AMALTHEA 4public

(ITEA 2 – 13017)

Enabling of Results from AMALTHEA and others
for Transfer into Application and
building Community around

Deliverable: D 4.3

Mechanisms to support certification of multi-core development

Work Package: 4
Safety

Task: 4.3
Evaluation and integration development of mechanisms to support certification of multi-core development

Document Type:	Deliverable	**Classification:**	Public
Document Version:	Final	**Contract Start Date:**	01.09.2014
Document Preparation Date:	June 21, 2017	**Duration:**	31.08.2017

ITEA 2
INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

EUREKA

History

Rev.	**Content**	**Resp. Partner**	**Date**
V 0.1	First version of document	TWT	22^{th} April 2016
V 0.2	Content completely available	TWT & OFFIS	9^{th} June 2017
V 1.0	First submitted version	OFFIS & TWT	21^{st} June 2017

Contents

List of Figures

List of Tables

Summary

This document comprises and discusses the results of Tasks 4.3 and 4.4 of the project AMALTHEA4Public and constitutes the main outcome and result of WP 4 “Safety”.

In this deliverable, the results are additionally described along the lines of two examples in order to show the practical relevance of the work conducted within the project. The focus within this deliverable is to describe the methods and tools used in order to enable APP4MC to be compliant with standards of functional safety as, e. g., ISO26262. Compliance with such standards can be interpreted in many different ways and the considerations within the project follow the considerations described in the following lines and whose technical details are reported in this deliverable.

For the work done in AMALTHEA4public and reported in the deliverables, *compliant with standards of functional safety* refers to the ability of APP4MC and the AMALTHEA modelling language to

1. define development artefacts that are relevant for the development of, e. g., safety mechanisms in harware or software,
2. include safety-related information for development artefacts,
3. establish linking and traceability of safety-related development artefacts.

D4.1 [2] reports the gap analysis between AMALTHEA and ISO26262, i. e., *the* common standard for functional safety in the automotive domain. D4.2 [3] reports on methods and tools used within the safety-lifecycle of ISO26262 in industrial practice and makes proposals for the integtration of these methods and tools in AMALTHEA4public, i. e., in AMALTHEA’s modelling language and in APP4MC. Based on the considerations of D4.1 [2] and D4.2 [3], Tasks 4.3 and 4.4 are dedicated to define the concrete integration of methods and tools in order to enable functional-safety compliant development with AMALTHEA’s modelling language and with APP4MC. Moreover, the suitability of the approach has been validated by considering two use cases with their specific needs for safety-related development.

In this deliverable we describe how to use AMALTHEA’s modelling language, APP4MC and the tool Capra in order to enable functional-safety standard compliant development with APP4MC. We describe how modelling artefacts in APP4MC shall be decorated with, say, safety-related meta data. Moreover, we describe how modelling and development artefacts shall be linked togehter in order to establish traceability of safety-related information: this is a mandatory requirements of all safety-related developments and demanded by all standards on functional safety. The discussions in this deliverable are given along the lines of the safety-lifecycle of ISO26262, i. e., we consider the *item definition*, the *Hazard Analysis and Risk Assessment*, the *Functional Safety Concept*, the *System Design*, and the *general management of safety requirements.*

1. Overview

Based on the resuts reported in D4.1 [2] and D4.2 [3], we describe how the different tasks of the safety lifecycle of ISO26262 shall be handled within a development using AMALTHEA. As already stated in D4.2, we do only consider the branch of the V-model, i. e., specification and implementation.

For each phase of the safety lifecycle, i. e., *item definition*, *Hazard Analysis and Risk Assessment*, *Functional Safety Concept*, *System Design*, *hardware design* and *software design* we describe how these steps shall be handled during the development of a system with AMALTHEA, i. e., with APP4MC. For this, we describe firstly how elemtents of AMALTHEA's modelling notation shall be decorated with safety-related information, see Chapter 2.

The subsequent chapters describe the approaches for the different phases of the safety lifecycle and are all organised similarly. The structure of these chapters is as follows. First, we describe the approach to be followed in order to be, say, safety-standard compliant. Second, the approach is illustrated in one or two examples. Third, each chapter closes with a short conclusion to summarise the main points of the approach in a few words.

The examples that are used in the following discussion are:

1. An ACC system that is in many modern cars as one of the advanced driver assistance systems. The focus of this example is to show how AMALTHEA would be used in a serial development within the automotive industry.

2. A Quadcopter. The focus of this example is set on the development of a safety concept that relies on multicore systems. Hence, with this example it is shown that the considerations in WP4 are not restricted to developments that are in accordance with ISO26262: our results and approaches can be used within any development of safety-critical systems.

However, not in all chapters both examples are used to illustrate the belonging approach. For example, the *item definition*, see Chapter 3, is illustrated by means of the ACC.

2. Decoration of modelling elements

In this introductory chapter, the method used in order to decorate modelling elements within AMALTHEA models with safety-related information is described.

Development artefacts that are created during the development of the item and its safety mechanisms can roughly be discriminated by the following property: **either** an element is used in order to fulfill a safety goal **or** an element is not used to fulfill a safety goal. For example, the *distance sensor* can be considered as a development artefact that is not used to fulfill a safety goal in an ACC system. The distance sensor is a development artefact that is used for the nominal behaviour of the ACC. On the other hand, the *watchdog*[1] is a developmemt artefact that is used to fulfill a safety goal in the ACC system. More precisely, the watchdog is one of the safety-mechanisms that is realised for the ACC

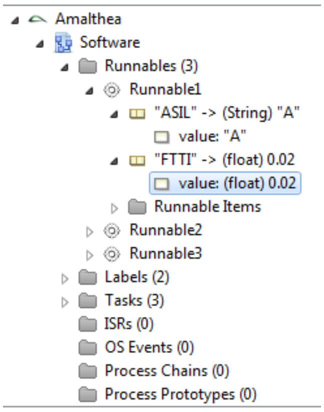
Due to the process prescribed by ISO 26262, each development artefact that is used to fulfill a safety goal will be linked to the safety goals. That kind of linking is either a direct link as, for example, for top-level development artefacts. Or a development artefact is linked to another development artefact that already is deemed to be used in order to fulfill a safety goal. In both cases, the information on, e. g., ASIL, can be traced back to the safety goal itself. However, it makes sense to have a possibility to decorate the development artefacts with, say, *ISO 26262 labels*. There might exist labels for the following information:

- ASIL;
- Diagnostic Test Interval (DTI);
- Fault Detection Time (FDT);
- FTTI;
- Fault Reaction Time (FRT);
- safe state.

ISO 26262 labels can directly be included in the AMALTHEA model by using the *custom property* attribut as, e. g., shown in Figure 2.1, for the ASIL of software-runnables. Hence, the meta model of AMALTHEA already provides a mechanism to include the necessary information.

However, this apporach has advantages as well as disadvantages. The main advantage is that high flexibility in the definition of the used ISO 26262 labels is given and the process that has to be followed by the developers can be tailored by the process authority as needed. For example, the ISO 26262 label *FTTI* might be left out due to process reasons without having empty labels at the development artefacts. As all of the ISO 26262 labels are covered as *custom property* the main disadvantage is the following. Automatic processing of the ISO 26262 labels requires an alanysis of all custom properties although these might be used also for other labels than ISO 26262 labels. Hence, the logic that has to be implemented for automatic processing

[1] No matter whether realised as software- or hardware-component.



```
◢ Amalthea
  ◢ Software
    ◢ Runnables (3)
      ◢ Runnable1
        ◢ "ASIL" -> (String) "A"
            value: "A"
        ◢ "FTTI" -> (float) 0.02
            value: (float) 0.02
        ▷ Runnable Items
      ▷ Runnable2
      ▷ Runnable3
    ▷ Labels (2)
    ▷ Tasks (3)
      ISRs (0)
      OS Events (0)
      Process Chains (0)
      Process Prototypes (0)
```

Figure 2.1.: Custom property for ISO 26262 labels: ASIL and FTTI

needs to be richer in comparison to approaches where the ISO 26262 labels would be given within the meta model.

3. Item Definition

According to ISO 26262 the ([9, 5.1]) “objective is to define and describe the item, its dependencies on, and interaction with the environment and other items.” The *required information* is the necessary input for the item definition and covers:

- the functional concept;
- the operational and environmental constraints;
- legal requirements, national and international standards;
- behaviour achieved by similar functions, items or elements;
- assumptions on behaviour expected from the item; and
- potential consequences of behavioural shortfalls including know failure modes and hazards.

In D4.2 [3] it is highlighted for this *required information* whether tool and method support external to AMALTHEA has to be considered. In a nutshell, the conclusion drawn in D4.2 [3] has been that no external tooling is necessary for the item definition.

However, it definitely makes sense to enable linkage to other modelling formalisms as, e. g., Systems Modelling Language (SysML). The different stakeholder of the development chain usually use different tools and modelling notations for the description of their system under development. Moreover, the definition of the item, or more precise its *description*, covers different levels of detail. In order to be able to describe different levels of detail in a process that, for example, follows the Object-Oriented Systems Engineering Method (OOSEM)[1], different development tools will be used. This is necessary in order to cover the description of the item itself and also to realise the description of the boundary of the item. SysML is one of the modelling languages that is used in Model-Based Systems Engineering (MBSE) in that context. Other involved methods cover, for example, AUTomotive Open System ARchitecture (AUTOSAR)[2] and Electronics Architecture and Software Technology - Architecture Description Language (EAST-ADL)[3].

Hence, in order to enable connections to the AMALTHEA plattform, i. e., APP4MC[4], the methods and languages used in order to describe the item shall not be restricted. The challenge to connect the artifacts, e. g., requirements, block definition diagramms, diagrams describing dynamic behaviour, of different development tools to APP4MC shall thus be met. Links from development artifacts of one tool to another tool can be established by using Capra[5]. Capra is one of the results of Amalthea4public and is used for managing traceability links. Additionally,

