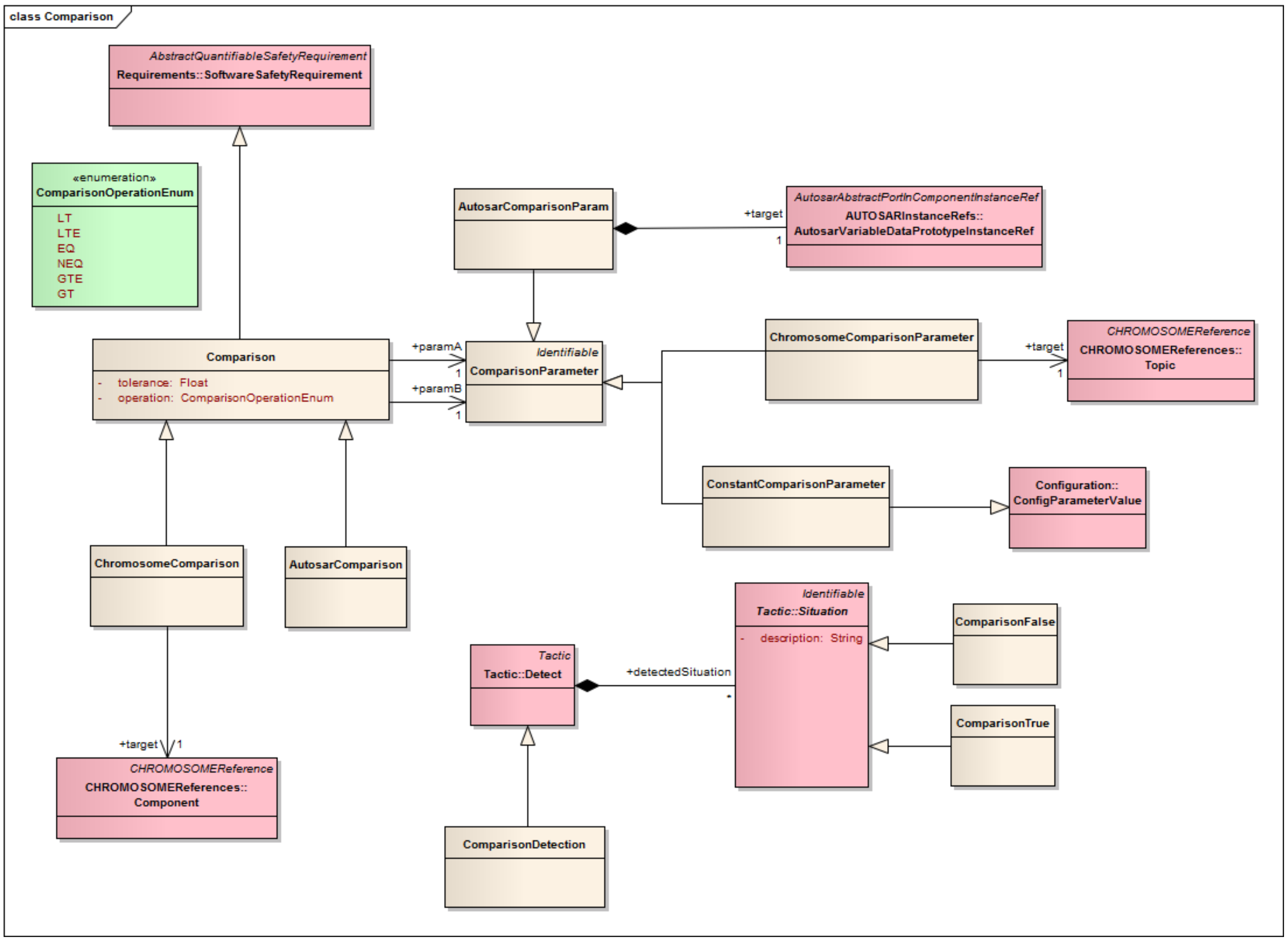


Figure 11 – Comparison meta-model

Semantics: the generation of a comparison operation has to take into account what kind of input values are provided to the operation (e.g. where do values come from) and the related interfaces for obtaining these values shall be generated. The generated code has to take care of realizing the operation taking into account the configured tolerance for the comparison. Furthermore depending on what kind of reaction is specified the realized mechanism shall either provide an interface for writing the error or a value if one is to be provided by the mechanism.

6.4.5 CRC

Description: the cyclic-redundancy-check shown in Figure 12 detects errors occurred between operations on a given value. Usually additional information will be added to the value in order for the check algorithm to perform validation operations on the value. CRC algorithms require certain basic parameters which are defined in the meta-model. Additionally it is possible to define requirements on the possible reactions to CRC errors. These are done through the specification SSRs having a handling tactic.

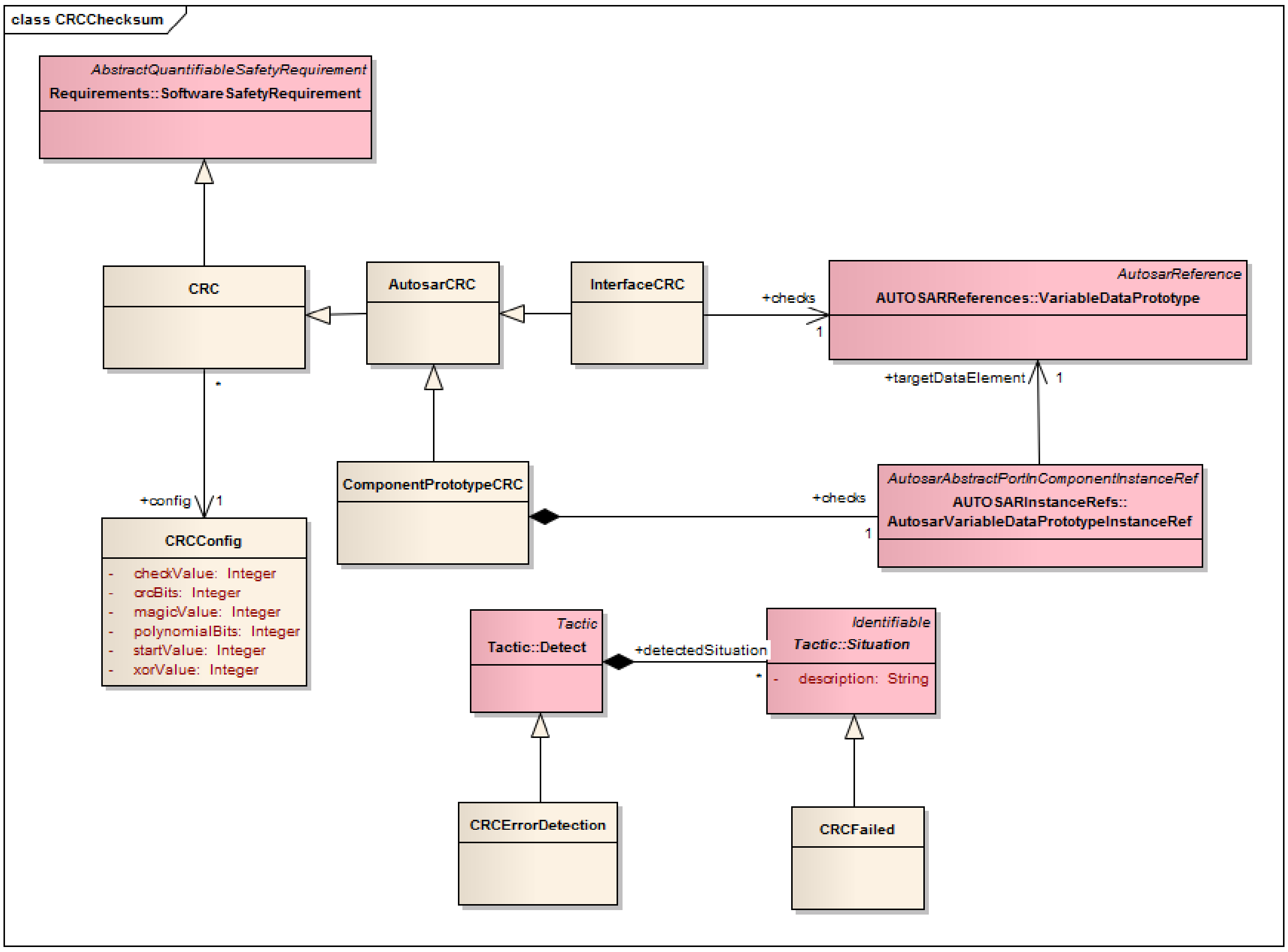


Figure 12 – CRC meta-model

Semantics: the specification of CRC software safety requirements provides flexibility to code generators regarding the implementation of CRC algorithms. The generators have only to take into account the CRC configuration parameters defined by the SSRs and to provide the correct interfaces for the implementation to work. This mean for example that if some kind of preexisting CRC library is to be used the correct interfaces for the mechanism and for storing the configuration information of the mechanism have to be generated and the library code does not need to be generated. Moreover, the interfaces for communicating CRC errors to the target technology platform have also to be generated for the mechanism and in the case of using a third party library the interfaces for obtaining error information from the library must also be generated.

6.4.6 Filter

Description: the filtering of values, SSR shown in Figure 13, provides an error handling possibility for specifications where an error state is only achieved after a given temporal frame of anomalous behavior. For example, the error state of a temperature sensor is only achieved if for five cycles the sensor delivers unreliable values. Hence, filtering the values delivered between the first time an error is seen until the fifth time the error is seen (sequentially in time) can provide the system certain robustness against sporadic errors (de-bounce). The filter computes a value to be forwarded to the required interfaces of a given system element according to a given expression. This expression can take into account, for example, the current value the filter has been provided with and previous values (e.g. the average value from last 5 samples).

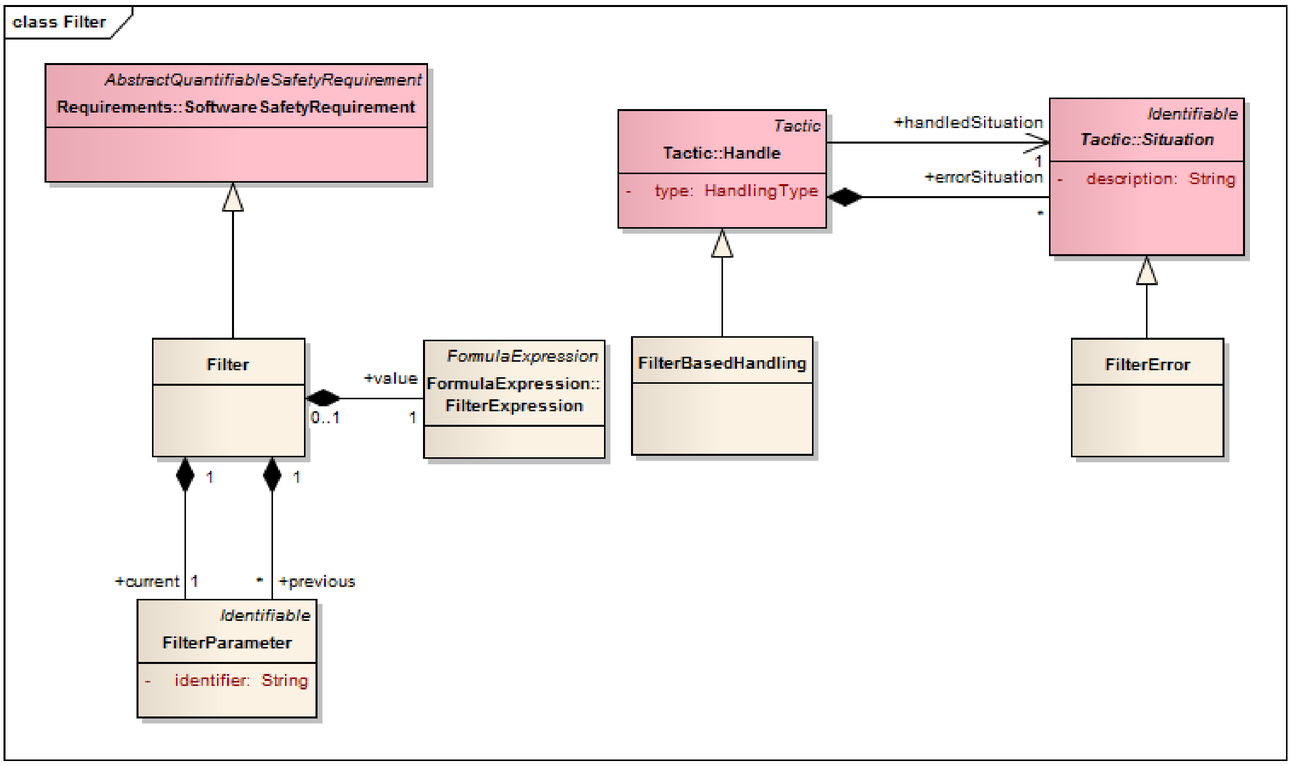


Figure 13 – Filter meta-model

Semantics: the requirements on generator implementations posed by the specification of filter SSR are mainly related to how the filter mechanism is generated. Normally the filtering is highly coupled with the SSR requiring filtering. Therefore it is recommended that generators implement one of two cases for generation: coupled generation and stand-alone generation. On the coupled version of the filter mechanism the filter behavior is generated integrated into the mechanism implementation requiring filtering. This implies that the mechanism using the filter will also output the filter value. Hence, an appropriate interface shall be generated within the filter-requiring mechanism. On the other hand the generation of stand-alone filters requires the interfaces for receiving input values and sending the filtered values have to be generated. Safety code generators must observe the definition of previous values for a given filter specification since the definition of multiple previous values might imply in a temporal relation, meaning that a buffer for storing the values provided within a time window has to be generated.

6.4.7 Actuator Monitor

Description: the monitoring of actuators, defined via the SSR meta-model depicted in Figure 14, provides the ability to determine if there are issues with the control loop of a given system. By observing the feedback read from actuators it can be determined if the actuator is behaving as the controller expects it to behave. This requires the definition of relationships between input data from sensors and controller output data. Since the algorithms used to monitor the behavior of actuators vary significantly the approach taken in the SAFE meta-model is to allow the definition of monitoring algorithms using external functions (a function in this context is an algorithm executed by a function call at code level). In this way the monitoring mechanism functionality can be defined using, for example, Simulink ® [12].

