Project acronym:	*SafeAdapt*
Project title:	*Safe Adaptive Software for Fully Electric Vehicles*
Grant Agreement number:	608945
Coordinator:	*Dr.-Ing. Dirk Eilers*
Funding Scheme:	*FP7-2013-ICT-GC*

Deliverable 4.1

Concept for Modelling Safe Adaptive System Behaviour

Due date of deliverable:	30th June 2015
Actual submission Date:	01st July 2015
Lead beneficiary for this deliverable:	CEA

Dissemination level		
PU	**Public**	**X**
PP	**Restricted to other programme participants (including the Commission Services)**	
RE	**Restricted to a group specified by the consortium (including the Commission Services)**	
CO	**Confidential, only for members of the consortium (including the Commission Services)**	

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013)

This document contains information which is proprietary to the members of the SafeAdapt consortium. Neither this document nor the information contained herein shall be used, duplicated or communicated by any means to any third party, in whole or in parts, except with prior written consent of the members of the SafeAdapt consortium.

Document Information	
Title	Concept for modelling safe adaptive system behaviour
Creator	CEA: Ansgar Radermacher, Önder Gürcan
Description	This document describes the modelling of safe adaptive system behaviour for the SafeAdapt project.
Publisher	CEA
Contributors	Tecnalia: Alejandra Ruiz, Mª Carmen Palacios, Maite Álvarez ESK: Alexander Stante, Dulcineia Oliveira da Penha, Gereon Weiss Ficosa: Andrea Saccagno (Review) Duracar: Ken Lam (Review)
Language	en-GB
Creation date	05/01/15
Version number	0.7
Version date	23/06/15
Audience	☐ internal ☒ public ☐ restricted

Table of Contents

List of Figures 4
List of Tables 5
Glossary 6
Executive Summary 7
1 Introduction 8
1.1 Document scope 8
1.2 Document outline 8
2 Classification of adaptive systems 9
3 Modelling Requirements 11
4 Related Work 12
4.1 Feature modelling in EAST-ADL 12
4.2 DiVA 13
4.3 DySCAS 13
4.4 Design patterns 14
4.5 Summary and conclusions 14
5 Modelling of Adaptation 16
5.1 Anticipated adaptations – predefined configurations 17
5.1.1 Cluster definition for modelling degraded functionality 19
5.2 Triggers, conditions and actions 21
5.3 Fault Tolerance 22
5.3.1 Fault tolerance patterns 23
5.4 Modelling design patterns 25
5.5 Modelling of constraints 27
5.5.1 Modelling allocation constraints 27
5.5.2 Modelling of resource constraints 27
5.5.3 Modelling of timing constraints 27
5.5.4 Modelling of adaptation constraints 27
5.5.5 Constraint evaluation 28
6 Models@runtime 29
7 Validation of Safe Adaptive System Behaviour 31
8 Summary 33
9 Bibliography 34

List of Figures

Figure 1: Hierarchy of self-X properties (from [4])....... 9
Figure 2: Feature tree (l) and configuration of this tree (r) – [source EAST-ADL variability]....... 12
Figure 3: Configuration of feature tree....... 13
Figure 4 : Different configurations....... 18
Figure 5: Cluster modelling in case of an automatic cruise control....... 19
Figure 6: Overview of an UML-based profile for the degradation concept....... 20
Figure 7: DegradationMode for modelling system degradation....... 20
Figure 8: Different ways to model fault tolerance (from D3.1, [Salewski et.al 2008])....... 22
Figure 9: Model of passive fault-tolerance pattern with diversification....... 25
Figure 10: Modelling of a design pattern – identifying roles and applying constraints....... 25
Figure 11: EAST-ADL2 model describing the failure propagation in a hot standby system [13]....... 26
Figure 12: Self-X profile, static part....... 29

List of Tables

Table 1: Requirements for modelling adaptability ... 11
Table 2: SAPC fault causes and associated remedies ... 23

Glossary

Term	Definition
Configuration	A configuration is a set of software component (instances) along with their parameter settings and allocation to execution resources which in turn are executing on a specific ECU. Distinction between system wide configuration and projection on single ECU.
EAST-ADL	Automotive modelling language; exists in form of a dedicated language as well as a UML profile. Contains sub-profiles for different issues, e.g. variability and timing.
ECU	Electronic computation unit
MARTE	UML profile dedicated for "Modeling and Analysis for real-time and embedded systems", contains several sub-profiles and a generic value extensions allowing for value and type tuples
model@runtime	Projection of the design model (including the adaptation issues), used by the safe adaptation core to execute reconfigurations
Quiescent state	States in which adaptations may be safely performed
SAPC	Safe Adaptation Platform Core – A piece of software instantiated on every core node to control the local runtime reconfiguration
TIMMO	Timing extensions of EAST-ADL
UML	Unified modeling language

Executive Summary

This document describes the concept for *modelling* safe adaptive system behaviour. This means that the model provides information that allows the safe adaptation core to execute the adaptation at runtime in a way that assures the requirements with respect to timeliness, resource constraints and safety.

Different modelling approaches are chosen depending on nature of an adaptation and the criticality of the affected components, i.e. there is no "one mechanism fits all" approach. The main reason is to achieve the objective of scalability (avoid combinatorial explosion) and that subsystems of different criticality have different requirements with respect to off-line validation.

Besides the modelling itself, the document describes tooling aspects, i.e. how tools support the modelling and which artefacts can be generated from the adaptation model, notably the information required by the safe adaptation core. This information is a projection of the adaptation modelled at design time and is also denoted as a model@runtime.

1 Introduction

This document describes the concept for *modelling* safe adaptive system behaviour. The model describes how the system will react to events such as faults, environmental conditions or the wish to integrate third party components.

The modelling is specified as an additional and complementary aspect of the system model which itself is modelled using UML in combination with the EAST-ADL profile. EAST-ADL already provides a way to specify timing constraints, including end-to-end deadlines.

1.1 Document scope

The document describes how adaptation is modelled at design time, which artefacts are generated from the design model and how tools support this process. The document is intended for system architects that need to define adaptation conditions and constraints as well as implementers of the runtime mechanisms (D3.1 – Concept for Enforcing Safe Adaptation during Runtime).

1.2 Document outline

The document is structured as follows. In the next section, we introduce the requirements for modelling adaptation, for instance scalability, the possibility to express allocation constraints and so on. The next session is the main part of this deliverable. It describes how adaptation is actually modelled and classifies different ways of modelling adaptation. The last section describes a model at runtime, i.e. the runtime artefacts of the modelling.

2 Classification of adaptive systems

In this section, we classify safe adaptive systems according to the literature. In the sequel, we consider in particular self-adaptive systems, since the systems within a car have to make autonomous reconfiguration decisions (which is an important difference between adaptive an self-adaptive systems). Thus, we start with definitions from literature what self-adaptive systems are. Laddaga [3] defines these as

"Self-adaptive software evaluates its own behaviour and changes its behaviour when the evaluation indicates that the system is not accomplishing what is intended to do, or when better functionality or performance is possible […] This implies that the software has multiple ways of accomplishing its purpose and has enough knowledge of its construction to make effective changes at runtime."

The last part of the quotation above refers to what is today denoted as model@runtime. i.e. additional information about the software (its components) that are running on a system.

Self-adaptive systems have multiple self-X properties, shown in the overview article from Salehie and Tahvildari [4]. Figure 1 shows the hierarchy as proposed in this article. At the basic level, a self-adaptive system needs to be aware of itself (its structure) and its context / environment. This enables properties such as self-optimisation and self-healing.

In the context of SafeAdapt, we are interested primarily in the self-healing (i.e. increase robustness with respect to faults, self-protecting is more related to security instead of safety) and self-optimising with respect to energy consumption.

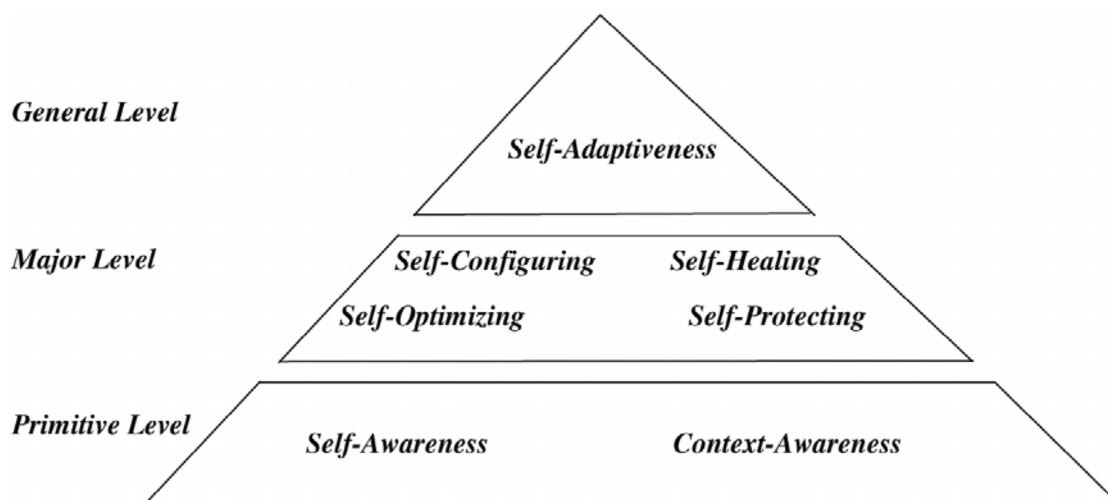

Figure 1: Hierarchy of self-X properties (from [4])

Adaptation actions can be further classified with respect to the time at which they are executed. There is a distinction between *reactive* and *proactive* actions. In case of the former, the system performs an action as a direct reaction to a change. In case of the latter, the system adaptation is based on a prediction of (environmental) changes. Thus, it may eventually reduce the number of adaptations or react before a (critical) event occurs. An example in SafeAdapt is the driver-

drowsiness detection – it predicts a loss of attention and executes counter measures. Eventually, it may be possible to predict hardware failures based on the number of previous, temporary faults.