[1] http://www.omgwiki.org/MBSE/doku.php?id=mbse:incoseoosem

[2] http://www.autosar.org

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

[4] http://www.eclipse.org/app4mc/

[5] https://projects.eclipse.org/projects/modeling.capra

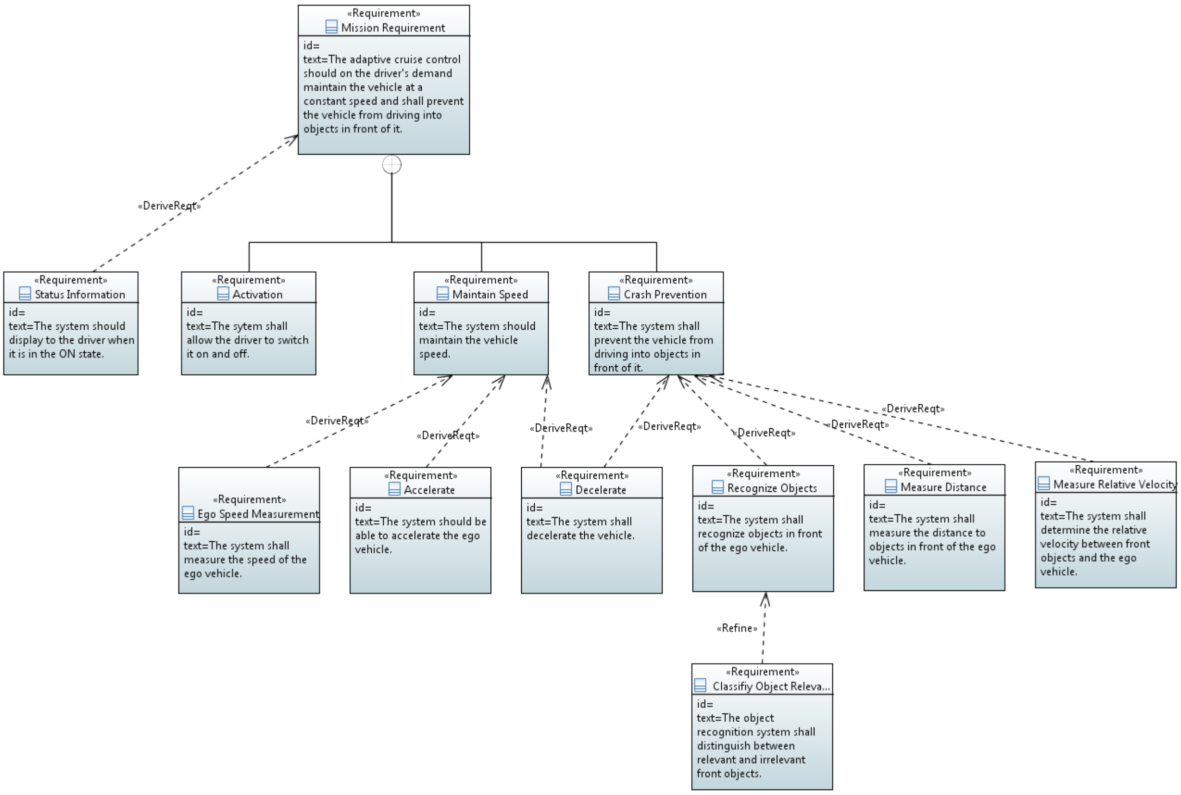

«Requirement»
Mission Requirement
id=
text=The adaptive cruise control should on the driver's demand maintain the vehicle at a constant speed and shall prevent the vehicle from driving into objects in front of it.
«DeriveReqt»
«Requirement»
Status Information
id=
text=The system should display to the driver when it is in the ON state.
«Requirement»
Activation
id=
text=The sytem shall allow the driver to switch it on and off.
«Requirement»
Maintain Speed
id=
text=The system should maintain the vehicle speed.
«Requirement»
Crash Prevention
id=
text=The system shall prevent the vehicle from driving into objects in front of it.
«DeriveReqt»
«DeriveReqt»
«DeriveReqt»
«DeriveReqt»
«DeriveReqt»
«DeriveReqt»
«DeriveReqt»
«Requirement»
Ego Speed Measurement
id=
text=The system shall measure the speed of the ego vehicle.
«Requirement»
Accelerate
id=
text=The system should be able to accelerate the ego vehicle.
«Requirement»
Decelerate
id=
text=The system shall decelerate the vehicle.
«Requirement»
Recognize Objects
id=
text=The system shall recognize objects in front of the ego vehicle.
«Requirement»
Measure Distance
id=
text=The system shall measure the distance to objects in front of the ego vehicle.
«Requirement»
Measure Relative Velocity
id=
text=The system shall determine the relative velocity between front objects and the ego vehicle.
«Refine»
«Requirement»
Classifiy Object Releva...
id=
text=The object recognition system shall distinguish between relevant and irrelevant front objects.

Figure 3.1.: Item definition ACC: requirements diagram

in order to be compliant with safety standards, the linkage of, e. g., safety requirements to elements of the functional concept of the item and the implemented safety mechanisms, and its traceability is of major interest and importance.

Example

In Figure 3.1 the requirements to the ACC are given as a requirements diagram in SysML. This diagram shows the top-level requirement, i. e., the element on top of the requirements, say, tree. This top-level requirement, for example, *contains*[6] requirments on *Activation*, *Maintain Speed* and *Crash Prevention*.

In Figure 3.2 the context of the item ACC is given as BDD. Roughly speaking, the diagram shows the context in which the ACC is, say, embedded. For example, the ACC is part of the *EgoVehicle* and the *Drive Train* is also part of the *EgoVehicle*.

In Figure 3.3 the interface specification of ACC is given as BDD. Here, it is shown which interfaces are needed in order to implement the ACC.

[6] *Containment link* is a connection type defined for and used in SysML.

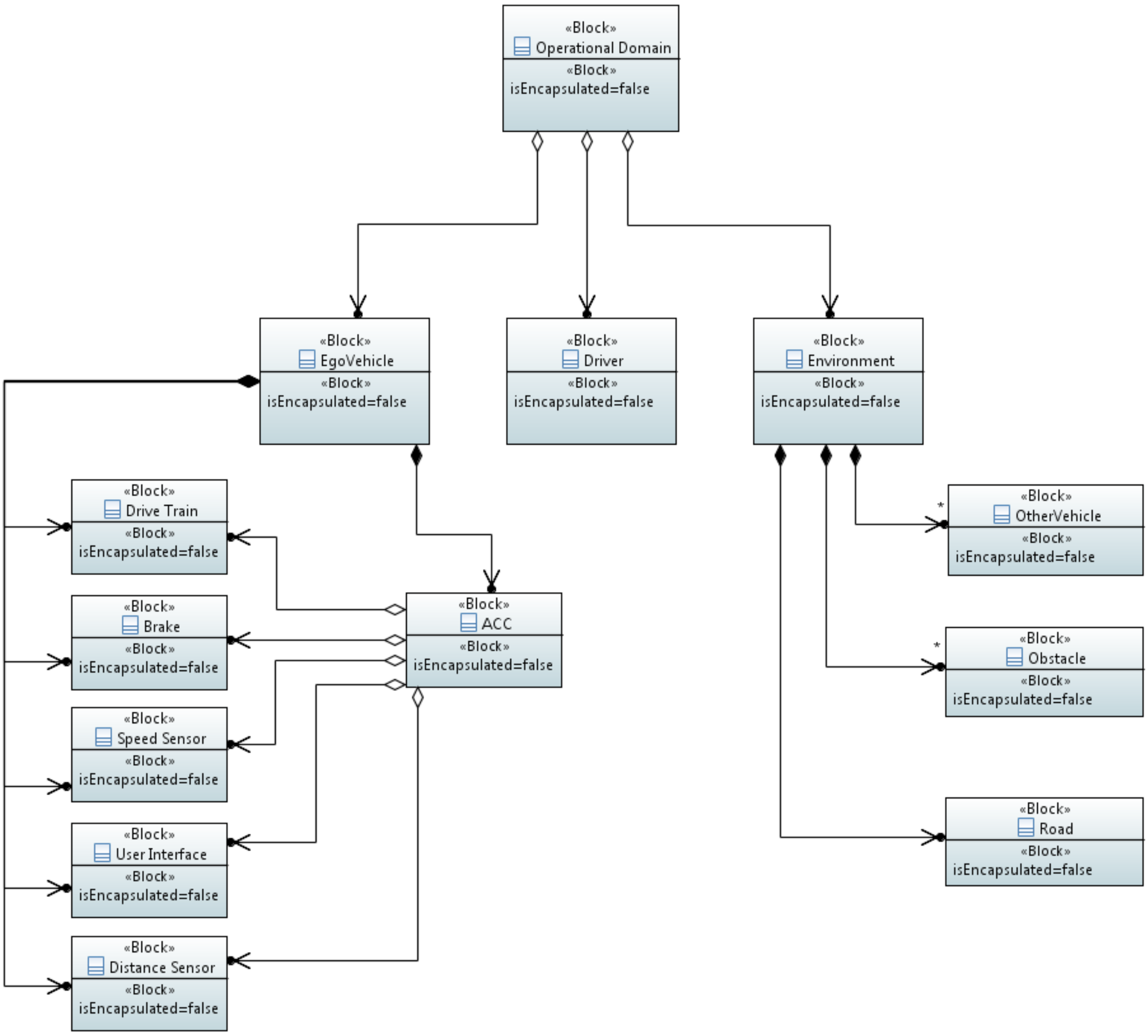


Figure 3.2.: Item definition ACC: BDD context defintion

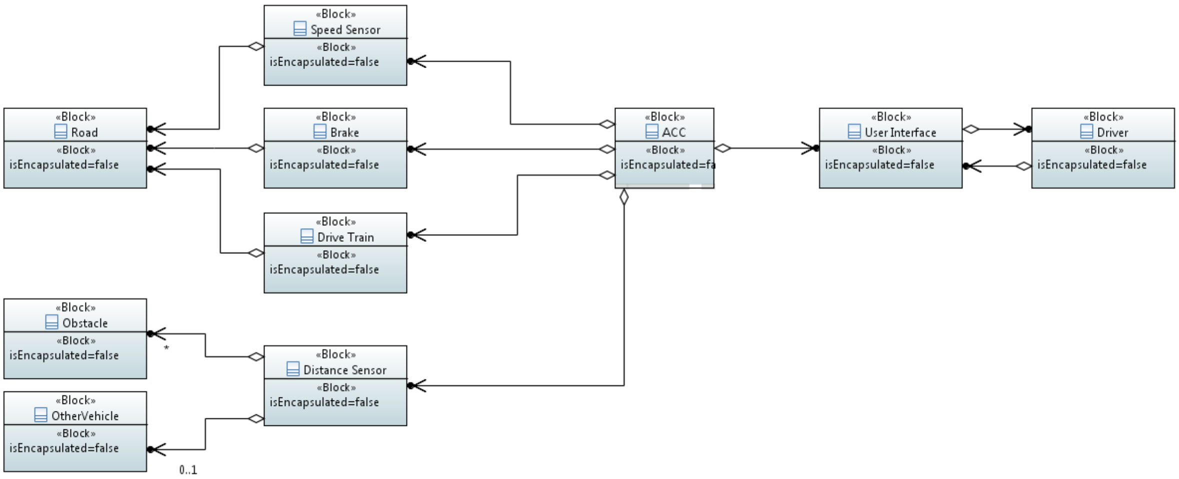


Figure 3.3.: Item definition ACC: BDD interface specification

Conclusion

Dedicated interfaces from tools and methods used for modelling and designing the item into APP4MC are not necessary. Rather, the level-of-detail that will be modelled for the item will cover different abstraction levels. Some of these abstraction levels will be covered by languages as, e. g., SysML, while others will directly be covered in APP4MC. Hence, the previously described possibility to link these development artifacts to each other via Capra is deemed to be sufficient in order to support certification of multi-core development with respect to (w.r.t.) ISO 26262.

4. Hazard Analysis and Risk Assessment

For Hazard Analysis and Risk Assessment (HARA) it is highlighted in D4.2 [3] that no external tooling or methods will be included in the design flow of AMALTHEA. Rahter, the tooling already used in industrial practice for both identifying hazards and documenting hazards shall be linked to the corresponding development artefacts in APP4MC. This kind of linkage can be established by using and tailoring Capra. As soon as the development of safety mechanisms arise, it makes sense to include information on, e. g., ASIL directly in the AMALTHEA modelling elements. This can be done as pointed out in Chapter 2.