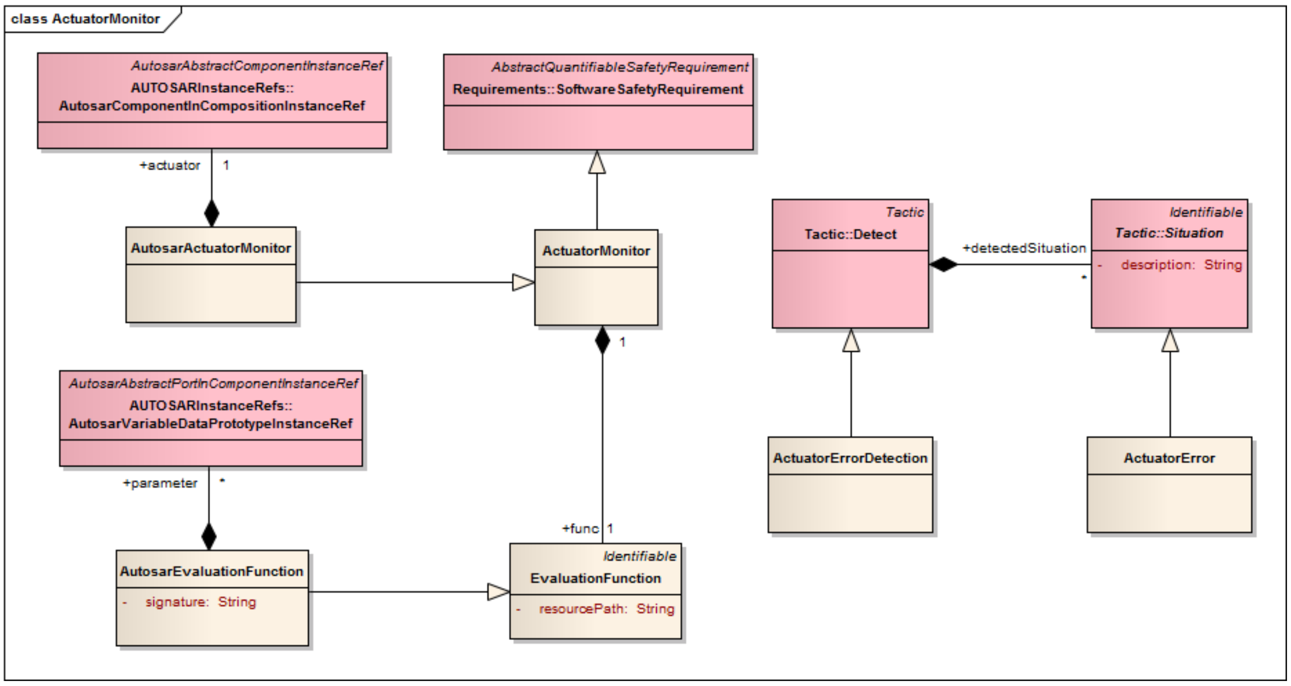


Figure 14 – Actuator monitor meta-model

Semantics: the main consequence for safety code generators implementing actuator monitors is that the behavior of the mechanism might be specified using different technologies (e.g. C-code or UML models). The generated artifacts have to integrate such technologies and also the target technology used to execute the system (e.g. AUTOSAR). The safety code generator shall generate a structure for controlling the execution of the mechanism (buffers for delayed values, etc.) and interfacing with the behavior realization. Furthermore, interfaces for obtaining the required data have to be generated. There is a special characteristic regarding actuator monitors since the required information comes from sensors and from controlling components. This means that the interfaces have to gather input data coming from the environment and output data being provided by the system. This characteristic influences how the generated artifacts (code or model elements) interact with preexisting artifacts. It is also important to achieve the integration at the scheduling level since the input-output correlation is a temporal correlation and might affect the monitoring of the system.

6.4.8 CPU Self-test

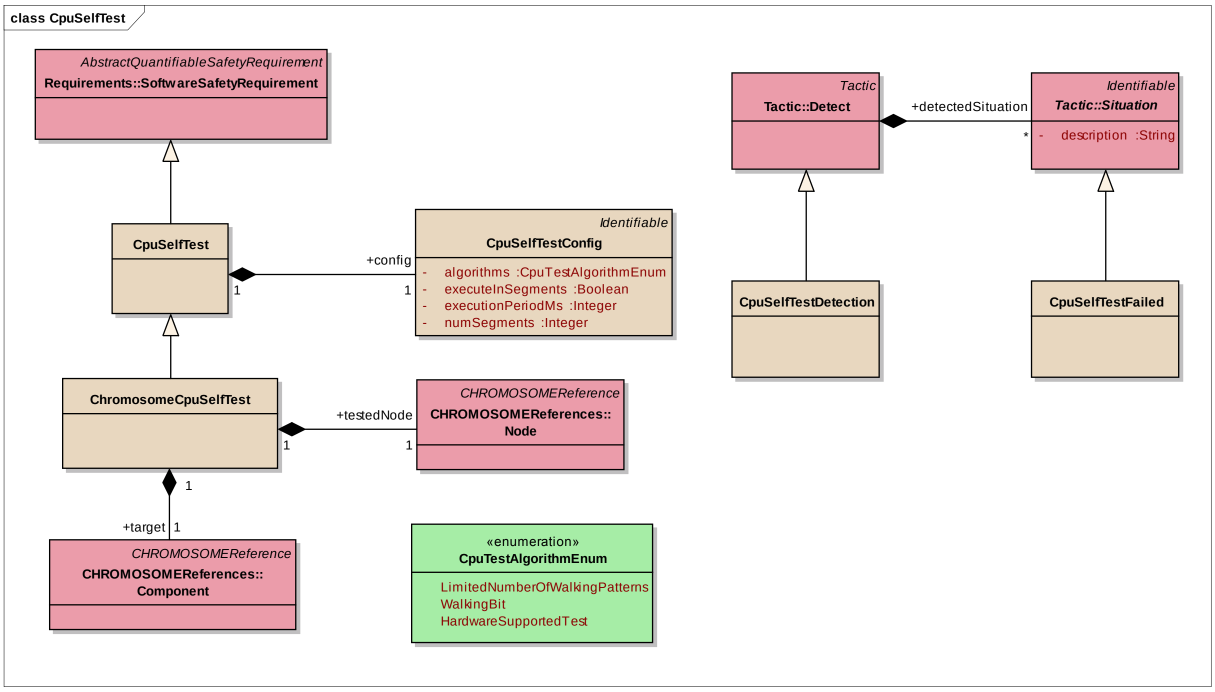


Figure 15 – CPU Self-Test meta-model

Description: The CPU Self-test (Figure 15) meta-model allows the periodic testing of CPU on the specific node and detects when the response to a test is not matching expected. Its configuration allows performing CPU tests in segmented mode, so that no large timeslot is occupied in the schedule by the test. The total number of segments and full execution period are then specified as configuration parameters.

Semantics: the specification of the CPU self-test requirement implies that generators shall transform the SSR into a component instance and schedule it accordingly to the specified configuration parameters (to be executed completely or in chunks) and perform the specified CPU test. Only one instance of such a component instance per ECU is generated, provided that the highest requirement is satisfied. The generated component shall have necessary interfaces to communicate errors to other components of the system, or to execute reactions as a direct response to the detected error. CPU Self-test is specified with a direct reference to the monitored entity (i.e., target ECU / node for deployment).

6.4.9 RAM Self-test

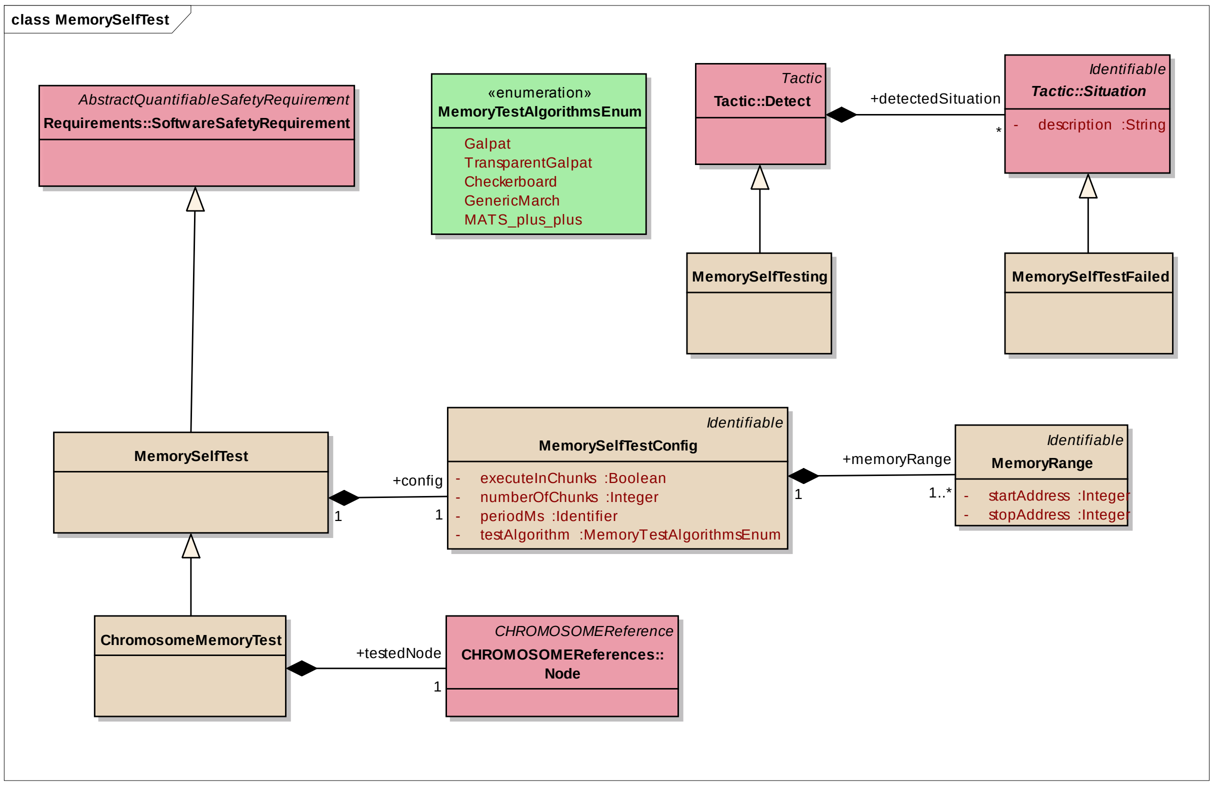


Figure 16 – Memory Self-Test meta-model

Description: RAM self-test depicted in Figure 16 performs periodic execution of RAM test of specified memory range with specified memory test algorithm. It allows performing RAM tests in segmented mode, so that no large timeslot is occupied in the schedule by the test. Total number of segments and full execution period are then specified as configuration parameters.

Semantics: the specification of the RAM self-test requirement implies that generators shall transform the RAM self-test detection SSR into a component instance and schedule it accordingly to the specified configuration parameters to execute (completely or in chunks) with specified period and perform the specified CPU test. The generated component instance shall have necessary interfaces to communicate errors to other relevant components of the system.

6.4.10 Voting

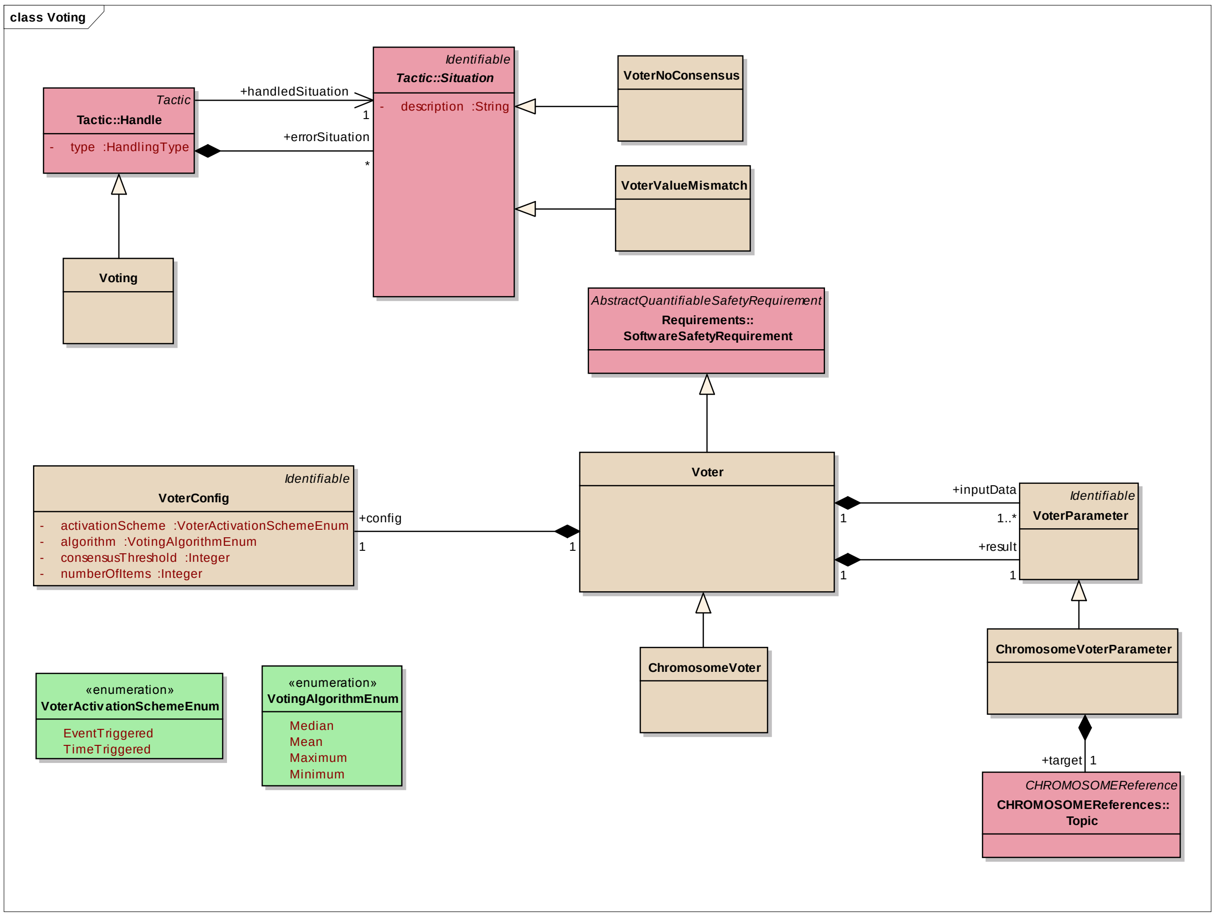


Figure 17 – Voter meta-model

Description: Voting (Figure 17) defines a simple redundancy-based monitoring mechanism. Being configured by number of items to vote on and consensus threshold, voting mechanism allows binary pair-wise comparison of multiple input items, producing an output item as consensus value. It features discrepancy reporting via error notifications, different for two cases. If one value is different in the input array, but consensus could be reached, a “value mismatch” notification is issued. If consensus can’t be reached on the input set of data elements, “no consensus” notification is issued, which generally should lead to advanced error handling methods, such as ECU / node restart or reconfiguration.

Semantics: the specification of the voting requirement implies that generators shall transform the SSR into a component instance. The generated component instance shall be inserted into the data path, and inter-component data paths shall be rerouted accordingly. The comparison function is specified in the mechanism configuration in SSR model, and transformed into a corresponding configuration of the component instance.

Implementation on target platforms could be performed in two ways. One option is to generate a separate component instance, and modify the data paths accordingly. The second option is to generate a wrapper for a component instance consuming data from the voter.

For data-centric platforms like CHROMOSOME generators will transform the SSR in form of a component instance on its own, so it has to be scheduled accordingly to match input and output data rates / delays. In a different implementation the generated code becomes a wrapper for a runnable entity consuming data from the voting component. Additionally, depending on activation mode the scheduling info (priority / slots / scheduling strategy) need to be generated for the software item. The generated component instance or wrapper shall have necessary interfaces to communicate errors to other components of the system. It will also be configured by the generator to match the consensus threshold specified in the voting SSR.

6.4.11 Health Monitor

Description: Health Monitor (Figure 18) performs centralized supervision of the system / subsystem state and allows execution of preconfigured reactions in response to events, such as a specific combination of component instance modes, or error notifications from error detection mechanisms. Corresponding reactions can be configured by specifying Tactics of type Handle associated with the SSR being source of error notification.