The time dimension has an additional aspect: at which moment in the system lifetime is a reconfiguration possible? The literature distinguishes between design-time, configuration-time (during start-up phase), in a safe state[1] (e.g. when stopped at a traffic light) or at a arbitrary point In the context of SafeAdapt, we target re-configuration at runtime – whereas the modelling of adaptations could be partly at design-time, as shown later.

Another classification is whether the adaptation action is an optimization or a corrective adaptation, i.e. an action that masks a fault. In the context of SafeAdapt, we target both kinds of adaptations.

Optimisation use cases are given in deliverable 5.1 – "Evaluation Methodology for the SafeAdapt results": (1) the shutdown of auxiliary systems if the battery level drops below 35% and (2) the energy recuperation by using electrical brakes whenever possible. A use case similar to the former the minimization of the energy consumption when the system is shut-off but needs to retain a few functions (such as keyless entry).

[1] Note that the term safe-state is also used in the work of runtime reconfiguration: it denotes a state in which no requests are active therefore enabling *consistency* during reconfiguration.

3 Modelling Requirements

The requirements for modelling adaptability comprise several aspects, notably **which** information is modelled and **how** this information is stored. Additional aspects are the usability and efficiency to create and store this information. A part of the information has been captured in deliverable D2.3 (requirements of the design process and tools), although the latter focusses more on the individual requirements of the tools.

Table 1 shows requirements extracted from D2.3:

ID	Category	Sub Category	Description	Conflicts
CEA-001	Functional	System	Modelling of system architecture in EAST-ADL. Use Timing extensions of EAST-ADL	
CEA-002	Non-Functional	System	Modelling of non-functional requirements using UML/MARTE	
CEA-003	Functional	System	Modelling of global modes, but respect CEA-004	CEA-004
CEA-004	Non-Functional	Efficiency	Scalability – Avoid combinatorial explosion, i.e. avoid enumerating all possible system configurations in detail.	
CEA-005	Non-Functional	Tools	Depending on criticality, we must be able to assure schedulability even during reconfiguration at all times (offline-analysis) possible conflict: doing schedulability analysis requires detailed configuration information and **validation** of **all** possible transition	CEA-004
CEA-006	None	Process	Respect tool flow picture from D4.2	
		System	Modelling of adaptation is a separate issue that should be orthogonal to system definition	

Table 1: Requirements for modelling adaptability

There are two particular requirements that are important in the context of modelling adaptability:

The modelling must be **scalable** (CEA-004). This means that the size of the adaptability model may only grow linear or slightly faster (e.g. n*log(n)) with the system size. This requirement is important since it enables developers to specify the adaptation model in an efficient way, and since it also limits the storage requirements of the adaptation information in the running system.

Critical components or subsystems require that all resource and timing requirements are met. While the validation of a new configuration after and during adaptation is in principle possible based on information at runtime, the results could not be guaranteed and the delay for the calculation could be important. Therefore, it is necessary to **pre-calculate** some information (CEA-005), implying a potential conflict with the requirements to be scalable.

The specification of timing constraints, including end-to-end deadlines is done using the TIMMO extension of EAST-ADL.

4 Related Work

In this section, we list related work in the context of modelling adaptation.

4.1 Feature modelling in EAST-ADL

EAST-ADL offers the possibility to model at multiple levels of abstraction. The most abstract level is the **feature level**, in which a desired functionality is represented by a feature. Features may be structured in a tree-like way, including feature groups with cardinalities. The main idea is that different *variants* of a car differ in the sense that some features are present in a certain variant while others are not. Often a choice needs to be made. On the feature tree level, such a choice is represented by a so-called feature group offering an m out-of n choice. A configuration represents a selection of features for a product that is going to be built. Figure 2 shows a feature and a concrete configuration. In the configuration, the option ClimateCtrl is active, with respect to the wiper the mutual choice for an advanced wiper has been chosen but there is no rain sensor. Although the feature model is *originally intended for making design time choices*, it could be interesting to allow for adaptations in the sense of re-evaluating these choices at runtime.

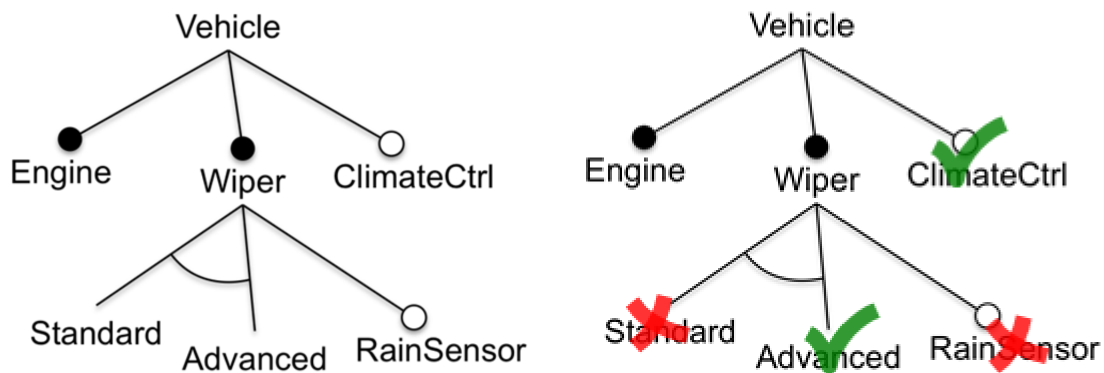

Figure 2: Feature tree (l) and configuration of this tree (r) – [source EAST-ADL variability]

The functional analysis architecture represents a set of components that realize a feature. These components already have ports and are assembled as parts within a system representing the analysis architecture

The detailed design architecture represents the architecture of the system that is deployed on the hardware. There is no instance modelling in EAST-ADL (most components exist exactly once, the cardinality of parts controls the number of instances). Figure 3 shows a feature tree at the design level, the alternative choice between two options of a wiper control results in two optional components. One of these could be activated at runtime.

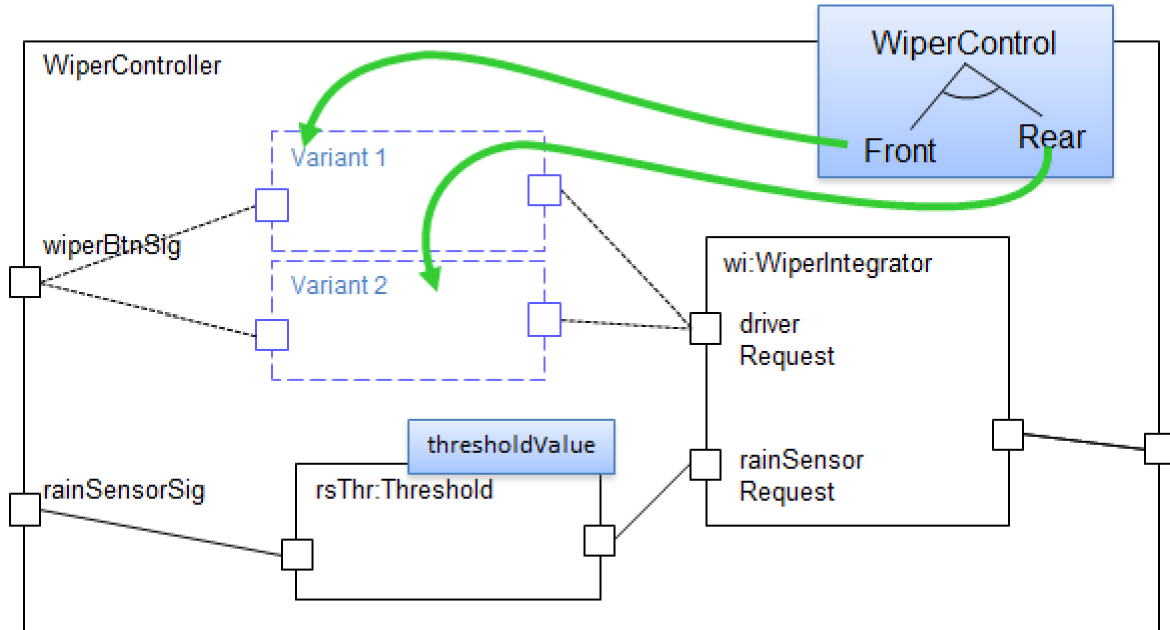

Figure 3: Configuration of feature tree

Thus, the combination of feature trees and configurations is a part of the adaption modelling which may be used to make decisions at runtime, i.e. to toggle at runtime between configurations.