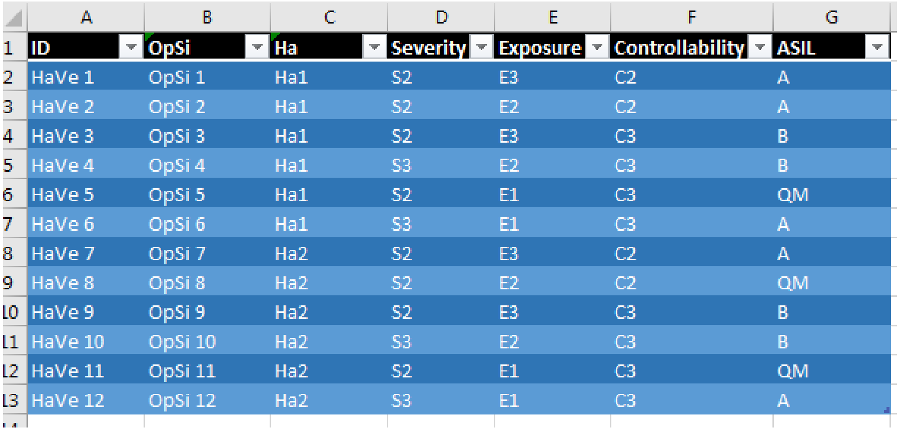
Example

For the ACC system, an exemplified classification of HaEvs and the determination of the ASIL is given in Figure 4.1. Recall, that a according to ISO26262 a HaEv is the combination of an Operational Situation (OpSi) and a *hazard*. The OpSis for this example are given in Figure 4.2 and the hazards are given in Figure 4.3.

The corresponding development artefacts that shall be linked to information on HaEvs are given in the subsequent chapters of this deliverable.

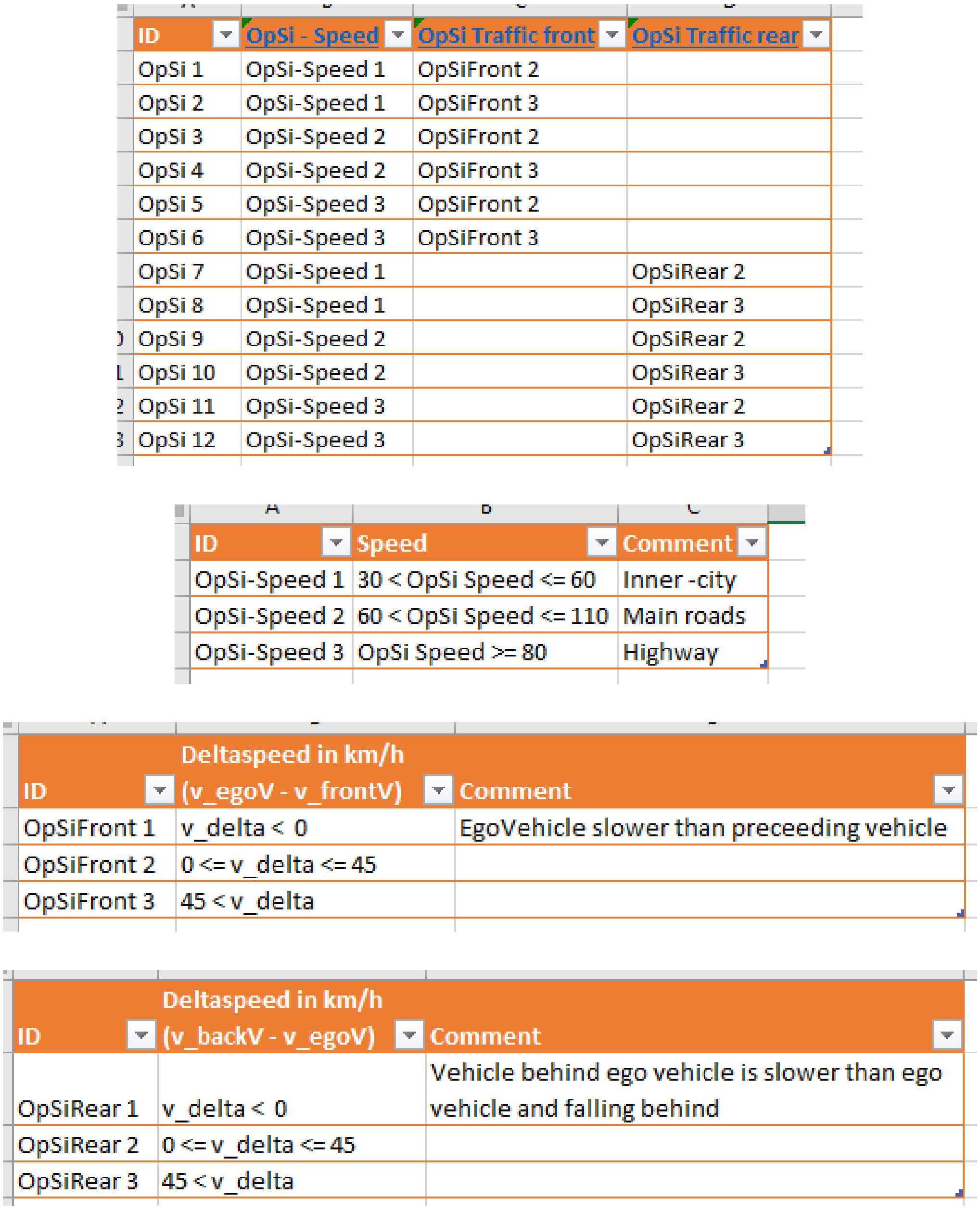
Conclusion

Capra shall be used in order to link information from dedicated tooling and methods that are external to APP4MC to modelling artefacts in APP4MC. With such linking, AMALTHEA will be enabled to support certification of multi-core development.



ID	OpSi	Ha	Severity	Exposure	Controllability	ASIL
HaVe 1	OpSi 1	Ha1	S2	E3	C2	A
HaVe 2	OpSi 2	Ha1	S2	E2	C2	A
HaVe 3	OpSi 3	Ha1	S2	E3	C3	B
HaVe 4	OpSi 4	Ha1	S3	E2	C3	B
HaVe 5	OpSi 5	Ha1	S2	E1	C3	QM
HaVe 6	OpSi 6	Ha1	S3	E1	C3	A
HaVe 7	OpSi 7	Ha2	S2	E3	C2	A
HaVe 8	OpSi 8	Ha2	S2	E2	C2	QM
HaVe 9	OpSi 9	Ha2	S2	E3	C3	B
HaVe 10	OpSi 10	Ha2	S3	E2	C3	B
HaVe 11	OpSi 11	Ha2	S2	E1	C3	QM
HaVe 12	OpSi 12	Ha2	S3	E1	C3	A

Figure 4.1.: HaEv classification and ASIL determination



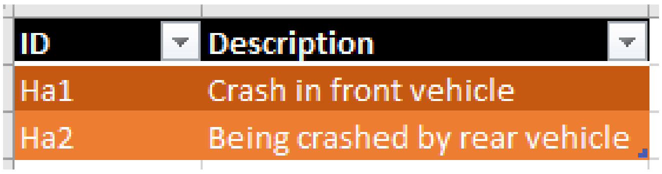
ID	OpSi - Speed	OpSi Traffic front	OpSi Traffic rear
OpSi 1	OpSi-Speed 1	OpSiFront 2	
OpSi 2	OpSi-Speed 1	OpSiFront 3	
OpSi 3	OpSi-Speed 2	OpSiFront 2	
OpSi 4	OpSi-Speed 2	OpSiFront 3	
OpSi 5	OpSi-Speed 3	OpSiFront 2	
OpSi 6	OpSi-Speed 3	OpSiFront 3	
OpSi 7	OpSi-Speed 1		OpSiRear 2
OpSi 8	OpSi-Speed 1		OpSiRear 3
OpSi 9	OpSi-Speed 2		OpSiRear 2
OpSi 10	OpSi-Speed 2		OpSiRear 3
OpSi 11	OpSi-Speed 3		OpSiRear 2
OpSi 12	OpSi-Speed 3		OpSiRear 3

ID	Speed	Comment
OpSi-Speed 1	30 < OpSi Speed <= 60	Inner -city
OpSi-Speed 2	60 < OpSi Speed <= 110	Main roads
OpSi-Speed 3	OpSi Speed >= 80	Highway

ID	Deltaspeed in km/h (v_egoV - v_frontV)	Comment
OpSiFront 1	v_delta < 0	EgoVehicle slower than preceeding vehicle
OpSiFront 2	0 <= v_delta <= 45	
OpSiFront 3	45 < v_delta	

ID	Deltaspeed in km/h (v_backV - v_egoV)	Comment
OpSiRear 1	v_delta < 0	Vehicle behind ego vehicle is slower than ego vehicle and falling behind
OpSiRear 2	0 <= v_delta <= 45	
OpSiRear 3	45 < v_delta	

Figure 4.2.: Operational Situations



ID	Description
Ha1	Crash in front vehicle
Ha2	Being crashed by rear vehicle

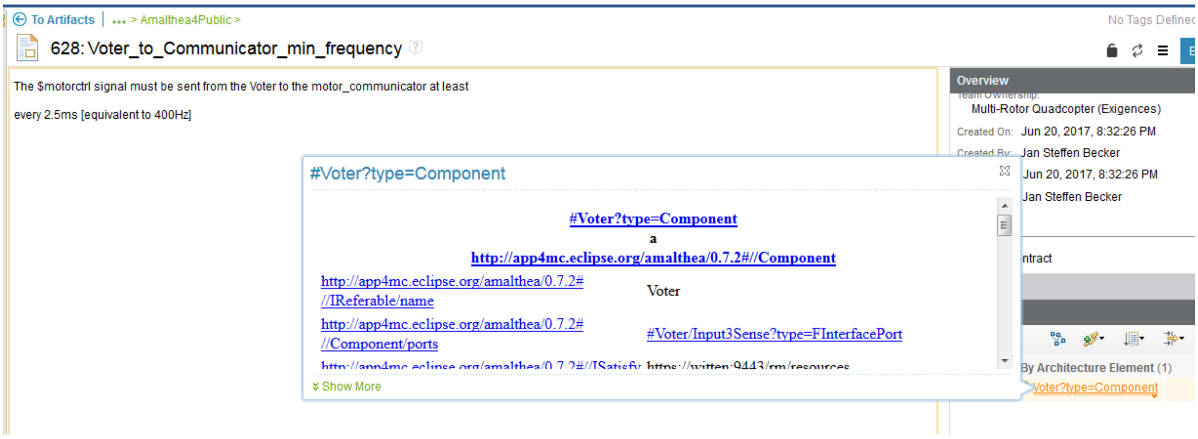
Figure 4.3.: Hazards

5. Functional Safety Concept

The functional safety concept contains the functional safety requirements and the allocation of these requirements to preliminary architectural elements. Hence, two challenges arise: safety requirements need to be documented either in APP4MC or in external tooling; secondly, the safety requirements shall be allocated to preliminary architectural elements of the item. In D4.2 [3] we have already highlighted, that different possibilities exist but that we focus on the following solutions. For the first challenge, i. e., the documentation of functional safety requirements, two different solutions are appropriate depending on the tooling that is used for, say, functional safety requirements management. In case that Rational Doors by IBM (DOORS) is used for functional safety requirements management, a direct interface from DOORS to APP4MC shall be used in order to tackle the second challenge, i. e., the allocation of functional safety requirements to preliminary architectural elements. Here, we implemented an OSLC [5] server that also implements the necessary interfaces to the IBM Jazz platform [7]. Since DOORS is also part of IBM Jazz, we can use OSLC and Jazz technology to link requirements located in DOORS to architecture elements located on the APP4MC OSLC server. Our OSLC server implements simple HTML-based selection and preview interfaces. These are displayed inside the DOORS web frontend and facilitate the selection of architectual elements to be linked and a preview of link targets. The latter is shown in Figure 5.1: One of the requirements from Example 2 below has been linked to the 'Voter' component. The link is listed in the bottom right corner of the screenshot and the preview embedded into the speech bubble in the front.

In case that any other requirements management tooling is used for functional safety requirements, the integration of these requirements into APP4MC shall be realised with Capra. That is, allocation of functional safety requirements to preliminary architectural elements of the item shall be done by using links in Capra.

