Kernel-based ARchitecture for safetY-critical cONtrol

KARYON

FP7-288195

D4.4 – Safety constraints and safety predicates

Work Package	WP4		
Due Date	M27	Submission Date	2014-03-31
Main Author(s)	Kenneth Östberg (SP) Rolf Johansson (SP)		
Contributors	Renato Librino (4SG), Jörg Kaiser (OVGU), Sebastian Zug (OVGU), Eric Vial (FFCUL), António Casimiro (FFCUL)		
Version	1.0	Status	Final
Dissemination Level	Public	Nature	Report
Keywords	Safety constraints; Safety requirements; Safety integrity level; Information quality		

Part of the Seventh

Framework Programme

Funded by the EC - DG INFSO

Version history

Rev	Date	Author	Comments
V0.1	2014-03-04	Kenneth Östberg (SP)	Copy from the initial report.
V0.2	2014-03-25	Kenneth Östberg	Update after internal review.
V0.3	2014-03-27	Kenneth Östberg Eric Vial (FFCUL) Antonio Casimiro (FFCUL)	More updates and a new §9 from FFCUL
V0.4	2014-03-28	Kenneth Östberg	Added one picture about co-operative safety.
V0.5	2014-03-31	Kenneth Östberg	Updated according to SP review process.
V1.0	2014-03-31	António Casimiro (FFCUL)	Final revision and delivery.

Glossary of Acronyms

ADL	Architecture description language
ASA	Aircraft safety assessment
ASIL	Automotive safety integrity level
CCA	Common cause analysis
DAL	Development assurance level
ETSI	European Telecommunications Standards Institute
FDAL	Function development assurance level
FHA	Functional hazard assessment
IDAL	Item development assurance level
ITS	Intelligent Transport System
IQ	Information quality
LDM	Local Dynamic Map
LoS	Level of service
PASA	Preliminary aircraft safety assessment
PSSA	Preliminary system safety assessment
QM	Quality management
SEooC	Safety element out of context
SIL	Safety integrity level
SOP	State of practice
SOTA	State of the art
SSA	System safety assessment

Executive Summary

This delivery is the report from the work performed in task 4.1 on safety constraints and safety predicates. The focus is on defining the problems in this area needed to be solved to achieve the overall KARYON goals.

There are two main strategies to achieve a safe system. One is that the safety kernel implements redundancy in such a way that the limited level of integrity of the complex application components is still sufficient (this is what is referred to as ASIL decomposition in the ISO 26262 standard). The other is that the safety kernel enforces the vehicle to a situation where the requested level of integrity is lower (this is what we refer to as degraded functionality). These two strategies can be combined and this report discusses both approaches and then proposes a way to unify them inside the KARYON architecture.

In order to guarantee safety, we both need to express all the applicable safety requirements from design time, and all the evidence showing fulfilment of the requirements in run time. In order to show that all requirements are fulfilled, the evidence and the safety requirements need to be expressed in a comparable way, and with clear sufficiently clear semantics. An abstract type system is proposed which is compliant with the theorem prover PVS. PVS allows for predicates and constraints on its types to be able to form very precise properties and requirements. We are thus able to have a formal notation to support our descriptions of safety requirements and our analysis of safety cases.

To identify our need on safety constraints and predicates, we analyse both applicable standards, and implications from the KARYON use cases and architecture pattern.

From existing safety standards and from our understanding of the implications of introducing cooperative functions, we draw the following general conclusions for Safety Requirements:

- Safety Requirements shall be possible to express on different levels in a reference life cycle
- Safety Requirements shall be possible to allocate onto elements of an architecture on the different levels of a reference life cycle
- Safety Requirements shall have a Safety Integrity Level attribute
- Safety Requirements shall be possible to break down from one level in a reference life cycle to a level below
- The Safety Integrity Level attribute is inherited when breaking down a Safety Requirement, unless redundancy is applied and a lowering of Safety Integrity Level may occur.

How to estimate level of safety integrity in run-time is a field that is not foreseen in the current safety standards. Hence, this is a new area of research that hopefully can give input to future revisions of safety standards where cooperative functions also are addressed. This is a major topic for research in the KARYON project.

So far, we conclude that there is a need to identify attributes of Information Quality that can serve as a source for estimating provided Safety Constraints in a system. Estimating Information Quality may be done on as well information available during design time as on information available only during run time, as is required according to the basic KARYON concepts. An example of a design time source is documented quality of a particular sensor. An example of a run time source is to what extent redundant sensors show the same value.

Some identified basic requirements on Information Quality attributes are the following:

- Information Quality is defined with respect to a certain failure to avoid, i.e. applicable for a certain safety requirement or safety constraint.

- Information Quality shall be possible to aggregate together with the aggregation of the data for which it is telling the quality.
- The order in which aggregation is performed shall not be of importance.
- Rules applicable for aggregation of Information Quality shall be consistent with rules for applying redundancy when Information Quality is transformed to safety Integrity.

One important conclusion from this report is that somewhere in the architecture, there shall be elements capable of aggregating values and corresponding attributes of information quality for each failure having a safety requirement. The safety manager shall be able to compare this evidence for achieved level of integrity with the required level of integrity for each applicable safety requirement. This implies that evidence and requirements both follow a common pattern of a safety constraint.

Table of Contents

List of Figures

1. Introduction

1.1 Purpose & Scope

KARYON focuses on the predictable and safe coordination of smart vehicles that autonomously cooperate and interact in an open and inherently uncertain environment. The method to achieve this is to identify a hybrid architectural pattern in which a safety manager in run-time has the responsibility to guarantee safety. The purpose of this document is to identify what is needed for formulating the problem, and also valid solutions, of the tasks of the safety manager.

In general, the task of the safety manager is to "guarantee safety". To identify what safety means, we can for today's vehicles look into what is prescribed in standards. For the scope of KARYON we also need to conclude what future scenarios of cooperative autonomous vehicles may imply.

The scope of this document is to formalize the problem of "guarantee safety" in such a way that it can be used for specifying and validating a safety manager and hence a complete KARYON architecture. This includes identifying the formalism needed for the safety requirements defining what constitutes the definition of safety for different parts of a KARYON design. Furthermore it also includes identifying the formalism needed for collecting the run-time evidence that the safety manager uses for assessing safety.

This is the first report on this topic and its purpose is more to define the problems than finding all the solutions

1.2 Methodology

The work presented in this deliverable has been performed in very tight cooperation with several other tasks. Many of the research questions in KARYON project are tightly connected to one each other, and because of the small consortium size with only seven partners, it is a practical method to let several tasks go on more or less together.

For this task about safety constraints and safety predicates, there has been especially tight connections to the following other tasks so far:

Task 1.1 on safety constraints and safety predicates, where the identification of the six last requirements (R4.2.80- R.4.2.130) of the generic requirements on a KARYON architecture are a direct consequence of a deeper analysis of the needs from Safety constraints and safety predicates.

Task 1.2 on assessment of generic requirements, where the identification of general requirements from the safety standards is essential for finding the unified view on safety requirements.

Task 2.1 on hybrid system architecture, where the essential part here is the identification of the role of the safety manager and what kind of evidence that are to be collected and aggregated in run-time.

Task 2.2 on failure modes and semantics is perhaps the one where some large research questions are currently under common investigation. In particular, this is in the field of defining attributes for "validity" or "quality" of information in an E/E architecture.

Task 4.2 on safety kernel design specification, where the work on the hybrid system architecture is further detailed. As for that task, it is especially the question of what evidence to collect in run-time, and how that may be compared to design-time defined safety requirements and that is investigated together.

Besides the tight cooperation with other work tasks in KARYON, the state-of-the-art (SOTA) in the research field, and the state-of-practice (SOP) in the industrial community, has been input.

For the SOP it is obvious that the safety standards are important, and the method is involving analyses of these to identify common pattern and also to identify patterns valid in a future scenario with cooperative autonomous vehicles. For SOP the references [1] - [5] have been of particular interest. The SOTA includes related research in the field of safety requirements such as the ones in the list of references [6] - [11] Some results have already been subject for publication: [12] [13]
.

1.3 Structure of the document

In Section 2.1 requirements on safety requirements from the safety standards in the domains are analysed. Together with implications from adding cooperative scenarios this leads to general requirements on safety requirements. The chapter ends with a list of five properties that need to be well-defined with clear semantics in order to reason about safety requirements, as is the case for the safety manager in the KARYON architecture.

While Section 2.1 has the top-down perspective of breaking down safety requirements, Section 2.2 has the bottom-up perspective of collecting and aggregating evidence needed for the argumentation whether all safety requirements are met. This is done by describing the problem of collecting some safety evidence in run-time, especially the problem of aggregating this and transform it to something comparable to how the safety requirements are expressed.

In Section 2.3 is discussed how a common pattern for expressing safety requirements and safety constraints is related to expressing failure models with well-defined semantics.

Chapter 2 is concluded in section 2.4 where is presented a general structure for modelling safety requirements and safety constraints with enough precision to be used for a safety manager of a KARYON architecture.

In Chapter 3 the bottom-up problem of collecting run-time evidence and transforming this to something comparable to the safety requirements are further elaborated. This is a major field of on-going research and the content of this chapter is mostly focused on defining the problem and identifying necessary criteria for valid solutions.

Chapter 4 summarizes how the results in Chapters 2 and 3 affect the safety manager.

Chapter 5 discusses information systems in general and then argues that co-operative systems are by nature very complex and that there is a need for a more formal information model for systematically specifying and managing the concerns of such system. Such an information-model would allow a systematic and formal definition of context-awareness data in each individual system participating in the system. To support such an information model there is a need for an abstract type system that allow us to specify safety requirements, failure modes and fault models together with Safety Integrity Levels.

Chapter 6 goes further and gives a more precise meaning of syntax and semantics in relation to the information model and the need for dedicated metrics in order to be able to measure any quality metrics in the system.

In Chapter 7, co-operative scenarios are discussed and how to perform the design time analysis so the safety assessments later on can be brought into run time and monitored by the safety manager. The KARYON architecture is instantiated into an concrete example by looking at an ITS-Station in the ITS context as being specified by ETSI

In Chapter 8 we are developing a type system that is compatible with many existing notations and programming languages of today and has especially been designed with the theorem prover PVS in mind. The tight connection with PVS allows us to use very precise types to specify safety requirements at design time and assure that we have well defined semantics for that type system. It will also allow us to perform formal verification of safety requirements which are attributed

with ASIL D. Parts of the KARYON architecture will certainly achieve that integrity level. The type system is designed to be used from start at the co-operative design and analyse level right down to the run time level to be used for specifying run time safety rules for the safety manager to supervise. There is also a discussion how the type system can be used as the foundation of the domain specific language EAST-ADL which is an architectural design language specifically designed to handle specification of safety requirements on vehicle level and with a tight integration to ISO26262. There are efforts to extend EAST-ADL into the co-operative domain and the proposed abstract type system will support such initiative.

Chapter 9 is giving a concrete example of the structures for the safety rules that will be stored in the rule database for the demonstrators in WP 5. The used notation is XML which is supported by the type system in Chapter 8. The rules which may be produced by a tool chain compatible with EAST-ADL can thus easily be exported in XML format to suite the KARYON architecture.

Finally, in Chapter 10 the report is briefly summarized.

2. Expressing Safety Requirements and Constraints

The task of the safety manager in the KARYON architecture is to compare required safety with achieved safety, and during run-time make sure that the system always is safe. In order to perform this comparison we need to specify how to express both "required safety" (Safety Requirement) and "achieved safety" (Guaranteed Safety or Safety Constraint). The assumption is that the comparison can be done if they are expressed in a similar way.

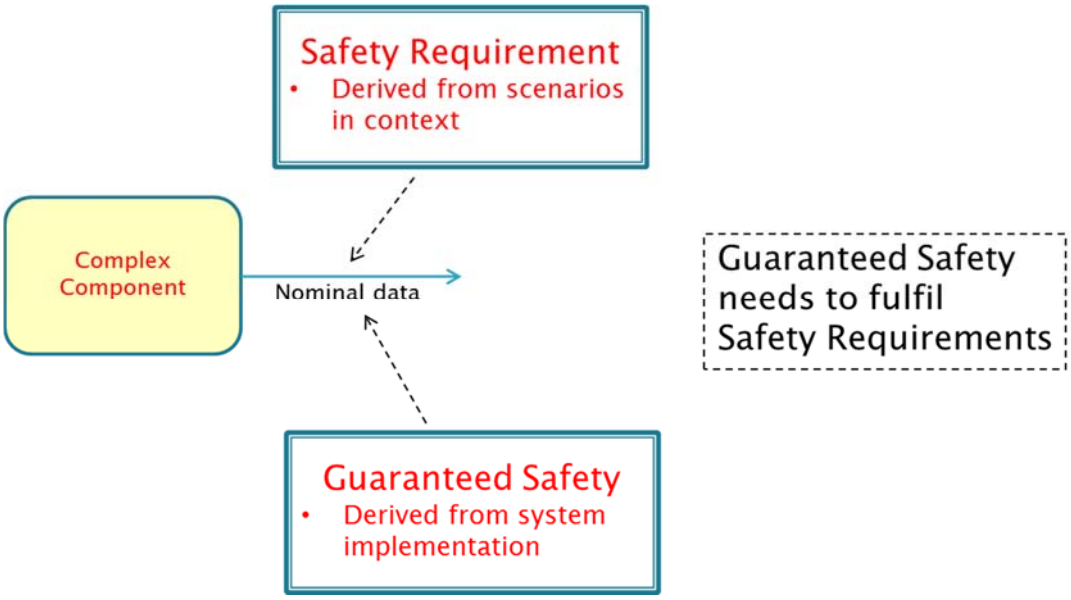

Figure 1: Challenge from Safety Standards: Prove that all safety requirements are fulfilled by the implementation.

In [6] is described a general modelling pattern for non-functional requirements/properties, like safety. Here the general pattern is called safety constraint. The safety constraint can then be used for expressing as well safety requirements as "achieved safety". In the domain of architectural description languages (ADL), the safety constraint is a meta class in the domain model and safety requirement is a role name. In section 2.1 below is identified what is needed for expressing safety requirements applicable to the safety manager. In section 2.2 is then the other role of the safety constraint elaborated with respect the needs of the safety manager. Both these perspectives are then concluded in section 2.4, where is summarized what is needed for a safety constraint covering both the different roles.

In this report the focus of relation between safety requirements is in the vertical direction (pattern for break-down and for aggregation) and not in the horizontal direction (safety contracts). The latter question is (currently) considered as out of scope but can be studied in for example [10] .

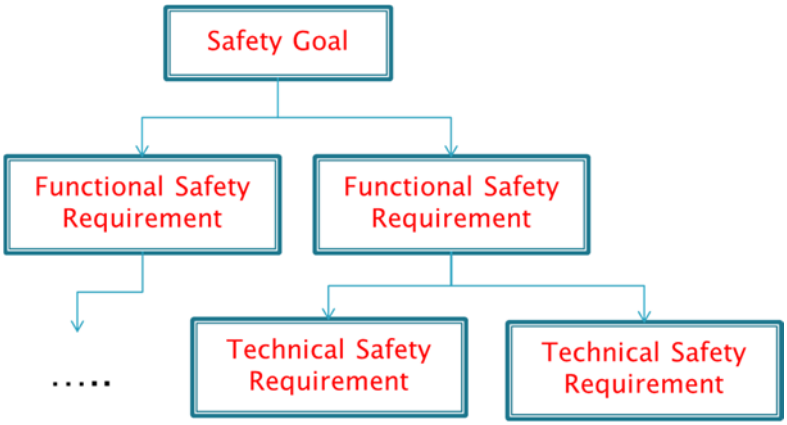

Figure 2: Automotive Example of Vertical Relation between Safety Requirements.