Semantics: specification of Health Monitor implies that Health Monitor shall be generated as a platform-specific component instance, which is capable of executing various error reactions, both predefined and user-specified. This means it needs to have sufficient rights to execute signed user code, internal reactions such as node restart, and so on. Health Monitor configuration shall be generated as a table / *struct array*, where each row corresponds to one error condition to be monitored. Health Monitor also requires generation of communication interfaces not only to receive notifications, but also to announce state changes for the related system parts.

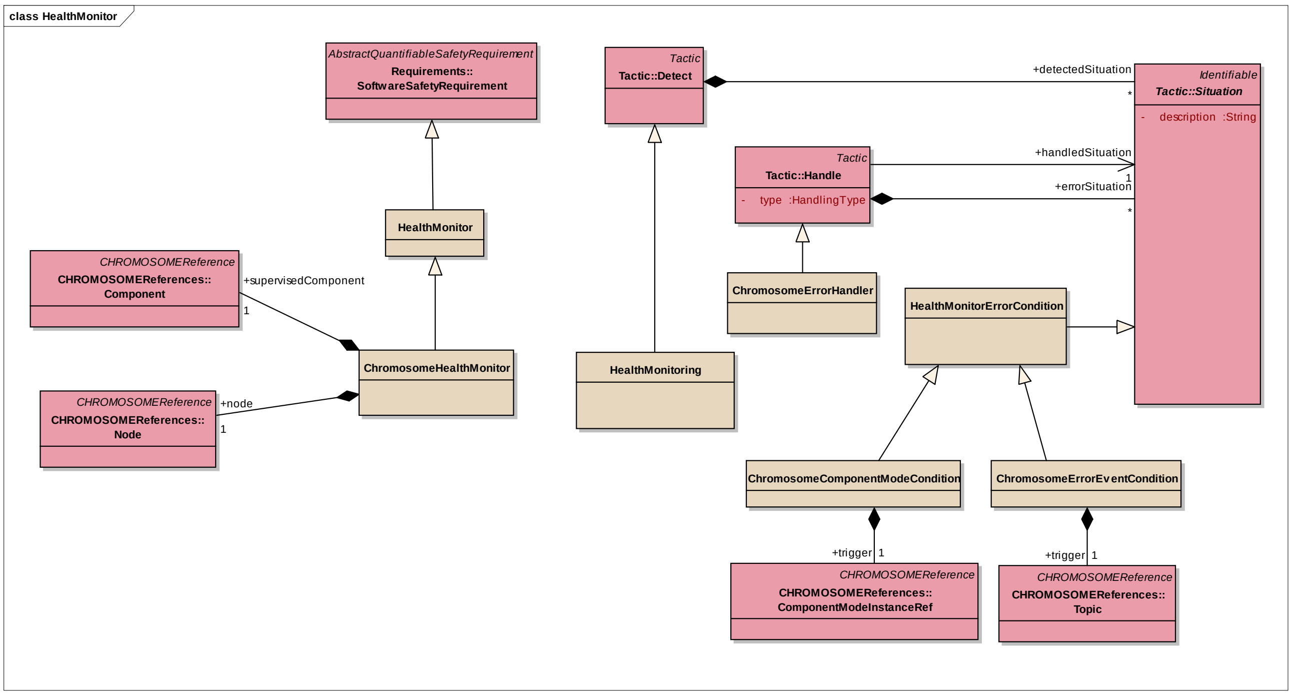


Figure 18 – Health Monitor meta-model

6.4.12 Heartbeat

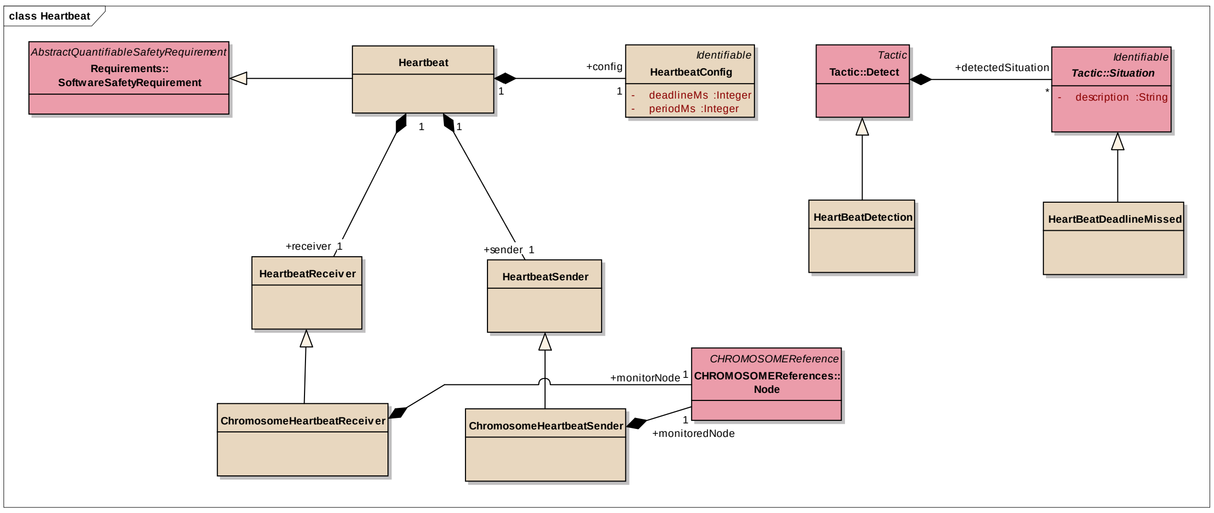


Figure 19 – Heartbeat meta-model

Description: Heartbeat (Figure 19) represents a classical pattern of error detection. It performs periodic check of reachability of one computing node / ECU from another one. It consists of heartbeat receiver, a primitive implementation of an aliveness monitor containing one checkpoint, and an implicit event generator, called heartbeat sender. So, heartbeat sender issues a heartbeat signal once per ‘period’ milliseconds, and heartbeat receiver checks arrival of this signal within ‘deadline’ milliseconds. It should be noted explicitly that deadline computation should include jitter, clock drift and correspond to a worst case estimate to avoid false positives.

Semantics: the specification of a heartbeat SSR in a SAFE model implies that the generators shall transform this SSR into a pair of component instances to be deployed on specified target ECU’s. Schedules of the target ECU’s need to be modified to reach execution rates specified by the SSR configuration parameters ‘period’ and ‘deadline’. ‘Sender’ component instance shall be configured to be executed every ‘period’ milliseconds, and shall have an interface allowing to send data to receiver in a non-blocking manner. ‘Receiver’ shall then be either executed on event arrival and shall require an interface which allows receiving data from the receiver in a blocking manner with a timeout ‘deadline’. Alternatively, ‘Receiver’ is executed periodically with a period allowing detection of ‘deadline missed’ events. In addition, the generated runnable component shall have necessary interfaces to communicate to the system and especially to report errors to other components of the system.

7 Code generation use case for software safety requirements for AUTOSAR

In this chapter the requirements for the generation of software safety requirements based on the AUTOSAR target platform are provided. The information for software safety mechanisms contains a detailed description of the necessary input, the generated artifacts and the integration strategy for preexisting artifacts.

7.1 Use case specification: Control-flow Monitor

In this section the details regarding the usage of control-flow software safety requirements are given.

7.1.1 Description

In order to guarantee the correct behavior of software elements in a given system the engineer might specify safety requirements which define what the control-flow of such software elements should be. The decision of defining such a requirement depends on the technical safety concept defined for the element in question and on the ISO 26262 requirements for a given ASIL.

In this document, the detailed requirements for the realization of a control-flow monitoring mechanism using AUTOSAR are described in detail. The requirements provide a solid base for the implementation of a code generator.

There are two aspects related to control-flow monitoring requirements. One is the set of requirements towards the implementation of a control-flow monitoring mechanism and the other is the set of requirements towards the specification of control-flow monitoring requirements. This section describes the former while the latter is described within section 7.2.

A control-flow monitor specification is decomposed as following:

- **Checkpoint specification:** state machine like specification of checkpoints and transitions.
- **Monitoring element:** software component or hardware device responsible for receiving notifications and checking for valid transitions defined by a valid checkpoint specification.
- **Monitored element:** element whose control-flow is monitored by the monitoring element
- **Control-flow monitoring interface:** interface specification for the communication between monitored elements and the control-flow monitoring element.
- **Control-flow monitoring specification:** the specification describing a monitoring mechanism for a given monitored element.

7.1.2 Checkpoint specification

The checkpoint specification is responsible for defining the valid states and transitions to be used by the monitoring element in order to observe if the monitored element has a valid control-flow. The state machine is completely independent from the monitored element and can be reused to monitor different elements given the same control-flow is expected. The checkpoint specification must contain the following properties according to the:

- States: addressable points within the logical and/or temporal execution of software elements
- Transitions: elements relating the allowed flow between states

- Limits for the number of times a state (checkpoint) is reached (maximum, minimum)

Time window in which the checkpoint can be achieved (time unit is monitoring cycles and not seconds or milliseconds) options are: *not before, not after, at*

7.1.3 Control-flow monitoring specification

The control-flow monitoring specification determines which and how elements have to be monitored. In the context of AUTOSAR three possible monitoring approaches are foreseen:

7.1.3.1 Software component monitoring

- The monitoring of a SWC within AUTOSAR regards the observing of the interaction of a given SWC with the system. In this case the specification must allow the definition of interaction rules (e.g. sequence of read/write on the component’s ports).
- **Requirements:**
    - The specification shall allow the referencing of ports and data elements of ports related to a given software component in AUTOSAR.
    - The specification shall allow referencing states in a given checkpoint specification.
    - The specification shall allow specifying the relation between ports and data elements and states in checkpoint specifications based on the kind of access being realized (e.g. read, write, call, send and receive). See AUTOSAR port interfaces for more details [13].
    - The specification shall allow the definition of read and write (or send and receive) blocks which have to be related to checkpoint definitions.

7.1.3.2 Internal behavior monitoring

- The monitoring of internal behavior of software components regards the sequence in which runnable entities, defined in the given internal behavior of AUTOSAR software components, are executed.
- **Requirements:**
    - The specification shall allow referencing runnable entities within the internal behavior definition of AUTOSAR software components (also within compositions).
    - The specification shall allow the referencing of states defined within checkpoint specifications.
    - The specification shall allow defining the relation of runnable entities and states within checkpoint specifications regarding the possible operations realized over runnable entities (e.g. on start runnable entity, on stop runnable entity).
    - The specification shall allow the referencing of runnable entity execution constraints from the AUTOSAR Timing specification.

7.1.3.3 Runnable entity monitoring

The monitoring of runnable entities is the most fine-grained monitoring of all kinds of monitoring defined in this document. It allows the monitoring of the behavior of the runnable entity itself, namely, the implementation of its behavior. However, this depends on the interface between safety engineers and software developers. Since the checkpoints are not defined by the developer, some kind of exchange of information has to happen between the safety engineer and the software developer in order to define how the checkpoints related to the safety concept and to the source code. The AUTOSAR part of the development (e.g. runnable entity variable access) can be generated automatically, but calling interface from the runnable implementation has to be explicitly considered by the developer.

- **Requirements:**
    - The specification shall allow references to runnable entities within AUTOSAR software components and software component compositions.
    - The specification shall allow references to variable access elements within AUTOSAR runnable entities specifications.
    - The specification shall allow referencing states defined within checkpoint specifications.
    - The specification shall allow the definition of identifiers to be used within the runnable entity’s code for reporting a checkpoint event
        - Checkpoint events are for example "checkpoint reached" and possible additional parameters.
    - The specification shall allow the definition of relations between the events on runnable entity’s variables (read/write), points within the code of runnable entity and states defined within checkpoint specifications.

7.1.4 Control-flow monitoring interface

In order to realize control-flow monitoring within AUTOSAR a communication between the monitored elements (SWC, runnable, internal behavior) and the monitoring element (e.g. watchdog manager) is necessary. This is realized through port prototypes. These port prototypes reference port interfaces. These port interfaces must conform to the specification of the AUTOSAR watchdog manager interfaces.

The interfaces for control-flow monitoring have to be generated automatically by code generators. The requirements to be fulfilled by code generators are:

- An interface for the notification of checkpoints reached has to be generated. The interface shall provide a client server operation named according to the AUTOSAR specification for the notification of checkpoints.
- An interface for the notification of aliveness events has to be generated. The interface shall provide a client server operation named according to the AUTOSAR watchdog manager specification for the notification of an update to the aliveness counter.
- A mode group and the corresponding modes for a given supervised element, for more details refer to the AUTOSAR watchdog manager specification.

- A mode switch interface for the notification of mode changes related to the monitoring of a given supervised element, referencing the corresponding mode group.

7.1.5 Monitoring element

The monitoring element, that is, the component responsible for observing the control-flow of supervised elements, is realized using the AUTOSAR watchdog manager. The Service Software Component for the watchdog manager is responsible for providing the necessary interfaces for the application software components to report their status. The monitoring and check of correctness of control-flow information is realized within the watchdog manager service software component.

The code generator shall generate the following elements for the monitoring element:

- A service software component type defining the watchdog manager
    - For each monitored/supervised element a provided port prototype on this component has to be generated. The interface of the port prototype depends on the type of supervision (control-flow or aliveness).
    - For the service software component an internal behavior element has to be generated
        - For each type of monitoring specified for the monitoring element a runnable entity shall be generated.
        - For each provided port of the service software component, an operation invoked event element within the internal behavior shall be generated, the corresponding runnable entity shall be referenced and the corresponding port and operation referenced.
        - For each of the provided ports of the service software component a port api option element shall be generated, the corresponding provided port defined and a port defined argument value generated with a unique ID identifying the referenced port.
        - In case there is a reaction for given control-flow monitoring errors the following elements shall be generated:
            - For each element specified within the reaction block a Provided Mode Port for status reporting has to be generated.

7.2 Code Generator Inputs

In order to specify the requirement for a control-flow monitor, the engineer needs to provide different resources to code generators. In the following sections these resources are described in details.

7.2.1 Control-flow monitor software safety requirement

The first resource to be provided by the engineer is the control-flow monitor software safety requirement. The input is a model corresponding to the SAFE control-flow monitor meta-model, shown in Figure 20.