As already briefly mentioned in Chapter 3, architectural elements of the item might be present at different levels of detail and thus, a cascade of linking has to be used. In this cascade, the



628: Voter_to_Communicator_min_frequency

The $motorctrl signal must be sent from the Voter to the motor_communicator at least

every 2.5ms [equivalent to 400Hz]

Team Ownership: Multi-Rotor Quadcopter (Exigences)

Created On: Jun 20, 2017, 8:32:26 PM

Created By: Jan Steffen Becker

Jun 20, 2017, 8:32:26 PM

Jan Steffen Becker

ntract

By Architecture Element (1)

Voter?type=Component

#Voter?type=Component

#Voter?type=Component
a
http://app4mc.eclipse.org/amalthea/0.7.2#//Component

http://app4mc.eclipse.org/amalthea/0.7.2#//IReferable/name Voter

http://app4mc.eclipse.org/amalthea/0.7.2#//Component/ports #Voter/Input3Sense?type=FInterfacePort

http://app4mc.eclipse.org/amalthea/0.7.2#//ISatisfy https://written:9443/rm/resources

Figure 5.1.: Linking from DOORS to APP4MC

functional safety requirements shall be linked to the top-level of the architecture. In a typical development process, linkage from these top-level architectural elements to lower-level architectural elements has to be enabled. As soon as the level of APP4MC is reached, linking directly works with Capra. For links between tooling used on higher levels of abstraction, we propose to use and possibly extend Capra in order to establish linkage. In summary, this has the benefit that, e. g., SysML and EAST-ADL models, can be linked to functional safety requirements as well as SysML modelling blocks to EAST-ADL modelling blocks. Subsequently, APP4MC modelling elements can be linked via Capra to modelling blocks of SysML or EAST-ADL.

Example1

We demonstrate one way of specifying the Functional Safety Requirement (FSR) for the ACC system, which was partially definded in Chapter 3 as a SysML model. In particular we focus on how the FSR specification and the SysML model can be linked with each other, such that traceability between the different models of our functional safety concepts can be established.

Requirements

We specify requirements for the ACC with the Requirements Management Framework (RMF) for Eclipse while using the ProR tool.[1] RMF is a framework for working with textual requirements while using the Requirements Interchange Format (ReqIF), the industry standard for exchanging requirements.[2] The ProR tool is an Eclipse plugin, providing a GUI that allows working with requirements under the ReqIF standard.

Figure 5.2 shows a snippet of the requirement definition for the ACC. The requirements document shows two safety goals:

1. The ego vehicle must not stay below the safety distance;

2. The ego vehicle must not break unintendedly

Further, Figure 5.2 depicts six functional safety requirements w.r.t. these two safety goals. Note that some requirements only refer to one safety goal, while others can be assiciated with both. In this case *REQ-1* and *REQ-2* refer to safety goal 1, *REQ-19* to safety goal 2, and *REQ-4*, *REQ-5* and *REQ-20* refer to both. Besides, Figure 5.2 also shows some resulting technical and hardware safety requirements.

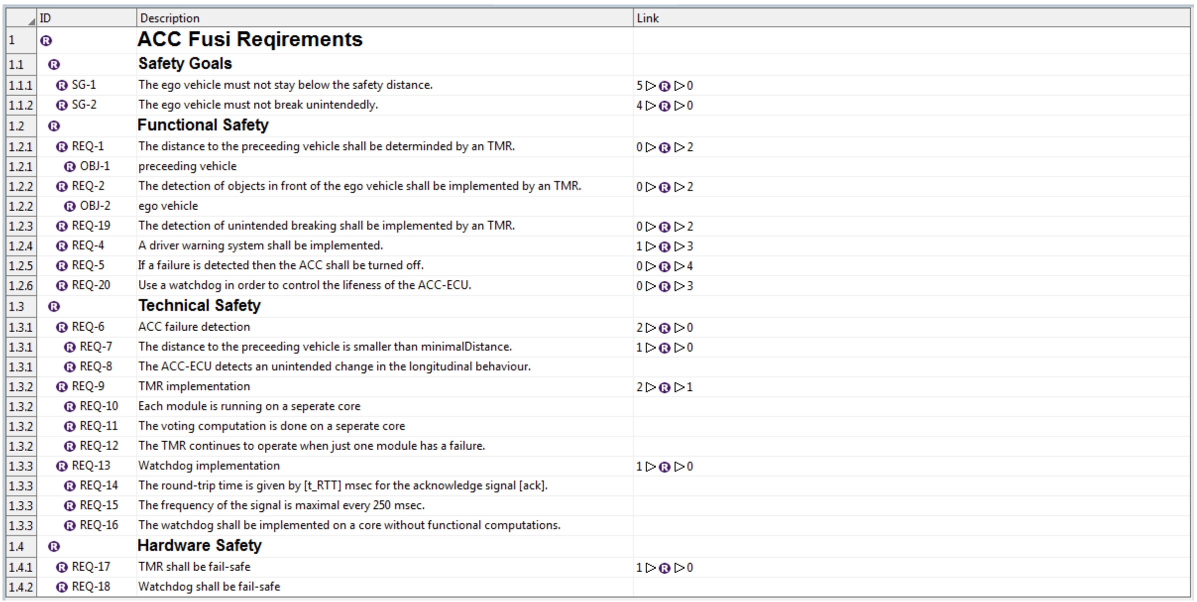
Linkage

To link requirements with each other, we can create associations between them within the ReqIF specification. Figure 5.3 shows how such linking is represented within ProR workspace. For example, consider the requirement *REQ-4*. Besides the description it shows a 1 → 3 linking relation. That means, that *REQ-4* is linked to 3 other specifications within the ReqIF specification, and has 1 dependency, i.e. another specification is linked to *REQ-4*.

While it is possible to link specifications within the ReqIF specification, we also need a way to link amongst different models within our functional safety concept. In this case, we want to link our FSRs with our SysML model. Therefore, we will use Capra to establish traceability

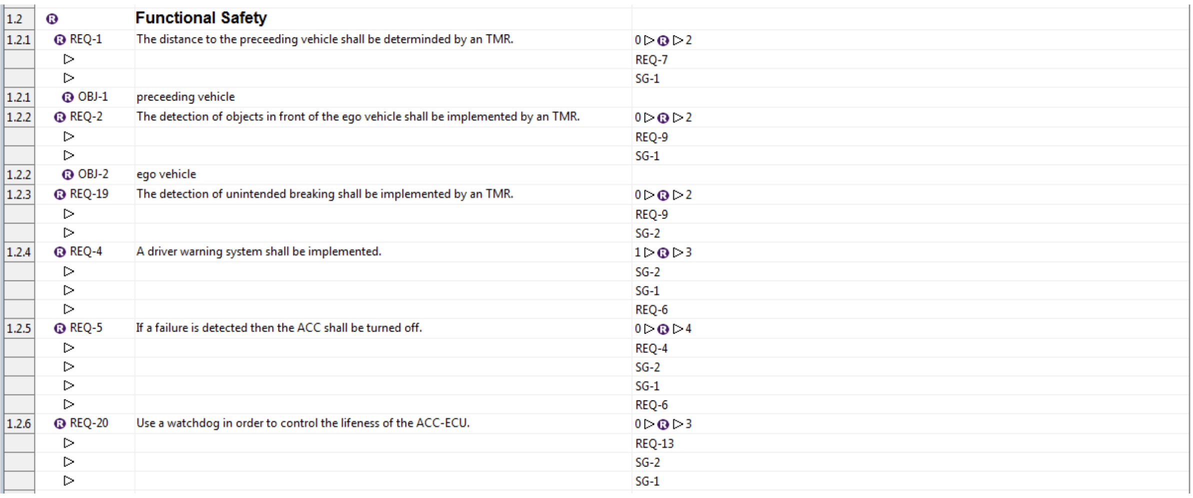
[1] http://www.eclipse.org/rmf/

[2] http://www.omg.org/spec/ReqIF/



	ID	Description	Link
1	Ⓡ	**ACC Fusi Reqirements**	
1.1	Ⓡ	**Safety Goals**	
1.1.1	Ⓡ SG-1	The ego vehicle must not stay below the safety distance.	5▷Ⓡ▷0
1.1.2	Ⓡ SG-2	The ego vehicle must not break unintendedly.	4▷Ⓡ▷0
1.2	Ⓡ	**Functional Safety**	
1.2.1	Ⓡ REQ-1	The distance to the preceeding vehicle shall be determinded by an TMR.	0▷Ⓡ▷2
1.2.1	Ⓡ OBJ-1	preceeding vehicle	
1.2.2	Ⓡ REQ-2	The detection of objects in front of the ego vehicle shall be implemented by an TMR.	0▷Ⓡ▷2
1.2.2	Ⓡ OBJ-2	ego vehicle	
1.2.3	Ⓡ REQ-19	The detection of unintended breaking shall be implemented by an TMR.	0▷Ⓡ▷2
1.2.4	Ⓡ REQ-4	A driver warning system shall be implemented.	1▷Ⓡ▷3
1.2.5	Ⓡ REQ-5	If a failure is detected then the ACC shall be turned off.	0▷Ⓡ▷4
1.2.6	Ⓡ REQ-20	Use a watchdog in order to control the lifeness of the ACC-ECU.	0▷Ⓡ▷3
1.3	Ⓡ	**Technical Safety**	
1.3.1	Ⓡ REQ-6	ACC failure detection	2▷Ⓡ▷0
1.3.1	Ⓡ REQ-7	The distance to the preceeding vehicle is smaller than minimalDistance.	1▷Ⓡ▷0
1.3.1	Ⓡ REQ-8	The ACC-ECU detects an unintended change in the longitudinal behaviour.	
1.3.2	Ⓡ REQ-9	TMR implementation	2▷Ⓡ▷1
1.3.2	Ⓡ REQ-10	Each module is running on a seperate core	
1.3.2	Ⓡ REQ-11	The voting computation is done on a seperate core	
1.3.2	Ⓡ REQ-12	The TMR continues to operate when just one module has a failure.	
1.3.3	Ⓡ REQ-13	Watchdog implementation	1▷Ⓡ▷0
1.3.3	Ⓡ REQ-14	The round-trip time is given by [t_RTT] msec for the acknowledge signal [ack].	
1.3.3	Ⓡ REQ-15	The frequency of the signal is maximal every 250 msec.	
1.3.3	Ⓡ REQ-16	The watchdog shall be implemented on a core without functional computations.	
1.4	Ⓡ	**Hardware Safety**	
1.4.1	Ⓡ REQ-17	TMR shall be fail-safe	1▷Ⓡ▷0
1.4.2	Ⓡ REQ-18	Watchdog shall be fail-safe	

Figure 5.2.: Snippet of the ACC requirements document.