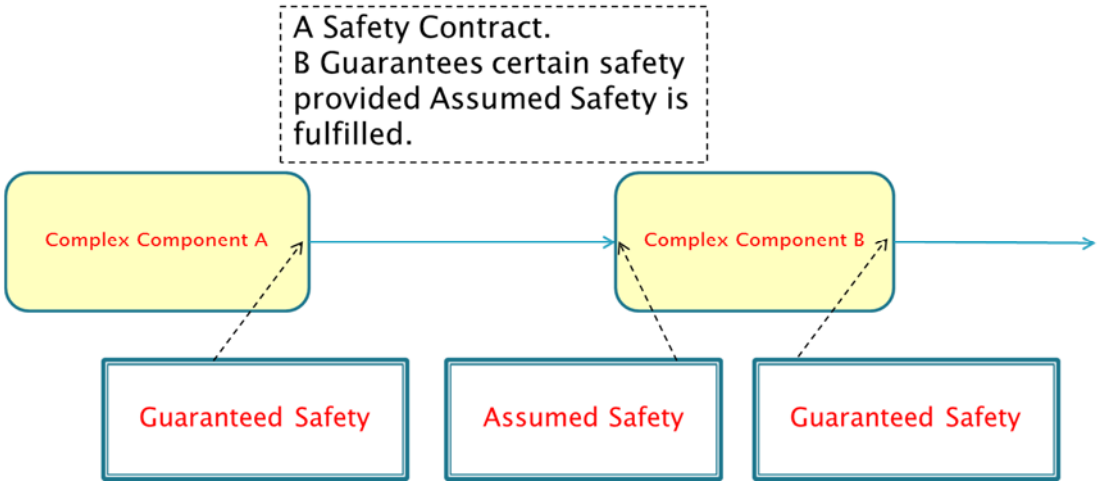

Figure 3: General Example of Horizontal Relation between Safety Requirements.

2.1 Needs from Safety Standards

In both the automotive and in the aeronautic domain there are safety standards that are built around some common concepts. In the following sections these are elaborated for both of the domains, but from a very general perspective the following common concepts can be identified:

- A reference life cycle depicting a V-model with hierarchical levels
- Safety requirements in each phase (on each level)
- Safety requirements from one phase broken down to the level below
- Safety requirements allocated on architectural elements on each level
- A safety integrity attribute on each safety requirement
- Rules for lowering required level of safety integrity when applying redundancy

These common concepts above are elaborated below for the two domains. Furthermore it discussed what will be needed in future revisions of the safety standards when also cooperative functions are explicitly addressed.

2.1.1 Aeronautic domain

In the combination of [1] [4] and [5] all the above general concepts can be identified.

In [1] the general reference life cycle is called Safety Assessment Process Model and the nominal development activities start with Aircraft Function Development, then follows Allocation of Aircraft Functions to Systems, Development of System Architecture, Allocation of System Requirements to Items, System Implementation, and System/Aircraft Level Integration & Verification. The associated safety life cycle phases are Aircraft Level Functional Hazard Assessment (FHA), Preliminary Aircraft Safety Assessment (PASA), Common Cause Analyses (CCAs), System-Level FHA, Preliminary System Safety Assessment (PSSA), System Safety Assessment (SSA) and Aircraft Safety Assessment (ASA). [1] then further references for the lower levels to [4] for the software development lifecycle and to [5] for the electronic hardware development lifecycle. How to perform FHA is further described in [3] .

Depending on phase, the safety requirements are denoted Safety Requirements (allocated on system architecture) or Item Requirements (allocated on Items).

The general concept of safety integrity levels is defined as Development Assurance Level (DAL) in the aeronautic domain. Furthermore, DAL can be specialized as FDAL (Development Assurance Levels for aircraft functions and systems) or IDAL (Development Assurance Levels for electronic hardware and software items). Like the general concept of Safety Integrity Level (SIL), DAL is also considered as a measure to what extent safe behaviour can be guaranteed. In [1] this is expressed as "The Development Assurance Level is the measure of rigor applied to the development process to limit, to a level acceptable for safety, the likelihood of Errors occurring during the development process of aircraft/system functions and items that have an adverse safety effect if they are exposed in service". The highest level of safety integrity is called DAL A, and the lowest DAL E (outside scope of safety).

In [1] it is also described how redundancy may lower the required level of integrity. On the aircraft level this is described as "The members within a given Functional Failure Sets may be assigned their own FDALs which may be lower than the top-level Failure Condition severity classification provided the functional independence attribute is satisfied." This is according to a general pattern for lowering requirements on safety integrity level, which is considered ok when applying redundancy, given that the redundancy is implemented with independency. The specific rules are also listed in [1]

2.1.2 Automotive domain

In [2] all the above general concepts can be identified.

In [2] the general reference life cycle is called Safety lifecycle. It starts with a Concept Phase, then follows Product development at the system level (first part: 26262-4: 5-7), Product development at the hardware level (26262-5), Product development at the software level (26262-6), Product development at the system level (second part, 26262-4: 8-11), and Production and operation (26262-7).

Depending on phase, the safety requirements are denoted Functional Safety Requirements (allocated on conceptual architecture in the concept phase), Technical Safety Requirements (allocated on system architecture in the phase of Product development at the system level), Hardware Safety Requirement (allocated to hardware architecture in the phase of Product development at the hardware level), and Software Safety Requirement (allocated to software architecture in the phase of Product development at the hardware level).

The general concept of safety integrity levels is defined as Automotive Safety Integrity Level (ASIL) in the automotive domain. Like the general concept of Safety Integrity Level (SIL), ASIL is also considered as a measure to what extent unsafe behaviour can be guaranteed. In [2] this is expressed as denoting "necessary requirements of ISO 26262 and safety measures to apply for

avoiding an unreasonable residual risk". The highest level of safety integrity is called ASIL D, and the lowest QM (outside scope of safety).

In part 9 of [2] it is also described how redundancy may lower the required level of integrity. This is described as "During the allocation process, benefit can be obtained from architectural decisions including the existence of sufficiently independent architectural elements. This offers the opportunity: to implement safety requirements redundantly by these independent architectural elements, and to assign a potentially lower ASIL to these decomposed safety requirements. If the architectural elements are not sufficiently independent, then the redundant requirements and the architectural elements inherit the initial ASIL". This is according to a general pattern for lowering requirements on safety integrity level, which is considered ok when applying redundancy, given that the redundancy is implemented with independency. The specific rules are also listed in part 9 of [2] .

2.1.3 Future needs for Cooperative Functions

As automotive is concerned, the present safety standard ISO 26262 is applicable to any on-board electric/electronic system, independently from the applications. Therefore, in principle, also the cooperative driving functionalities and the vehicle systems that implement them shall be dealt with according to the standard.

The general approach required by the standard is to consider the functional safety of the "item", which is the system that is under development. The item, according to the definition given by the standard, relates sensors, processing units and actuators. Therefore, following the standard, the development of a cooperative driving system, in general, will be conducted only taking into account a vehicle-centric point of view, while the operation of the other cooperating vehicles should be considered only as an operational situation or "external measures" that could intervene to mitigate the effects of the hazards in the case of failure.

The result of the development activity, if performed meeting the requirements of the standard, will be functional safety of the item. Although this result can be acceptable, it does not take completely into account that the considered functionalities are cooperative and should be analysed from a more general point of view including the interactions with the other vehicles. The concept of level-of-service (as pursued in KARYON) is more oriented to a system approach implicitly based not only on information sharing, but also on operational strategies agreed among the participating vehicles.

Therefore, it is quite evident that, at present, cooperative driving is not well covered by the standard. In fact this is a point under discussion in the ISO 26262 working group to evaluate the need or the opportunity to address that kind of functionalities in a more specific way.

From the aeronautic perspective all the above apply as also that domain has a vehicle centric view in current standards.

2.1.4 General Conclusions

From existing safety standards and from our understanding of the implications of introducing cooperative function, as elaborated above, we draw the following general conclusions for Safety Requirements.

- Safety Requirements shall be possible to express on different levels in a reference life cycle
- Safety Requirements shall be possible to allocate onto elements of an architecture on the different levels of a reference life cycle
- Safety Requirements shall have a Safety Integrity Level attribute

- Safety Requirements shall be possible to break down from one level in a reference life cycle to a level below
- The Safety Integrity Level attribute is inherited when braking down a Safety Requirement, unless redundancy is applied and a lowering of Safety Integrity Level may occur.

2.2 Aggregation of Safety Constraints

In the reference life cycle of the safety standards, a general pattern is that on the left leg of the V the Safety Requirements are broken down, and on the right leg the Safety Requirements are verified on more and more integrated levels.

In the development of such complex products like aeroplanes and cars, there is a large amount of suppliers and sub-suppliers involved, each having their own innovation and product development. In practice, this means that the ideal reference life cycles described in the standards do not reflect a time line in the real world. Sub-systems and/or components may be specified and developed before their overlaying system is defined. In a similar way as the Safety Requirements are used to communicate from customer to supplier in a supply chain, we can formulate assumed Safety Requirements (or Safety Constraints) as a kind of data sheet to communicate from supplier to customer about something that is ready for integration into new possible contexts. When integrating a system on the right leg of the V, and performing safety assessment activities for each step upwards, we then need to aggregate the safety constraints already assessed, for each of the parts to be aggregated.

This approach is in line with ISO 26262, which includes the possibility to develop components in compliance with the safety lifecycle even if those components are not directly devoted to a specific vehicle, i.e. they are not part of an item under development.

The development of such components is addressed by the standard with the concept of the elements SEooC (Safety Element out of Context). To follow the relevant part of the safety lifecycle, it is fundamental to make assumptions on the intended application and on the item configuration, in order to specify the safety requirements that are assumed to be met at item level and those that will be allocated to the SEooC. In the case of KARYON, the safety requirements above mentioned as a kind of data sheet of the Safety Kernel, have to be complemented with the requirements that will be specified at item level, which represent the expected behaviour (in terms of functional safety) of the item. Both sets of requirements shall be assessed during the development of a specific vehicle system to verify that the assumptions made are correct and to eventually define the proper modifications of the SEooC (in our case, the Safety Kernel) or of the item to ensure the compliance with the requirements specified to develop the specific application.

This leads to a conclusion that we require a general common way to formulate both Safety Requirements that can be allocated and broken down on the left side of the V, and Safety Constraints that can be aggregated and assessed on the right side of the V. In the same way as the redundancy pattern prescribes how Safety Integrity Levels can be lowered when breaking down safety Requirements, the same rules guides how to higher the Safety Integrity Level when applying redundancy to aggregation of Safety Constraints.

The possibility of aggregating safety constraints is even more important in a KARYON architecture. The reason for this is that here, part of the safety assessment is to be done in run time. This implies that it shall be possible for the Safety Manager to aggregate during run time varying amount of redundancy of the Safety Constraints of the available architecture elements, and compare this to the required Safety Requirements for different levels of service (LoS).

2.3 Relation to Modelling of Faults and Failures

In all the safety standards there is a direct relation between safety requirements and the failures that are required not to happen. Depending on the level in the reference life cycle, the failures are expressed differently.

On the vehicle (aircraft/concept) level the failures are the hazards for which the risk assessment is done implying the Safety Integrity Level (DAL/ASIL) used in the break down to lower levels of safety requirements.

On lower levels, the failures are those that can happen to the architectural elements to which the safety requirements are allocated. This means that a safety requirement says what shall not happen to that element (failure to avoid), and how sure it shall be that this does not happen (Safety Integrity Level).

2.4 A General Pattern for Modelling Safety Constraints

We identify a general pattern to model safety constraints and safety requirements that can be used in both the application domains, also including the special needs in the KARYON architecture. The information to be modelled is the following:

- Failure to avoid
- Safety Integrity Level telling how sure it is that the failure is avoided
- Reference to architecture element for which the failure is applicable

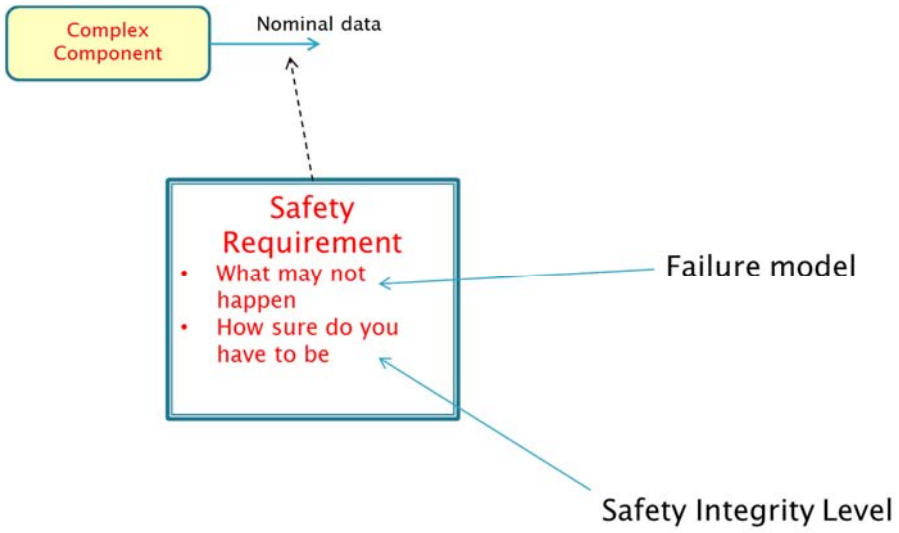

Figure 4: Requirements on a Safety Constraint.

Extending an Architectural Description Language (ADL) with such safety requirements could look like the following class diagram depicting a domain model.

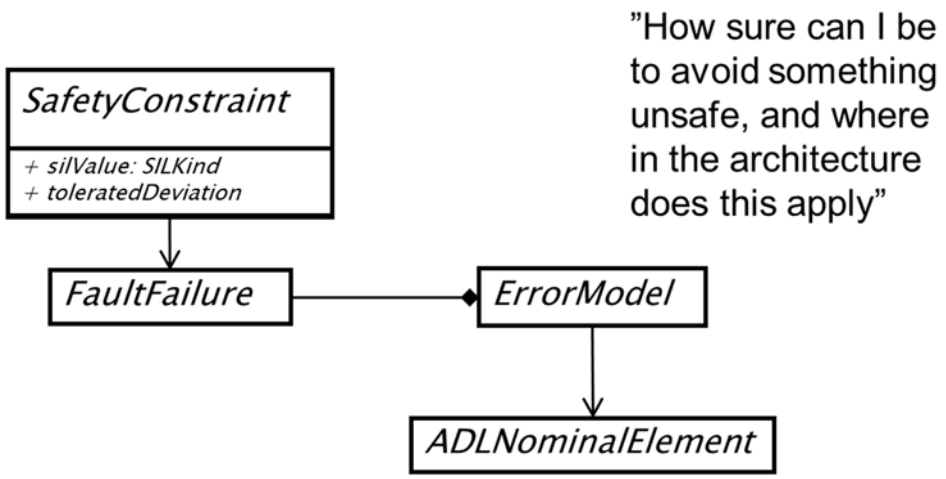

Figure 5: General Domain Model Picture.