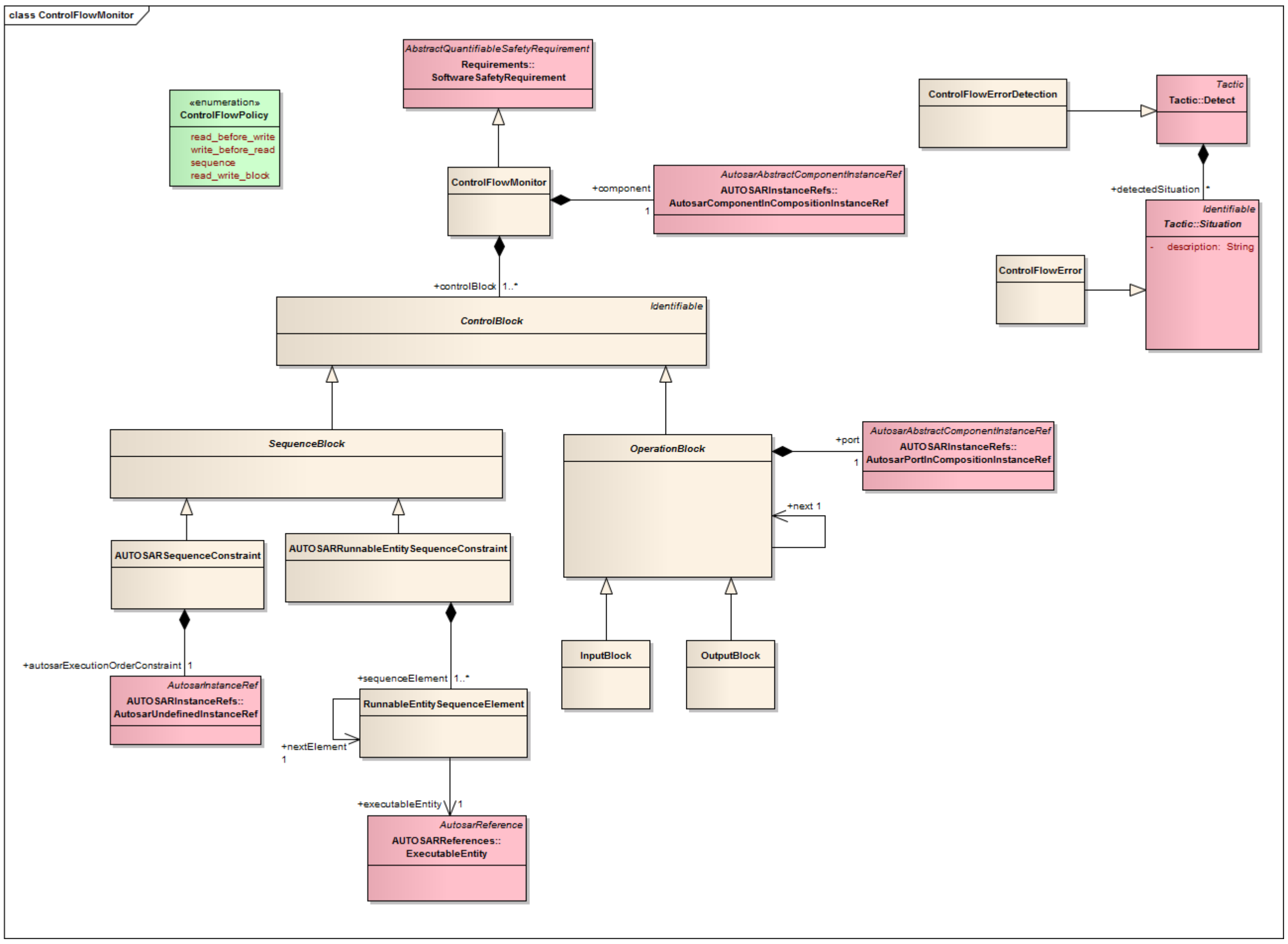


Figure 20 – SAFE meta-model for the specification of a control-flow monitor software safety requirement

Tool implementers might provide the safety engineer with support for specifying a control-flow monitoring requirement. In the next section some DSL constructs which map to the SAFE meta-model for control-flow monitor specification are described. These constructs can be taken as example for the definition of tooling support.

The final decision on what notation to take is to be taken by the tool implementer. The description of the DSL constructs are based on the different kinds of monitored described previously.

7.2.1.1 Software component control-flow monitoring

In the case of monitoring the control-flow of a software component the abstraction level is the interaction of this component with the system. This means the reading and writing of values to the component's ports. This monitoring could lead to the fact that the orchestration of the operations in the ports of different software components within a single software component composition can be specified by the engineer. In the list below some possible ways to model such a requirement are described:

- **Variant 1:**

  ```
  monitor software component <<reference to autosar SWC>> { policy =
  read-before-write }
  ```

- **Variant 2:**

```
monitor software component <<reference to autosar SWC>> { policy = write-before-read }
```

- **Variant 3:**

```
monitor software component <<reference to autosar SWC>> { policy = rw-block
           read_block b1 { (<<reference to autosar required port>>) (, <<reference to autosar required port>>)* }

            write_block  b2  { (<<reference  to  autosar  provided port>>) (, <<reference to autosar provided port>>)* }

           read_block b3 { (<<reference to autosar required port>>) (, <<reference to autosar required port>>)* }

            write_block  b4  { (<<reference  to  autosar  provided port>>)  (,  <<reference  to  autosar  provided  port>>)*  }
     }
```

- **Variant 4:**

```
monitor software component <<reference to autosar SWC>> { policy = sequence

            (<<reference to autosar provided port>> | <<reference to autosar required port>>) ( -> <<reference to autosar provided port>> | <<reference to autosar required port>>)*

     }
```

- **Variant 5:**

```
monitor software composition <<reference to autosar composition>> { policy = sequence

            (<<reference to autosar component>>::<<reference to autosar provided port>> | <<reference to autosar component>>::<<reference to autosar required port>>) ( -> <<reference to autosar component>>::<<reference to autosar provided port>> | <<reference to autosar component>>::<<reference to autosar required port>>)*

     }
```

The type of interaction can also be defined based on port properties, for example, send / receive or call operations. Furthermore, in order to support compositions the DSL could allow the referencing of the composition in question at the beginning of a statement and within the definition of the monitor requirement, the engineer would refer to a specific component within the composition, as shown in Variant 5.

7.2.1.1.1 Internal behavior control-flow monitoring

In the case of monitoring the control-flow of a component's internal behavior it is interesting for the user to refer to runnable entities and to define a certain ordering of execution of the runnable entities which shall be ensured / monitored. The definition of a (partial) order of runnable entities execution could be specified within the DSL. The user has to be able to refer to the timing specification from within the DSL.

Options:

- **Variant1:**

  ```
  monitor internal behavior of <<reference to autosar SWC>> :: <<reference to internal behavior of the SWC>> { control-flow = <<reference to AUTOSAR Timing specification>> }
  ```

- **Variant 2:**

  ```
  monitor internal behavior of <<reference to autosar SWC>> :: <<reference to internal behavior of the SWC>> { control-flow = <<reference to state machine specification>> }
  ```

  - State machine specification provides events related to runnable sequences (e.g. startEvent, stopEvent). This can be used by the engineer to define transitions -> (st1, startEvent(runnable1)) > st2

7.2.1.1.2 Runnable control-flow monitoring

The challenge for monitoring the control-flow of runnable entities is that the instrumentation of the runnable entity's code is necessary. Furthermore, the correct control-flow of the code is not known a priori, that is, at AUTOSAR level, and the safety engineer is possibly not able to determine what a correct control-flow from the runnable entity's implementation perspective looks like.

One possible solution for getting around this kind of issue is to allow a checkpoint specification to be defined and offer the developer of the runnable entity's code to call checkpoints which are automatically mapped to the checkpoint specification. This specification would be a kind of exchange format between the safety engineer and the software developer responsible for a given runnable entity's implementation. This however implies that the safety engineer is aware of implementation details of the runnable entity in question.

Since it is not completely clear how this kind of monitoring could be seamlessly integrated into the development process of safety critical systems, it will not be described in detail in this document. The suggestions described previously could be used by tool implementers in order to create code generators able to integrate the monitoring requirements of safety engineers with the code produced by software developers.

7.2.1.2 System/Software architecture

Besides the requirement specification for control-flow monitoring the engineer has to provide the AUTOSAR model containing the software architecture and the AUTOSAR model containing the system specification determining the deployment of software components to ECUs.

7.2.1.3 Configuration information

The engineer has also to provide configuration information to the code generator in order to specify where generated artifacts are kept, for example a target source folder for generated source code, target model folder for generated model elements and a documentation target folder for documentation related to the generated artifacts.

7.2.2 Code generation

The code generation step for the control-flow monitoring software safety requirement produces the implementation of the logic for controlling the correct transitions specified by the requirement. The code shall provide the necessary interfaces for integration with the AUTOSAR Watchdog Manager Service Software Component Type described in the generated artifacts section (7.3).

Code generators are allowed to define how the meta-model elements are mapped to code. The only restriction is that the generated code is able to validate the control-flow using the Control-flow Monitor SSR defined by the engineer.

Furthermore, since the definition of a control-flow monitor is supported by the AUTOSAR 4.0 standard it is possible to rely on the Watchdog Manager implementation of an AUTOSAR stack complying with the 4.0 version of the standard. In this case there is no need for generating code on the side of the safety code generator, rather only the necessary AUTOSAR model elements.

7.3 Generated artifacts

The specification of control-flow requirements using a DSL abstracts the engineer from the realization details of such requirements in a given target platform. The DSL constructs described in the previous chapter provides the engineer with simple constructs which refer to some AUTOSAR elements in order to allow the specification of control-flow monitoring requirements. These requirements can then be automatically realized within AUTOSAR through the use of a code generation framework.

In this section the artifacts created by the code generation framework are described in more detail. It shall serve as a guideline for tool implementers creating code generators. The following description is however not to be taken as base implementation since it might not provide all the elements necessary for realizing control-flow monitoring in a real AUTOSAR environment.

7.3.1 Service software component type

A ServiceSWComponentType AUTOSAR element is generated in order to define the WatchdogManager AUTOSAR component type. It provides the interfaces necessary for the interactions of regular AUTOSAR software components with the watchdog manager for control-flow and aliveness monitoring.

Name: WdgM.

Ports:

- For each supervised entity specified using the DSL an AUTOSAR provided port is generated.
    - Each provided port provides the corresponding interface for the specified monitoring (either control-flow or aliveness).
- For each error which the watchdog manager is able to communicate a provided port is generated.
    - Each provided port provides the interface corresponding to the mode (error) which is being reported.

Internal behavior:

- For each supervised entity a PortAPIOption has to be defined in order to inform the corresponding runnable entity which control-flow structure has to be used for the monitoring activities.
    - For each port API option a unique ID shall be defined, used internally by the AUTOSAR WdgM BSW module code.

- For each supervised entity an OperationInvokedEvent has to be generated and mapped to the corresponding runnable entity.

7.3.2 Software component prototype

A software component prototype instantiating the watchdog manger in a composition software component type in AUTOSAR has to be created in order to provide the monitoring mechanisms to application software components. The corresponding connections of each supervised entity (monitored by the watchdog manager) to the component's port are also created.

7.3.3 Client server interface

For each type of monitoring (control-flow / aliveness) a client-server interface is generated.

- Control-flow monitoring: provides the operation for notifying about checkpoint reached events. See AUTOSAR watchdog manager specification for the requirements on the operation name.
- Aliveness monitoring: provides the operation for notifying the update of the value of aliveness counters. See AUTOSAR watchdog manager specification for the requirements on the operation name.

7.3.4 Mode switch interface

For each possible error to be communicated from the watchdog manager a mode switch interface is created and provides the corresponding mode group for the interface.

7.3.5 Mode group

For each set of modes used for error reporting by the watchdog manager a ModeDeclarationGroup is generated, the errors related to this group are generate within it. The mode group is referred to by mode switch interfaces used for error reporting.

7.3.6 Mode declaration

For each error reported by the WdgM a ModeDeclaration element is generated.

7.3.7 ECU Configuration

The code generator is responsible for generating the ECU configuration file necessary for the AUTOSAR watchdog manager to work. This file contains different elements and the generation of each element is described below.

WdgM ECUModuleDef

In order to configure the AUTOSAR watchdog manager as a control-flow monitoring mechanism an ECU Configuration file is generated. The file specifies the configuration of a WdgM ECUModuleDef element.

WdgMSupervisedEntity

For each supervised entity (monitored element) specified by the safety engineer a WdgMSupervisedEntity entry is generated in the ECUConfiguration file. It receives a unique ID controlled by the code generation framework.

Checkpoints

Within each WdgMSupervisedEntity entry, the corresponding checkpoints are generated. The checkpoint generation is realized depending on the control-flow monitoring specification. The generation framework takes as input in most cases a state machine representation of the control-flow and generates the necessary checkpoints in the watchdog manager configuration.

WdgMInternalTransition

Each checkpoint is connected to the subsequent one through a set of transitions. Each transition is created as a WdgMInternalTransition by the generator. The transitions are derived from the control-flow monitoring specification.

If the control-flow specification defines an initial state the generator defines within the ECU configuration file which initial checkpoint is to be used as initial checkpoint by the watchdog manager monitoring code.

7.4 Modification to existing AUTOSAR elements

Depending on the type of monitoring specified different adaptations to the AUTOSAR model are necessary. Since the adaptations are highly dependent on the code generation utilized to realize the control-flow monitoring requirement, the adaptations to existing AUTOSAR models are not described in detail in this document. The following sections described in prose how such modifications could look like for a code generation framework.

7.4.1 Software component monitoring

In order to monitor the interactions of a software component (mainly the activity on provided and required ports) either an adaptation to the RTE generation process has to be done in order to intercept calls to the RTE related to a given port. Or the monitored component has to be integrated into a wrapper component which will handle the interception before the call gets to the RTE.

Variant 1 - RTE Adaptation

In this case the RTE is adapted to trigger the watchdog manager monitoring mechanism before the real call to the RTE function for reading/writing sending/receiving values to ports is called. This implies that implicitly the RTE calls the watchdog manager to inform that a certain checkpoint has been reached.

Variant 2 - Wrapper component

When monitoring a software component using a wrapper component, extra wrapper software component type and prototype are generated. It provides the same interfaces as the ones provided by the SWC being monitored. Furthermore, it intercepts the communication between the monitored component and the RTE in order to realize the monitoring. For this reason, the wrapper component is generated with runnable entities which are responsible for the interception of RTE interactions, triggering of the watchdog manager component through the corresponding monitoring ports and forwarding the value to the real target of the monitored RTE interaction (e.g. delivering a received value to the required port of the monitored component).

7.4.2 Internal behavior monitoring

When defining the monitoring of the internal behavior of a given AUTOSAR SWC the engineer defines the possible valid sequences in which runnable entities can be executed. In order to monitor this sequence, the SWC being monitored has to be adapted so that the required ports for the control-flow monitoring are defined within the SWC. Furthermore, the runnable entities have to be adapted to interact with the watchdog manager. If the integration with the RTE is planned – checkpoint notifications done by the RTE – there must be no adaptation to the monitored software component whatsoever.

In case no adaptation of the RTE is possible the following SWC adaptations are necessary and will enable the monitoring of the runnable sequence:

SWC modifications:

- Generator creates new ports required for the monitoring on the given software component.
- Runnable entities are modified to access ports and report to the watchdog manager at the start and end of the runnable.
- A macro for the runnable entities is generated and configured as main function for the runnable entities.
    - The macro is then the function which is called by the RTE when the runnable entity is supposed to be executed and could have the following structure:

```
void runnable_entity_being_monitored_<<name>> () {
     CALL_WDGM_START_(RUNNABLE_ID);
     original_runnable_function();
     CALL_WDGM_STOP(RUNNABLE_ID);
}
```

    - The macro is generated within header files by the generator

7.4.3 Runnable entity monitoring

For the monitoring of the internals of runnable entities a state machine is generated and a set of macros for checkpoint notification are also generated. The software developer of responsible for the runnable entity is also responsible for calling the corresponding macros in the correct sequence and the macros report the checkpoints to the watchdog manager realizing the control-flow monitoring.

8 Detailed Specification of code generation for software safety requirements for CHROMOSOME

In this chapter the requirements for the generation of software safety requirements based on the CHROMOSOME target platform are provided. The information for software safety mechanisms contains a detailed description of the necessary input, the generated artifacts and the integration strategy for preexisting artifacts.