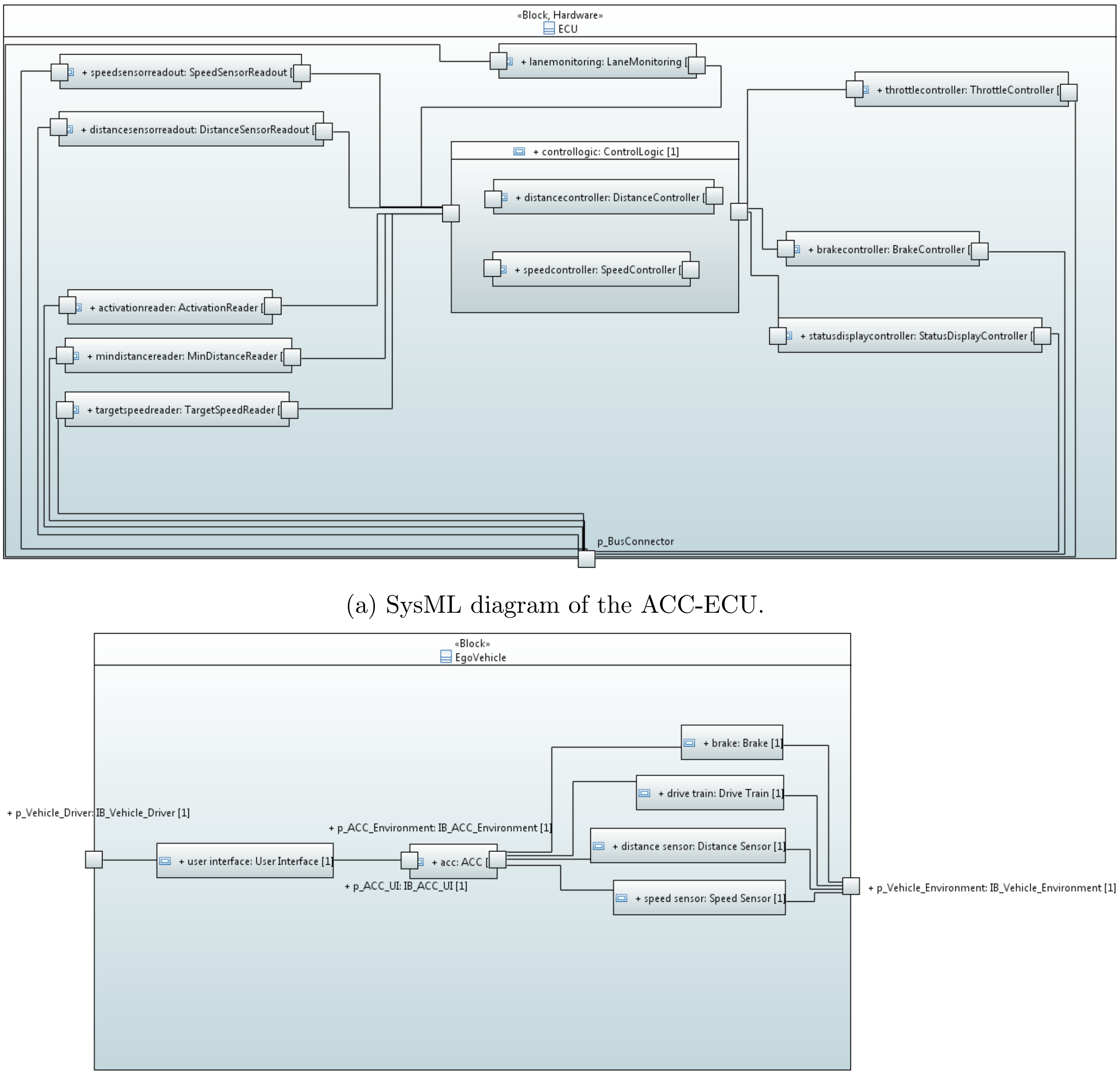
1.2	Ⓡ	**Functional Safety**	
1.2.1	Ⓡ REQ-1	The distance to the preceeding vehicle shall be determinded by an TMR.	0▷Ⓡ▷2
	▷		REQ-7
	▷		SG-1
1.2.1	Ⓡ OBJ-1	preceeding vehicle	
1.2.2	Ⓡ REQ-2	The detection of objects in front of the ego vehicle shall be implemented by an TMR.	0▷Ⓡ▷2
	▷		REQ-9
	▷		SG-1
1.2.2	Ⓡ OBJ-2	ego vehicle	
1.2.3	Ⓡ REQ-19	The detection of unintended breaking shall be implemented by an TMR.	0▷Ⓡ▷2
	▷		REQ-9
	▷		SG-2
1.2.4	Ⓡ REQ-4	A driver warning system shall be implemented.	1▷Ⓡ▷3
	▷		SG-2
	▷		SG-1
	▷		REQ-6
1.2.5	Ⓡ REQ-5	If a failure is detected then the ACC shall be turned off.	0▷Ⓡ▷4
	▷		REQ-4
	▷		SG-2
	▷		SG-1
	▷		REQ-6
1.2.6	Ⓡ REQ-20	Use a watchdog in order to control the lifeness of the ACC-ECU.	0▷Ⓡ▷3
	▷		REQ-13
	▷		SG-2
	▷		SG-1

Figure 5.3.: Snippet of the ACC functional safety requirements with linking.

links between our requirements specified in ProR and our SysML specification modelled with Papyrus.[3]

Linking between models. Figure 5.5 depicts some relations between our FSRs as depicted in Figure 5.3 and the SysML diagrams of the ACC-ECU depicted in Figure 5.4a, and the *ego vehicle* depicted in Figure 5.4b. Now lets consider the Capra linking depicted in Figure 5.5c. Here the linking shows that *REQ-1* is related to the *distancecontroller* and *speedcontroller* of the ACC-ECU as well as the *distance sensor* and *speed sensor* of the *ego vehicle*. This relation helps us to navigate between the FSR and the SysML model. Hence, by clicking on

[3] https://eclipse.org/papyrus/


«Block, Hardware»
ECU
+ speedsensorreadout: SpeedSensorReadout [
+ lanemonitoring: LaneMonitoring [
+ throttlecontroller: ThrottleController [
+ distancesensorreadout: DistanceSensorReadout [
+ controllogic: ControlLogic [1]
+ distancecontroller: DistanceController [
+ brakecontroller: BrakeController [
+ speedcontroller: SpeedController [
+ activationreader: ActivationReader [
+ statusdisplaycontroller: StatusDisplayController [
+ mindistancereader: MinDistanceReader [
+ targetspeedreader: TargetSpeedReader [
p_BusConnector
«Block»
EgoVehicle
+ brake: Brake [1]
+ drive train: Drive Train [1]
+ p_Vehicle_Driver: IB_Vehicle_Driver [1]
+ p_ACC_Environment: IB_ACC_Environment [1]
+ distance sensor: Distance Sensor [1]
+ user interface: User Interface [1]
+ acc: ACC [
+ p_Vehicle_Environment: IB_Vehicle_Environment [1]
+ p_ACC_UI: IB_ACC_UI [1]
+ speed sensor: Speed Sensor [1]

(a) SysML diagram of the ACC-ECU.

(b) SysML diagram of the ego vehicle.

Figure 5.4.: SysML diagrams of the ACC example.

the requirement *REQ-1* in ProR we will obtain the relation graph from Figure 5.5c. Hence, we can directly associate the requirement with the corresponding elements in the SysML model, and vice versa.

However, these traceability links between the models are not generated automatically, but rather manually by the user. That means, while writing the FSRs in ProR or modelling the SysML diagram in Papyrus, the user has to establish the relation by adding elements to the Capra selection. Then, Capra is able to establish relations between the elements and e.g. provide graphs like in Figure 5.5. Thus, the user is responsible to to relate elements between the different safety concepts, while Capra provides the functionality to do so.

Importance of traceability. As described prior and briefly mentioned in Chapter 3, architectural elements of an item might be present at different levels of detail. In our example, the ACC, we have represented the top-level architecture with SysML, while the FSRs are specified at a different level under the ReqIF standard. Without traceability between these two levels, it would be much harder to relate architectural elements with their role in the functional safety of the system.

For example, consider the *statusdisplaycontroller* in Figure 5.4a. By inspecting its relation as depicted in Figure 5.5d one can observe that it is connected to the FSRs *REQ-4* and *REQ-5*. Hence, we have a direct relation of the `statusdisplaycontroller` and FSRs. Thus, if we would change the architecture and make changes to the `statusdisplaycontroller`, we also have to reevaluate the related FSRs. Without this traceability, the relation could easily be overseen, and thus we could end up with an incompatible safety concept, since the FSRs and the SysML model may not be compatible anymore.

Furthermore, since the user is responsible for initiating the relations between the different models, i.e. relate FSRs and SysML elements, gaps in the architecture are more easily spotted. For example, consider the FSRs *REQ-1*, *REQ-2* and *REQ-19*. All refer to the implementation of a Triple Modular Redundancy (TMR). However, when trying to link these FSRs to the architecture, the user will observe that the current SysML specification does not describe a TMR. Thus, by executing the step of establishing traceability links, we have discovered a gap in our architectural design that can now be fixed.

By observing the previous two examples, we can conclude that traceability between models is a crucial property for establishing a coherent safety concept over different level of detail. On the one hand, by having links between different levels it is possible to easily keep track between the relations of e.g. the architectural model and the FSR. Therefore, when e.g. a change is performed on either, the relation will tell the user what other parts of the safety concept have to be considered. On the other hand, the workflow of creating the relations between the different models already provides a first level of validation. Hence, the process of linking between e.g. the architectural model and FSRs can reveal gaps.

Example2

We show our safety concept using a quadcopter as a demonstrator for general multi-core platforms. The original internal hardware architecture can be found in Figure A.1 in the Appendix.

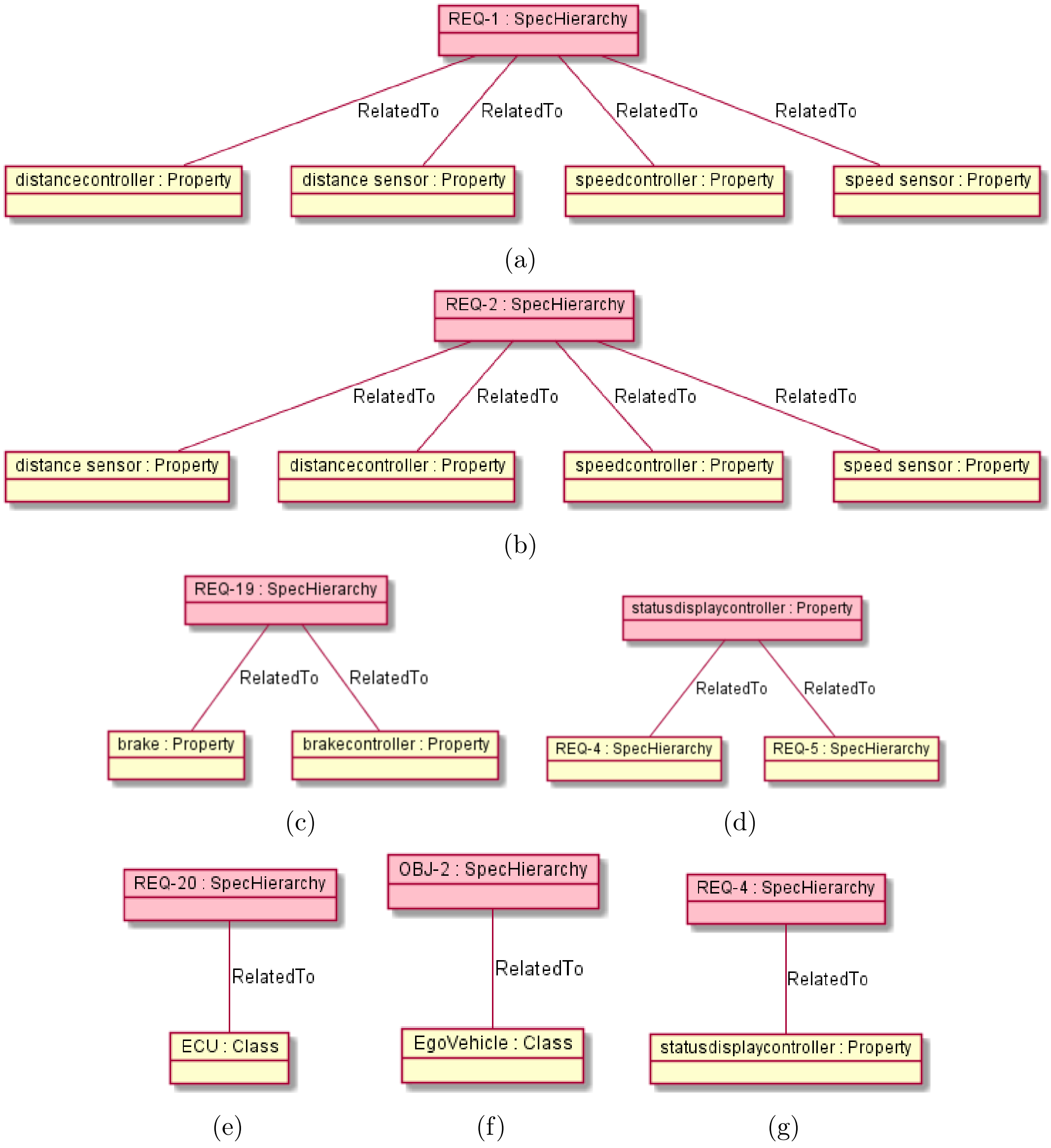
Hardware Architecture

For the sake of simplicity, we have simplified the hardware architecture of the quadcopter subject to the following