The conclusion is that in order to get more precise semantics of safety requirements and safety constraints, we need precise semantics of all the three: failure, level of safety integrity, and architectural element being the candidate to fail.

The first of these three topics is about failure modelling. In this preliminary phase, is identified that precise failure models are critical for getting clear safety requirement semantics and some preliminary results are shown in D2.2. However, this work needs to go further in later version of the KARYON project deliverables in order to cover all the needs from the KARYON concept.

A major research area yet to be investigated is how to derive information needed to conclude on available (aggregated) level of safety integrity. This problem is crucial for the safety manager as assumed in the KARYON architecture in [D2.1]. Estimation of safety integrity in run time is a new research field, and the question to be investigated is presented in the next section.

3. Estimating Available Safety Integrity

3.1 The Problem of Transformation Rules

According to the general architecture pattern [D2.1], the safety manager shall at run time be able to estimate available levels of safety integrities, to determine for what service levels the safety requirements are met. The safety requirements to be checked, are all those allocated on all architectural elements. These requirements are defined at design time and are expressed according to the general pattern as expressed in the previous paragraph.

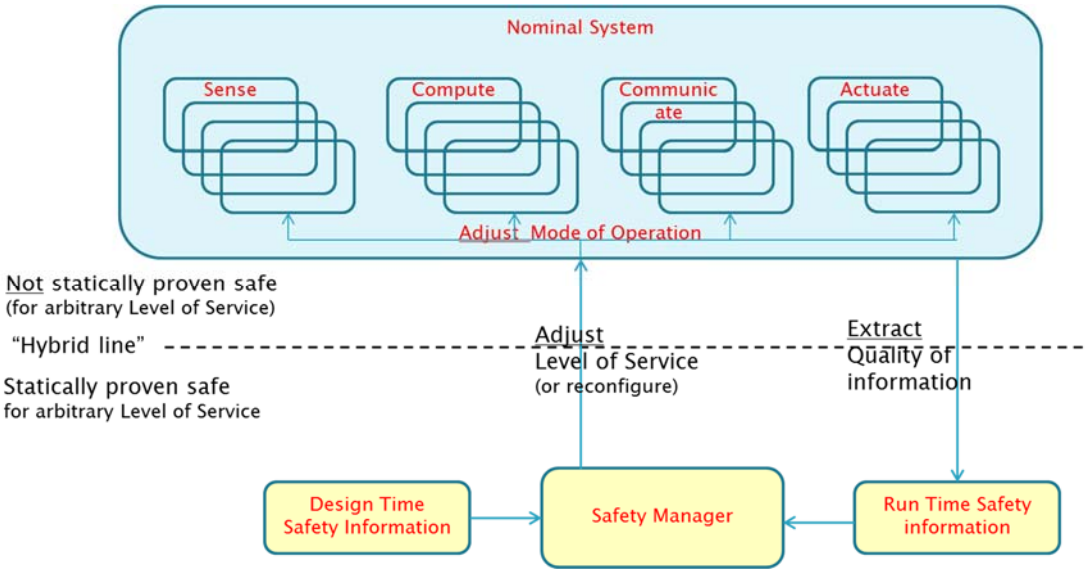

Figure 6: Hybrid Architecture.

The task in run time is to collect all kinds of available data that can be used as evidence to calculate the available Safety Integrity Level with respect to each applicable Safety Requirement.

There are essentially two ways to estimate the level of safety integrity. The first is to apply redundancy rules when redundancy is present. This is a typical KARYON situation. When several vehicles cooperate, they typically share data with one another that may serve as a source of redundancy. Depending on the current quality of sensor data and of communication links, the amount of redundancy may vary during run-time. This implies that the aggregated level of safety integrity also may vary, because the amount of redundancy defines how much the level of safety integrity may get higher in an aggregation.

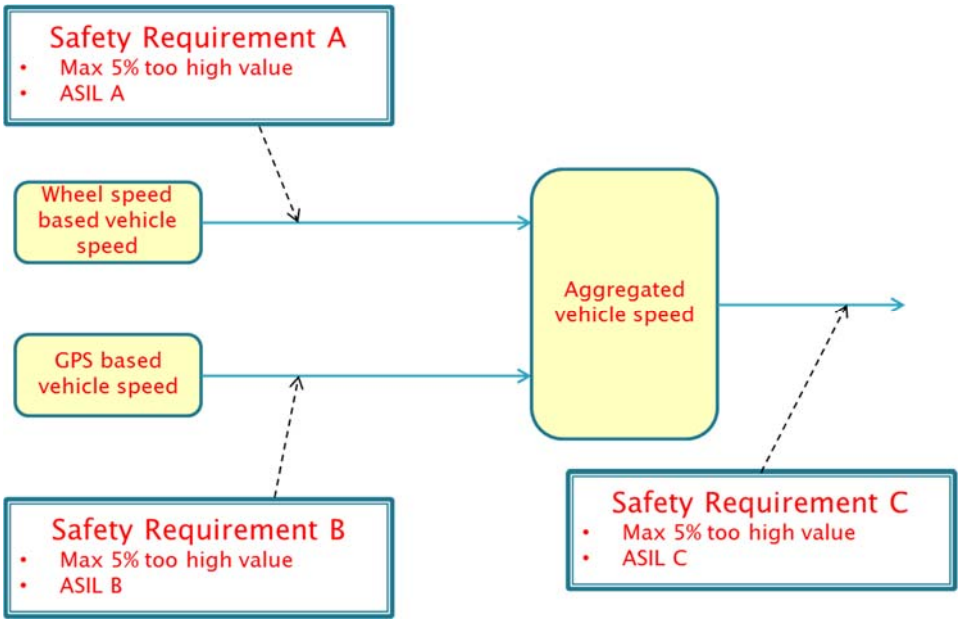

Figure 7: Automotive Example of Aggregation of Redundant Safety Constraints.

It is evident that the safety integrity cannot be directly measured at run time. Instead other measurable quality attributes have to be identified, that then can be transformed to safety integrity. The problem of such quality attributes and transformation rules is to a certain extent formulated in the following sub-section. The solutions to these problems are yet to be found.

These two ways for estimating available safety integrity are in the D1.1 formulated as a need for possibilities to estimate the amount of information, and to estimate the quality of information, respectively.

3.2 Estimating Information Quality

Estimating information quality is a major topic for research in the KARYON project. In this final version, there are still lots of research needed before any substantial results can be presented. Some preliminary results are found in D2.2. This will be further elaborated in later deliverables of the project.

Some basic requirements from the safety requirement perspective are the following:

- Information Quality is defined with respect to a certain failure to avoid, i.e. applicable for a certain safety requirement or safety constraint.
- Information Quality shall be possible to aggregate together with the aggregation of the data for which it is telling the quality.
- The order in which aggregation is performed shall not be of importance.
- Rules applicable for aggregation of Information Quality shall be consistent with rules for applying redundancy when Information Quality is transformed to safety Integrity.

The first rule says that we can use error detection mechanisms to estimate Information Quality, given that the errors detected corresponds to the failures of interest, and given that the efficiency of the mechanisms can be estimated. If the error detection mechanism is not perfect, the proportion of false negative results is of interest for the safety argumentation (not the proportion of false positive). The false negative results from the mechanism tell how often this critical failure

may pass undetected through, and this is the evidence we need to match to the required level of safety integrity for the associated safety requirement. A false positive result only says that we act more conservatively than needed, because we then act as if there are failures even if that is not the case. This may in the KARYON architecture lead to a lower level of service than needed, but it is not in conflict with any safety requirement.

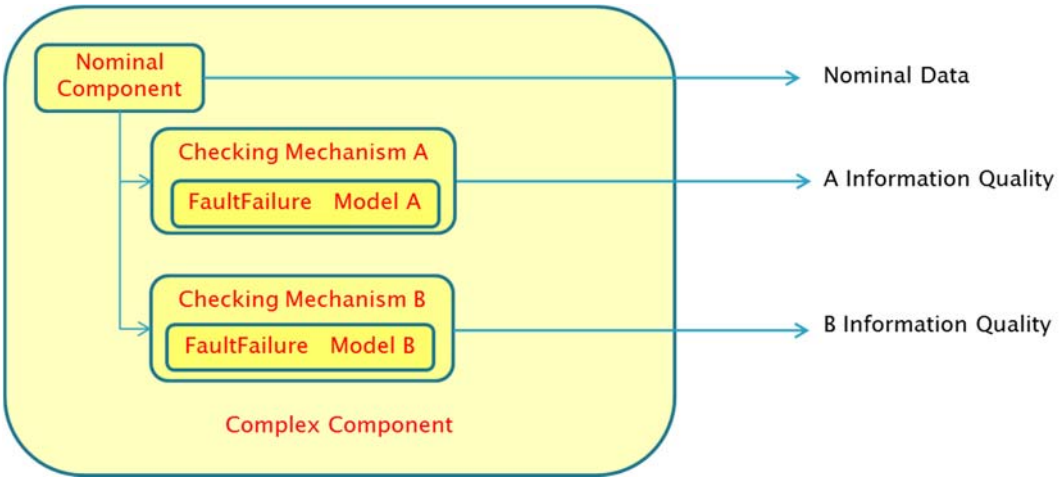

Figure 8: Estimating Information Quality by Detection Mechanisms.

The second rule is because we have highly dynamic scenarios measuring Information Quality in a system of systems. We do not know in advance if we may get yet another redundant source of information because of a new vehicle joining the cooperation. This implies that we always have to be prepared to aggregate an already aggregated value of Information Quality to a new estimation of Information Quality.

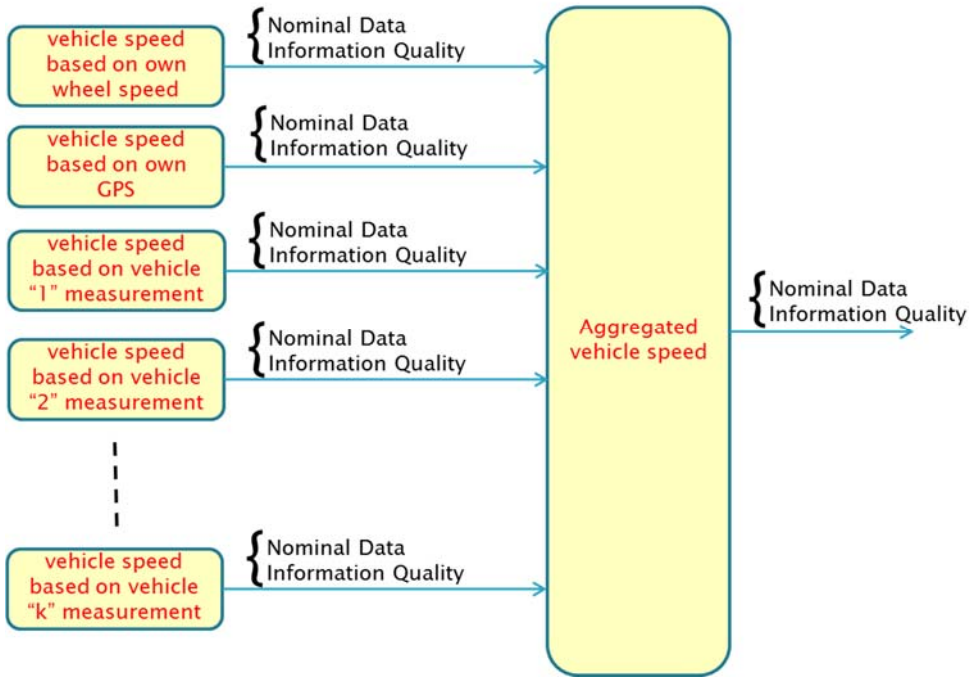

Figure 9: Aggregation of Data with Associated Information Quality Attribute.

Another implication of the second rule is that that aggregation of Information Quality is not only dependent on the Information Quality of the sources for aggregation; it is also dependent on to

which degree the data itself is consistent. The underlying assumption for aggregation is that the sources to aggregate are redundant and sufficiently independent. But if they differ between one another, it means that there is a lack of redundancy, which in turn implies lower aggregated Information Quality than if there was perfect redundancy.

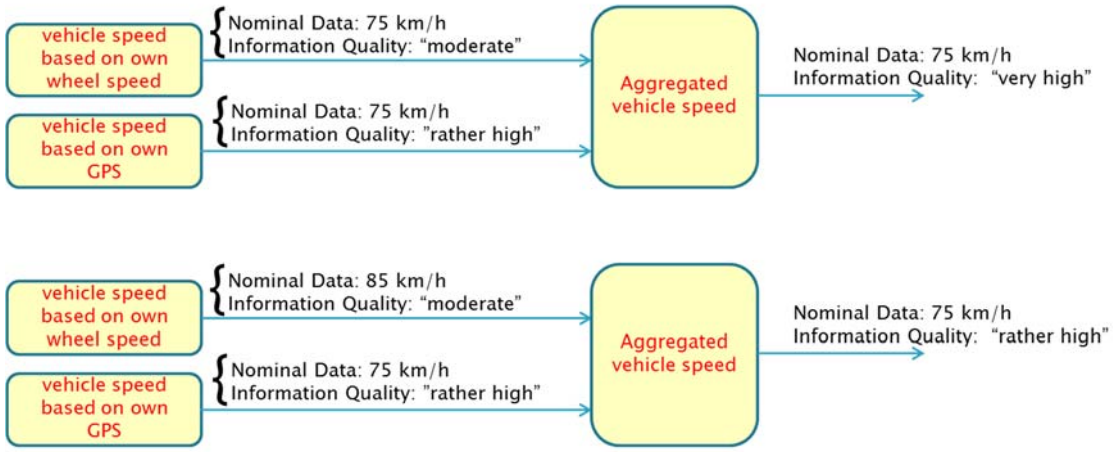

Figure 10: Example Showing Aggregated Information Quality Depending on Consistency between Aggregated Data.

An interesting question in this area is the following: When is it better not to get a sensor value at all and when it is better to get a redundant value with low Information Quality? A hypothesis is that adding more values should never cause the total Information Quality to become lower, even if the added values have a low validity compared to what is known before.

The third rules says that for three vehicles, each providing redundant information with a certain Information Quality, the resulting estimation of Information Quality shall be the same regardless if first the values of vehicles A and B are aggregated and finally C, or if first B and C are aggregated and then A. This can also be expressed as the Information Quality property shall be independent of the order of the aggregation operator.