8.1 CPU Self-Test

The CPU Self-test SSR (meta-model depicted in Figure 15) provides the possibility to detect errors of the CPU. It is one of the mechanisms providing the startup proof test and allowing periodically scheduled testing of the CPU in case latent fault detection is also critical for the mission duration.

CPU on the specific node is required to perform a challenge computation, and results are checked against expected output. Detection of an error results from a mismatch of response and expectation.

8.1.1 Inputs

The SSR Specification allows performing CPU tests in one of two main modes: proof test mode and segmented mode, so that no large timeslot is occupied in the schedule by the test. Total number of segments and full execution period are then specified as configuration parameters.

Parameterization provides some flexibility in specifying the mechanism SSR for CHROMOSOME. SSR has to be configured according to the meta-model (Figure 15). One or more *algorithms* can be selected to specify, which types of tests have to be included. Execution of test in segments can also be specified by setting *executeInSegments* flag to true, then *numSegments* determines how many segments should be scheduled within *executionPeriodMs* time interval. Otherwise (*executeInSegments* is *false*) the test is required to be performed as a whole uninterruptable sequence once per *executionPeriodMs* or more often. For CHROMOSOME, the *testedNode* parameter references specific CHROMOSOME node, where the test should be scheduled.

Software Architecture: CpuSelfTest shall be integrated as an instance of a *depend_cpuSelfTest* component, already implemented in CHROMOSOME and described in the safety-specific manifest model. Alternatively, a specialized variant of *depend_cpuSelfTest* shall be generated as a C module, and an instance of this specific variant shall be integrated into the existing CHROMOSOME system.

8.1.2 Code generation

As specified in the Subsection 5.3.2, code generation happens on the basis of a SAFE model and CHROMOSOME model. One component instance of CpuSelfTest is generated per requirement.

Code generator is required to perform the following transformations:

- If *executionPeriod* is equal to zero, the SSM is considered to be a startup proof test, otherwise as a periodic test. In the former case, the schedule *S*, within which the SSM will be scheduled, is *S = {STARTUP}*, and in the latter case it is *S = {NORMAL}*.
- In case of specified *executionPeriod* a component instance shall be scheduled with period *Texec* < *executionPeriodMs* / *numSegments*.
- The *algorithms* field along with *numSegments* should be transferred to the configuration of component instance implementing the SSM.

- Deployment of the generated SSM should occur on the hardware node, specified by the *testedNode* reference.

Constraints for code generation:

Only one instance of a periodic CpuSelfTest is generated. Whenever a concurring requirement is found, the configuration of an existing CpuSelfTest instance is updated to meet the strictest requirement (all specified algorithms and smallest period).

8.1.3 Generated artifacts

The implementation of code generator shall provide the following model elements that make further code generation and traceability possible:

CHROMOSOME deployment model

- An instance CpuSelfTest1 of *depend_cpuSelfTest* with the following parameters:
  - Execution period = *Texec*
  - Active schedule set = *{S}*
  - Number of segments = *numSegments*

SAFE model

- Reference to the component instance CHROMOSOMEReferences::Component with the same name (“CpuSelfTest1”).
- A Satisfy relationship is generated with *satisfiedElement* aggregating the SSR and *satisfyingTargets* referencing the CpuSelfTest1 component instance.

Further process of code generation happens within target technology configuration tool and is described in Appendix B – CHROMOSOME.

8.2 RAM Self-Test

The Memory Self-test SSR (meta-model depicted in Figure 16) provides the possibility to detect permanent errors in RAM cells or logic. It is one of the mechanisms providing the startup proof test and allowing periodically scheduled testing of RAM in case latent fault detection is also critical for the mission duration.

CPU on the tested node executes the specified test on the preconfigured memory address range. As periodic execution of such tests at runtime requires copying of data and is a costly operation, it is possible to specify that the test is executed in chunks, thus splitting the address range into multiple chunks and executing the test in smaller portions.

8.2.1 Inputs

The SSR Specification allows performing RAM tests in one of two main modes: proof test mode and segmented mode, so that no large timeslot is occupied in the schedule by the test. Total number of segments and full execution period are then specified as configuration parameters.

The SSR has to be configured according to the meta-model (Figure 16). One or more *algorithms* can be selected to specify, which types of tests have to be included. Execution of test in segments can also be specified by setting *executeInChunks* flag to *true*, then *numberOfChunks* determines how many segments should be scheduled within *executionPeriodMs* time interval. Otherwise (*executeInChunks* == *false*) the test is required to be performed as a whole uninterruptable sequence once per periodMs or more often. One or more memory ranges limited by *startAddress* and *stopAddress* can be specified to be tested. For CHROMOSOME, the *testedNode* parameter references specific CHROMOSOME node, to which the testing mechanism should be deployed.

Software Architecture: MemorySelfTest shall be integrated as an instance of a *depend_memorySelfTest* component, already implemented in CHROMOSOME and described in the safety-specific manifest model.

8.2.2 Code generation

As specified in Subsection 5.3.2, code generation happens on the basis of a SAFE model and CHROMOSOME model. One component instance of MemorySelfTest is generated per requirement.

Code generator is required to perform the following transformations:

- If *periodMs* is equal to zero, the SSM is considered to be a startup proof test, otherwise as a periodic test. In the former case, the schedule *S*, within which the SSM will be scheduled, shall be set to *S = {STARTUP}*, and in the latter case it is set to *S = {NORMAL}*.
- In case of specified *periodMs* the execution period should be *Texec* < *periodMs / numberOfChunks*.
- Generate the component instance. The ‘algorithms’ parameter values along with *numberOfChunks* and *startAddress*/*stopAddress* should be transferred to the configuration of component instance implementing the SSM.
- deployment of the generated SSM should occur on the hardware node, specified by the testedNode reference.

8.2.3 Generated artifacts

The implementation of code generator shall provide the following model elements that make further code generation and traceability possible:

CHROMOSOME deployment model

- An instance MemorySelfTestX of *depend_memorySelfTest* with the following parameters:
  - Execution period = *Texec*
  - Active schedule set = *{S}*
  - NumberOfChunks = *numberOfChunks*
  - StartingAddress = *startAddress*
  - EndAddress = *endAddress*

SAFE model

- Reference to the component instance CHROMOSOMEReferences::Component with the same name (“MemorySelfTest1”).
- A Satisfy relationship is generated with *satisfiedElement* aggregating the SSR and *satisfyingTargets* referencing the MemorySelfTest1 component instance.

Further process of code generation happens within target technology configuration tool (XMT) and is described in Appendix B – CHROMOSOME.

8.3 Voter

Voter is an SSR (meta-model depicted in Figure 17) providing the possibility to specify comparison of multiple values with each other and provide a consensus value based on a specified voting algorithm.

Voter is usually applied in context of redundant data items being collected from different sources, so that certain tolerance between the values can take place. The values are compared against each other, and discrepancies are reported. At the same time a valid consensus value can be provided in most cases.

8.3.1 Inputs

Specification: Voter is specified with a configuration and specification of input data and resulting data item.

Configuration of Voter should be performed in conformance to the meta-model. *inputData* are parameters, on which voting should happen; *result* specifies Voter output. In CHROMOSOME systems, *inputData* and *result* contain references to CHROMOSOME topics. *result* is correctly selected using algorithm if number of inputs (matching with specified *tolerance*) is larger or equal to *consensusThreshold*.

Software architecture: The basis for Voter implementation is a generic implementation of voting functionality within the *depend_voter* CHROMOSOME component. The component is configurable and implements all the voting algorithms specified.

The *depend_voter* implementation allows snapshot comparison with a specified algorithm of an arbitrary number of input ports *VoterIn* and output of a result into a single output port *VoterResult*. Error management interface is also provided as specified in Section 5.1.

8.3.2 Code generation

As specified in Subsection 5.3.2, code generation is performed on the basis of a SAFE model and CHROMOSOME model.

While transforming the models, safety code generator is required to perform the following model transformations:

- Generate a CHROMOSOME component model “VoterNooM”, where N is the value of *consensusThreshold* parameter, and M is the value of *numberOfItems*. The generated component will have M required input (subscription) ports *{Input1, Input2 …}* and two output (publication) ports*: {result, errorIndication}*.
- Select a node for deployment. In case of one sink for the topic referenced by the result parameter.
- Generate an instance X (incremental serial number) of the component VoterNooM on the selected node. Execution period *Tx* of the voter is set to least common multiple of execution periods of the sinks of the topic referenced by the result parameter.
- Transfer the parameters *algorithm* and *tolerance* to the configuration of the generated instance
- Generate relevant SAFE model entities and references

8.3.3 Generated artifacts

The implementation of code generator shall provide the following model elements that make further code generation and traceability possible:

CHROMOSOME manifest model

- A component VoterNooM as described above

CHROMOSOME deployment model

- An instance VoterNooMX of VoterNooM on *monitoredNode* with the following parameters:
  - Execution period = *Tx*
  - Active schedule set = *{NORMAL}*
  - Algorithm = *algorithm*
  - Tolerance = *tolerance*

SAFE model

- A reference to the component instance CHROMOSOMEReferences::Component with the same name (“VoterNooMX”).
- A Satisfy relationship is generated with *satisfiedElement* aggregating the SSR and *satisfyingTargets* referencing the VoterNooMX component instance.

Further process of code generation happens within target technology configuration tool (XMT) and is described in Appendix B – CHROMOSOME.

8.4 Health Monitor

The Health Monitor SSR (meta-model shown in Figure 18) makes centralized supervision possible and allows execution of preconfigured reactions in response to events, such as a specific combination of runnable modes, or error notifications from error detection mechanisms.

Health monitor is not an implicitly mandatory requirement, but its use is strongly recommended, while the execution of reactions to detected situations requires privileges that other components could fail to possess (to execute signed user code, internal reactions such as node restart, and so on).

8.4.1 Inputs

The SSR specification is trivial, and almost the entire important configuration relies on safety code generator. The SSR for CHROMOSOME is specified with a single parameter *node*, which specifies, on which instance of CHROMOSOME runtime the monitoring should take place.

Software architecture: The implementation of the SSR shall instantiate a separate CHROMOSOME component for the SSM and shall schedule it appropriately. The corresponding implementation of the SSM in CHROMOSOME *depend_healthMonitor* is configured by providing a table of *errorConditions* and corresponding *errorHandlers*. While *errorHandlers* are simply function pointers, *errorConditions* (also “monitors”) are Boolean functions that can be of two predefined types: *stateMonitor* and *indicationMonitor*. When executed, *depend_healthMonitor* iterates through the table, evaluating *errorConditions*, and executes the corresponding *errorHandler* if the condition fires (evaluates to *true*).

The functionality of *indicationMonitor* is based on a subscription to a *healthIndications* topic, whose attribute *eventId* should match the id configured in *indicationMonitor* to fire the *errorCondition*. Thus, any detector SSM can publish *healthIndication*’s with a globally unique *eventId*, and if HealthMonitor is configured to process such indications, necessary reaction will be executed.

The functionality of *stateMonitor* is based on user components publishing their state, and conditions defined in *stateMonitor* which receive it and evaluate the state changes. This mechanism is currently out of safety code generation scope.

8.4.2 Code generation

As specified in Subsection 5.3.2, code generation is performed on the basis of a SAFE model and CHROMOSOME model. Health monitor generation differs from generation of code for detection SSMs, while the error condition/handler table needs to be filled based on the model of the system as a whole, so safety code generator shall process HealthMonitor SSRs before the other ones.

One CHROMOSOME component instance is generated per node.

While transforming the models, code generator is required to perform the following transformations:

- Iterate through all SSR specifications in SAFE model, filtering out the components deployed to the node where HealthMonitor is to be deployed
- Optional: generate the *monitorTable:*
    - For each SSR find all *Tactic*s of type *Handle*
    - Find matching *Detect* tactics
    - For each matching *Tactic* generate an entry in the SSM configuration parameter *monitorTable*:
        - [*indicationMonitor* (with a new unique *eventId*), <*matching ChromosomeHandler function*>].

8.4.3 Generated artifacts

The implementation of code generator shall provide the following model elements that make further code generation and traceability possible:

CHROMOSOME deployment model

- An instance HealthMonitor of *depend_healthMonitor* at the specified CHROMOSOME node with the following parameters:
    - Active schedule set = *{NORMAL}*
    - Monitor table: filled as described above

SAFE model

- References to the component instance CHROMOSOMEReferences::Component with the same names (“HealthMonitor”).
- A Satisfy relationship is generated with *satisfiedElement* aggregating the SSR and *satisfyingTargets* referencing the HealthMonitor component instance.

Further process of code generation happens within target technology configuration tool (XMT) and is described in Appendix B – CHROMOSOME.

8.5 Heartbeat

The Heartbeat SSR (meta-model shown in Figure 19) allows for monitoring reachability and responsiveness of one computing node / ECU from another one.

It consists of heartbeat receiver, a primitive implementation of an aliveness monitor containing one checkpoint, and an event generator, called heartbeat sender. So, heartbeat sender issues a heartbeat signal once per ‘periodMs’ milliseconds and heartbeat receiver checks arrival of this signal within ‘deadlineMs’ milliseconds. It should be noted explicitly that deadline specification should take possible jitter and clock drift into account and correspond to a worst case estimate to avoid false positives.

8.5.1 Inputs

The SSR specification implicitly defines two parts of the heartbeat pattern that need to be allocated to (typically) different CHROMOSOME nodes. The SSR for CHROMOSOME is specified with a sender period *periodMs*, receiver deadline *deadlineMs. monitorNode* specifies, on which node the receiver should be located, and *monitoredNode* specifies, which CHROMOSOME node will become a sender of the heartbeat signals.

8.5.2 Code generation

While transforming the models, safety code generator is required to perform the following transformations:

- the parameter *deadlineMs* is to be transferred into the configuration of HeartbeatReceiver instance;
- execution period is set for HeartbeatSender $Ts = periodMs$ and HeartbeatReceiver: $Tr < deadlineMs / 2$;
- two CHROMOSOME component instances are generated: one of HeartbeatSender component, and one of HeartbeatReceiver. Deployment of the generated SSM component instances should occur on the hardware nodes, specified by the respective *monitorNode* and *monitoredNode* references.

Constraints:

- $2 * periodMs < deadlineMs$

8.5.3 Generated artifacts

The implementation of code generator shall provide the following model elements that make further code generation and traceability possible:

CHROMOSOME deployment model

- An instance HeartbeatSenderX of *depend_heartbeatSender* on *monitoredNode* with the following parameters:
  - Execution period = *Ts*
  - Active schedule set = *{NORMAL}*
- Am instance HeartbeatReceiverX of *depend_heartbeatReceiver* on *monitorNode* with the following parameters:
  - Execution period = *Tr*
  - Active schedule set = *{NORMAL}*
  - Deadline = *deadlineMs*

SAFE model

- References to the component instances CHROMOSOMEReferences::Component with the same names (“HeartbeatSenderX”, “HeartbeatReceiverX”).
- A Satisfy relationship is generated with *satisfiedElement* aggregating the SSR and *satisfyingTargets* referencing the HeartbeatReceiver and HeartbeatSender component instances.

Further process of code generation happens within target technology configuration tool (XMT) and is described in Appendix B – CHROMOSOME.

9 Validation of generated artifacts

There are different approaches to validate the artifacts generated by code generators. The approaches vary basically depending on the type of artifact generated: source code, model elements, documentation and traceability information.