The former remains on the feature level, the latter is done on the design level.

4.2 DiVA

The DiVA project [21] focuses on dynamic variability in complex systems following a component-based approach, close to Fractal. The actual configurations of the application are built at runtime, based on Aspect-Oriented Modelling (AOM) and models at runtime. A main objective of the DiVA approach is to model adaptive systems without having to enumerate all possible configurations statically. The application is modelled using a base model, which contains the common functionalities and a set of variant models, which can be composed with the base model. An adaptation model specifies which variants should be selected at runtime according to the adaptation rules and the current context of the executing system. However, the DiVA approach is not targeting distributed embedded systems, e.g. due to the large resource consumption overhead at runtime.

4.3 DySCAS

The DySCAS (Dynamically Self-Configuring Automotive Systems) project [18][19] focused on developing a middleware enabling dynamic self-configuration of automotive software systems. Adaptation of the system is realized through decision points, at which a policy has to be evaluated. Policy evaluation at runtime poses a significant overhead.

A main use case is the addition of a new device. [20] describes a UML based modelling approach for the DySCAS reference architecture, consisting of analysis and design of application and middleware functions. The DySCAS reference architecture contains an Information Model which is

interesting in the context of modelling adaptability, as it describes the *architectural variability* of configurations.

4.4 Design patterns

The concept of design patterns [12] became one of the widely used and universal approaches for describing and documenting recurring solutions for design problems. The idea of design patterns was original proposed by the architect Christopher Alexander [6] who wrote several books on the field of urban planning and building construction. He defined a design pattern as a construct that expresses a relation between three parts: context, problem and solution (what is called three-part rule). Ever since, this concept has been applied to many different domains, including hardware and software design.

In software domain, design patterns became popular in software architecture and development after the success of the book Design Patterns: Element of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides [9] frequently referred to as the Gang of Four (GoF). They defined a design pattern as “Description of communicating objects and classes that are customized to solve a general design problem in a particular context”. This book, which includes a collection of patterns for object-oriented software design, was not the first effort to use the concept of design pattern in software; however, it is considered as the foundation for design patterns in software construction.

Like in software development, design patterns have been also adapted for hardware design to provide implementation-independent and abstract views for recurring hardware design solutions. The more general benefit of using design pattern is to improve the hardware development process through importing the advantages of object oriented modelling techniques (see e.g. [7][8][10][11]).

In general, we can summarize a design pattern as an abstract representation for how to solve a general design problem that occurs over and over in many applications. The main purpose of design patterns is to support and help designers and system architects to choose a suitable solution for a recurring design problem among available collection of successful solutions. Moreover, design patterns can simplify communication between practitioners and provide a good way for documentation of proven design techniques.

4.5 Summary and conclusions

We have shown existing approaches to deal with variability and adaptability. Although the variability modelling on the feature level is primarily intended for the static configuration of a system, it is useful for runtime adaptations as well. In particular, it can capture several (mutually exclusive) realization options of a function, which we will use in the context of degraded functions as shown later.

The projects DiVA and DySCAS use and evaluate models at runtime in order to execute autonomous adaptation decisions. This approach is also chosen in SafeAdapt: runtime decisions are executed by the safe adaptation core (SAPC) based on model information. The main difference is that SafeAdapt needs to make some decisions at design time in order to guarantee the offline validation of safety-critical functions. SafeAdapt will take real-time considerations into account (on

the one hand by explicitly modelling timing constraints, on the other by taking care of performance aspects in the runtime representation of these models)

Design patterns are a general way to associate design problems with a well-known solution. We will primarily use the patterns in the context of fault-tolerance patterns in section 5.3.

5 Modelling of Adaptation

Before we can introduce different ways of modelling adaptation, we want to introduce the relevant terminology.

Definition 1: a system-configuration defines a set of components, along with their interconnections and their allocation to tasks which in turn are allocated to a hardware node (ECU).

Definition 2: A node-configuration is a subset of a system-configuration. It consists of the tasks for a certain hardware node and the components allocated to it.

Please note that an ECU needs to know at least its node-configuration; it is currently not clear whether a single ECU knows the complete system configuration.

*Definition 3: a configuration is **valid**, if it respects resource and timing constraints. The latter can be verified by means of a schedulability analysis. A configuration may be accompanied by a scheduling plan that defines task priorities (in case of fixed priority scheduling). This definition can be applied to system and node configurations.*

*Definition 4: An **adaptation** (or reconfiguration) is the transition from one configuration to another. This transition may be triggered by different events ranging from optimisations to failures.*

Definition 5: The health vector (see D3.1) provides status information about current configuration, it may trigger adaptation (or reconfiguration).

The **modelling** of adaptations as defined above is to find a suitable level of abstraction to define system-configurations as well as the transitions between these. Node configurations can be derived automatically from system configurations.

In the sequel, we define different ways to model adaptation at different abstraction levels, i.e. requiring more (or less) detailed information from the developer. In this section, we only have a brief description of each modelling aspect which is discussed in more detail in its own section within this chapter.

The modelling of adaptation is often based on rules or policies that consist of a triple (event, condition, action), denoted as *ECA*: if the event is fired, the associated condition is evaluated. If the result is true, the associated (reconfiguration-) action is executed. Rule based adaptation is frequently used, for instance used in the robots domain.

1. Define configurations and transitions explicitly. If this approach is chosen, a configuration of the system is defined in detail. An **anticipated** event, e.g. the failure of an ECU or a low-battery condition is used as trigger for the move towards a new configuration. Defining a configuration explicitly within the model does not mean that the developer has to provide

this information. For instance, the allocation to a task (and the configuration of this task) is the result of schedulability analysis.

A particular case of anticipated events are classical fault tolerance mechanisms: critical components need to be replicated to achieve a required availability. By defining a (passive) replication strategy, we define implicitly a set of configurations in which some replicas are active while others are not (and some used degraded variants of a rudimentary functionality). Fault detection is a trigger for transitions. The likelihood of a failure of a replica (or the ECU hosting it) is explicitly modelled in order to derive a suitable number of replicas. Thus, fault-tolerance is also within the class of anticipated fault recovery.

2. Define constraints. Constraints impose restrictions on configurations and on reconfiguration actions. Constraints are typically combined with explicit configurations, since they reduce the search space of valid configurations. The latter is important in the case that the system has to handle a **non-anticipated** failure and needs to calculate a new configuration at runtime. Constraints can be applied to difference aspects of the system, for instance to timeliness (timing constraints) or memory consumption (resource constraints).

Adaptation execution needs an underlying runtime mechanism, in case of SafeAdapt, the safe adaptation core as identified in D3.1. The adaptation core supports a set of runtime mechanisms, notably the activation of a stored configuration. Its use imposes implementation restrictions, for instance the number of pre-validated configuration that can be stored. Thus, it may be possible that a failure has been anticipated, but a predefined configuration reacting to this fault is not stored and the failures must therefore be handled like an unanticipated failure. The decision whether to use a stored configuration needs to take the importance of events into account, e.g. a combination of likelihood and safety impact. The use of a suitable mechanism thus becomes a derived information.

5.1 Anticipated adaptations – predefined configurations

Highly critical parts of the system must meet their availability requirements as well as resource and timing constraints. The latter must also hold while the system is executing a reconfiguration. Certification requires that the system design guarantees these properties by means of a validation at design time. Therefore, it is not feasible to calculate a new configuration at runtime, but all possible configurations and during transitions between these need to be validated.

Event-condition-action rules can describe the activation of a new configuration, i.e. the action part of the ECA rule activates a certain configuration. The rules correspond naturally to transitions in a state machine, since the latter have a trigger (event), a guard (condition) and an effect (action). The main difference is that the firing of a transition depends on the source state (corresponding to a configuration) whereas the ECA rules can always fire and would need a suitable filtering condition.

The number of possible configurations can potentially grow quickly, since each additional component can be active or inactive in a certain configuration and allocated to different ECUs. There are several ways to reduce the combinatorial explosion.

1. Restrict predefined configurations to single hardware failures. This means that the case of a second failure after a first is not part of the set of predefined configurations.