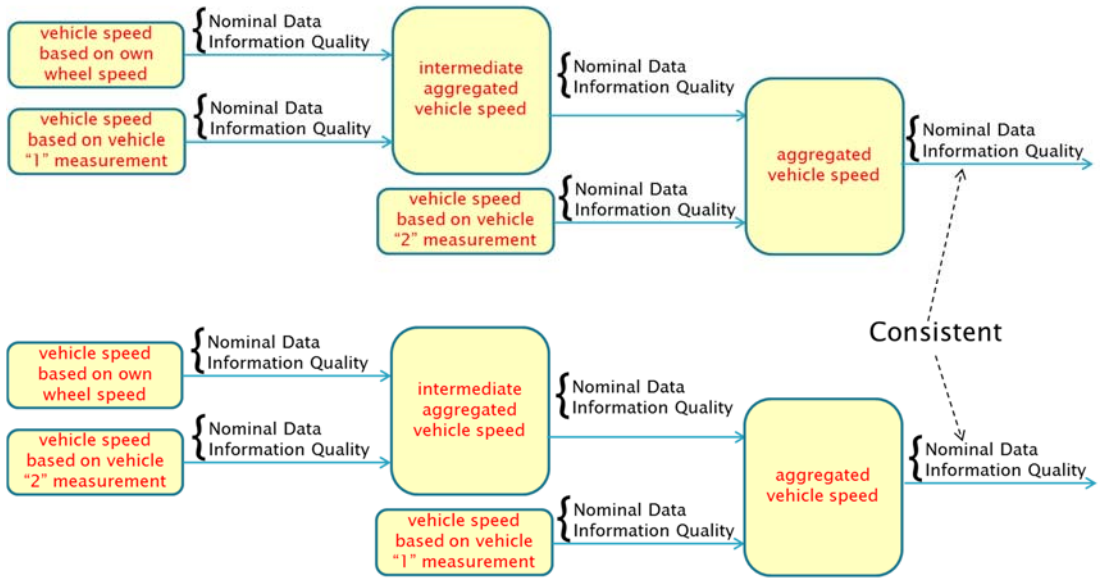

Figure 11: Order of Aggregation not Significant for Value of Aggregated Information Quality Attribute.

The fourth rule says that on any level of aggregation there might be an applicable safety requirement needed to be checked against the required level of safety integrity. To achieve

consistency, the effect of applying the redundancy rules shall be the same independently if they are applied on the Information Quality attribute or on the Safety Integrity Level attribute.

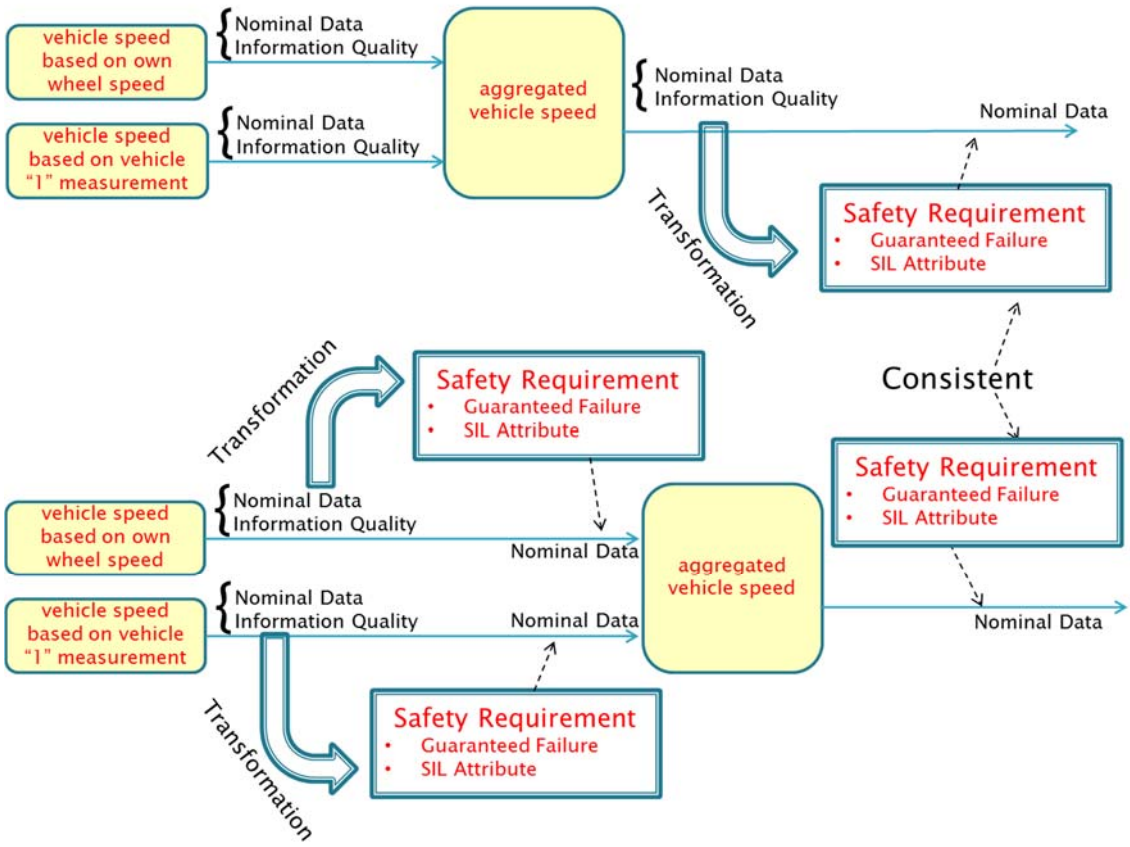

Figure 12: Rules for Aggregation of Information Quality Consistent with Rules for Aggregation of Safety Integrity.

A very interesting question concerning the rules for aggregation of redundant sources of information with associated quality attributes is if it is always "better" to get another redundant source of information even if that value deviates a lot from the other sources. In other words, is it always the case that adding another source of redundant information will never decrease the aggregated Information Quality attribute, regardless of how much that value deviates from the other and how low or high its Information Quality attribute is? A hypothesis is that this is true. The key for this to hold is that the design-time rules for aggregation must anticipate what pairs of value consistency and Information Quality attribute values may appear during run time. Then it is possible to formulate rules for aggregation in such a way that getting more pieces of information (another redundant source) always represent a non-negative information addition. This is a key question to further investigate.

4. Implications on the Safety Manager

In the previous chapters is defined how to formulate safety requirements and how to formulate the evidence to be aggregated and used when assessing that all safety requirements are met. This is of outmost importance for defining the task of a safety manager as specified in the KARYON hybrid architecture.

In short, we can describe that the safety manager has three tasks. First, it shall keep track of all safety requirements applicable in run-time. Second it shall collect evidence for assessing all these safety requirements. Third, it shall adjust the level of service for any functionality to the maximal level that still can be considered safe.

Independently of whether the safety manager is implemented in a centralized or in a distributed way, it has to be capable to perform the run-time safety assessment. This implies that it shall be able to keep track of all the safety requirements allocated on all the architectural elements being part of the same hybrid architecture as the safety manager itself. From the above sections we can conclude that this implies that the safety manager also shall keep track of all applicable failure models of all the architectural elements to which any safety requirement is allocated.

The safety manager shall be able to collect the necessary evidence for assessing all the run-time applicable safety requirements. The important thing for the safety manager is to get the evidence that is aggregated to such a level where it can be used for comparing the achieved level of safety integrity with the required level of integrity. If some of the aggregation of evidence can be performed “above the hybridization line” is a question to be further elaborated in Task 4.2, concerned with the Safety kernel design specification. The important conclusion from this report is that somewhere in the architecture, there shall be elements capable of aggregating values and corresponding attributes of information quality for each failure having a safety requirement. The safety manager shall be able to compare this evidence for achieved level of integrity with the required level of integrity for each applicable safety requirement.

5. Information system

Co-operative systems of tomorrow will probably be the most complex systems humans have ever built. It is highly likely that not all initial design decisions will be correct but careful design decision form start will be of most importance to be able to adjust and correct early mistakes.

Previous chapter discussed of how to understand and handle safety and risks from a perspective of what standards prescribe and the relation and implications on the safety manager. The understanding is that there is a need to extend them into the co-operative perspective where redundant and shared information potentially allow risk reduction beyond what is possible with a standalone system. The following discussion will be a more concrete discussion how to handle this from a more theoretical perspective and an architectural solution that is part of the KARYON architecture. As a concrete example we will use the Intelligent Transport System (ITS) as specified by ETSI [14] and show how the KARYON architecture may look like instantiated in such a system. The discussions though are very general and not tied to ITS or automotive area.

A co-operative system is most likely to be introduced in steps and to maintain public acceptance we therefore suggest the creation of a methodology for systematically specifying and managing the concerns of such system. This would require:

- An information-model that allows a systematic and formal definition of context-awareness data in each individual system participating in the system.
- An information-model for a standardized parameterization and structuring of a wide range of concerns in the System-of-system context.
- A methodology for enabling a systematic approach to preliminary hazard analysis for system safety.

Based on such a methodology for specifying and managing the concerns of a co-operative system, the feedback loops through design-time modeling/simulation of safety, deployment, run-time decisions and post-crash information can be sped up. This would minimize the impact of safety by invalid assumptions and enable an increasing public trust in co-operative systems.

The reason for the forthcoming discussions about type systems is our attempt to extend a type system not only for software program themselves but also for the description of a co-operative system as a whole, the information-model. The goal is thus to extend what is normally defined as a type system with concepts and properties to handle physical information and safety information as well.

This discussion will also serve a purpose in understanding how to perform safety analysis for co-operative system and harmonize that effort with today's functional safety analysis.

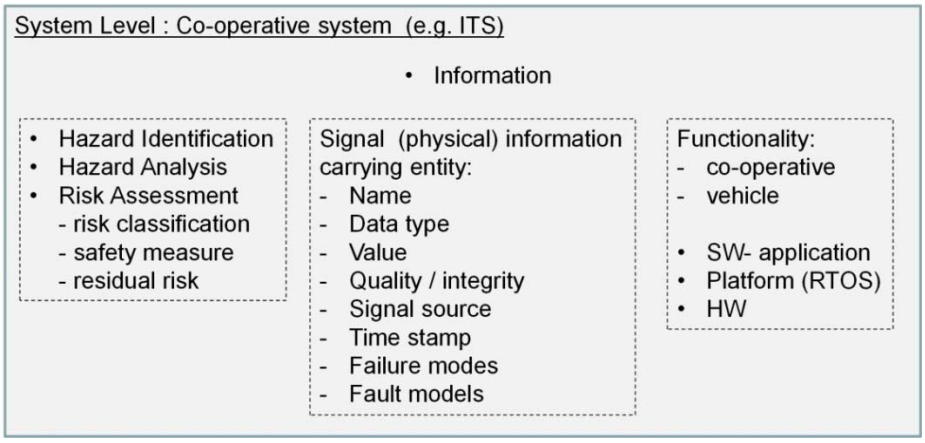

Figure 13: A theoretical discussion model of a co-operative system.

The central idea is that every system has, or is built upon, information, in one form another. It is awareness of information that makes it possible to identify potential source of harm. An embedded system makes it possible to handle information and perform safety measures which serve the purpose to mitigate risk.

Traditionally in physical system one has the terminology of signals & systems which are based on formal mathematics. A signal can simply be defined as an information carrying entity. Such simple definition is adequate for KARYON and enables the use of that term throughout the abstraction levels in a co-operative system. At physical level a signal is either an effect or an energy signal which may be converted into a data unit by a sensor. In the software abstraction a signal is an information unit that is communicated in the system e.g. the OSEK operating system [15] is using the term signal for data units that are sent between processes. In hardware description languages like VHDL [16] the term is used for information that is sent on the hardware abstraction layer between hardware processes.

For our purpose we simply understand that any information processing system consists of three different main activities: storing and retrieving of information, communication and sharing of information and lastly processing information. Separating an embedded computer system into these domains will aid in architectural explorations and also safety analysis.

In order for information to be handled by an embedded system the information must be encoded into a data structure. An information processing system is only dealing with encoded data structures. Every data structure can be associated with a type which is defining the value domain and the operators that can examine and manipulate the data structure. This data structure can be understood (decoded) wrongly or manipulated wrongly. There is thus a possibility to handle the structures in an incorrect way. We term that systematic fault i.e. faults that are resulting from the design process.

We propose to extend a simple type system, like the C programming languages, with some constructs that enables us to formulate safety requirements in a more formal notation. We term such a type an abstract data type.

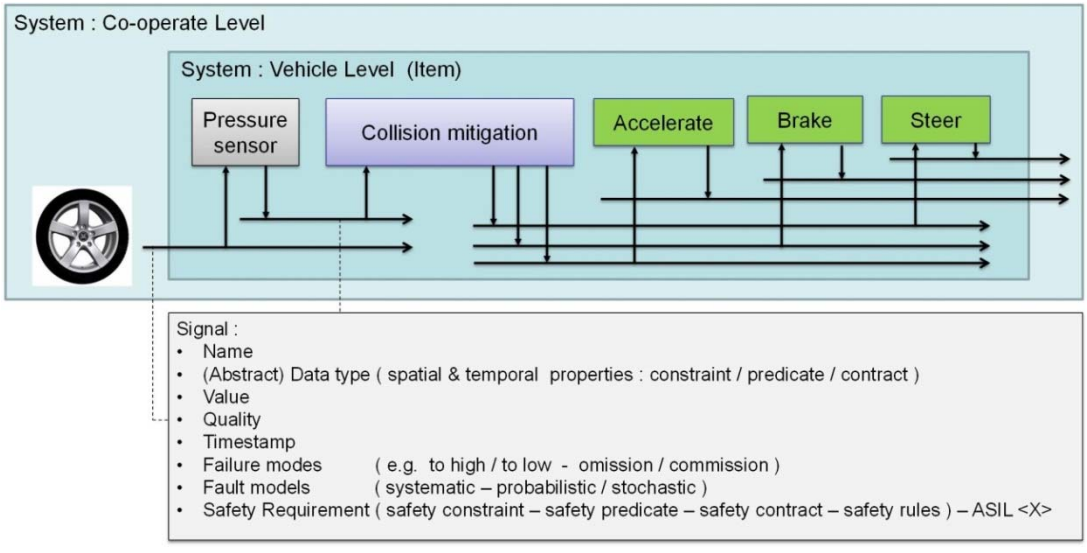

Figure 14: The concept of a signal and abstract data type.

For our purpose we need to attribute the data types with additional information that is not normally needed when programming software systems. We add information about quality (validity), timestamp, failure modes & fault models and safety requirements with ASIL information.

To understand how to use these extra information attributes we need to look at how a normal type system is used in software development process.

Programming is the art of designing data structures and algorithms that process those structures. The semantics is the meaning of information and syntax is its encoding into a technical representation. Any processing of information is done in the syntactical domain. This includes any verification or supervision process. A safety manager thus works in the syntactical domain with safety predicates.

To show these relations we need to look a bit deeper into a programming language. A programming language has two domains, the syntactic domain (the visual notation) and the semantic domain which attach a meaning to the syntactic domain. One may say that a safe programming language is one that protects its own abstractions [33] . An important abstraction is data abstraction which lets a programmer ignore details about data structure manipulations. The most common way to protect this abstraction for a programming language is by using a type system. The current discussion will be about type system for programming languages but the goal is to extend our understanding of type systems so we may use the same concept for safety constraints and safety predicates for co-operative systems in general and Intelligent Transport Systems (ITS) specifically.