Depending on the target platform being used for system, software and hardware modeling, the output of safety code generators for the validation of the generated artifacts might vary. For example, if the validation of the generated elements is realized by external tools safety code generators might generate either source code for tests or test models which are used as input by external tools used in the validation process.

9.1 Source code validation

In order to validate the generated source code, code generators can also provide support for the generation of unit tests or component tests for the software safety requirements provided as input. In most cases the values provided within the software safety requirement model can also be used to generate tests for in-range and out-of range tests.

For example, the specification of a range-check software safety requirement will contain the range boundary values which in turn can be used for the generation of tests.

9.1 Model element validation

In the case of generated model elements, code generators can either provide the validation of the generated models according to the meta-models of these models, or using external model validation tools. Furthermore in case the code generator is integrated into a modeling tool which supports the generated models, additional information can be added to the generated models in order to provide visual or textual feedback on the implications of the changes triggered by the output of the code generator.

9.2 Traceability information validation

Regarding traceability information, code generators can provide the expected coverage information from generated elements. This information can in turn be used by tools in order to demonstrate coverage or traceability in a more general context, such as the argumentation of coverage and traceability within a safety case.

10 Conclusions and Discussion

In this document the approach for modeling software safety requirements (SSRs) developed during the SAFE Project was described. The approach has been consolidated with the SAFE meta-model, which allows capturing technical safety requirements, refining technical safety requirements into SSRs and providing traceability information between system artifacts and requirements.

Furthermore, the document provides a specification for the implementation of safety code generators which (semi-)automatically transform SSRs into safety critical code and model artifacts. This specification is provided in a generic way so that different target technology platforms can be used for the realization of the SSRs. For reference purposes, the document also provides examples for the realization of code generators for specific target platforms such as AUTOSAR.

The SAFE meta-model for software safety requirement specification was defined in order to formalize given software safety mechanism patterns which are commonly used while developing safety critical systems. The patterns were identified through a state of the art research and classification of safety mechanisms types. Afterwards, the patterns were captured within the SAFE Project as meta-model elements. The meta-model elements capture the essential information for the configuration and deployment of safety mechanisms and allow code generators to produce the necessary artifacts (code and model elements) to realize the requirements.

The formalization of such patterns eases the argumentation of safety and the execution of safety analysis still in an early development phase since analyses can be realized on requirements level specified based on the preliminary system architecture. Furthermore, not only the automatic generation of code and model artifacts is possible, but also the automatic consolidation of safety arguments, also based on the SSRs, into the safety case argumentation.

With more fine-grained SSRs formalized in the meta-model a more structured argumentation on safety can be done. Moreover, activities related to analysis such as FTA can be eased since SSRs can be linked to other model elements indicating that a safety measure is provided in order to cope with errors defined using the SAFE error model.

11 References

[1] www.autosar.org

[2] http://www.fortiss.org/en/research/projects/chromosome/

[3] http://www.misra-c.com/Activities/MISRAC/tabid/160/Default.aspx

[4] International Organization for Standardization: ISO 26262 Road Vehicles – Functional Safety Part 6

[5] http://www.artop.org/arunit/

[6] http://www.east-adl.info/

[7] http://www.autosar.org/download/R4.0/AUTOSAR_EXP_VFB.pdf

[8] http://www.artop.org/artext/

[9] http://www.eclipse.org/xtend/

[10] SAFE Full Project Proposal (FPP)

[11] http://www.autosar.org/download/R4.0/AUTOSAR_SWS_WatchdogManager.pdf, Page 8, Section 1.1 – Supervised Entities and Checkpoints

[12] http://www.mathworks.de/products/simulink/

[13] http://www.autosar.org/download/R4.0/AUTOSAR_TPS_SoftwareComponentTemplate.pdf

[14] Weihang Wu and Tim Kelly. 2004. Safety Tactics for Software Architecture Design. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01* (COMPSAC '04), Vol. 1. IEEE Computer Society, Washington, DC, USA, 368-375.

[15] http://www.hq.nasa.gov/office/codeq/doctree/871913.htm - NASA software safety handbook

[16] http://www.autosar.org/download/R4.0/AUTOSAR_TR_SafetyConceptStatusReport.pdf

[17] http://download.fortiss.org/public/xme/xme-0.6-tutorial.pdf

[18] http://www.autosar.org/download/R4.0/AUTOSAR_SWS_WatchdogManager.pdf

[19] http://www.eclipse.org/xtend/

[20] http://www.stack.nl/~dimitri/doxygen/index.html

[21] http://www.gtk.org/gtk-doc

12 Acknowledgments

This document is based on the SAFE and SAFE-E projects. SAFE is 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). SAFE-E is part of the Eurostars program, which is powered by EUREKA and the European Community. The work has been funded by the German Ministry of Education and Research (BMBF) and the Austrian research association (FFG) under the funding ID E!6095. The responsibility for the content rests with the authors.

Appendix A – Classification of Software Safety Mechanisms

In Sections 6.4, 7 and 8, a detailed specification of software safety requirements has been presented for a selected group of safety mechanisms. However, there are many other mechanisms which allow the realization of software safety requirements.

SAFE has done an extensive research [14][15][16], review of ISO 26262 recommendations and internal discussions amongst project partners regarding safety mechanisms which can be implemented in software, namely, which could be specified as SSRs. For these mechanisms a classification has been done together with a technical description and semantic specification for each of the mechanisms considered in the classification.

The classification results obtained during the activities of WT 3.6 are presented in this section in order to allow further extensions of the SAFE meta-model to incorporate mechanisms not yet modeled but already researched during the SAFE project.

General Classification of Mechanism Types

The safety mechanisms were initially divided into three groups. The definition of each group has to do with the separation of concerns of each of the mechanisms. The first group is fault avoidance and it regards mechanisms which implement fault avoidance techniques. For instance, a given system might fail if a failure manifests itself as a fault to another co-existing (e.g. executed within the same controller) although independent (e.g. belonging to different protected memory partitions) part of the system. These mechanisms prevent faults from happening and are therefore classified as fault avoidance mechanisms. In order to clarify the terminology the following considerations have to be made:

Fault – abnormal condition that can cause an item to fail (ISO 26262 – Part 1 – Vocabulary) (e.g. bit flip in memory leads to erroneous address in memory).

Error – manifestation of a fault (ISO 26262 – Part 1 – Vocabulary) (e.g. invalid address in pointer variable).

Failure – termination of the ability of an item to perform a function as required (ISO 26262 – Part 1 – Vocabulary) (e.g. crash due to invalid pointer).

Errors are the manifestation of faults and therefore cannot be avoided. In order to prevent an error from happening one must avoid the fault that causes the error to appear. Therefore the term "**Fault avoidance**" is used to describe mechanisms which prevent the manifestation of faults in the form of errors.

The second group of mechanisms is responsible for the detection of errors, namely error detection mechanisms. The error occurs because the fault which manifests itself as an error could not be avoided. In this case the system has to provide means to detect that an error has occurred. This group has been subdivided into two categories. The first category groups mechanisms where the error detection depends only on the current data, i.e. the current port input value. Therefore this group is named stateless error detection. The second category contains mechanisms which in order to be implemented need previous information about the computation process. In this case these mechanisms are called stateful error detection, e.g. logical monitoring of program sequence.

The third group represents mechanisms which handle errors once they have been detected. There is a dependency between error detection and error handling since the error handling mechanisms must become aware of errors detected by the error detection mechanisms.

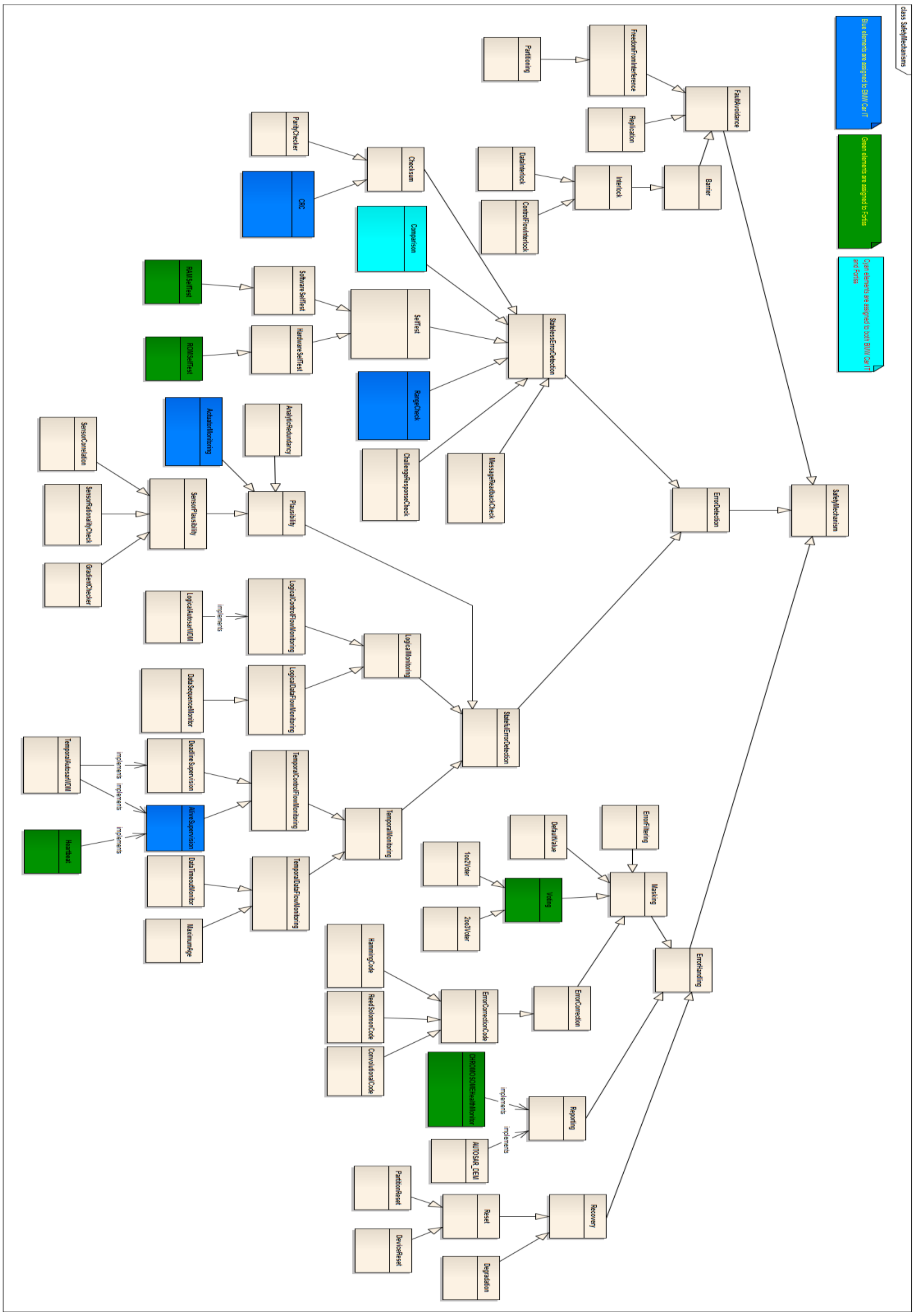


Figure 21 – Software safety mechanism structuring and classification

Semantics

In the following the semantics of each mechanism that can be automatically generated by a safety code generator is described. The semantics serves as a base for specification of the generation process and also for the definition of meta-model constructs which can be used by engineers to specify the safety mechanisms used in a given system. In the semantics specification, when the result of a computation or mechanism is false, the interpretation shall be that an error has occurred. A generated safety mechanism can react to detected errors e.g. by reporting the error to the error forwarding or communication system which could then in turn trigger error handling mechanisms for the given error.

Fault Avoidance

Fault avoidance mechanisms prevent errors triggered by faults from manifesting. The mechanisms are responsible for impeding the execution of actions (e.g. reading memory regions, activating runnables) which if not impeded would lead the system to an unsafe state.

Partitioning

With partitioning the reader shall understand memory partition and the deployment of functions (runnable or task) to this partition. This allows the implementation and argumentation of freedom from interference. Since this technique prevents errors related to illegal memory access and the interference between software elements it is classified as a fault avoidance safety mechanism.

Capabilities:

Prevents errors caused by invalid or illegal access to certain memory regions. This mechanism prevents independent elements from causing/suffering interference from arbitrary system elements.

Semantics:

partition *X,Y,…*

----

system element *X,Y,…* will be executed in the same partition

Interlock

The interlock mechanism avoids errors by requiring a pre-defined sequence of actions to be executed before the targeted action can actually be executed. Through an interlock an error is triggered when incorrect sequence of actions for a given resource protected using this mechanism are executed.

DataInterlock

A Data interlock mechanism computes the plausibility of two or more data values from distinct channels which refer possibly to different temporal states and feeds the result to a software component, such that the result value is interpreted as limiter or enabler. Thus, hazardous effects as a result data failure are avoided. The forwarded result could also be non-discrete.

Capabilities:

Prevent erroneous data communication when the current state is an error state

ControlFlowInterlock

A Control flow interlock provides one data value (discrete or non-discrete) as enabler for an activity, e.g. the execution of a runnable entity. Thus, a runnable can be blocked as long as its execution could have hazardous effects. The interlock is defined as the correct sequence or set of control flow actions which must be executed in order to produce a valid state for the controlled activity.

Capabilities:

Prevent erroneous activation/execution of system element when the current state is an error state

Replication

A replication mechanism is responsible for copying data multiple times in order to reduce the probability of a fault manifesting itself. A clear example that describes this concept is message replication on a bus. The replication requires no computing on the receiver side and a simple repetition on the sending side. The mechanism realizes fault avoidance by reducing the probability that a bus fault manifests itself as a “message not received” error. The main characteristic of this mechanism is that it does not require any processing on the receiver side, contrary for example of a mechanism which saves the same data on different memory regions whereby in this case the receiver has to check different locations and compare data.

Capabilities:

Avoid random faults during the transmission of values from manifesting

Semantics:

$$\frac{\text{replicate } X\ Y \text{ times}}{\text{repeat } Y \text{ times the action executed for } X}$$

Error Detection

The mechanisms described below comprise stateless and stateful safety mechanisms. The text structure will not be subdivided into these classes due to readability of section numbering. For a structural overview please refer to Figure 21. In the following an explanation about the stateful and stateless definitions is given.

Stateless error detection is the term used in this document to describe error detection mechanisms which detect system errors independently of system state or state history. The mechanisms do not require information about previous system to determine if a given error occurred. An example is a CRC check which does not require information about previous values to determine if data has been corrupted.

Stateful error detection refers to mechanisms where the error detection does not only depend on the computation performed by the mechanism but also on previous and current system state. For example, a gradient check mechanism will always depend on system timestamp (be it discretized in steps or other time units) and system state (i.e. previous data value).

ParityCheck

Parity checkers might check values for different numbers of parity bits. The most common case is one bit parity. However this only enables the detection of odd bit errors. More parity bits can be used to increase error detection capabilities.

Capabilities:

Detect data corruption during data communication or data storage.

Semantics:

```
checksum X with Y bits parity
------------------------------------------
parity(X,Y) ? true : false
```

MessageReadbackCheck

A message read back check provides support for validating if sent date was correctly sent. This is usually done at hardware only or hardware dependent levels. The reason is that the validation of sent data happens at the "transport" layer. For example, while transmitting CAN controllers must read back the bits sent on bus ("listening") to make sure there are no collisions and that bus arbitration worked correctly. However at this level, the mechanism is not controlled by software but actually by the transmission hardware at physical level. The software part can control the execution of such mechanism.