2. Restrict predefined configurations to critical subsystems which in turn requires that the operating system isolates subsystems of different criticality from each other to avoid mutual influences, as done by partitioned operating systems used in the avionics domain. However, this solution has two disadvantages. The first is that a SafeAdapt solution needs to work in the context of the commonly used operating systems in the automotive domain (AUTOSAR-OS based on OSEK/VDX) that do not support partitions. The second is that the assignment of fixed time budgets to partitions reduces overall effectiveness (difficult to tune budget), in particular for a large number of partitions.

3. Hierarchical approach. Model global modes and certain failure situations. Possible sub-states can be explicitly modelled. In such an approach, the modelling effort remains scalable while the number of different fine-grained configurations (as well as transitions between these) may explode. While the latter is costly, if may be acceptable, since it can be largely automated.

4. Define additional constraints that reduce the number of possible configurations. For instance, allocation constraints (e.g. component may only be co-located) and dependencies exclude configuration from the search space. Constraint modelling is also important for unplanned adaptations, as it allows for online validations of possible configurations.

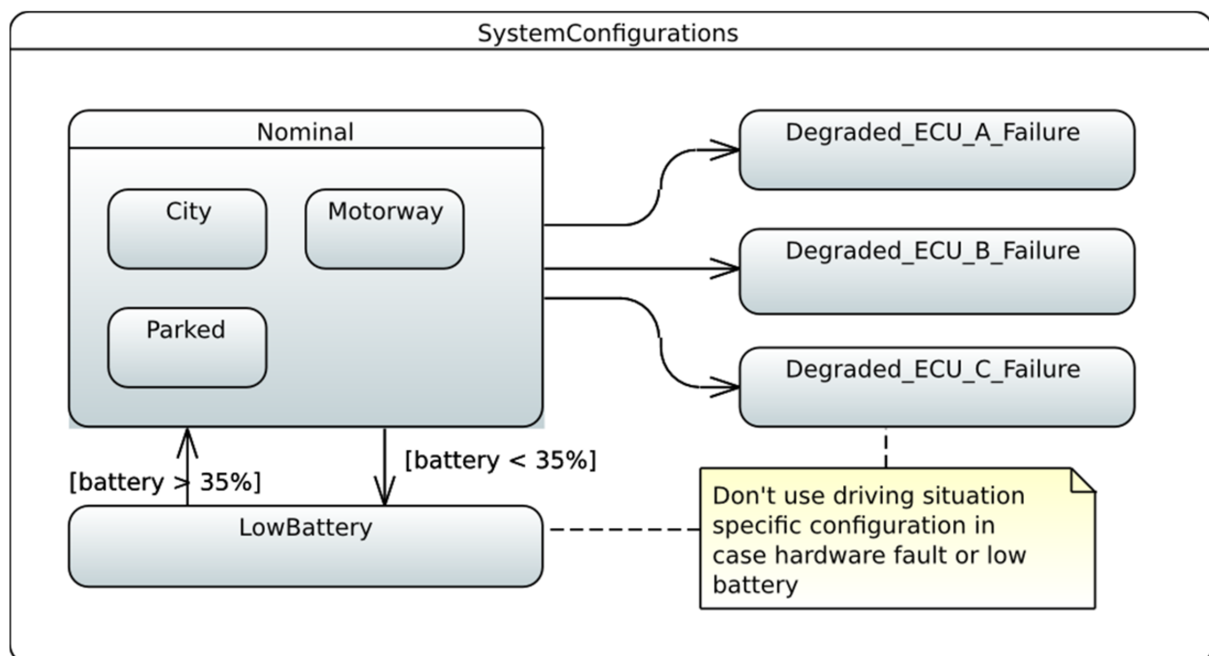

Figure 4 : Different configurations

Figure 4 shows an example of a state machine that represents a combination of the strategies (1) and (3). It considers three degraded states corresponding to the failure of ECU_A, ECU_B and ECU_C respectively. There is an additional low battery state/configuration which reconfigures the system to save energy (switching off auxiliary components with a relatively high energy consumption, as outlined in D5.1). The degraded modes do not support a specific low-battery state, as we would have a specific sub-state for each of them. The nominal state has nested sub-states corresponding to the driving situation. The use of the state-machines hierarchy avoids to specify multiple triggers to the degraded or low-battery states.

5.1.1 Cluster definition for modelling degraded functionality

Some automotive functions can offer different degrees of functionality depending on the availability of sensors or the presence of failures. Figure 5 shows an example of an automatic cruise control function that comes in three different variants. A fully working cruise speed, a nominal ACC and a variant called Speed Control. The coloured boxes enclose the components that are required by the different variants. The complete cruise control requires all sensors, whereas the simple variant requires a single component. The different variants can be captured with the EAST-ADL variability modelling shown in the right-hand side of the figure: instead of making a choice at system configuration time, the choice depends on the availability (at runtime) of the sensors.

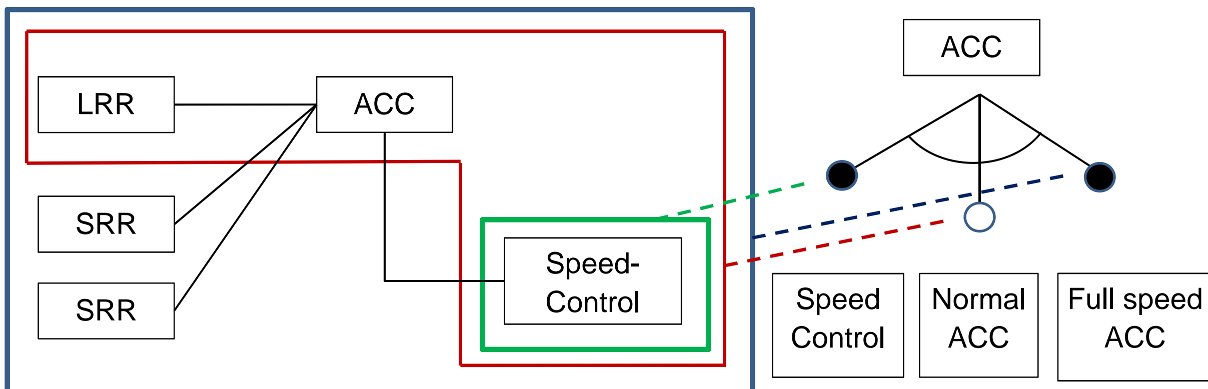

Figure 5: Cluster modelling in case of an automatic cruise control

The full-speed ACC depends on the functionality of the limited-speed ACC and two Short Range Radars (SRR) and both depends on the functionality of the traditional cruise control and Long Range Radar (LRR). These can be seen as a case in which a limited or degraded functionality can be offered even after a malfunction in parts of the system.

To model the degradation we propose to cluster software functions (e.g. EAST-ADL *FunctionPrototype*) in a *FunctionCluster*. This *FunctionCluster* represents a set of software functions which are necessary to deliver the functionality of the cluster. In the case of the example in Figure 5, *FunctionClusters* for the full-speed ACC, limited-speed ACC and Cruise Control can be modelled (illustrated by the coloured boxes). An overview of all elements in our proposal is shown in Figure 6 in form of a UML profile. During prototyping, we will evaluate which of the stereotypes are required and which can eventually be based on existing mechanism in EAST-ADL.

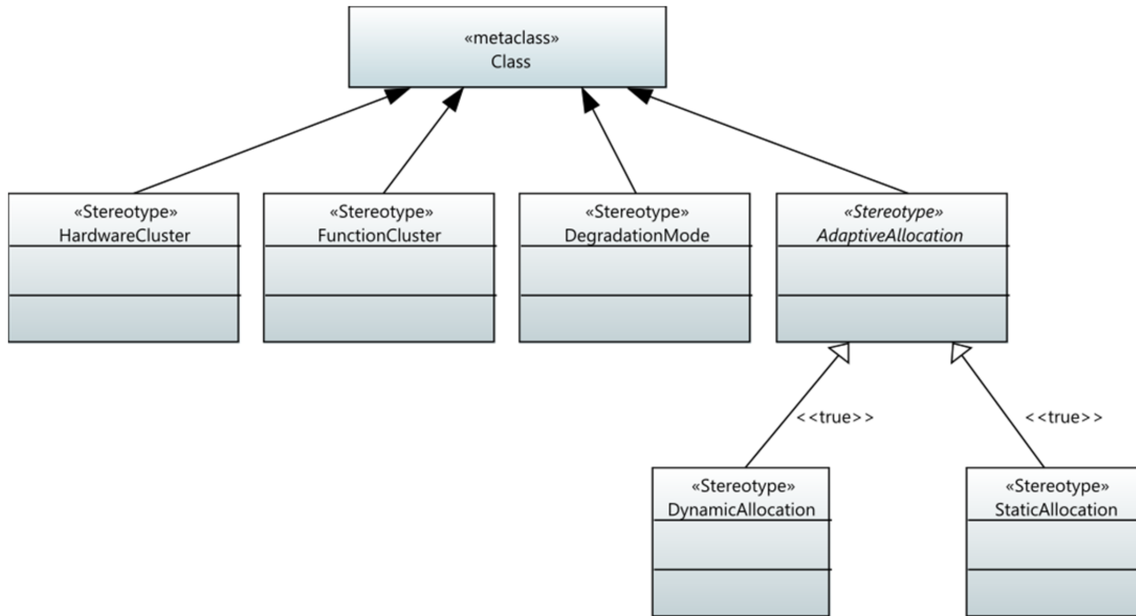

Figure 6: Overview of an UML-based profile for the degradation concept

To model the degradation between the *FunctionClusters* we propose to introduce a *DegradationMode* which refers to a *FunctionCluster* and also to a more degraded *DegradationMode*, as shown in Figure 7. The semantic is as follows: If a function in a *FunctionCluster* fails, the system can switch to a degraded mode to deliver at least limited functionality.