There are mainly three different methods to assert that a program is safe using types:

- Static checking. A type analyser check (often built into a compiler) at design time that the program is safe. This is a conservative approach; it checks for absence of bad behaviour but cannot check for their existence. The problem to check for type consistency is an undecidable problem. Hence the type checker will sometimes reject perfectly correct programs i.e. a false negative.
- Dynamic checking. The type analyser does not check for type consistency during design-time but inserts extra information like type tags and small code segments into the ordinary program code that will run type checking during run-time.
- Let the programmer be in control of type checking. This kind of checking is mostly for more advanced checks and the programming language must have built in support for this, e.g. “try ... catch” statements (java programing language) or explicit exceptions and exception handling (Eiffel programming language). These concepts are similar and are a kind of recovery block mechanism [34] . This means that the programmer is responsible for inserting assertions in the code at suitable places. There will be no check that the programmer has done this correctly. This is a run-time check.

It is of course possible to mix the three alternatives and this is the most likely scenario in future.

A type is simply a set of elements (e.g. integers) and the operators one have attached to the set (e.g. for integers the operators addition ‘+’, multiplication ‘*’ etc.). The operators can be divided into processing information and comparing information. For integers we have arithmetic operators and relational operators. Most of the arithmetic operators have the result type into the same set as the operands, i.e. the operators are closed. The relational operators have their result type into the Boolean set (a set that only consists of two elements, True & False). A type system is the combined sets of types and their interrelations. A type system may be very simple like the C programming language or more complex like the Ada programming language. In Section 8.3 there is a closer look at a simple type system and all its systematic failure models in the spatial domain (i.e. the value domain).

The two most important abstractions there is to protect are the data abstraction and the control flow abstraction. The reason behind this is that these two abstractions are the only abstractions that have concrete mechanisms in hardware to support them. This means that these two abstractions are sensitive to real execution errors and thus those who really can be observed by some safety mechanism and supervised by a safety manager.

Data abstractions are protected at the abstract level by the type system. The type system assures that values have correct type and that the program will not crash because of because of illegal values. The mechanism down at hardware layer that execute the operands on the data structures are the Arithmetic and Logic Unit (ALU). To protect the data structures from being inadvertently modified there might be either a Memory Protection Unit (MPU) or a Memory Management Unit (MMU). The MPU exists even on fairly simple microcontrollers while the MMU only exists on more expensive and complex microcontrollers.

The control flow abstractions may be protected at the abstract level by the programmer explicitly. This is the way for example how AUTOSAR is handling protection of control flow abstractions. There is though nothing that prevents an extended type system to handle control flow abstractions too. Such type system is not commonly used today because of their complexity. Functional programming languages like Haskell have a type system that in some respects handle control abstractions. The structured programming theorem asserts that a programming language only needs three control flow abstractions; sequence i.e. executing statements in sequence, selection e.g. if-then-else constructs and lastly iterations e.g. for-loops constructs. All these abstractions are handled by the program counter at hardware level in any microcontrollers. Incrementing the program counter by one handles executing instructions in sequences. Loading the program counter with a higher address allows by-passing of code blocks and thus handles selection. Loading the program counter with a lower address will repeat code blocks and handles iterations. To protect the control flow abstraction at low level there are watchdog timers. Even simple micro controllers have timers. A watchdog timer may check for infinite loops as it will be set to expire within some short time if not re-loaded at certain predefined check points in the code. If the timer is expired the run time system becomes aware of the probability of an infinite loop in the task and may choose to restart the task. The timer may also be used to time the execution of code blocks to assert that they are executed with certain time limits.

There are other abstractions to but they do not affect the execution aspect of the program and thus have no mechanism at lower level to support or protect them.

The most common abstractions are:

- Functional abstraction. The use of subprograms for commonly repeated code blocks. This abstraction has often support at lower level by the stack pointer.
- Abstract data type. This abstraction encapsulates the functions operating on a data structures together with the data structures themselves. This is the foundation for object based and object oriented languages.
- Component abstraction. This is a grouping abstraction to reference a collection of programming elements in one structure. This is a support construct for the separation of concern principle. Each component has a clear responsibility.
- Hierarchical abstraction. This abstraction allows for a vertical structuring of components. This is a support for the divide and conquers strategy. The problem is sub structured into lower level tasks.
- Architectural abstraction. This is more or less the use of all the other abstractions together.

These abstractions are used to describe the system and ease developing programs e.g. reuse of code structure for cut-and-paste etc. It is of course also possible to have a type system for declaring interfaces for sub programs and components which can be checked for consistency by a type checker.

Our conclusion is that communicating and sharing of information is the central point in co-operative systems by the V2V (vehicle-to-vehicle) and V2I (vehicle-to-infrastructure) technology commonly termed V2X. We term the information carrying entity, regardless of abstraction level, a signal. Information has to be encoded into data structures in order to be stored and retrieved, processed and communicated. The purpose of a type system is to protect these syntactic data

structures at the abstract level. At the concrete level there exists mechanism to manipulate and protect the data structures.

Having a more information centric view will allow for easier discussion about super vision of information, protecting information and system safety. This will allow us to extend a type system very similar to the programming language C (the most commonly used in automotive industry) with attributes to handle information in the physical domain and to handle safety. Safety is partly addressed by formally analysing the type system for systematic fault models and also to extend the types with information how serious those faults are. The type system is also extended with quality information to accompany the values. Thus a type may have static attributes to handle the physical domain, fault models and quality attributes. At run time there will be attributes that asserts what fault models are unacceptable and what quality is needed for the current operational situation and thus the type system can then also support run time fault models.

For the same reason that a type system for a programming language will be too conservative to check for all safety properties at design time any type checker for co-operative system will be too conservative and give false negatives at design-time. This would have a strong impact on the potential advantages with a co-operative system. There is thus a need to move parts of the safety assessments into run-time. The dynamic behaviour of a co-operative system like the ones KARYON addresses even reinforces that statement. There will be a need for a safety manager and safety mechanism to handle safety at run time.

One might argue that a theorem prover can assert safety at design-time but it will in the end rely on the hypothesis one put into such proof. That hypothesis can never be proved to be correct but has to be assumed correct [32] . This does not mean that theorem provers cannot be of use for arguing safety in a co-operative system. A theorem prover and a type checker works very similar, in fact a type checker is a specialized theorem prover, and can be used to check the consistency between different abstraction layers. In essence a type checker can prove the system to be *consistent*, i.e. there are no conflicting requirements, at design time but can never be used to prove all safety at design time.

The type system in mind is designed with the specific purpose to be compliant with an existing theorem prover, Prototype Verification System (PVS) [18] . The advantage with this theorem prover is that it has a very advanced type system which allows types to be extended with predicates to form very precise types and requirements. The tight connection with PVS will allow us in future to use formal verification techniques for safety requirements of ASIL D in ISO26262 at design time.

The initial design of the type system though is simple to later be extended piece by piece to become more advanced. The short term goal with the type system is to have a formal notation to support KARYON safety rules to be checked at run time and not to perform formal verification. However this approach will allow future research to leverage on the KARYON project and to take benefit of the PVS and gradually understand how to formulate more and more advanced safety requirements and also to experiment of how to formulate safety requirements for a theorem prover, The long term goal is to be able to support the design process and the safety analysis right from start at co-operative level down to run time and assert that there are no semantic gaps in the information model. The environment model and the requirements are therefore stored in the information model (i.e. they constitute the information-model) right from start at top, cooperative system design and analysis level and accompanied the design process down to the run time level. The needed part of the information model is then extracted and stored in the vehicles to be used at run time by the safety manager.

All safety requirements that can be proved or asserted in design time will be analysed there (not necessarily by PVS) and the rest will be handled at run-time. The main reason to put safety cases and safety requirements on a formal base is that it will enable us to use tool support in handling them as discussed in [17] and to perform safety monitoring in run time with high confidence in safety.

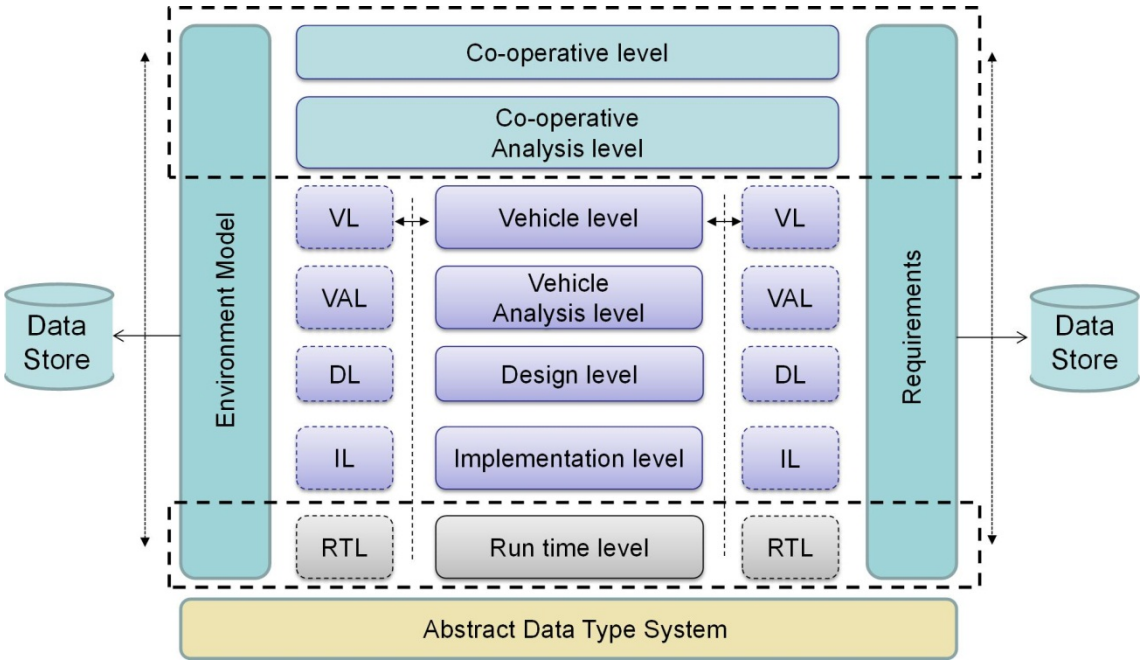

Figure 15: The abstract type system and the abstraction levels it supports.

6. Metrics and monitoring

Redundancy is important to achieve safety (and of course independency in the redundancy). In fact redundancy, in one way or another, is the only way to detect that something is not correct (or safe). A type system gives redundant information which is attributed to the data declarations which in turn is used to inform the compiler (type checker) about expected types.

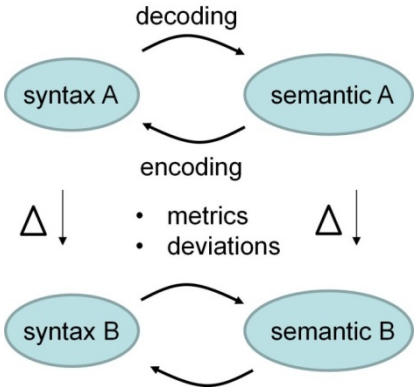

Figure 16: Relation about syntax and semantics.

Information is about meaning and belongs to the semantic domain. There might be several notations to syntactically represent the information, e.g. an integer can be represented in arabic numerals like "151" or in roman numerals like "CLI". So information is encoded into a syntactical domain for representation and decoded from a syntactical representation into the semantic domain as information. The most common errors in describing and interpreting programs occur here because of uncertain typing. Is the date "01/02/03" referring to 2001-feb-03 or 2003-jan-02? This is impossible to know without a type system. Is the length 10 referring to centimetres or inches? These simple type errors has been made in real life and cost a fortune because of system failure.

Processing in any form, like supervising information is always performed in the syntactical domain. Again, the only way to detect any deviation (difference) from normal (or safe) conditions is to have redundancy. So we may say that safety is a semantic property that is checked by a syntactic notion. So what we seek to understand is what kind of deviations in the syntactic domain is implying a change in the system which have such effect in the semantic domain that the system may become unsafe.

Deviations comes in two forms, the simplest form is just to assert that there is a difference. Questions which belongs to this problem class are: Is my program type safe? Is my system safe? The decision procedure to determine equivalence just returns the answer yes or no. What KARYON strives for is to find a more fined tuned metrics of deviation, i.e. what in KARYON is termed quality of information or validity of information. In order to do that there needs to be well defined metrics at our use.

To exemplify this: To protect data in memory one may use some form of checksum algorithm. The easiest algorithm is parity checking where one extra redundant bit is attached to a value. This will allow a yes/no question whether the data is valid (assuming a fault mode of one bit fault). It can though not describe how valid the data is. A better checksum is to use a hamming code. This code uses the metric Hamming distance. If there is a deviation between the actual value and what the hamming code prescribes it may be detected, and even in some cases be corrected. The mechanisms to do this supervision are the parity-checker respective the error –detection-and-correction-checker.

The metrics used in KARYON which are attributed to data will answer questions about validity in a more advanced way and the KARYON safety manager works with safety rules that monitors the validity of values and about timeliness of information processing.

6.1 Quality metrics

The simplest validity indicator for any data is a single bit denoting a valid or in-valid value, just like the parity bit discussed. This is though a very coarse metric, indicating useful or not useful data. The simplest validity metrics for physical information are related to standard deviation. The accuracy tells how well the value describes the correct value, i.e. the actual value. The precision is a metric which quantifies concepts as repetition and reproducibility of the result.

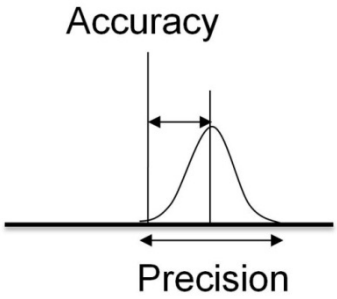

Figure 17: Validity metrics: accuracy and precision.

In KARYON we strive for finding more fine-tuned validity metrics for smart sensor data [19] but the two above may also be used. It all depends on the operational situation and what information is considered sufficient or necessary to achieve risk reductions.

In KARYON the terms quality and validity are used to denote the same semantics and will be used interchangeably.

7. Risks and safety for co-operative systems

Safety is about being safe, i.e. protected against events that are considered harmful. It can also be defined as achieving an acceptable level of risk. Risk assessment is about analyzing the system to identify the risks, and then the risk needs to be classified and prioritized. For co-operative system we have identified the need to monitor the system for the identified risks and to mitigate risks during run time.

There are two main strategies to achieve a safe system. One is that the safety manager implements redundancy in such a way that the limited level of integrity of the complex application components is still sufficient (this is what is referred to as ASIL decomposition in the ISO 26262 standard). The second is that the safety manager enforces the vehicle to a situation where the requested level of integrity is lower (this is what we also refer to as degraded functionality). These two strategies can then be combined. We have previously discussed the ASIL decomposition and following discussion is about separation in time and space from a risky situation. There can of course also be solutions which combines these two strategies.