1. we abstract away from the sensors and assume that all sensor values are always present and up-to-date, thus we abstract from potential sensor failures, from sensor value processing times and (polling) delays

2. in the same way, we abstract away from the remote control and assume that remote control information is always present and up-to-date

3. we abstract from the internal (computation) behaviour of components



(a)

(b)

(c)

(d)

(e)

(f)

(g)

Figure 5.5.: Capra links between the requirments document and the SysML model.

4. we disregard the battery and battery status information, thus we prove our safety concept under the assumption that the battery power is always sufficient

5. we abstract away from all `general_information` signals

The simplified internal (sub)architecture of the quadcopter is shown in Figure 5.6.

The components shown are found on one FPGA chip (as suggested by the dashed frame in Figure A.1). In our simplified (sub)architecture, the system consists of three cores (a `MicroBlaze` SoftCore, a `LEON` SoftCore and an `ARM` Core), a `Voter` and a `motor_communicator`. The latter serves as interface to the physical hardware controlling the quadcopter. Each core has its own set of sensors (not shown/considered here) providing the current position, magnetic field and pressure, as well as a receiver to receive the current user input (from the remote control).

Each of the two soft cores periodically and independently computes the control signal for the `motor_communicator`, i.e. the control signal used to control and operate the motors of the quadcopter. For this, the soft cores use the sensor's and remote control data as input. The `ARM` Core is responsible for other (non-safety critical) tasks, like for example processing camera data. The two control signals of the soft cores are sent to the `Voter`, whose task it is to compare the two signals. If the signals are identical (up to some threshold), the `Voter` forwards the control signal to the `motor_communicator`, which in turn transforms it into a control signal to be sent to the motors. If, however, the control signals of the two cores are not identical, the `Voter` sends an interrupt signal (`$ARMitrpt`) to the `ARM` Core. As soon as the `ARM` Core receives the interrupt, it has to stop its current task and start computing a third control signal. This is sent back to the `Voter`, who compares it to the other two control signals. It forwards the correct control signal to the `motor_communicator` based on a two-out-of-three decision.[4]

The interrupt handling and computation of the control signal done by the `ARM` Core needs to be fast enough to guarantee that the frequency of control signals from the `motor_communicator` to the motor hardware does not drop below 400Hz (i.e., one signal every 2.5ms). This is also the top level verification requirement.

Safety Requirements

Initially, for the original hardware architecture (cf. Figure A.1), a set of about 100 requirements covering all aspects (functional/technical/hardware/software, safety/functionality/comfort, ...) has been defined for the quadcopter. After first removing those constraints that are not needed in the simplified architecture anymore, among the remaining requirements we have identified those that are necessary to prove the top-level safety requirement, which we state formally as

> (If one of the soft cores fails, the switch over to the ARM core must be fast enough such that)
> The frequency of `motor_control_signals` sent to the `motor_communicator` must not drop below 400Hz
> (to guarantee safe operation of the quadcopter)

The result is a list of seven requirements, which can be found in Appendix A.[5] We formalised the requirements using the Simplified Universal Pattern (SUP), cf. [1, 4]. The formalisation can be found in Table A.1 in Appendix A.

In short, SUPs relate triggers/preconditions with actions/postconditions, both expressed as boolean expressions over the observables of the system. The semantics of SUPs is based on (simulation) runs of the system, SUPs can thus be viewed as observers. An SUP accepts a run if both trigger and action are traversed successfully, and rejects a simulation run (the simulation run is called violating) if the trigger is traversed successfully but the action is not. See [1] for a detailed introduction to SUPs.

[4] We do not consider the case where all three control signals are different, since that would require to consider a more complex (sub)architecture than the one shown in 5.6. The complexity of such an example would be beyond of the scope intended for the example here.

[5] Initially, the requirements were parametrised in the timing constants. Our verification approach is to check that a given set of timing constants—namely those chosen in Appendix A—satisfies the top-level requirement. Another verification approach would be to automatically derive the timing constants needed to satisfy the top-level safety requirement, but this is out of the scope of this example.

To provide some insight into the functionality of the SUP, we explain in detail the formalisation of the 3. requirement: "As soon as the `Voter` has received the signals `$MBctrl` and `$LEONctrl` and they are identical, it sends the `$motorctrl` signal to the `motor_communicator` in less than 2ms". The corresponding SUP fragments (including those that were left out for readability in A.1) are explained below.

Activation Mode: Cyclic Since we want the requirement to hold repeatedly, we use the activation mode Cyclic. This allows (re)activation of the SUP after a previous observation cycle with an inconclusive or passed result, i.e., as long as it is not violated

Interpretation Type: Progress The interpretation type Progress is used to express liveness: the action shall follow the trigger within a certain time interval (or at a certain point in time)

Trigger Start Event Identical to Trigger Condition since the Trigger Condition is characterised as a single point in time (see below)

Trigger Condition: $(\texttt{\$MBctrl} \neq 0) \wedge (\texttt{\$LEONctrl} \neq 0) \wedge (\texttt{\$MBctrl} == \texttt{\$LEONctrl})$ The trigger of this SUP is the point in time where the `Voter` receives both the control signals from the `MicroBlaze` SoftCore $(\texttt{\$MBctrl} \neq 0)$[6] and the `LEON` SoftCore $(\texttt{\$LEONctrl} \neq 0)$, and where both signals are identical $(\texttt{\$MBctrl} == \texttt{\$LEONctrl})$

Trigger End Event Identical to Trigger Condition since the Trigger Condition is characterised as a single point in time (see above)

Trigger Duration Interval: $[0, 0]$ The Trigger Duration Interval has length zero since the Trigger Condition is characterised as a single point in time (see above)

Trigger Exit Condition: `false` The Trigger Exit Condition guards the time interval between the system start (or the end of a previous SUP observation for subsequent occurrences) and the occurrence of the Trigger End Event. If it evaluates to `true` during this period, the observation cycle is terminated with an inconclusive result. But since the Trigger Condition is characterised as a single point in time (see above), we don't need such an "abort criterion"

Local Scope: $[0, 2]$ Time between the end of the trigger (Trigger End Event) and the beginning of the action (Action Start Event), this value is directly derived from the textual description above

Action Start Event Identical to Action Condition since the Action Condition is characterised as a single point in time (see below)

Action Condition: $(\texttt{\$motorctrl} \neq 0)$ The action consists of the single point in time where the `$motorctrl` signal is sent to the `motor_communicator`

Action End Event Identical to Action Condition since the Action Condition is characterised as a single point in time (see above)

[6] We use real-valued observables for the control signals, where a value $== 0$ indicates that the signal is absent, and a value $\neq 0$ indicates that the signal is present.

Action Duration Interval: $[0, 0]$ The Action Duration Interval has length zero since the Action Condition is characterised as a single point in time (see above)

Action Exit Condition: `false` The Action Exit Condition guards the time interval between the end of the trigger (occurrence of Trigger End Event) and the beginning of the action (occurrence of Action Start Event). The explanation is analogue to the Trigger Exit Condition above

Global Scope: ∞ Time frame for the observation of all occurences of the SUP. Since this interval is not bound (the requirement should hold "forever"), it is set to infinity

Verification

We use a Consistency Analysis Tool that is based on the work presented in [6] to verify that the set of requirements is consistent, i.e. that all requirements hold. Since this includes the top-level requirement, we were thus able to infer that the top-level requirement is met, i.e., that the frequency of `motor_control_signals` arriving at the `motor_communicator` does indeed not drop below 400Hz.

The Consistency Analysis Tool allows to specify requirements (amonst others) in the SUP format. A consistency check can then be performed on a chosen set of requirements. For the set of seven requirements,[7] consistency could be checked in a matter of seconds. A modification of the top-level requirement with a frequency of 1000Hz showed that indeed this frequency cannot be guaranteed.

Conclusion

The tooling used in industrial practice for safety-requirements management differs highly from company to company. Hence, when direct linking of safety requirments to architectural elements as for, e. g., DOORS linkage to SysML, is not possibly, Capra shall be used in order to establish allocation from safety requirements to preliminary architectural elements of the item. With either possibilities, AMALTHEA will be enabled to support certification of multi-core development.

[7] Since there is no explicite typing in the tool, we actually had to add some more requirements to guarantee type consistency; we omit the technical details here.

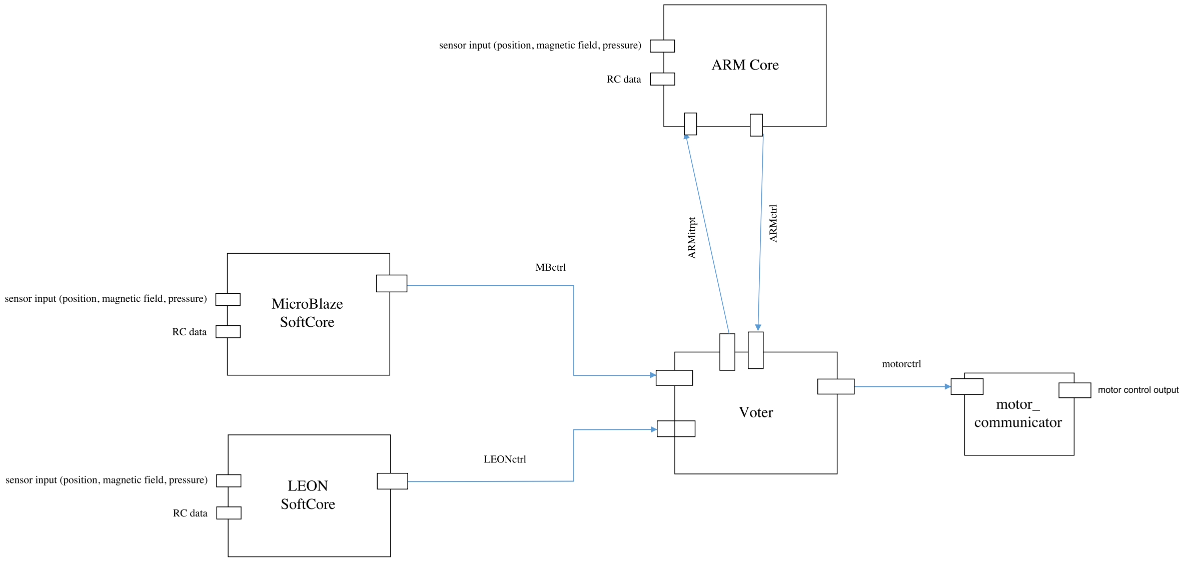

sensor input (position, magnetic field, pressure)
RC data
ARM Core
ARMirpt
ARMctrl
MBctrl
sensor input (position, magnetic field, pressure)
RC data
MicroBlaze SoftCore
motorctrl
Voter
motor_ communicator
motor control output
LEONctrl
sensor input (position, magnetic field, pressure)
RC data
LEON SoftCore

Figure 5.6.: The simplified hardware architecture of the quadcopter

6. System Design

The *product development at the system level*, i. e., part 4 of ISO 26262 [8], covers the definition of Technical Safety Requirements (TSRs) as well as the development of the System Design (SysD) in combination with the development of the Technical Safety Concept (TSC).