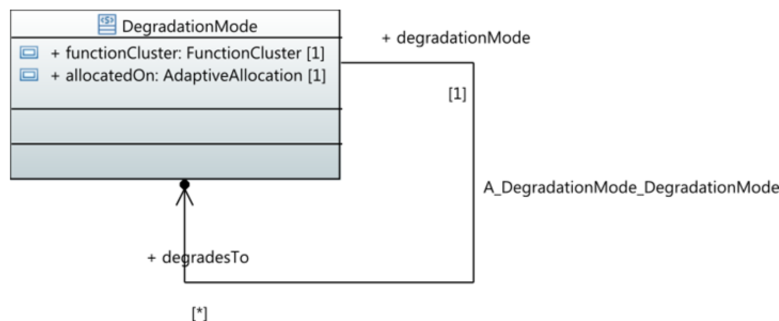

Figure 7: DegradationMode for modelling system degradation

We also extend the concept of clustering software functions to hardware. Therefore we introduce a *HardwareCluster* analogously to the *FunctionCluster*. The semantic of the *HardwareCluster* is that a cluster provides its functionality only if all hardware elements (e.g. EAST-ADL *HardwareFunctionPrototypes)* are available (i.e. operating). We furthermore propose a *DynamicAllocation* concept to provide a mechanism to model that a software function of a cluster can be allocated to any hardware element of a *HardwareCluster*. During runtime the system decides on which hardware element a software function is running. For some safety-critical functionality it is prohibited to determine the allocation dynamically during runtime. In that case we propose to model the static allocation of a *FunctionCluster* by referring the *DegradationMode* to a *StaticAllocation*. These allocations would be determined and validated offline and the system is only allowed to switch between statically validated allocations.

5.2 Triggers, conditions and actions

Adaptations can be triggered by several events. While the compensation of faults is very important in the context of SafeAdapt, it does not represent the only kind of triggering event. In this section, we list a set of possible triggers. We also add the associated conditions and actions. The resulting triple (ECA rule) can be represented by state machines and may also be associated with a design pattern, see section 5.4.

Event	Node failure (replica failure?)
Condition	
Action	- if active: none (remaining replica is already active and its results are immediately taken into account) - if passive: activate replica
Post-Action	Bring up new backup replica, i.e. execute a reconfiguration that re-establishes the original number of replicas and enables an eventual compensation of additional faults.

Event	Sensor failure
Condition	Condition: failed sensor is essential for correct component operation of a nominal function (this information is available in model@runtime)
Action	Activate degraded mode replica. See section 5.1.1 and 5.3.1.

Event	Replica state change
Condition	n-th change (objective: reduce traffic related to state changes)
Action	Broadcast state-update message

Event	Timer
Condition	State has been modified
Action	Broadcast state-update message

5.3 Fault Tolerance

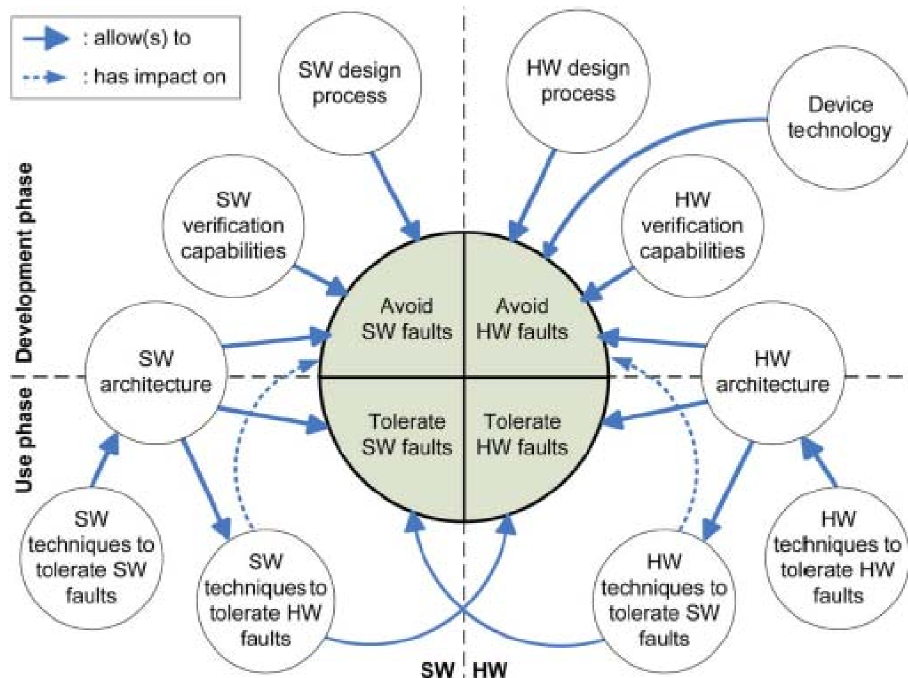

Figure 8: Different ways to model fault tolerance (from D3.1, [Salewski et.al 2008])

Figure 8 shows a classification of several fault tolerance mechanisms, distinguishing four different sectors. In the sequel, we will list each of the sectors briefly in the context of SafeAdapt.

1. Avoid SW faults: In the SafeAdapt project, we reduce the probability of software faults by means of a well-designed design process according to the ISO26262 methodology. Requirements are explicitly defined. The behaviour of software components is verified by means of analysis mechanisms and through simulation. We do not further describe these mechanisms in detail here, since they do not contribute to the modelling of adaptation.

2. Avoid HW faults: This aspect is not in the scope of the SafeAdapt project, no specific hardware is developed. However, the lockstep mechanism allows failure

3. Tolerate SW faults: Fault tolerance patterns use a set of replicas that may or may not run in parallel (see next section). The patterns provide the option to diversify the software, i.e. use different implementations of certain functionality. In case of an active replication, faults can be detected if there is a disagreement. However, in order to know which component is defective, at least three replicas are required. In the context of SafeAdapt, diversification is mainly done in form of using *degraded* components, i.e. components that provide less accurate results, but require fewer resources and use a simpler calculation which is therefore less likely to contain a software failure. Degraded components could be used to determine a plausible interval in case of active replication. In the context of SafeAdapt, degraded components are mainly used as backup components in a passive replication pattern. On the feature level, degraded variants can be modelled with the variability

mechanism in EAST-ADL, *function clusters* may capture the different dependencies, as shown in section 5.1.1.

4. Tolerate HW faults: The SafeAdapt hardware architecture enables a flexible routing of sensor as well as actuator data to each ECU. In combination with fault tolerance patterns and reconfiguration mechanisms, ECU failures can be tolerated. The lockstep mechanism enables failure detection and isolation. Deliverable D3.2 lists the used mechanisms in detail.

5.3.1 Fault tolerance patterns

Different kinds of faults have been classified in the Safe-Adapt deliverable D3.1, section 5.5 (Failure Detection, Classification & Remediation). The runtime uses a M.A.P.E-cycle (monitor, analyze, plan, execute, see Figure 4 of D3.1), failures are detected in the analysis phase of the M.A.P.E-cycle, implemented as part of the fault filter. Four different categories are identified.

SAPC fault cause class	Remediation
Maskable Transient Hardware Error	Remediation is directly performed in hardware or platform software. The SAPC remembers this occurrence in order to treat the error as permanent if it occurs more often.
Hardware Partition Error	Passivate partition and move application to other partition or alternatively to other core node.
General Hardware Error	Passivate core node and reactivate applications on other core nodes.
Application Design Fault	Passivate application (only QM) or use degraded variant

Table 2: SAPC fault causes and associated remedies

These faults can be captured by fault tolerance patterns. In the sequel, we list three different patterns how fault tolerance can be achieved within a software architecture.

- Active with vote: *n* components are running in parallel. Results are voted, typically *n* must be bigger or equal to three, otherwise it is not possible to tell who is “right”, if there is a divergence. Due to its relatively high cost, the pattern is less important in the automotive domain, but used in railways and avionics.
- Active without vote: *n* components are running in parallel. The result of one of the ECUs (components may not run on the same ECU. Results are not compared, each ECU performs regular self-tests and results are checked for validity.
- Passive: only one component is active at a given time. If the component detects that is fails (or other ECUs detect the failure/absence of data), a backup component is activated. There are different variants, how this backup component can be determined and whether other components are affected by this activation.