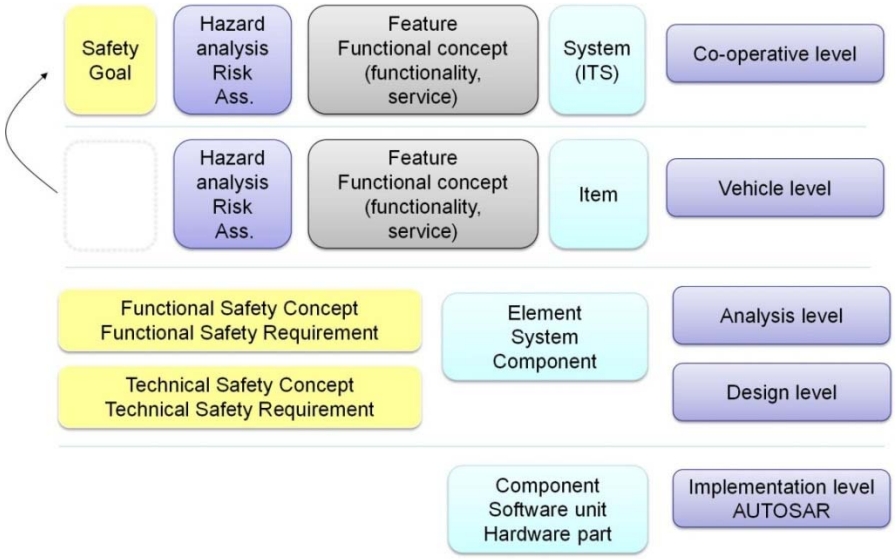

Figure 18: Safety goal from a vehicle perspective and a co-operative perspective.

Safety is intrinsically end-to-end in scope in that it has to treat the function of a component or subsystem as part of the function of the whole system. The implication is that the safety analysis and safety requirements needs to be started one level up than what is performed today. ISO26262 starts at vehicle level and assigns a top level safety requirement with an ASIL attribute to an item. For a co-operative system the top level safety requirement will be assigned at the co-operative level on the system and thus also the safety goals.

7.1 Co-operative safety analysis at design time

Safety and risks are much related to the services the system is providing. When designing these services the use of “use cases”, scenarios and operational situations are important as they drive the functional development. The use cases and the operational situations are also important in the process of identifying, classifying, prioritizing and handling the risks.

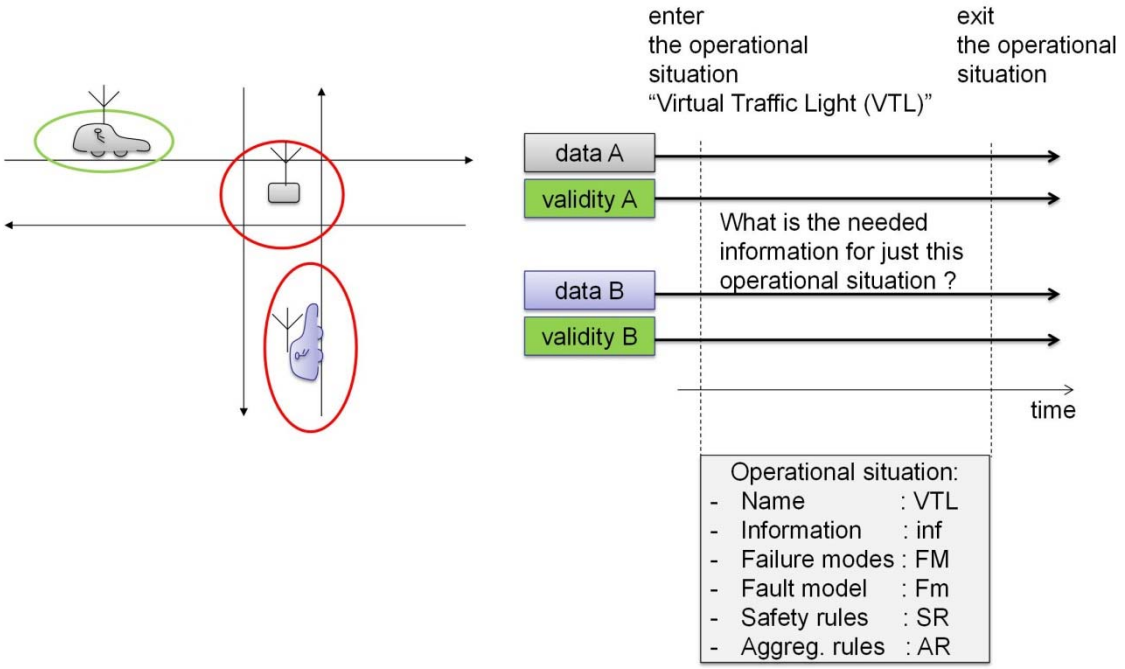

Figure 19: Co-operative operational situations and safety from design time perspective.

Normally a vehicle must always be in a state where the risk is acceptable from an ego perspective. The v2x infrastructure will allow vehicles to communicate vehicle to vehicle and vehicle to road side units. The sharing of information will allow risk reduction beyond ego perspective and enable the vehicles to co-operate in order to achieve a common objective, e.g. platooning while still being safe. In this scenario the safety measure is to separate objects in time and space in such a way that they are safe related to the information they perceive about its environment. So in a cooperative operational situation it is the ability of the participating vehicles to handle their common situation that matters. Today's standards on functional safety are more about the ego perspective than the cooperative perspective. To allow cooperative analysis a standard must state that a possible safety measure to reduce a risk is to separate in time and space physically from a hazardous situation. And also that, in some way, the shared information and its validity must be taken into account when estimating the degree of risk. A viable solution could be to add an extra parameter for calculating the Safety Integrity Level. These are small suggestions for modifications to existing standards but this has to be investigated further.

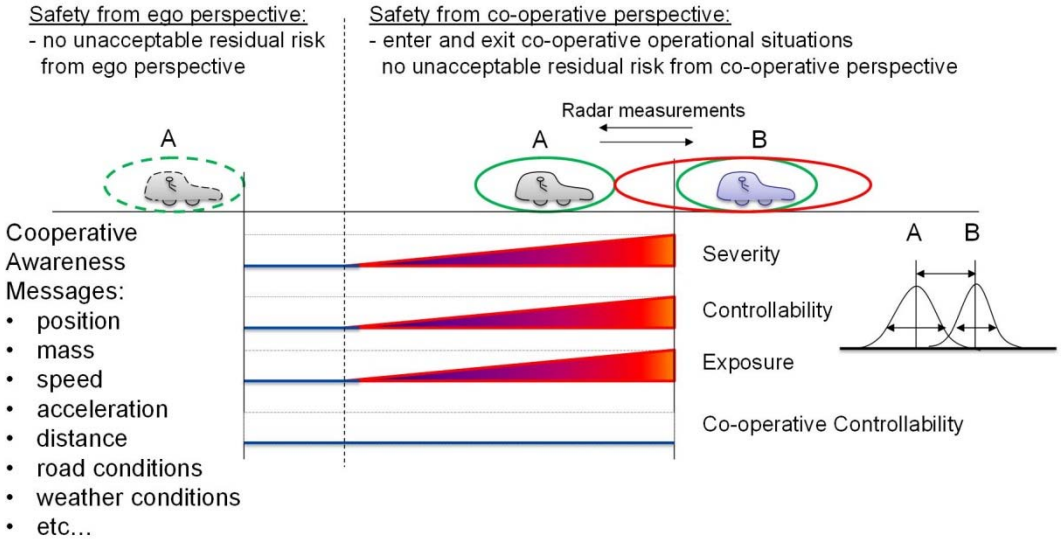

Figure 20: Operational situations and safety from vehicle perspective and co-operative perspective.

From an engineering perspective, one key topic for enabling advanced safety features is concerned with how to derive an information-model that allows a systematic parameterizations of a wide range of structural, functional and technical concerns in the co-operative context. By formally stipulating such concerns, the model constitutes a domain specific ontology that offers one of the most fundamental engineering supports towards solving the control and safety challenges brought about by the operational complexity. For instance, such a model will play a key role in plant-modelling for the design of autonomic control functions where runtime risk perception, prediction and treatment could be the themes. At design-time, such an information-model would also facilitate the communication and reasoning for safety analysis and risk management. For system run-time, the model would be used as the rules for the creation, analysis and consolidation of context-awareness data, as well as for potential online model-learning. For post-incident or accident analysis, the model is also valuable as it allows a formalized definition of the data recording the operational history and statistics.

Whatever the shared information is about it has extra attributes that is used for calculations of both the risks and validity in data aggregation (e.g. sensor fusion). Our abstract data type carries extra information about data type, quality (validity), timestamp, failure modes & fault models and safety requirements with ASIL information.

If the information should be directly monitored by a safety rule or aggregated for further use as fused senor data is dependent of the operational situation and internal use in the vehicles. The KARYON architecture will allow both options.

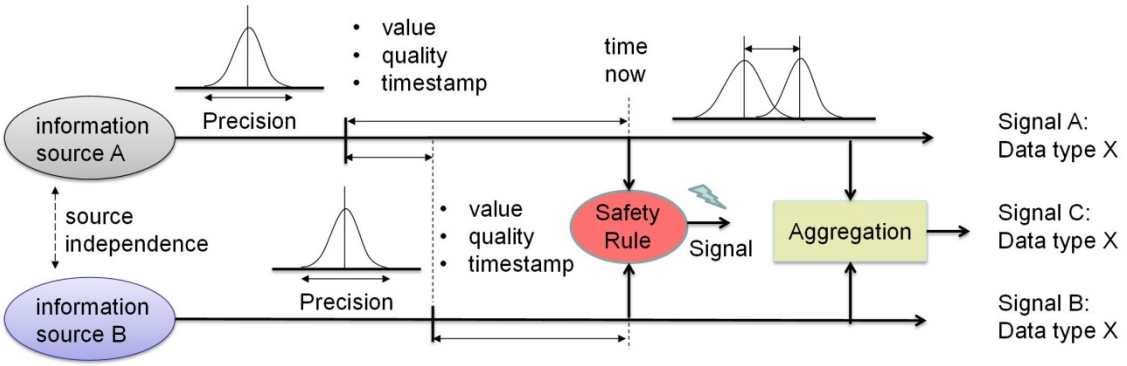

Figure 21: Information monitoring and fusion.

These are the attributes that will have an impact on risk calculations and sensor fusion calculations:

- Differences in values.
- Relative differences in timestamp, e.g. is the difference in time when the information was sampled small enough for them to be compared?
- Actual age of timestamp, e.g. to loss of information because of poor link quality might imply that the available information should be regarded as too old to be used.
- Quality (e.g. precision)
- Degree of source independence between A & B. If the information sources have something in common in how they have measured the information the degree of redundancy may be considered to be insufficient for risk reduction. Independency is a strong requirement in safety standards to achieve risk reduction by redundancy. Exactly how this attribute translates to shared co-operative information is not clear today and has to be investigated further.

There is noting that prevents the safety manager from monitoring individual signals for validity and timeliness i.e. it is the rules that determines what signals the safety manager shall monitor and what actions to take if a rule is violated.

7.2 Risk contours and dynamic safety shields

Risk contours are a way of describing the perceived risks from a vehicles ego perspective. Each vehicle scans its surrounding with sensors 360 degrees. This will give the vehicle a view of the nearby. In the same time other vehicles do the same. Their information is broadcasted on periodical bases and is received by the nearest vehicles. This will allow each vehicle to get a far better perception of the operational situation than what had been possible from an ego perspective. As long as information is shared and the information received is correlating with current perception it is possible to achieve a risk reduction which enables co-operative scenarios.

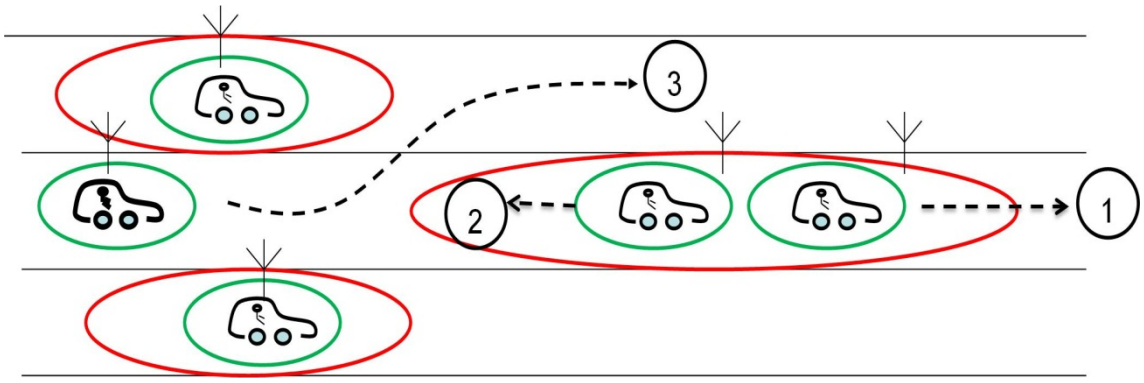

Figure 22: Risk contours.

7.2.1 Collision mitigation

Today's collision mitigation systems in cars use radar, laser and camera systems to get a perception of the current operational situation. In a co-operative context these systems will be extended with the information from other cars. Basically there are 3 main strategies to avoid collisions when the risk is eminent: 1) accelerating, 2) breaking or 3) steering away. They could of course be used in valid combinations.

Risk contours give a good estimation of the best action to take. The goal is to assert that the ego comfort zone will not come in contact with any other objects risk contour. This is very similar to the "Dynamic Safety Shield" [20] which is a concept in the co-operative application, Longitudinal Collision Risk Warning (LCRW), in ITS context.

For system safety the handling of an unsafe situation may not be what is most safe from an ego perspective but from the system safety perspective. A careful elicitation and analysis of the operational scenarios may lead to the requirement to support the following run-time decision making, ranging from strategic decisions during normal driving to tactical decisions to hit a certain part of another vehicle in a real critical situation:

- Braking as hard as possible may be the correct choice, for instance if traffic is sparse or all vehicles are autonomously driven.
- Smooth braking may be required if traffic is dense or weather conditions have made the road slippery.
- Driving off the road may be preferable if a truck is about to hit a bus, but the road environment consists of plains devoid of natural obstacles. At the same time, choosing to hit a car instead of a bus may be acceptable if the road environment consists of sheer cliffs.
- A front-to-side collision with a car may be preferable to hitting a pedestrian, but not if it will push that car into oncoming traffic and thereby create a collision involving several vehicles.
- A front-to-side collision may be preferable, but if a front-to-end collision is possible it may be a better choice. However, this might depend on the manufacture, model and year of both vehicles.

Again, to perform these kinds of decisions at runtime there is a need for an information-model that supports advanced analysis of operational situations at run time and evaluates the safety rules by extensive simulations. This information model must semantically be the same as the one each vehicle is maintaining at run time. In case of an accident it is vital that information has been logged by the system in such a way that the whole situation can be reproduced off-line to do a

post-crash analysis to assert what went wrong. The logging of information must be time stamped so it may be possible to merge information from several vehicles to get a more complete view.

For a fully autonomous operation the system strives to achieve optimal traffic flow while maintain safety of the system. The risk contours describes current options for the autonomous system. Hence the risk contours will be an aid in human driving, in coordinated driving and fully autonomous driving. It will also make it possible to perform smooth transitions between the modes by not allowing shifts if the situation is consider unsafe.