Technical Safety Requirements

The specification of TSRs and their documentation has to be done either in APP4MCor in external tooling as, e. g., DOORS or INTEGRITY by PTC (Integrity), or methods like, for example, SysML. As or the Functional Safety Concept (FSC), see Chapter 5, we propose to use toolings or methods external to APP4MCfor the specification and documentation of TSRs. With this, the currently used development workflows at the companies along the development chain only need minor changes in order to use APP4MCfor multi-core development. Additionally, the usage of Capra allows to link and trace requirements and any development artefacts as already highlighted in Chapter 5. Note, that this refines the solution envisaged and described in D4.2 [3]. There, we proposed to extend the meta model of AMALTHEA in order to represent TSRs.

System Design and Technical Safety Concept

One of the great challenges in the development of the SysD is that ([10, Sec. 7.2]) “safety-related and non-safety-related requiements are handled within one development process.” Hence, there needs to be a possibility to discriminate between elements of the SysD that are safety-related and those that are not safety-related. The mechanism to be used is described in Chapter 2: the feature of *custom properties* will be used as soon as the development continues in APP4MC. Elements between requiements tooling and development tools or between different development tools is realised by Capra and appropriate tailoring.

In D4.1 [2] and D4.2 [3] we already highlighted three main points to be considered within AMALTHEA4PUBLIC w.r.t. SysD:

SD1 allocation of TSRs to corresponding system elements;

SD2 ASIL attribution of system elements;

SD3 Hardware-Software Interface (HSI) specification and traceability of hardware resources to software as well as HSIs to TSRs

These main points are covered as follows.

SD1 and SD3 are covered by Capra and its linking and traceability properties. In D4.2 [3] we point out that it shall be possible to check whether each TSR is allocated to at least one system element. This can be realised by user and customer specific extensions of Capra’s implementation.

SD2 The attribution of system elements with ASIL will be realised by using the *custom property* attribute. In D4.2 [3] we point out that it that, for example, it shall be possible to check that ([10, Section 7.4.2.3]) "If an element is comprised of sub-elements with different ASILs assigned, or of non-safety-related sub-elements and safety-related sub-elements, then each of these shall be treated in accordance with the highest ASIL, unless the criteria for coexistence, in accordance with ISO 26262-9:2011, Clause 6, are met." This check can be easily realised by accessing the *custom property* when stringent syntax is used for the attribution with ASIL.

Example

At the time of finishing this deliverable, not all information on the ACC example has been created completely. Hence, we can not show the linkage of elements of the SysD to parts of the, say, SysD that is used for Safety Mechanisms (SaMes). However, linkage with Capra has already been shown in previous sections and linkage and traceability for SysD works in exactly the same way for ACC.

Conclusion

Specification of TSRs will be done with tooling and methods external to APP4MC. The SysD will be developed in different tools and with different methods. However, as soon as multi-core development becomes relevant, APP4MC is used. The linkage between requirements and development artefacts is realised by Capra and appropriate tailoring.

7. General management of safety requirements

In this chapter, the general management of safety requirements as well as the hardware and software design are covered. In particular and as pointed out in D4.2 [3] the hardware and software design is tightly correlated to the general management of safety requirements. From a functional-safety point of view, hardware as well as software design will be done with specialised tooling and the main challenge remains to establish traceability of development artefacts to design elements as, e. g., parts of a VHDL-AMS model or UML elements.

In D4.1 [2] and D4.2 [3] we already highlighted the following main points to be considered within AMALTHEA4PUBLIC w.r.t. the general management of safety requirements:

T1 Traceability from item definition to HARA;

T2 Traceability from Safety Goals (SGs) to FSRs;

T3 Traceability from FSRs to TSRs;

T4 Traceability from TSRs to Software Safety Requirements (SSRs);

T5 Traceability from TSRs to Hardware Safety Requirements (HSRs);

T6 Allocation of HSRs to hardware components implementing the HSRs;

T7 Allocation of SSRs to software components implementing the SSRs;

T8 Traceability from FSRs to elements of the SysD;

T9 Traceability from TSRs to elements of the SysD;

T10 The structure and dependencies of safety-related software shall be traceable.

These main points are covered as follows.

T1 Establishing traceability from the item definition to elements of HARA as, e. g., the determination of ASIL for each of the hazardous events or the SGs, shall be done via Capra, see Chapters 3 to 4. Recall, that HARA is usually done in company specific tooling and not within the meta model of AMALTHEA.

T2 In industrial practice, dedicated tooling for requirements management is used. Within thie tooling it is possible to establish links from one element to another. Hence, traceability from SGs to FSRs shall be established in such tooling.

T3 Requirements linking is realised in specialised requirements management tooling. In order to establish traceability from FSRs to TSRs that tooling shall be used as well.

T4 and T5 As long as SSRs and HSRs are denoted in requirements management tooling, traceability shall be established by using the linkage mechanisms of that tooling. Due to the increasing complexity of the developed items, modern development processes tend to use more model-based methods to denote even requirements on software and hardware. For this, model-based languages as, e. g., SysML, are used in order to denote requirements on software as well as on hardware. Traceability shall be established by using Capra in order to link elements of requirements management tooling to artefacts of tooling used for model-based design.

T6 and T7 In the development process, specialised tooling is used for hardware as well as software components. The linkage from HSRs and SSRs to those components shall be established by extending, if needed, the capabilities of Capra.

T8 We propose to follow a model-based approach in order to neatly establish, say, holistic traceability. However, the tools used in order to create the SysD vary in industrial practice from company to company. Therefore, traceability from FSRs to elements of the SysD shall be established via Capra, see Chapter 5.

T9 This is covered by the same mechanisms as explained for the traceability from FSRs to elements of the SysD.

T10 Here, two different challenges arise. First, elements of the software architecture and software design shall be linked for traceability. This is, say, simply possible by using tooling for software design and according standardised modelling techniques as, e. g., Unified Modelling Language (UML). Second, the architecture and design artefacts have to be linked to implementation artefacts that are given in a particular programming language. As most of today's used Integrated Development Environments (IDEs) do not support this kind of linkage, we propose (again) to use and extent Capra in order to establish that kind of linkage.

Example

For the example on ACC we have to state the same as in Chapter 6: not all parts are already finished and, hence, we cannot show details of the example here that show how we link, allocate and trace information as described in T1 to T10. However, linkage, allocation and traceability will be realised with Capra in exactly the same way as, e. g., dicsussed in Chapter 5.

Conclusion

Requirements management in general and management of (functional) safety requirements in special, is done in industrial practice with dedicated tooling as, for instance, DOORS or even Excel by Microsoft (Excel). However, the artefacts on requirements in this kind of tooling needs to be linked to development artefacts as, e. g., SysD or parts of the item definition. The conclusion for the general management of safety requirements is to use dedicated requirements management tooling in combination with traceability tooling like Capra. For the development of multicore systems with AMALTHEA, the usage of Capra is strongly recommended.

8. Conclusion

In this document we have summarised the main results of WP 4 of AMALTHEA4public. We have described how the different steps of the safety lifecycle[1] shall be implemented when multi-core systems are developed with AMALTHEA. We addressed, how modelling artefacts within AMALTHEA shall be decorated with safety-relevant information as, e. g., the ASIL or the WCRT. Moreover, we have discussed how information ranging from SGs, FSRs, TSRs to SaMes and architectural elements of the item shall be linked togehter in order to establish the demanded traceability of information. As we have also highlighted in D4.1 [2] and D4.2 [3] the focus is set on the left-side of the V-Model, i. e., the specification and implementation tasks.

For these tasks we have discussed how they shall be implemented within AMALTHEA. Additionally, we have shown the practical usage of our results in two examples: an ACC system and a quadcopter. The information on these examples given in this deliverable is just a small excerpt of the complete development artefacts that exist at the respective partners, i. e., the quadcopter at OFFIS and the ACC at TWT.

[1] As given in ISO 26262 and as already described and used in D4.1 [2] and D4.2 [3].

Acronyms and Tools

ACC	Adaptive Cruise Control
ASIL	Automotive Safety Integrity Level
AUTOSAR	AUTomotive Open System ARchitecture
BDD	Block Definition Diagram
DTI	Diagnostic Test Interval
DOORS	Rational Doors by IBM
EAST-ADL	Electronics Architecture and Software Technology - Architecture Description Language
Excel	Excel by Microsoft
ECU	Electronic Control Unit (dt: Steuergerät)
FDT	Fault Detection Time
FRT	Fault Reaction Time
FSC	Functional Safety Concept
FSR	Functional Safety Requirement
FTTI	Fault Tolerant Time Interval
HARA	Hazard Analysis and Risk Assessment
HSI	Hardware-Software Interface
HSR	Hardware Safety Requirement
HaEv	Hazardous Event
IDE	Integrated Development Environment
Integrity	Integrity by PTC
MBSE	Model-Based Systems Engineering
OpSi	Operational Situation
OOSEM	Object-Oriented Systems Engineering Method
ReqIF	Requirements Interchange Format

RMF	Requirements Management Framework
SaMe	Safety Mechanism
SG	Safety Goal
SSR	Software Safety Requirement
SysD	System Design
SysML	Systems Modelling Language
TSC	Technical Safety Concept
TSR	Technical Safety Requirement
TMR	Triple Modular Redundancy
UML	Unified Modelling Language
VHDL-AMS	Very high-speed integrated circuit Hardware Description Language with Analog and Mixed-Signal extensions
WCRT	Worst-Case Response Time
w.r.t.	with respect to

A. Safety Requirements for the Quadcopter

Prerequisites

For the formalisation of the safety requirements, we assume the following observables (cf. Figure 5.6 on Page 19):

`$MBctrl`: the control signal sent from the `MicroBlaze` SoftCore to the `Voter`

`$LEONctrl`: the control signal sent from the `LEON` SoftCore to the `Voter`

`$motorctrl`: the control signal sent from the `Voter` to the `motor_communicator`