- Statically assigned backup, no further changes of architecture. In case of critical subsystems, the backup component must be known in order to validate in advance that resources are available and timing constraints are met. The latter includes a schedulability analysis of the configuration in which the backup component is active. Since there are no further changes, it is possible to make a single schedulability analysis assuming that original and backup are active at the same time, i.e. reducing combinatorial explosion.
- Statically assigned backup with additional changes. In this case, the activation of the backup component implies a further reconfiguration, e.g. relocate or deactivate less critical components (in order to achieve resource constraints). In this case, a separate schedulability analysis of the new configuration is required.
- Dynamically assign a backup component and eventually perform additional changes.

All three variants of this patterns might be followed by a “repair action” that assures that a further backup will be available in case of second failure. This additional reconfiguration may be executed in a suitable delay after the first reconfiguration that activated the original backup (less time critical, might be planned at runtime).

Active and passive patterns can be used in combination with diversified algorithms, i.e. the software for each replica has been developed by different teams according to the same specification. Diversification addresses the application (software) design fault possibility in the table above. In the context of SafeAdapt, the diversified algorithm is typically degraded: (1) a simpler computation that provides less accurate but “good enough” results, requiring less resources (useful, since the overall number of resources is reduce after a hardware failure) or (2) a different implementation that may compensate failed sensors or actuators. An example is a steer-by-wire functionality based on individual braking/acceleration of the in-wheel motors on the left or right hand side, respectively.

Active fault tolerance patterns do not require any state transfer. Thus, the switch-over time is smaller compared to passive systems. In case of a voter, no reconfiguration at all is implied if one of the replicas fails. In the absence of a vote, only a different output needs to be selected, schedulability analysis already took into account that all replicas are running. Therefore, the transition in case of a failure is not considered as a reconfiguration.

Thus, only the passive fault-tolerance patterns require a reconfiguration. A failure of the primary component “only” implies the activation of a backup component. The activation of the backup-component corresponds to the activation of a new configuration. In case of a hardware fault, the main roles in the pattern are the hardware (and not the software) components. The link between pattern and adaptation specification is that the pattern may be accompanied by a set of event-condition-action rules – eventually in form of a state-machine. Thus, the application of the design pattern implicitly defines adaptation rules. From a tooling viewpoint, it either implies that the system architect does not have to specify specific transitions or that the consistency of explicit transitions with the design pattern application could be verified.

It is also possible that the backup components represent alternative components that are activated not only in case of a failure but due to other triggering events. In this case, a local state-machine (or ECA rule) should capture the transition. This case might differ with respect to schedulability

analysis. Backup components will never run on the same ECU while alternative ones might do that. Thus, we need to analyse each configuration (i.e. one with the normal component and one with the alternative component) separately.

5.4 Modelling design patterns

Design patterns capture the combination of a problem and a solution in which the participants play several ***roles***. We propose to model this by defining a UML package that captures the problem, solution and applicability in textual form (stereotyped UML comments) as shown in [22]. The roles are explicitly modelled by means of a UML collaboration. The solution of a pattern consists of a composite class that refines the roles of the collaboration by adding additional relations and implementation artefacts (for instance a voter). Constraints can be defined either on the level of the solution or collaboration.

The model of the passive fault-tolerance pattern with diversified replicas is shown in Figure 9. The package applies the stereotype “SafetyPattern”, a specialization of the “Pattern” stereotype. The screenshot shows four comments that contain textual descriptions. The solved problem is the systematic risk related to software, i.e. the “application design fault” row in the table from D3.1.

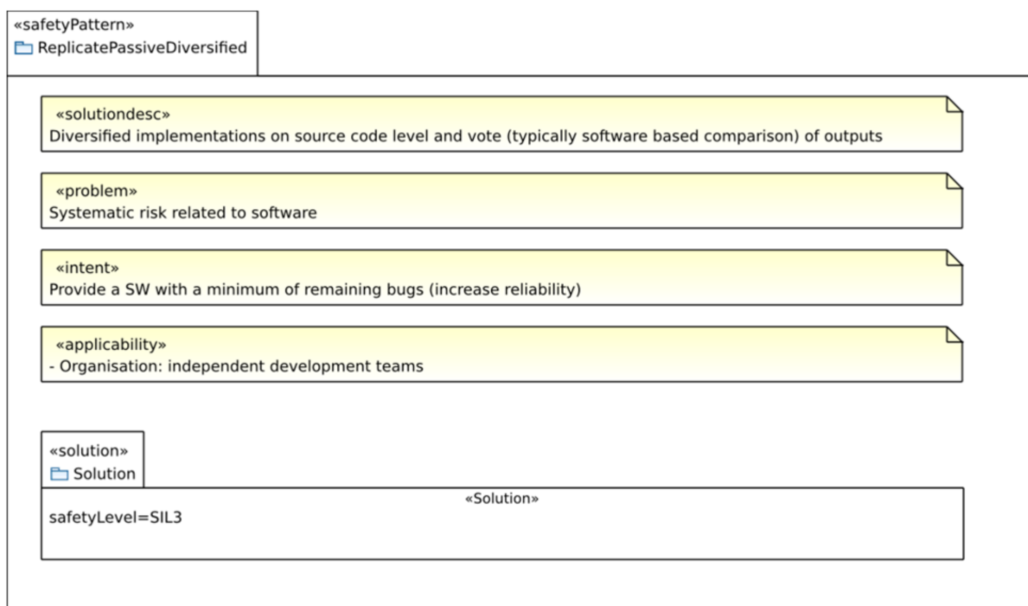

Figure 9: Model of passive fault-tolerance pattern with diversification

Figure 10 shows how the modelled roles within the collaboration. Nominal and degraded roles are identified. An allocation constraint is applied to the roles – they may not be allocated to the same ECU.

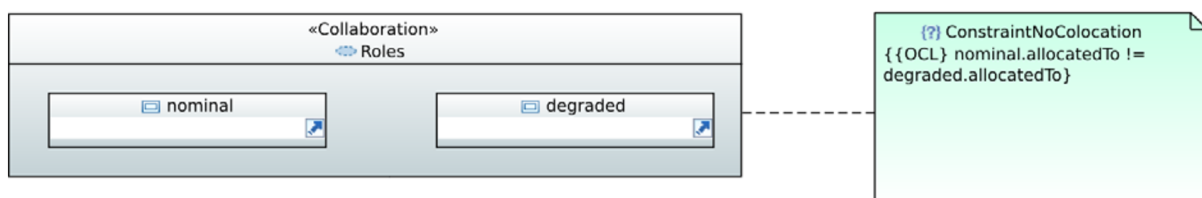

Figure 10: Modelling of a design pattern – identifying roles and applying constraints

Figure 11 illustrates the resulting architecture, when the pattern is applied. Consider an execution chain from F1 to F3 (to be exact from the parts that represent instances of F1, F2 and F3, respectively). F2 is also a composite using two replicas with different implementations (F21 and F22). In this example, the results of both replicas are fed into component F3 which is responsible for making a selection.

The application of design patterns helps to define configurations, since they determine a set of replicas (with an allocation constraint). This allocation of the replicas can be determined automatically, eventually be means of a constraint solver. Also possible: specify required availability (requirement), pattern determines number of required replicas.