Capabilities:

Detect hardware errors on communication components or transmission medium

ChallengeResponseCheck

This mechanism consists of a set of challenge-response pairs whose challenge is sent to another element on the system and the result of the computation is checked against the response value previously known to the sender.

Computations can be as simple as the storing of a value in a register and sending it back, to more complex schemes such as calculating some hash or cryptographic function or even consider system state, time and history and then sending the result back.

The specification of the mechanism can consist of a sequence of challenge-response values or of a single one.

Capabilities:

Detect hardware, state or communication errors

Semantics:

```
challenge comp with values=[(challenge,response)]+
--------------------------------------------------------------------------
foreach ch,re in values do:
    ret = let comp compute ch
    if (ret != re) then
        error detected
    endif
done
```

SensorCorrelation

This mechanism correlates the value of two similar sensors against each other. For example two sensors with inverted slopes will allow the detection of measurements due to corruption of one or both signals. The values must first be converted to the same slope.

Capabilities:

Detect errors near sensor operational limit or sensor measurement errors

Semantics:

correlate $s1^{[-x]}$ and $s2^{[-y]}$ with maximum tolerance X

where:

-x,-y = discrete time step in the past (or zero)

$s1^{-x}$ = The value of s1 at time current_time-(x time steps)

$s2^{-y}$ = The value of s2 at time current_time-(y time steps)

s1value(y) = <<user defined function>>

s2value(z) = <<user defined function>>

----

v1 = s1value(s1)

v2 = s2value(s2)

compare abs(v1-v2) lte X

SensorRationalityCheck

The rationality check of sensors is done using different input sources. The rationality comes from the comparison of different sensor values and the interrelation between their values. Example given by the ISO 26262 is the rationality check of air flow on the engine. The throttle position, manifold pressure and mass air flow values are taken and converted to the same unit of measure (air flow). The values are then compared to check for possible sensor errors.

Capabilities:

Detect sensor measurement errors

Semantics:

rationalize $s1^{[-x]}$,$s2^{[-y]}$,$s3^{[-z]}$ with maximum tolerance X

where:

-x,-y,-z = discrete time step in the past (or zero)

$s1^{-x}$ = The value of s1 at time current_time-(x time steps)

$s2^{-y}$ = The value of s2 at time current_time-(y time steps)

$s3^{-z}$ = The value of s3 at time current_time-(z time steps)

s1value(y) = <<user defined function>>

s2value(z) = <<user defined function>>

s3value(t) = <<user defined function>>

----

v1 = s1value(s1)

v2 = s2value(s2)

v3 = s3value(s3)

(compare abs(v1-v2) lte X and

compare abs(v2-v3) lte X and

compare abs(v1-v3) lte X)

AnalyticRedundancy

The analytic redundancy mechanism realizes redundancy through a model of the element realized redundantly. The redundancy is made through computation of values following the analytical model and comparing the computed value with the value received from the element for which the mechanism is providing redundancy.

Capabilities:

Detect erroneous values provided by system elements

LogicalControlFlowMonitoring

The logical monitoring (e.g. AUTOSAR WDM logical monitoring features) enables the engineer to specify the logical behavior according to which a given software element must be executed. The steps might be specified using, for example, state machines.

Capabilities:

Monitor logical behavior of elements and detect logical ordering errors

Semantics:

check logic of X with SM Y

(s,s') in Tx and (s,s') in Tsmy ? true : false

Where:

s,s' are states of *X*

(*s*,s') is a transition from *s* to *s'*

T*x* = the set of (*s*,*s'*) from *X*

T*sm* = the set of (s,*s'*) from *Y*

DataSequenceMonitor

This mechanism monitors the sequencing of data elements which arrive via a communication link and are forwarded to system elements. Data element sequencing requires a dedicated timestamp or sequence counter field in the monitored *VariableDataPrototype*.

The expected data sequence can also be defined as a more holistic data flow model, e.g. based on petri nets, which would allow the engineer to describe more complex consistency constraints for compositions of software components.

Capabilities:

Detect errors in the ordering of data messages

Semantics:

```
monitor sequence of element with f(x)
where:
f(x) = <<user defined function>>
--------------------------------------------------------
if compare f(*last_id*) eq *element.id* then
    *last_id = element.id*
else
    *error*
```

When the data sequence is defined through petri-nets or similar formalism the semantics shall be the following:

```
monitor sequence of E with P
where:
E = set of elements which have their sequence monitored
P = formalism which describes the interdependency between element sequences
----------------------------------------------------------------------------------------------------------
nE = compute(E,P)
for each ne,le,e in nE,lE,E
    if ne.id = e.id
        le.id = e.id
    else
        error
end for each
```

DeadlineSupervision

The deadline supervision mechanism monitors for the deadline of specific parts of a given software component. Deadlines can have a minimum and maximum parameter in order to define a time window in which the SW-C shall notify the WD that a given part of the component executed/is going to execute. Deadlines are seen as time windows between checkpoints by AUTOSAR 4.0. Therefore, one way to see deadlines is the time window between the reaching of a checkpoint *A* and the transition to checkpoint *B*.

This mechanism is a simple case of the LogicalControlFlowMonitoring and thus easier to implement and use.

Capabilities:

Detect the violation of deadlines for given functions

Semantics:

```
supervise X on deadlines [{cpA,cpB,min,max}]+
-------------------------------------------------------------------------------------------------------------------
configures AUTOSAR WDM to check X according to checkpoints and deadlines specified by
deadlines
```

DataTimeoutMonitoring

Data timeout monitoring is a mechanism which monitors the arrival interval of data. It differs from maximum age monitoring in the sense that data can have a short validity but a greater periodicity (e.g. data from sensors are valid only for 3ms and a given software unit expects valid data every 10ms). This means that the time for reaction (processing of data) is short (e.g. 3ms) but the periodicity of data is longer (e.g. 10ms).

Capabilities:

Detect timing violations on communications/data periodicity

Semantics:

timeout after X units

where:

units = system time unit (e.g ms, us, ns)

----

triggers error if no data is received within *X* time *units*

(could be implemented as annotation e.g. @timeout(5 ms))

MaximumAgeMonitoring

The maximum age monitor shall check if a given message/data is still valid according to an expiration date. This expiration date determines how long a message/data can be used before its age can influence the behavior of the system. There are implications when using this mechanism regarding the synchronization of clocks within the (possibly distributed) system architecture.

Capabilities:

Detect validity/expiration violations on data usage

Semantics:

maxage of A is X

----

cur_timestamp - A.timestamp < X ? A : invalid

Error Handling

In this section the mechanisms for error correction compiled for WT 3.6 are described. Once more, the document structure will not reflect the structure depicted in Figure 21 due to readability reasons.

DefaultValue

This mechanism realizes masking of errors in the sense that erroneous values are not forwarded to the system but substituted by a default and valid value instead. The mechanism can silently realize the feature or together with the masking notify / register an error.

Capabilities:

Prevent erroneous values from being forwarded to the system

Semantics:

X defaults to *last*

----

if an error occurs the value for *X* defaults to the last valid value received

X defaults to *Y*

----

if an error occurs the value for *X* defaults to *Y*

X defaults to *average*

----

if an error occurs the value for *X* defaults to the average of the valid values received

Reporting

A reporting mechanism handles errors by communicating them to interested parties. When a given error occurs and the reporting mechanism is assigned as handling mechanism the error shall be somehow persisted and communicated to other system elements which might be influenced by the error.

Capabilities:

Handle errors which do not directly impact a system element but may lead to system failure if not noticed by other components.

Semantics:

```
on X report to Y
where:
X := error event (detected)
Y := set of system elements (e.g. SWC)
---------------------------------------------------
for each y in Y do:
    notify y of X
end for each
```

Reset

A reset mechanism can be defined to help the system recover from an error. The reset can be applied to a given software partition (memory partition). Meaning the application deployed to this partition is reset. Or it can be defined for the whole system (ECU) and the MCU will be reset.

Capabilities:

Handle errors which require a system element or system restart

Semantics:

reset mcu on X

----

if event *X* happens reset mcu

reset partN on X

----

if event *X* happens reset partition partN

Where:

X is an event which might be triggered, for instance, when a supervised entity violates a checkpoint/deadline

Degradation

Systems can fail gracefully if the specification defines methods through which the functionality of the system can be reduced when error occurs. One idea is to specify a degradation tree, as shown in Figure 22, in which the different functionality levels (qualitative or quantitative) are specified. Upon an error functionality is degraded and when no solution is available an error is reported.

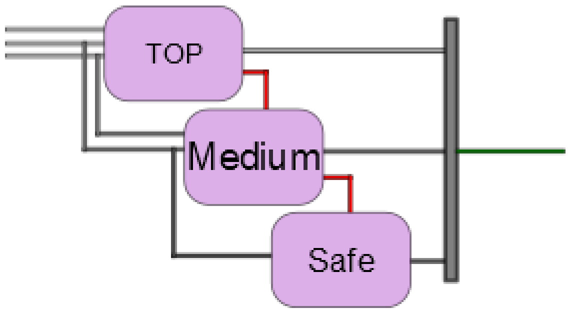


Figure 22 – Suggested approach for degradation modeling

The approach could be to specify the functionality of system components in a way that a logical sequence for degradation is achieved, degradation tree could be modeled in a manner similar to Boolean circuits. This could be in turn transformed into code that disables/enables given functions in order to maintain system quality or system safety level. However, the specification of degradation behavior might not be that simple.

Capabilities:

Keep system operational even when the normal operation of the system might be hindered by errors or damaged system elements.

Appendix B – CHROMOSOME

CHROMOSOME (often abbreviated by “XME”) is a domain-independent, data-centric middleware for cyber-physical systems. From the point of view of an application component, CHROMOSOME abstracts from basic functionality that is traditionally found in operating systems and middleware, like scheduling and communication. Apart from that, it offers model-driven design tools with code generation capabilities that allow a user to design the distributed system in an abstract way.

Apart from its configurable nature, CHROMOSOME sets a high goal to support adaptive system development by allowing dynamic reconfiguration of a distributed application at runtime. The configuration of an application is changed during a special “plug” phase, which ensures that runtime system is reconfigured consistently to keep real-time guarantees.

Some more information on core CHROMOSOME concepts can be found in CHROMOSOME tutorial [17].[1]

Generation of Runtime for CHROMOSOME

Chromosome development takes place in a manner similar to the AUTOSAR concept of VFB [7]. The model specifies some aspects of deployment, but allows low-level aspects to be generated by configuration tools without influencing the functional architecture (e.g., the data channels between component instances as well as the glue code to instantiate components are generated during configuration phase). Configuration of a distributed application developed with CHROMOSOME succeeds via a special configuration tool, which generates the final deployment in an automated model-based fashion.

The configuration tool co-developed and provided with CHROMOSOME is called CHROMOSOME Modeling Tool (XMT). XMT addresses in first place the needs of configuring the data-centric communication through generation of necessary marshaling and network communication components, which are not explicitly specified by the user. It also generates instance information for components configured to be executed as parts of distributed application. Scheduling of components is also done automatically (user specifies the period of application execution, and schedules, in which the component should be present).

CHROMOSOME Model types

In XMT, a model of a system is composed of multiple models defining specific aspects of the middleware configuration ([17], sec. 5):

- Topic dictionary model: defines the set of topics and associated attributes to be used by the component interfaces
- Manifest model: defines component blueprints (also referred to as “component types”), their configuration parameters and interfaces (input/output ports).
- Device types model: defines the types of hardware nodes available for deployment and their communication-specific properties like network interface types.
- The Deployment model links the other models together. Here the distributed application is specified: components described by the Manifest are instantiated at specific nodes, and

[1] All the presented information is valid for CHROMOSOME release 0.6.

ComponentInstance-specific configuration is done (execution period and component-specific parameters).

Not to go deep into details, it is worth mentioning that a component type modeled in the Manifest model corresponds to a component implementation in form of a C module within the CHROMOSOME source code or specified by user.

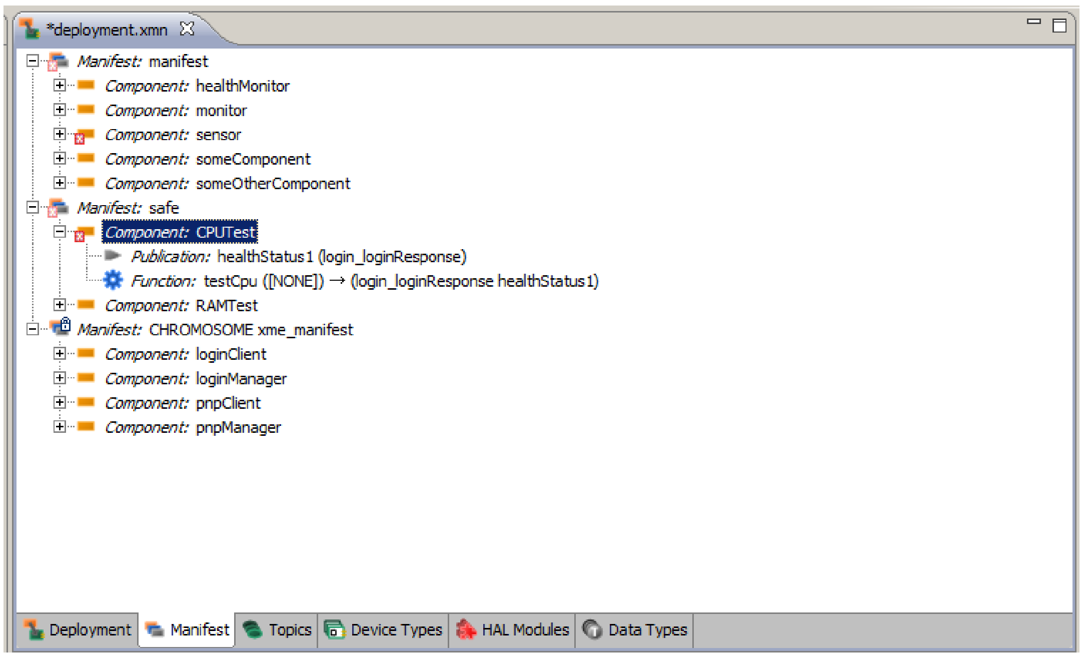

```
*deployment.xmn
Manifest: manifest
  Component: healthMonitor
  Component: monitor
  Component: sensor
  Component: someComponent
  Component: someOtherComponent
Manifest: safe
  Component: CPUTest
    Publication: healthStatus1 (login_loginResponse)
    Function: testCpu ([NONE]) → (login_loginResponse healthStatus1)
  Component: RAMTest
Manifest: CHROMOSOME xme_manifest
  Component: loginClient
  Component: loginManager
  Component: pnpClient
  Component: pnpManager
```

Figure 23 – CHROMOSOME Modeling Tool (XMT)

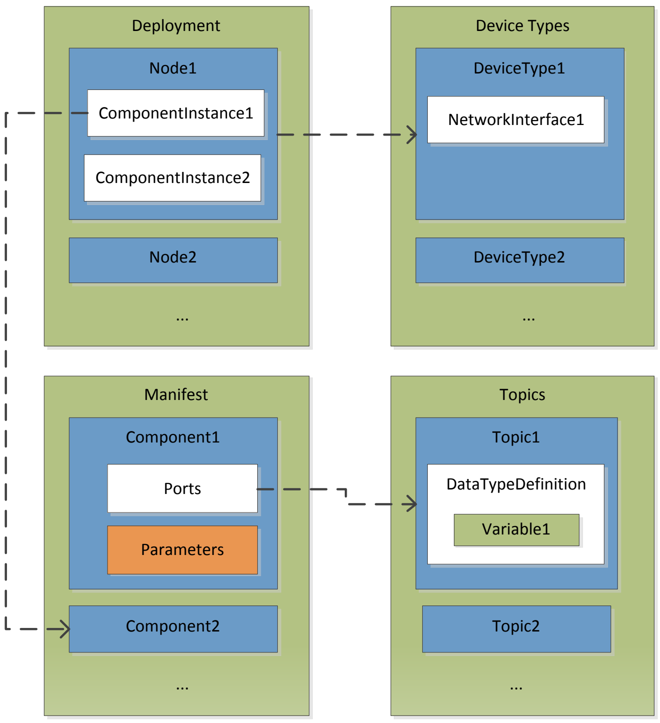


Figure 24 –Different meta-models within CHROMOSOME meta-model in XMT

An Example of Code Generation Workflow for CHROMOSOME

In this section a general outline of code generation scenarios is provided, where SAFE SSRs are implemented by CHROMOSOME SSMs.