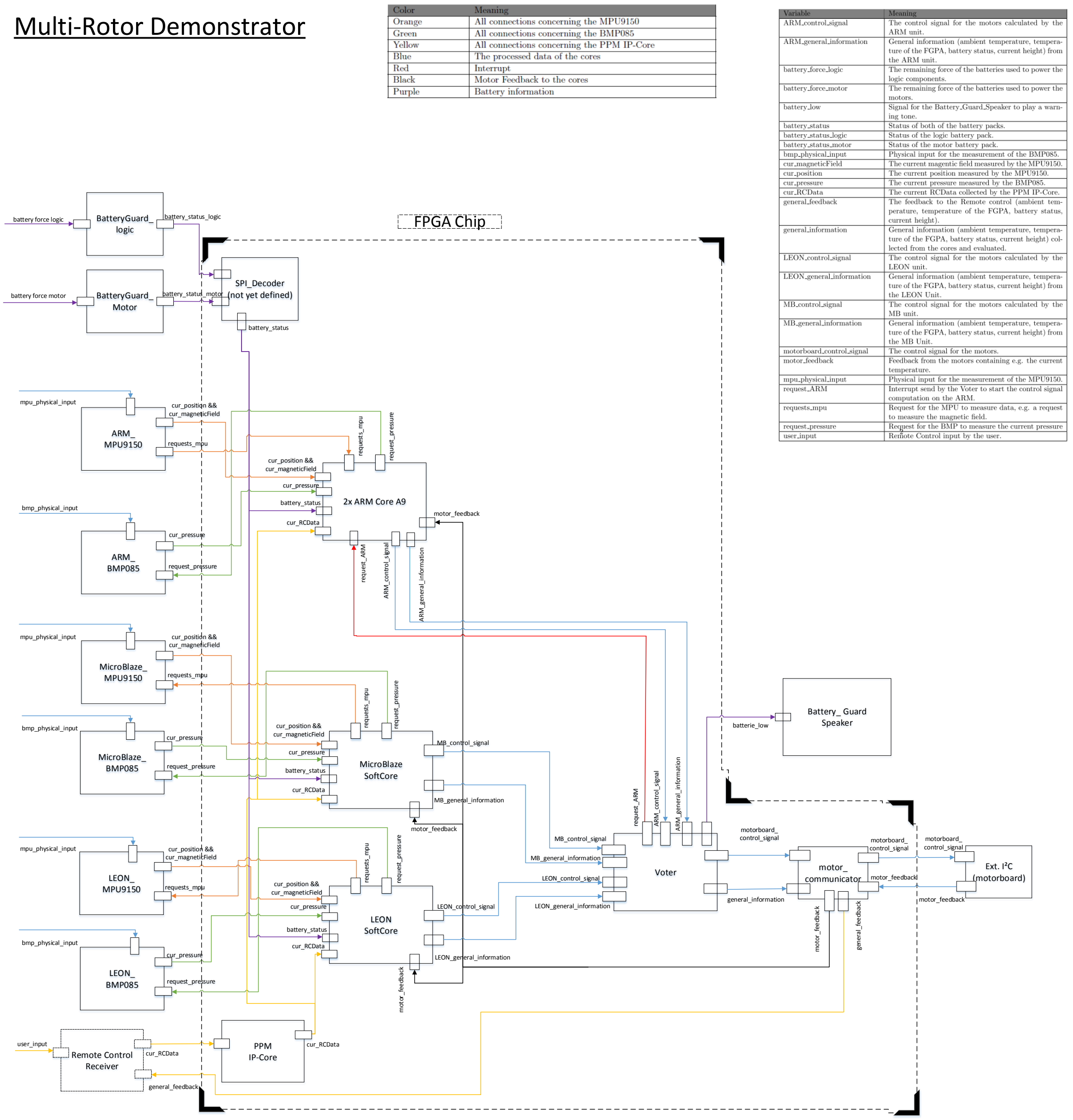
`$ARMitrpt`: the interrupt sent from the `Voter` to the `ARM` core

`$ARMctrl`: the control signal sent from the `Voter` to the `motor_communicator`

The `$ARMitrpt` signal is a binary signal, and the other signals are real-valued (assuming that all necessary control signal information can be encoded as a real number). For all signals, a value $== 0$ indicates that the signal is absent, and a value $\neq 0$ indicates that the signal is present.

Using the above variables, the textual description of the safety requirements is given in the following overview, where item 7 represents the top level safety requirement:

1. Every 2ms, the `MicroBlaze` SoftCore sends the `$MBctrl` signal to the `Voter`
2. Every 2ms, the `LEON` SoftCore sends the `$LEONctrl` signal to the `Voter`
3. As soon as the `Voter` has received the signals `$MBctrl` and `$LEONctrl` and they are identical, it sends the `$motorctrl` signal to the `motor_communicator` in less than 2ms
4. As soon as the `Voter` has received the signals `$MBctrl` and `$LEONctrl` and they are not identical, it sends the `$ARMitrpt` signal to the `ARM` Core in less than 2ms
5. As soon as the `ARM` Core has received the `$ARMitrpt` signal, it sends the `$ARMctrl` signal back to the `Voter` in less than 2ms
6. As soon as the `Voter` has received the `$ARMctrl` signal, it sends the `$motorctrl` signal to the `motor_communicator` in less than 2ms
7. The `$motorctrl` signal must be sent from the `Voter` to the `motor_communicator` at least every 2.5ms *[equivalent to 400Hz]*



Color	Meaning
Orange	All connections concerning the MPU9150
Green	All connections concerning the BMP085
Yellow	All connections concerning the PPM IP-Core
Blue	The processed data of the cores
Red	Interrupt
Black	Motor Feedback to the cores
Purple	Battery information

Variable	Meaning
ARM_control_signal	The control signal for the motors calculated by the ARM unit.
ARM_general_information	General information (ambient temperature, temperature of the FGPA, battery status, current height) from the ARM unit.
battery_force_logic	The remaining force of the batteries used to power the logic components.
battery_force_motor	The remaining force of the batteries used to power the motors.
battery_low	Signal for the Battery_Guard_Speaker to play a warning tone.
battery_status	Status of both of the battery packs.
battery_status_logic	Status of the logic battery pack.
battery_status_motor	Status of the motor battery pack.
bmp_physical_input	Physical input for the measurement of the BMP085.
cur_magneticField	The current magentic field measured by the MPU9150.
cur_position	The current position measured by the MPU9150.
cur_pressure	The current pressure measured by the BMP085.
cur_RCData	The current RCData collected by the PPM IP-Core.
general_feedback	The feedback to the Remote control (ambient temperature, temperature of the FGPA, battery status, current height).
general_information	General information (ambient temperature, temperature of the FGPA, battery status, current height) collected from the cores and evaluated.
LEON_control_signal	The control signal for the motors calculated by the LEON unit.
LEON_general_information	General information (ambient temperature, temperature of the FGPA, battery status, current height) from the LEON Unit.
MB_control_signal	The control signal for the motors calculated by the MB unit.
MB_general_information	General information (ambient temperature, temperature of the FGPA, battery status, current height) from the MB Unit.
motorboard_control_signal	The control signal for the motors.
motor_feedback	Feedback from the motors containing e.g. the current temperature.
mpu_physical_input	Physical input for the measurement of the MPU9150.
request_ARM	Interrupt send by the Voter to start the control signal computation on the ARM.
requests_mpu	Request for the MPU to measure data, e.g. a request to measure the magnetic field.
request_pressure	Request for the BMP to measure the current pressure
user_input	Remote Control input by the user.

Figure A.1.: The original hardware architecture of the quadcopter

No.	TSE = TC = TEE	LS	ASE = AC = AEE
1.	`true`	$[2, 2]$	(`$MBctrl` $\neq 0$)
2.	`true`	$[2, 2]$	(`$LEONctrl` $\neq 0$)
3.	(`$MBctrl` $\neq 0$) $\wedge$ (`$LEONctrl` $\neq 0$) $\wedge$ (`$MBctrl` $==$ `$LEONctrl`)	$[0, 2]$	(`$motorctrl` $\neq 0$)
4.	(`$MBctrl` $\neq 0$) $\wedge$ (`$LEONctrl` $\neq 0$) $\wedge$ (`$MBctrl` $\neq$ `$LEONctrl`)	$[0, 2]$	(`$ARMitrpt`)
5.	`$ARMitrpt`	$[0, 2]$	(`$ARMctrl` $\neq 0$)
6.	(`$ARMctrl` $\neq 0$)	$[0, 2]$	(`$motorctrl` $\neq 0$)
7.	`true`	$[0, 2.5]$	(`$motorctrl` $\neq 0$)

Table A.1.: SUP formalisation of the safety requirements.

TSE	Trigger Start Event	ASE	Action Start Event
TC	Trigger Condition	AC	Action Condition
TEE	Trigger End Event	AEE	Action End Event
LS	Local Scope		

Table A.2.: Abbreviations used in Table A.1

SUP Formalisation

We use the Simplified Universal Patter ([1, 4]) to formalise the above requirements.

In short, SUPs relate triggers/preconditions with actions/postconditions, both expressed as boolean expressions over the observables of the system. The semantics of SUPs is based on (simulation) runs of the system, SUPs can thus be viewed as observers. An SUP accepts a run if both trigger and action are traversed successfully, and rejects a simulation run (the simulation run is called violating) if the trigger is traversed successfully but the action is not. See [1] for a detailed presentation of SUPs.

In our formalisation, all requirements use the activation mode "cyclic"[1] and the interpretation type "progress"[2]. This combination of activation mode and interpretation type allows for an arbitrary number of sequential instances of each pattern. The formalisation of the above requirements is shown in Table A.1 (the abbreviations used are shown in Table A.2). To improve readability, we omit the information about activation mode and interpretation type from the table, since these are identical for all requirements. For the same reason, we also omit the information about Trigger Duration Interval and Action Duration Interval (the intervals are $[0, 0]$ for all requirements), Trigger Exit Condition and Action Exit Condition (`false` for all requirements), as well as the Global Scope (∞ for all requirements), and we unite columns with identical content.

[1] The activation of the SUP (re)starts after a previous observation cycle with an inconclusive ore passed result, i.e., the SUP is cyclically (re)activated as long as it is not violated.

[2] Used to express liveness: an action shall follow a trigger at a certain point in time or within a certain time interval.

Bibliography

[1] *BTC EmbeddedPlatform Concepts and Use Cases (user manual)*

[2] AMALTHEA4public: Deliverable D4.1 - Gap analysis against ISO 26262. July 2015. – Forschungsbericht. Deliverable Report

[3] AMALTHEA4public: Deliverable D4.2 - Integration of existing tools for safety development process. July 2016. – Forschungsbericht. Deliverable Report

[4] Bienmüller, Tom ; Teige, Tino ; Eggers, Andreas ; Stasch, Matthias: Modeling Requirements for Quantitative Consistency Analysis and Automatic Test Case Generation. November 2016 (CS-TR-1503). – Forschungsbericht. Proceedings of the Workshop on Formal and Model-Driven Techniques for Developing Trustworthy Systems (FM&MDD16

[5] contributors, Various open-source: *Open Services for Lifecycle Collaboration.* July 2015. – `http://open-services.net/`

[6] Ellen, Christian ; Sieverding, Sven ; Hungar, Hardi: Detecting Consistencies and Inconsistencies of Pattern-Based Functional Requirements. In: Lang, Frédéric (Hrsg.) ; Flammini, Francesco (Hrsg.): *Formal Methods for Industrial Critical Systems - 19th International Conference, FMICS 2014, Florence, Italy, September 11-12, 2014. Proceedings* Bd. 8718, Springer, 2014, S. 155–169. – URL `http://dx.doi.org/10.1007/978-3-319-10702-8_11`. – ISBN 978-3-319-10701-1

[7] IBM: *About Jazz - Our Story - Jazz Community Site.* – `https://jazz.net/story/about/`, (2017-06-21)

[8] ISO: *ISO 26262 - Road vehicles — Functional safety.* November 2011

[9] ISO: *ISO 26262 - Road vehicles — Functional safety — Part 3 Concept phase.* November 2011

[10] ISO: *ISO 26262 - Road vehicles — Functional safety — Part 4 Product development at the system level.* November 2011