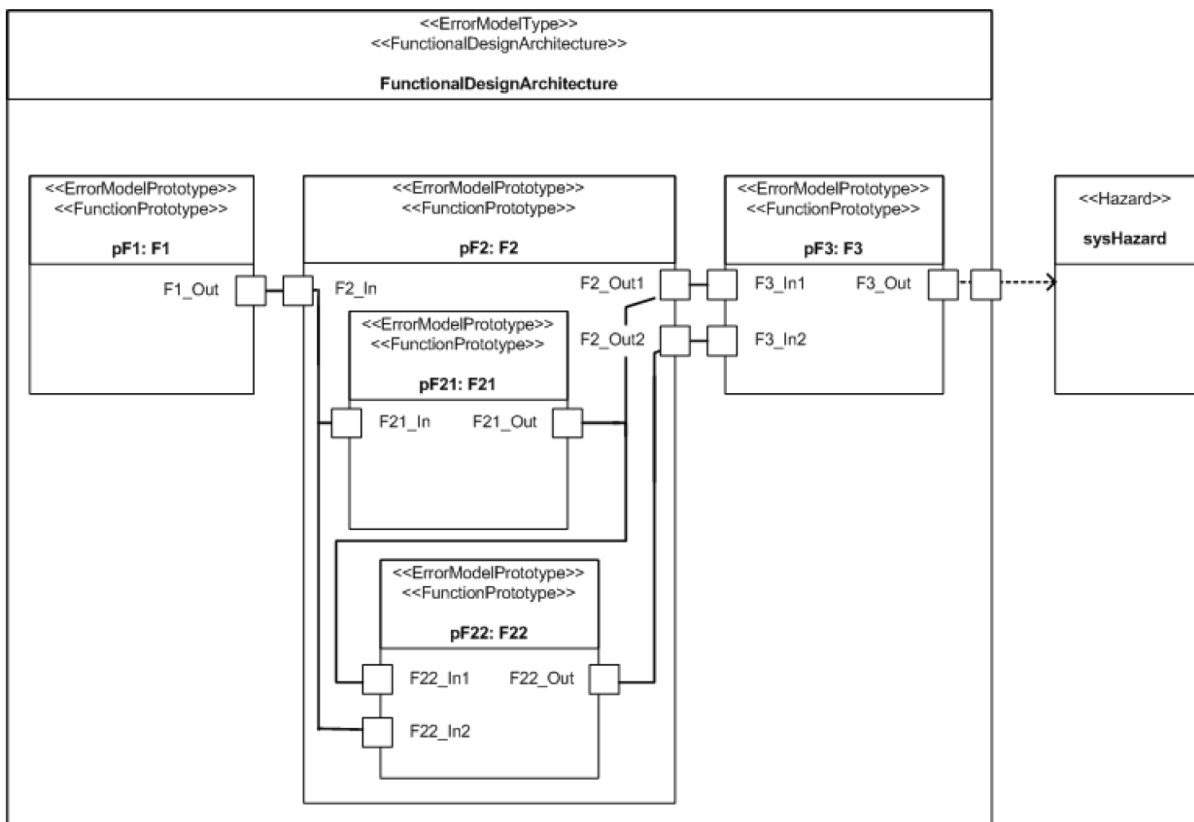

Figure 11: EAST-ADL2 model describing the failure propagation in a hot standby system [13]

The event-condition-action rules shown in section 5.2 can be associated with a design pattern: if a certain pattern is applied the runtime-adaptation-core executes the adaptation behaviour that is specified in the rule for the roles within the pattern. Thus, the use of the design pattern implies a known adaptation behaviour. This has the advantage that a developer does not need to specify reconfiguration actions manually for the associated components.

5.5 Modelling of constraints

In order to *verify* that a configuration is valid, it is necessary to check a set of rules or constraints. Constraints can apply to different aspects, for instance to the resource consumption, the timing, allocation or adaptation. Some constraints must be met by all applications (e.g. timeliness) while some are specific for an application (e.g. allocation constraints). The importance of constraints in the context of adaptation is twofold: they are used to verify configurations and to create new valid configurations automatically.

In the sequel, we discuss common constraints in embedded and real-time systems.

5.5.1 Modelling allocation constraints

An allocation constraint restricts the allocation of a component to a set of ECUs. Some components can only run on certain ECUs, e.g. due to the processor architecture or the availability of I/O signals or busses. Besides a direct constraint related to a specific ECU, it is common to have restrictions that depend on the mutual allocation of components. In the context of replication patterns, replicas should not run on the same ECU, i.e. to forbid co-localisation. For performance reasons the contrary is often required as well: application components that exchange a large amount of data, need to be co-located in order to meet deadlines.

5.5.2 Modelling of resource constraints

Application components require a certain amount of resources from the underlying platform, e.g. a certain amount of RAM, OS resources such as tasks, timers or semaphores. The generic constraint is that *the requirements of the application must meet the capabilities of the platform*. Therefore, both, application requirements and capabilities need to be modelled. For the former, we propose the use of the Self-X profile combined with EAST-ADL mechanisms. This profile describes for instance the memory resoures required by a component and its (worst case) execution time. It is the basis for creating a runtime model described in chapter 6.

For the latter the use of the hardware resource modelling part of MARTE, i.e. describe the resources that are offered by the hardware.

5.5.3 Modelling of timing constraints

Timing constraints can be modelled in different UML extensions, notably the profiles MARTE and EAST-ADL. In the context of this document, we will focus on specifying timing requires using the latter, notably the TIMMO specification within EAST-ADL.

The validation of timing constraints includes the execution of schedulability analysis: a plan how to applicate processor resources to the difference tasks within a system. The execution of such an analysis is a non-linear problem, when executed at runtime, heuristics are used to obtain a (likely sub-optimal) results in a short time frame.

5.5.4 Modelling of adaptation constraints

The adaptation is characterised by the transition from a source to a target configuration. Source and target configuration must respect the constraints defined so far. Timing constraints must also hold during the transition. In addition, specific adaptation constraints might explicitly forbid the evolution of certain applications.

Constraints are both evaluated for anticipated and non-anticipated adaptations. This means that constraint evaluation needs model information at runtime in order to verify a constraint. Examples of non-anticipated adaptations include the addition of a new component not been known at design time or failure scenarios that are not part of a predefined. It is also possible that a first failure could be handled by a planned reconfiguration but a subsequent failure has not been anticipated (while this is out of certification scope, it might be interesting to study this case).

Runtime decisions need to be taken for these non-anticipated adaptations and the runtime decisions are guided by constraints available in the model at runtime.

5.5.5 Constraint evaluation

An orthogonal aspect is the evaluation of constraints. If configurations are pre-configured, they can be completely evaluated offline or at runtime. The choice which algorithm is used can be derived automatically based on security level, recovery time, likelihood, and space restriction. In SafeAdapt, we will evaluate constraints at design time (offline) whenever possible and when required for safety critical components.

6 Models@runtime

In order to execute reconfiguration decisions, the safe adaptation core needs to know (among other information) which component runs on which ECU and how the components are interconnected. It also needs to know how many resources are available on an ECU and the resource consumption of each component. A part of this information is available in the runtime model – a combination of a projection of the model that is used at design time and additional information that can only be computed at runtime.

The information required information can be classified in the following way. Platform information/capabilities are provided by a specific component representing the platform.

- Static information on component level including resource and timing requirements.
- Static information instance level: in principle same information as for the preceding bullet. But this information is attached to an instance of a component, e.g. the required RAM is information shared by all instances (i.e. on the component level), while a configured period length or priority is specific to a concrete instance.
- Dynamic information: information that can be computed at runtime. An example is the current resource consumption.

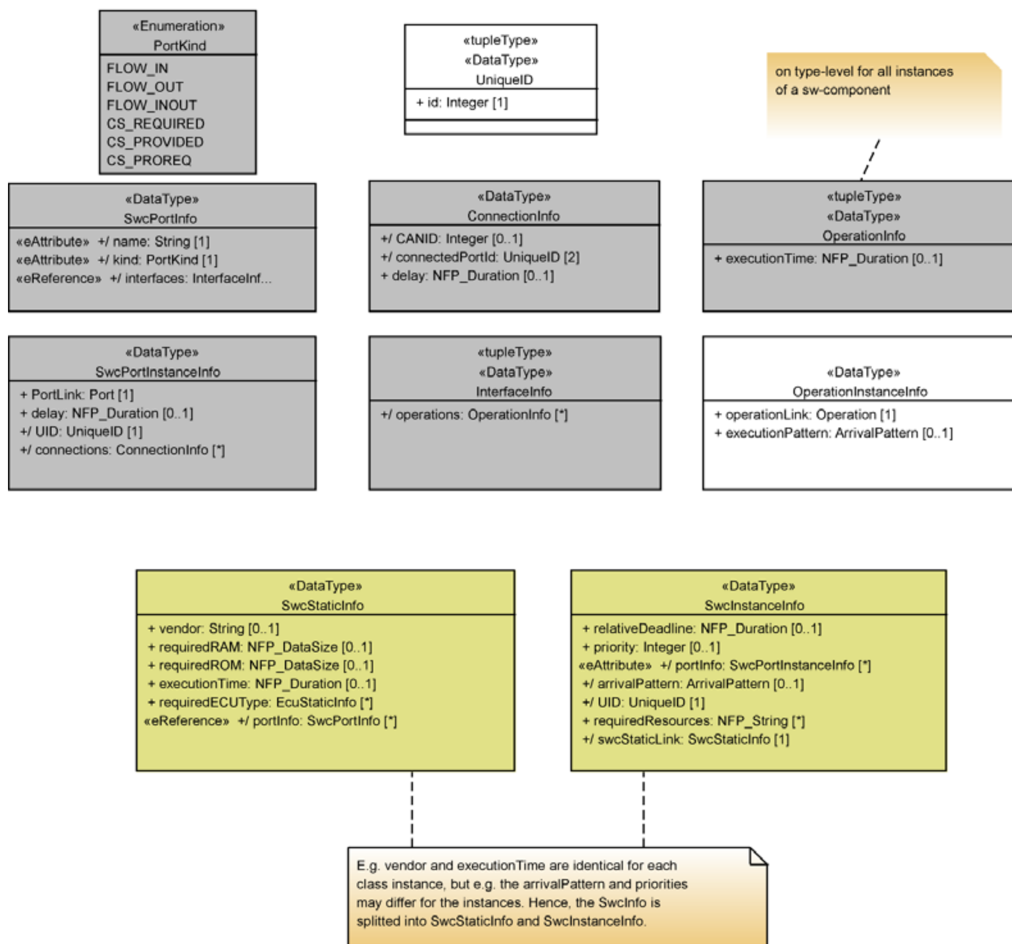

Figure 12: Self-X profile, static part

The static part of the information is modelled by means of the Self-X profile shown in For instance, the data type SwcStaticInfo captures additional information of software components regarding resource consumption, execution time and required ECU type (the ECUs on which the components can run, e.g. with respect to the processor architecture).