7.3 Safety monitoring at run time

As concluded in Chapter 5 there a different strategies how to handle and perform the safety monitoring and assessment during run time. The monitoring can be inserted manually in the code, e.g. a function call to the safety manager at specific places in the application code. This is the way the watchdog manager is working in AUTOSAR [35] . The software application is sending signals to the watchdog manager at checkpoints. The watchdog manager in turn checks that the software application reaches its check points in correct order and within time limits. This is a temporal integrity check. The safety manager in KARYON is checking for data validity and timeliness of information processing so the same concept might be used here also.

Another way is to attach hooks at the run time operating layer which is activated when monitored information is changing status, i.e. an event take place. Each time an event is triggered the safety manager is called by the hook mechanism and given the updated information. In this case the safety requirements and its safety rules needs to be stored in a data storage for the safety manager to access.

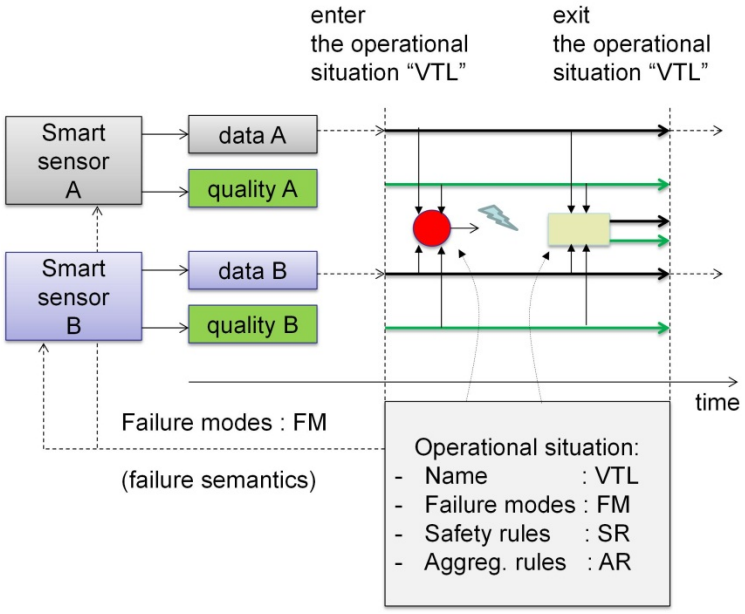

Figure 23: Data & validity to be monitored for an operational situation.

The safety rules may thus be statically inserted in the software application code but need only to be actively enforced during the time when its operational situations are active. The other approach is that the hooks are inserted and removed accordingly to the entering and exiting of operational scenarios.

When information needs to be aggregated the same concepts can be used. Either the aggregation function is instantiated permanently or it may be instantiated dynamically according to the current

operational situation. In ITS context it is in the facility layer that sensor data and quality metrics are fused (or aggregated). Even the history of data may be stored for the fusion algorithms.

In either case the aggregated function is activated by a triggered event on its inputs to produce a new aggregated value together with a new validity at its output.

The formalized description of an operational situation contains description of needed information and the quality attributes but also the failure modes that must be considered. The information about failure modes can be communicated to the smart sensors for two purposes. One is that it might be possible to fine tune the sensors at run time to activate some the relevant detection mechanisms. The other is that it might be possible to confirm that the smart sensors have relevant fault detection mechanisms that are activated. Otherwise it would not be possible to enter an operational situation that is not supported by the smart sensors.

7.3.1 Run time safety monitoring in an ITS context

In the ITS context sensor information is stored in a Local Dynamic Map [21] This is a very similar concept to the KARYON environment model [22] . The sensor information comes either from internal ITS-Station sensors or from the broadcasted information from another ITS-Station. All safety related Cooperative Awareness Messages (CAM) are broadcasted autonomously at best effort on a periodical time base.

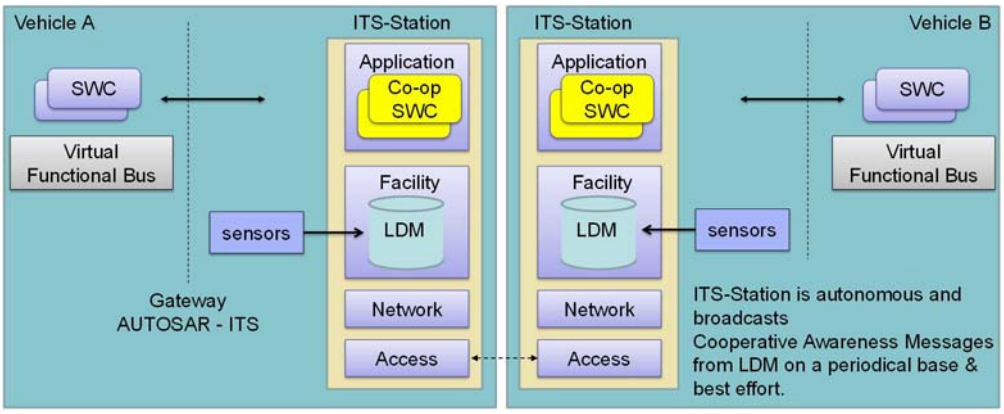

Figure 24: Two ITS Stations implemented in two vehicles that communicates.

It is the information in those CAM that allows for a risk reduction in a co-operative operational situation. The data is supported with a confidence interval of 95 % as a quality metric. This means that the Service Data and the Quality Data is stored in the same data storage physically. In the KARYON architecture the Service Data is separated from the Quality Data at the abstract level but at the concrete level instantiated in the ITS context into an ITS-Station they will be placed together.

In [23] the concept of run time safety analysis for automotive systems based on AUTOSAR and ITS-Station is looked at closer.

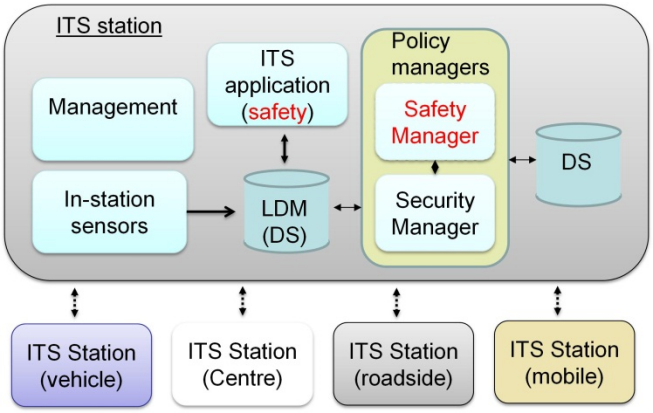

Figure 25: The safety manager, the environment model and the database storage for safety rules instantiated in ITS context.

At the abstract level of the KARYON architecture there is an environment model where smart sensor data are fused and stored and a second storage for the safety rules.

In the ITS context all the different storage can be combined into the LDM at the concrete level: the service data, the quality information & the safety requirements and safety rules. This would enable a very tight coupling between these parts for the safety manager to work with.

LDM contains data relevant to the operation of an ITS co-operative applications:

Types of data:

- Type1, Permanent: Geographic information.
- Type2, Transient static: Example: speed limit.
- Type3, Transient dynamic: Example: weather conditions.
- Type4: Highly dynamic: Cooperative Awareness Messages (CAM).

If a topological data base is used for both the LDM data storage and the safety rule database they might be considered as a unified data storage unit where the information in the LDM has been extended with a 5th type: the safety requirements described in our abstract type system .

In the KARYON scenario the LDM is thus extended with a 5th type:

- Type5, Information-model & Abstract-type-system:
  - Operational-situations
  - Safety-rules
  - Safety-properties
  - Safety-contracts
  - Safety-predicates

In [26] - [28] there are discussions about the different data storages and pros and cons for different applications. From this we conclude that a graph data base is a good option for storing the information model. It thus makes sense to have one and the same data storage for both the environment model and the safety requirements in both design time and run time.

8. Safety constraints and safety predicates as a type system

The aim with the type system described below is to be able to type the information in the information model with safety requirements. There are already efforts to support the design and safety analysis process with domain specific languages. EAST-ADL [24] is such an effort and is an architecture description language dedicated to automotive embedded electronic systems.

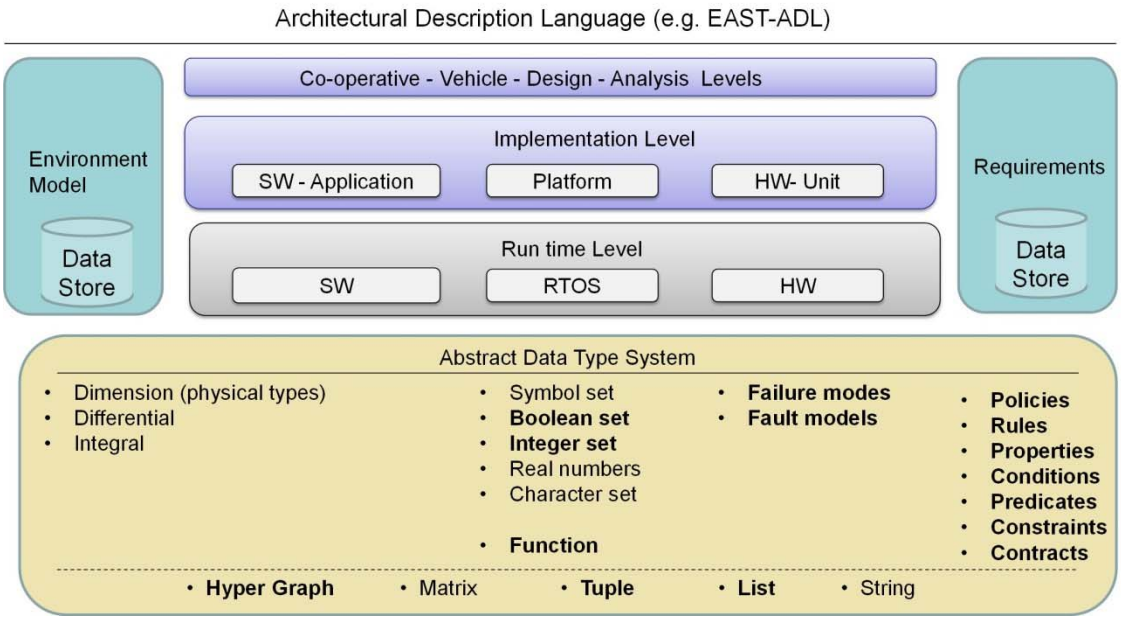

Figure 26: The abstract type system in relation to EAST-ADL.

One might wonder why then go to the effort of developing a new type system. The first reflection is that this type system is not replacing EAST-ADL. In fact EAST-ADL can be built on top of the described type system. EAST-ADL has been implemented in UML and other meta languages like MetaEdit+ [25] . The second reflection is that the type system is extending EAST-ADL with the possibility of having much more advanced types like predicate types used by PVS to fine tune the safety requirements. EAST-ADL stops at the implementation level while our type system is designed to be used also at run time to describe safety rules. Thus the type system supports the handling of safety requirements from the top level at design time down to the run time level.

Another important issue is that the proposed types system is so small that it is possible to define all the systematic fault models that can be inadvertently inserted at design time. This means that all the information in the information model has known systematic fault models at the basic level. The type system is advanced enough to build a very small executable language like LISP on top of it and that will mean that also the systematic faults for an operational semantics can be defined completely. It is important to understand that this alone will not encompass all possible run time failure modes and fault models but at least we have a good foundation to stand on.

Another important property with the type system is that it is possible to implement in PVS, the functional language Haskell and the programming language C. NICTA is a research institute from Australia that has developed a formal verification method that seems to have potential industrial relevance. They verified a small operating system, seL4 [29] , using the method in Figure 27.

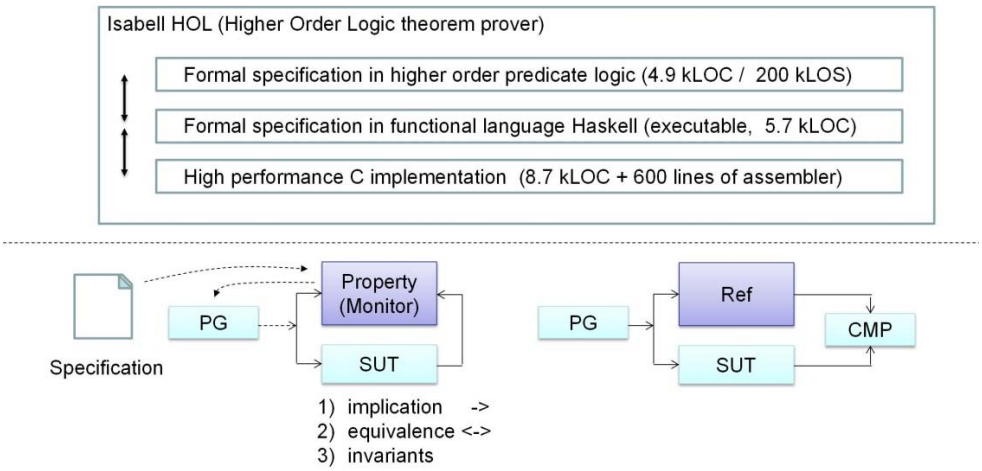

Figure 27: Property monitoring.

The topmost part of Figure 27 shows the tree different models that were used. In the top model only requirements where specified in Higher Order Logic (i.e. predicate logic of higher order). In the next phase an executable specification was implemented in the functional Language Haskell. Finally a highly efficient implementation was coded in the programming language C. The theorem prover Isabell HOL was used to prove that the properties specified at top level were consistent and that the lower models were fulfilling those properties.

It cannot be foreseen that this will become the normal standard for verification process in the software industry but for highly safety critical applications it has now become state-of-the art. NICTA is now researching on how to implement seL4 in the AUTOSAR Basic Software layer while another research group has taken the approach to directly verify an OSEK compliant operating system [30] .

A huge benefit with this verification method is that the requirement may be monitored at the design time during the verification process. During simulation a pattern generator (PG) provides test vectors for the software under test (SUT) and the simulator uses the properties to monitor the simulation execution. This is a very similar situation to our run time scenario where the pattern generator is substituted with the actual real life situation data and our safety manager is the monitoring device of the safety predicates. In the lower left part of Figure 27 the Haskell model is used as an executable oracle for the SUT which is another approach that becomes viable with this method. So this method opens up for a variety of verification options that can be tied to the exact needs for the safety assessment.

So even if one does not use a theorem prover there is still a meaning full task to derive the properties in a formal notation which makes it possible to proceed with the verification process with the best techniques for the situation at hand.

8.1 Logic systems

Safety predicates and safety constraints can be considered as an advanced type system where the value of one data can affect the allowed value domain of another data. Hence the types of the data are varying during runt time. As an example, when the parking brake has the value “On” the speed is not allowed to be over 5 km/h. And vice versa, the parking brake cannot take the value “On” if speed is greater than 5 km/h. Such safety rules are really a type system but an advanced one which is very difficult to analyse at run time.

To specify predicates and constraint one need logic systems. There are several logic systems that may be used ranging from the simplest one which is pure Boolean algebra to the most advanced

on which are higher order logic systems. Rushby [31] has investigated what kind of logic properties one can supervise with a safety manager.