To preserve source model traceability, the model transformation tool shall not operate on input model files, but rather on a copy of the SSR model expressed in terms of SAFE meta-model.

An overview of the workflow for the code generation process is depicted in Figure 25.

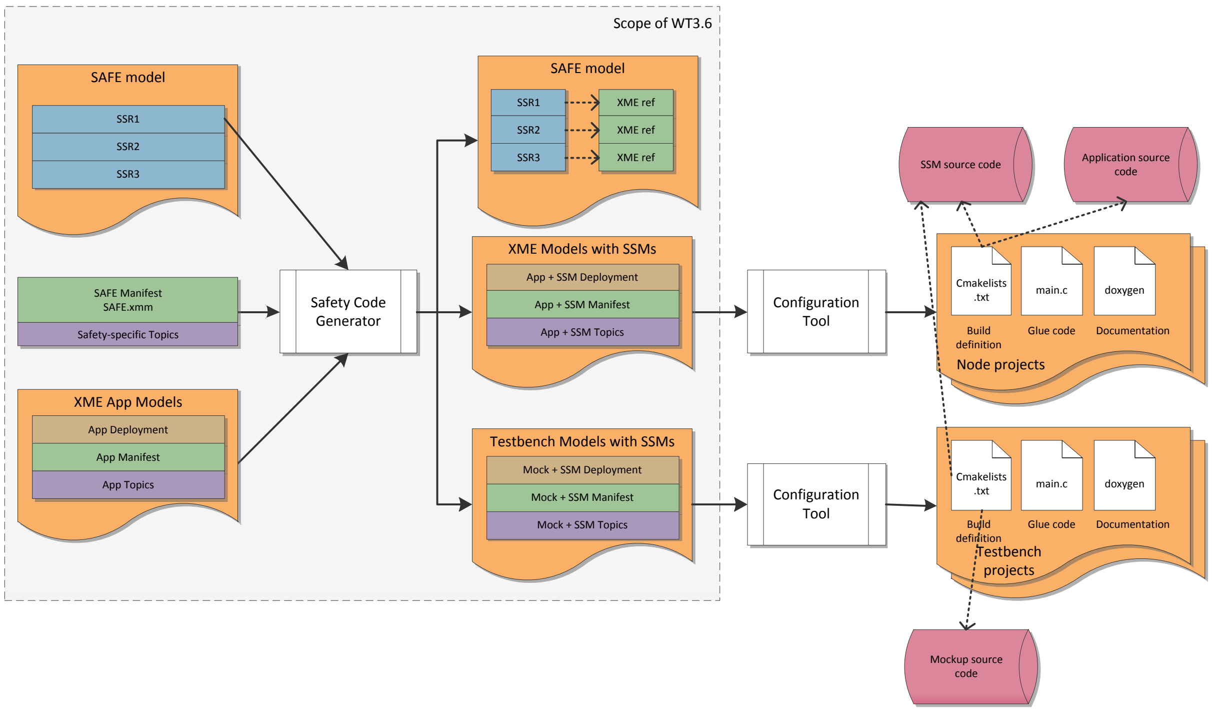


Figure 25 – Illustration of the code generation workflow for CHROMOSOME

The models and other inputs specified in the workflow need to be described in finer detail. Typical transformations performed on those models also need to be specified.

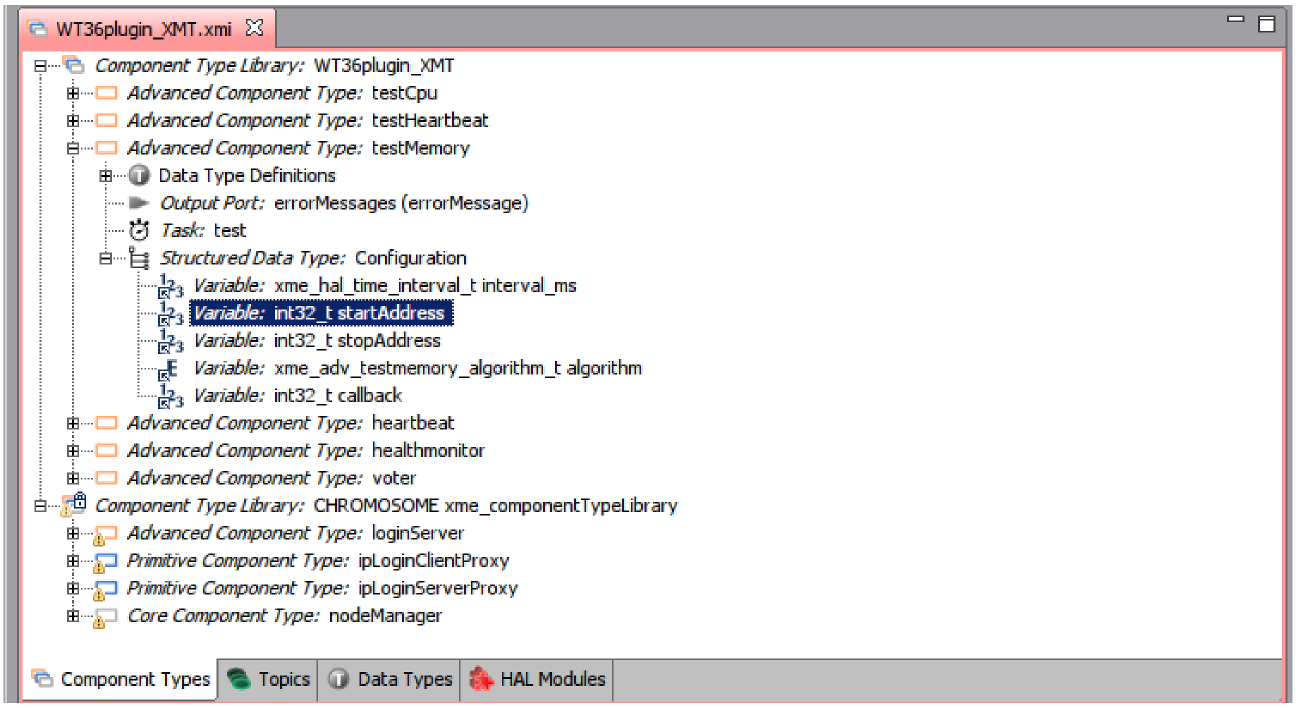
CHROMOSOME models

The model of a CHROMOSOME system is transformed into C code by a CHROMOSOME-specific code generation tool (to distinguish from safety code generator, this tool is further called configuration tool). A configuration tool supplied with CHROMOSOME is CHROMOSOME Modeling Tool (XMT). Configuration tool performs integration of predeveloped C components with custom configuration of such components provided by the CHROMOSOME deployment model and instantiation of data flow components between them with generated CHROMOSOME runtime.

CHROMOSOME by initial design features no safety view on the system as a whole, and provides only implementations of component behavior in form of a library of software modules implemented in C, which integrate into a health management subsystem through tool-supported code generation process (e.g., through code generation with XMT).

To improve the situation and simplify the development of safety code generators, a CHROMOSOME manifest model defining safety components has been created. It can be imported in the form of “SAFE.xmm” manifest by any other CHROMOSOME model, and then used to specify instances within the deployment model.

In our preliminary approach the generated CHROMOSOME models always import “SAFE.xmm” and rely on the components modeled in this manifest model.



```
WT36plugin_XMT.xmi
Component Type Library: WT36plugin_XMT
    Advanced Component Type: testCpu
    Advanced Component Type: testHeartbeat
    Advanced Component Type: testMemory
        Data Type Definitions
        Output Port: errorMessages (errorMessage)
        Task: test
        Structured Data Type: Configuration
            Variable: xme_hal_time_interval_t interval_ms
            Variable: int32_t startAddress
            Variable: int32_t stopAddress
            Variable: xme_adv_testmemory_algorithm_t algorithm
            Variable: int32_t callback
    Advanced Component Type: heartbeat
    Advanced Component Type: healthmonitor
    Advanced Component Type: voter
Component Type Library: CHROMOSOME xme_componentTypeLibrary
    Advanced Component Type: loginServer
    Primitive Component Type: ipLoginClientProxy
    Primitive Component Type: ipLoginServerProxy
    Core Component Type: nodeManager
```

Figure 26 – Manifest model with components specification to support WT3.6 SSM generation

SAFE model

A model acting as an instance of SAFE meta-model shall be used as input for SSR generation. This model can be presented in SAFE interchange format, in terms of a subset of SAFE meta-model implemented in an isolated modeling tool, or in a textual language derived from such a meta-model subset.

For prototyping reasons within WT3.6 the relevant subset of SAFE meta-model has been implemented using EMF (Eclipse Modeling Framework) partially using SAFE Technology Platform's ecore as a basis for building such a prototype.

Safety code generator

The relationship between the SAFE and CHROMOSOME models is established within the safety code generator logic based on matching entity names and enforcement of a global naming convention. The safety code generator mainly acts as a model transformation tool rather than code generator.

Transformation of multiple input models (predefined CHROMOSOME model and SAFE SSRs) can be implemented using one of the numerous model transformation frameworks. Almost every modeling framework or tool today is accompanied by at least one model transformation framework. To be mentioned as an example only, Eclipse Modeling Framework and the Xtend framework [19] may be used, which provides rich capabilities in specification of meta-models, model transformation and code generation.

The following subsections discuss the necessary steps that safety code generator shall perform:

Generation of additional artifacts into the CHROMOSOME model

New model entities shall be generated in the CHROMOSOME models, such as new topics, CHROMOSOME components and CHROMOSOME component instances. The possible generated entities are specified in this section in a generalized form, and defined more precisely in the following section, where safety code generation for specific SSRs is discussed.

1. Chromosome component instances are normally generated to implement an SSR. Such component instances perform required detection or handling of errors.
   For the sake of example, let us take a look at a simple system with two components, Sensor and Actuator, which are connected to each other. Sensor publishes topic pressureValue, and Actuator subscribes to this topic. All of the components are running on a single node. An example SSR on such a system is to perform periodic CPU tests of the node hardware. Such a requirement leads to generation of an instance of xme_core_depend_cpuTest component with configuration parameters inferred from the SSR specification.
2. If the number of input data items for an SSR is not known in advance (e.g., for Voter SSR it is a part of SSR specification), models of components with required number of input ports need to be first generated into the manifest model.
3. Some SSRs require for their implementation, that the data flow between the component ports is changed. In this case before a component instance is generated, new topic definitions need to be generated, manifest and deployment model for the related component(s) need to be updated with the re-routed topic as port data item.
   For example, instantiation of plausibility checking components like RangeCheck requires such an update of the dataflow. In the example case the generator shall create a new topic with name "rangeNotChecked_pressureValue", and update the deployment model and manifest, so that the source component generates the new topic, and an instance of Actuator subscribes to the original one. RangeCheck instance has then a subscription to rangeNotChecked_pressureValue and publishes pressureValue.
4. Some components require more complex queries and modifications being performed on the input models. For example, HealthMonitor requires iteration through all other SSRs for the reaction configuration table to be generated.

Generation of error detection SSRs is performed sequentially one-by-one, allowing an easy query-and-update workflow for safety code generator. One exception is error management components.

Error management for CHROMOSOME is centered on using HealthMonitor. All CHROMOSOME components matching the detector SSRs have the following configuration parameters: a Boolean flag *localHandling* indicates whether detected errors should be handled by the specified function, and *handler* – name of the provided function. Thus there has to be a common step in generation of all detector SSRs: if a HealthMonitor is present on the node, where the detector SSR will be implemented, localHandling is set to false. Otherwise, localHandling is set to true, and handler is filled in by the corresponding ErrorHandler, which is found by exploring the Tactics of subclass Handle, which refer to the same Situation as the Detect Tactics associated with the detector SSR. It is the job of safety code generator to acquire error management information from the model and apply it consistently throughout the generation process.

Generation of references to the generated artifacts within the SAFE model

SAFE meta-model allows referencing CHROMOSOME entities. Such references shall be created by the code generator. The references created in the SAFE model shall be bound to the related SSRs by the Satisfy requirements relationships.

In the above example, the CpuSelfTest SSR should be bound to the instance of the xme_core_depend_cpuTest component, as shown in the below figure.

Generation of tests

As it is not possible to directly test the code generated within such a setting, generation of testbench models is proposed. Generated safety component implementing SSM specified by a respective SSR is tested within a testbench project, which is a CHROMOSOME system model with publishers and subscribers mocked. SSM components that detect hardware errors only (like CpuSelfTest realization) shall be testable through introduction of a private interface for injecting errors by the testbench.

Additionally, to support test generation, library of mockup component implementations in form of C code has to be provided.

Generated documentation

Generated component instance models in CHROMOSOME models shall be annotated in a way that the configuration tool is capable of generating source code comments or documentation based on the annotations. In case of generated comments, documentation could be later build by document generation tools, such as, for example, Doxygen [20], gtk-doc [21] or similar. The annotation shall include information on the source SSR and other related component instances (if any).

Deployment generation

The output of safety code generator is a deployment specification in form of a CHROMOSOME deployment model (and possibly topic dictionary and manifest, if required). The code generator performs necessary conversions from the SSR specification to extract information necessary for scheduling and extension of data flow. The deployment model specification for every generated component instances includes:

- Scheduling parameters
  - Schedules (node states), in which the component should be executed
  - Execution period in milliseconds
- Component-specific parameters defined by the component manifest. For specific components implementing SAFE SSMs the configuration parameters are discussed in respective subsections of the next Section.

The specification from the deployment model is satisfied by XMT configuration tool. It performs schedule generation and generates data flow components, thus enabling the seamless integration of data-centric components into the system in a fully automated manner. Generation of source code from the CHROMOSOME model results in the following items for each CHROMOSOME node:

- Code for configured components in form of component/function wrappers (see [17] for details)
- Configured CHROMOSOME runtime code (“main.c”):
  - Configuration of schedule
  - Configuration of the data-centric communication (including data flow components for marshaling/demarshalling and communication over network/bus)
  - Glue code to start and initialize all configured components
- Build definition files to enable project compilation and documentation generation from source code (Make / CMake / other build tool).