The information at the instance level is captured by the data type SwcInstanceInfo, the period length we mentioned before is captured by a MARTE arrival pattern (which also adds a classification whether the execution occurs periodically, aperiodically or sporadically).

The design of the profile was done in a specific way to assure the consistency between additional information in the design model and a model at runtime. The additional information is not directly captured by means of stereotype attributes. Instead, the stereotypes inherit from the data types shown in Figure 12. The rationale is that the data types can be considered as the meta-model elements of the runtime model. They contain only a subset of the information already in UML such as the name of a port. Note that this information is marked as derived (prefixed with a "/" in the figure): in the design model, the stereotype attribute is taken from the UML base element, in case of the runtime model it is stored.

An automatic transformation extracts information from the design model in a format that can directly be used for the instantiation of the runtime model.

It is expected that the profile will still evolve during the prototyping phase, for instance we may need to add attributes for making suitable reconfiguration decisions at runtime.

7 Validation of Safe Adaptive System Behaviour

There is a need for an approach that can be used to validate the adaptive behaviour to avoid the adaptation errors that can lead to an undesired system state while it is in operation [14]. This approach should take into account the detection of errors that can happen during the adaptive behaviour specification and need to be detected [15][16][17].

To validate the system adaptive behaviour we need:

- Local and global properties / Include local versus global concepts
- Specify the properties that need to be checked against the adaptive behaviour model (i.e. the errors that need to be detected) at design time
- How to evaluate the feasibility of the new allocation at runtime?
- Include example of properties (including safety properties)
- Include boundaries of properties will be specified by constraints?

The System Adaptive Behaviour Errors at design time

In large scale software systems where there are a large number of adaptations, the system adaptive behaviour is subject to errors such as inconsistency, redundancy, cycles, and incompleteness.

- Adaptation Behavior Inconsistency: The inconsistency means that the adaptation actions that need to be applied into the system contradict each other. The possible system adaptation actions are to add, remove, and replace a system element. The inconsistency between these actions can happen in the following situations. First, the required adaptation actions are to add and remove the same system element (Type1 error). Second, the required adaptation is to change (i.e. replace) the system element twice (Type2 error). For instance when there are two replacements actions of the same component in the adaptation request (e.g. replace component 1 with component 2 and replace component 1 by component 3).
- Adaptation Behaviour Redundancy: The redundancy appears when a rule is repeated, or one rule is a sub-part of another. For example, two rules have the same condition(s), and the adaptation action(s) of a rule is a part of the other rule adaptation action(s) (i.e. Type3 error). This error is detected by looking for an adaptation action that is repeated twice in the required adaptation actions.
- Adaptation Behaviour Cycles: In context-aware systems, the context model changes initiate a system adaptation (e.g. when the context model has the driver preferences entity active, the route planning algorithm one is selected). In addition, the functional system changes can lead to a context model adaptation (e.g. in response to the driver selection to use the route planning two, the context model is changed by activating the congestion information context entity). As such, the adaptation rules for changing the functional system in response to context model changes and vice versa should be written carefully to avoid the

cycles. A cycle happens when the adaptation rules evaluation leads to adaptation actions that make the same chain of rules firing to be performed again (i.e. Type4 error).

- Adaptation Behaviour Incompleteness: In large scale systems, there are a large number of adaptation behaviours. As a consequence, there is a possibility of missing adaptation behaviours (i.e. Type5 error). These missing behaviours are appeared when there is a context situation without having an adaptation action to it or the rule conditions cannot be evaluated to true (i.e. the rule cannot be fired). For example, an adaptation rule is based on an and-condition (e.g. A and B), but the condition A and B cannot be evaluated to true in the same time.

8 Summary

This deliverable shows how adaptive behaviour can be modelled. The modelling is based on existing approaches such as a rule based adaptation and proposes means to model the rules on the level of the UML/EAST-ADL system model. Design patterns are a complementary approach to specify adaptive behaviour. We distinguish between anticipated adaptations that are based of offline validated configurations with respect to resource and time requirements and not anticipated ones that are validated at runtime based on runtime model information. The former is important for safety certifications, the latter an additional robustness outside the certification scope and primarily interesting for less critical functionality.

9 Bibliography

[1] EAST-ADL association, EAST-ADL Specification V2.1.2, Nov. 28th, 2013, http://www.east-adl.info/Specification.html

[2] ESK et al, Concept for Enforcing Safe Adaptation during Runtime, SafeAdapt, Deliverable 3.1

[3] R. Laddaga, Guest Editor's Introduction: Creating Robust Software through Self-Adaptation, IEEE Intelligent Systems Journal, Volume 14 Issue 3, pages 26-29, May 1999,

[4] M. Salehie and L. Tahvildari, Self-adaptive software: Landscape and research challenges, ACM Transactions on Autonomous and Adaptive Systems (TAAS), Volume 4 Issue 2, May 2009

[5] F. Salewski, S. Kowalewski, *Hardware/Software Design Considerations for Automotive Embedded Systems*, IEEE transactions on industrial informatics, vol. 4, no. 3, August 2008

[6] C. Alexander. A Pattern Language: Towns, Buildings, Construction. Oxford University Press, New York, 1977.

[7] R. Damaševicius, G. Majauskas, and V. Štuikys. Application of design patterns for hardware design. In DAC ’03: Proceedings of the 40th annual Design Automation Conference, pages 48–53, New York, NY, USA, 2003. ACM.

[8] R. Damaševicius and V. Štuikys. Application of UML for hardware design based on design process model. In ASP-DAC ’04: Proceedings of the 2004 Asia and South Pacific Design Automation Conference, pages 244–249, Piscataway, NJ, USA, 2004. IEEE Press.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston, MA, USA, 1997.

[10] F. Rincon, F. Moya, J. Barba, and J. C. Lopez. Model reuse through hardware design patterns. In DATE ’05: Proceedings of the conference on Design, Automation and Test in Europe, pages 324–329, Washington, DC, USA, 2005. IEEE Computer Society.

[11] N. Yoshida. Design patterns applied to object-oriented SOC design. In 10th Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI 2001), Nara, Japan, Oct. 2001.

[12] Armoush, A., "Design Patterns for Safety-Critical Embedded Systems," PhD Thesis , 2010.

[13] Biehl, M., DeJiu, C., Törngren, M. (2010) Integrating safety analysis into the model-based development toolchain of automotive embedded systems. LCTES 2010. ACM, New York, pp. 125-132

[14] Hussein, Mahmoud, Han, Jun, Colman, Alan, Yu, Jian. IEEE; 2012. An approach to specifying and validating context-aware adaptive behaviours of software systems.

[15] J. Zhang and B. H. C. Cheng, "Model-based development of dynamically adaptive software," presented at the Proceedings of the 28th international conference on Software engineering, Shanghai, China, 2006.

[16] D. Sykes, et al., "From goals to components: a combined approach to self-management," presented at the Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems, Leipzig, Germany, 2008.

[17] Y. Zhao, et al., "Model Checking of Adaptive Programs with Mode-extended Linear Temporal Logic," in 2011 8th IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems (EASe), 2011, pp. 40-48.

[18] I. Jahnich, I. Podolski, and A. Rettberg, "Towards a middleware approach for a self-configurable automotive embedded system," Software Technologies for Embedded and Ubiquitous Systems, pp. 55–65, 2008.

[19] A. Rettberg, R. Anthony, D. Chen, I. Jahnich, G. de Boer, and C. Ekelin, "A dynamically reconfigurable automotive control system architecture," in Proceedings of the 17th IFAC World Congress, 2008, vol. 17, 2008.

[20] D. Chen, M. Törngren, M. Persson, L. Feng, T. Qureshi, "Towards Model-Based Engineering of Self-configuring Embedded Systems. In H. Giese, G. Karsai, E. Lee, B. Rumpe, and B. Schätz (eds.), Model-Based Engineering of Embedded Real-Time Systems, volume 6100 of Lecture Notes in Computer Science, chapter 17, pp. 345–353. Springer Berlin, Heidelberg, 2011.

[21] B. Morin, O. Barais, and J. Jézéquel, "K@RT: An Aspect-Oriented and Model-Oriented Framework for Dynamic Software Product Lines," in 3rd International Workshop on Models@ Runtime, at MoDELS, 2008.

[22] A. Radermacher, B. Hamid, M. Fredj, J.L. Profizi, "Process and tool support for design patterns with safety requirements", Proceedings of EuroPlop'2013, ACM 978-1-4503-3465-5/13/07, to appear