Another way to understand what kind of logic system to use is to separate the properties into the spatial domain and the temporal domain. The spatial domain is the same as the value domain. A value domain can be constrained by the normal arithmetic and logical operators as described by the type system below. The temporal domain is used for constraining how the values may evolve over time. There are logic notations for specifying temporal behaviour like regular expressions and temporal logic system like Linear Temporal Logic (LTL) and Computational Tree Logic (CTL). These are used by model checkers to prove temporal properties at design time. It is though hard to incorporate the spatial domain into the same analysis. There are more advanced temporal logic systems that incorporate the value domain into the temporal domain e.g. Temporal Logic of Actions (TLA). It should be possible to use TLA to specify dynamic system behaviour that can be supervised at runtime but there is little research in this area so far.

Our safety predicates and constraints will be formed from the type system in Section 8.3. In KARYON though we only monitor validity of information and timing failures.

8.2 Run time storage for safety rules

One way to ensure flexibility and future improvement is to start with an information model that has few data types and a flexible structure to describe and relate information.

Looking at the LDM at type1 information we have geographic information. This kind of information is best described with a data graph structure containing metric information. A graph database is very appropriate for storing this kind of information.

There are two ways to describe an information model. One is to use a purely textual notation like an ordinary programming language. The other way is to rely on a graphical representation, more like an Abstract Syntax Tree. As our information model will constitute a lot of relations a graphical notation is preferred. This has the advantage that our notation language will be very flexible and extendible. It will be easy to export and import into a standardized XML schema to check for consistency and sharing between tools. The whole abstract type system can be exported to other tools via a XML and the safety rules will be exported in the XML format.

For these reasons we suggest that the safety predicates are described with a simple type system and stored in a hyper graph data storage. This implies that the storage structure becomes a vital part of the notation itself. A hyper graph is built with two simple elements, nodes and relations. Nodes may have several relations attached and a relation may have several nodes attached. The relations of a node are formed by a linked list and vice versa the nodes of a relation are formed by a linked list. This is indeed the most simple storage structure one can envision but it is very powerful in its simplicity. Simplicity and speed in accessing information are preferred properties and the hyper graph fulfills both these properties [28] .

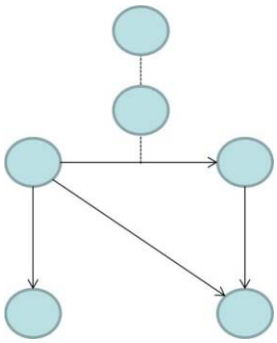

Figure 28: A hyper graph.

Simplicity is highly wanted as the information model in itself may have systematic faults that need to be found. The information model is a declarative model but has all the attributes of a normal programming language and is therefore submitted to the same faults.

To conclude the purpose of the type system:

- it is to be as simple as possible with well-defined systematic fault models,
- it can be used to build a hyper graph,
- it will allow us to specify safety requirements and safety rules,
- it will allow us to build Geographical Information System for our environment model if wanted.

The purpose is not to be efficient in terms of size; on the contrary the graph will be attributed with information to ease the manipulation of the structures and searching in the structures. The information-model will be constituted by the nodes and the relations in the hyper graph so during simulation it will be easy to change the information model to run iterative simulations to explore and evaluate impacts of different safety predicates. All safety requirements that has not been evaluated and assessed at design time will be exported as safety rules in XML format and stored in the safety rule data storage for run time checks.

Graph databases are fairly new concepts and have emerged as replacements for relational databases in situations where relations are important. It is believed that a graph database will be the best choice for describing operational scenarios at run time as the relations are highly dynamical, individual and changes fast [27] . Incorporating the safety predicates into the same database may have huge impact on speed of safety assessments during run time. Further research is though needed to support those claims but the demonstrators developed in KARUON WP5 can be used as research objects.

The type system in Section 8.3 with its value domains and arithmetic and relational operators are sufficient to build any information model and logical system. There are efforts to build different ADLs, like EAST-ADL, for system specifications but with this approach KARYON is not tied to any specific notation.

If the nodes only consist of the values and operators from the type system, then the hyper graph really constitutes an abstract syntax tree, like the ones used for internal representations in a compiler [41] . This means that the relations form the control flow abstractions and the graph is really executable. If the node is allowed to be more complex the graph may be used as a file system where the nodes may be similar to index nodes, i.e. inodes in a Unix style file system [42] .

So initially the nodes and their relations may be detailed low level objects but can later on easily be transformed into efficient binary code and placed in a more complex node as an executable code. This is referred to as just-in-time compilation (JIT) [43] and is used for example by the java programming language. An even further extension would be to transfer part of the safety rules directly to hardware which may be necessary for performance issue if there are many safety predicates to supervise at the same time. There are on-going researches about reconfigurable hardware [36] .

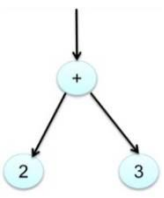

Figure 29: An abstract syntax tree.

There are more benefits with this architecture than KARYON is exploring but we will not investigate that in the KARYON project.

8.3 Type system

A type is concerning a set of values and the operations that may be applied to elements on those sets. Most programming languages have a static type system. A type system may be simple, e .g the C programming language, or more complex, e.g. ADA. There are other programming languages that have a dynamic type system, e.g. LISP or Erlang. A programming language that has a static type system may be checked at design time. The compiler does a basic type checking but there are explicit tools for static program analysis which performs more advanced checks like buffer over-run etc. A program that has been checked for type consistency is called type safe i.e. there will not be any failure during execution for those cases considered. Programming languages that does run time type checking, i.e. dynamic type checking, must attribute data with its type, i.e. tagged types and include the type checking code into the actual code. This makes the program much larger and it will execute much slower. The benefit is that it may allow more exact checking.

Another way to view this is that types are properties assigned to information (or data). The more advanced properties the more advanced type system. It is possible to construct type system where correct typing implies that the function is correct. These type systems are more or less what are termed contracts in other languages. Mathematically (and logically) these two views are identical. It is more a convention of terminology than how the underlying logic and mathematical systems are working.

To be able to conform to C, which a very common programming language for embedded systems and with a simple type system, the type system described in this section is extremely small but it has all the needed constructs for building any data type. The type system can in fact be used to build a Turing complete language.

Only four basic types are needed. They all have different properties and can be used together to encode any information. As the type system is so small and natural we will not define any concrete syntax for it but rely on our hyper graph to build abstract syntax trees. The notation used in this document is trivial and can be mapped to any programming language.

8.3.1 Symbol set

A symbol is simply just a name. It is used to represent different non-numerical constant values, e.g. it can be an XML tag. The symbols are built by concatenating character from the ISO/IEC 8859-1 character set. The only restriction is that the first symbol may not be a number or any other start character from the concrete syntax of any data types defined below

Example of symbols are: Stop, Rule, Action etc.

They can thus be used to build and give sematic meanings to data structures that are not in the basic definition e.g. (Date, (Year, 2014), (Month, 10) (Day, 2)). This is very similar to the way the programming language Lisp is defining more advanced data structures. The same data structure described in XML would look like:

<Date> <Year>2014</Year><Month>10</Month><Day>2</Day></Date>

The failure model is concerning selecting values from the set or selecting operators. For any set that is not ordered, the only failure model is to select the wrong element, i.e. value or operator.

- Values: { <all the allowed symbols> }

- Relational operators: { = , != }

8.3.2 Boolean type

All programming languages need the Boolean type. The reason is that this type is the result of any relational operation and is used in selections e.g. “if (expression: Boolean) then <true statements> else <false statements>”.

It is a simple set consisting of only two members: { True, False }. The set is not ordered.

The operators are grouped into two groups, relational operators and logical operators.

The relational operators are equality (=) and inequality (!=). As the set is not ordered we cannot perform a comparison in order like greater than (>)

The logical operators are the normal Boolean operators: AND, OR, XOR, NOT, IMPLY

The failure model are concerning selecting values from the set or selecting operators. For any set that is not ordered, the only failure model is to select the wrong element, i.e. value or operator.

- Values: { True, False }
- Relational operators: { = , != }
- Boolean operators : { AND; OR, XOR, NOT, IMPLY }

8.3.3 Integer type

All programming languages have the integer set.

- Values:] -∞, ∞ [
- Relational operators: { = , !=, >, < }
- Arithmetic operators : { +, -, *, integer_div }

The failure modes from the Boolean type still apply to the integer type. As the value set is ordered we extend the failure modes with: to_high, to_low.

8.3.4 Tuple type

Both types above are simple types. We also need types for aggregating types. The tuple types are used for this. It is similar to the record type in ADA or the struct type in C. It is an ordered list of elements and it is static. It is not possible to extend or reduce the number of elements of a tuple.

- Values: The tuple type can be defined after how many elements it has. The range is [0, , ∞ [e.g. a tuple with 3 elements could be notated (a, b, c).
- Selection operators: {first, second, third, ….}
- Other operators : { size }

The failure mode for this type is that the elements are in wrong order or that one selects the wrong element. Each element has a failure mode that depends on its type.

It is convenient to have the parenthesis as notation for grouping elements in a tuple, for example (1, True, (False, False), 6). The selection can be defined as (6, 1, 5).3 i.e. select the 3rd element which is 5.

The notation of using parenthesis for grouping elements makes it very similar to most programming languages call to subroutines e.g. function calls or procedure calls.

8.3.5 List Type

The list type is very similar to the tuple type but it is dynamic. It is possible to add and remove elements. If all the elements are required to be of the same type, this can be statically checked. On the other hand, if one allow the elements to be of different types the elements must be tagged during run time and be typing must be done at run time, i.e. dynamic typing.

- Values, The list type can be defined after how many elements it has.
- Constructor operators: { concatenate (or prepend), NIL }
- Selector operator : { head, tail }
- Other operators : { size, reverse, replace }

The failure modes are that elements are in wrong order, elements are missing or extra elements are included. Each element has a failure mode that depends on its type.

As a notation we may use the curly bracket parenthesis {} e.g. { 1, 2, 3, 4 } which is a list of integers or { 1, 2, False, (1, 2) } which is a list where each element may be of different type than other elements.

8.3.6 Concrete syntax notation for types and safety rules

In the examples that follows we denote information with its type coding likes this: <name> : <type name> e.g. “Speed: Integer” or “Engine on : Boolean”

As our type system has no limit in the domain values we may constraint them further by some predicates that must be true.
Example; Speed : Integer · [0, 200]
This is interpreted as Speed is coded in the integer set with a constraint of only talking values in the interval of 0 to 200. There could of course have been another concrete syntax.
Example; Speed: Integer · Speed >= 0 AND Speed <= 200

We will allow several constraints and predicates for information by separating them with “·”. The only restriction is that we cannot change the basic type set or allowing the information to take values from several type sets.
Example; Speed: Integer · [0,200] · Parking_break_on = On IMPLY Speed < 5
That is, when the parking break has the value ON the speed value should not be above 5 km/h.

It is easy to extend the Speed type with more properties which describes the failure modes which are relevant and their ASILs.
Example; Speed : Integer · Speed > 0 AND Speed < 200 · (omission, ASILD)

As the types and safety rules probably will be imported and exported in the XML format between tools it is up to the individual tool to select a concrete syntax for the users.

Note that the aim with the abstract type system is to describe the information in our information-model with a type system in a general sense, for the purpose of storing information and communicating information and to be very precise in the semantics.

8.3.7 Physical types

We can easily extend the above type set with a Real numbers to be able to handle physical quantities. We may also extend the type system to denote a type with dimension, unit and prefix.

For example, “Speed: Real · [L / T] mm/s”.
This should be read as Speed is encoded with a real number having dimension Length divided by dimension Time. The unit for length is meter with a prefix of milli and the unit for time is second.

This precise typing will prevent Speed to be assigned an integer that has dimension Length.

8.4 Policies, rules and properties

It is a good design principle to separate policies from the mechanism that enforces them. This separation will make the system more adjustable and maintainable as rules and policies may be updated or modified much easier than low level mechanism which may require a change at hardware level. It is hard to foresee all the safety properties needed by co-operative system at forehand during its life time. Safety rules may even be uploaded dynamically at run time in the future and a clear separation from start between rules and mechanisms will allow that [37] .

Separating the safety rules from the mechanisms that uses them also has implications for safety analysis. The safety manager is a vital part of the KARYON architecture and will probably be attributed with safety requirements having Safety Integrity Levels of class ASILD. The rules may not necessarily have the same integrity levels as the safety manager itself but may be verified according to lower needs, e.g. ASILB.

Our safety rules are described by the type system and stored in a data storage and the safety manager in the KARYON architecture is the mechanism that enforces them. Taking AUTOSAR as an example, the watchdog manager (WDM) is the mechanism that enforces temporal checking of software components [35] . The policies are encoded in the software components as explicit calls to the WDM.

In the AUTOSAR case we may say the policies are not actively enforced by the system. Instead, the policies are passively checked. It would be possible to have mechanisms in the hardware layer which actively monitors a temporal abstraction in the future.

The meaning of the words conditions, constraints, and predicates are often used in KARYON and sometimes with the same semantics. The semantics of these words are also often context dependent. For our type system we try to comply with how those words are used in safety community but also try to be compliant with the meaning in the PVS context. As our type system can be used to denote any information, not only safety related we remove the prefix safety in the definitions. In some context adding the prefix safety may change the semantic meaning slightly from those used here but then there should be a more clear definition of the terminology at those instances.

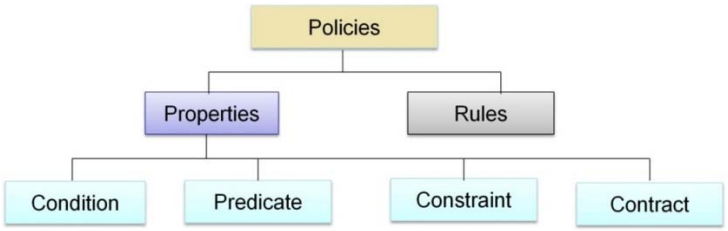

Figure 30: Taxonomy.

- Condition
  A condition is an expression with the result in the Boolean type and with no use of any Boolean operators (i.e a leaf Boolean expression).
  - Example: Speed > 10
- Predicate
  A predicate is an expression with the result type in the Boolean type and Boolean operators are allowed. To start with, the type system does not handle the predicate logic quantifiers for all (∀) and there is (∃) but this is a small limitation for the KARYON

project. These will be needed when using a theorem prover at design time or when building state machines that monitors more advance rules [40] .

- Example: Speed > 10 AND Speed < 300

- Constraint
  A constraint is an expression limiting the value domain of some data.
  - Example: Speed : Integer · Speed > 10 AND Speed < 300

  The differences with a constraint from the above definitions is that the constraint cannot be used in a test, its purpose is to limit the value domain i.e. it is a declarative property not an testing property.

- Contract
  Contracts comes in two flavours, see Figure 31. It can be used for specifying the properties of a component. Then a contract has the form of a triple:
  - (pre-conditions, invariants, post-conditions)

  This is in accordance with the Design-By-Contract principle [38] . The pre-conditions, invariants and post-conditions are all a list of predicates. In the other scenario a contract is between two components. Then the contract is formed by a pair:
  - (post-conditions of component A, pre-conditions of component B)

  Here the contract is fulfilled if component A provides data restricted according to its post-conditions and those post-conditions is in compliance with component B's pre-conditions. Another way to put it is that the type of the component A's output interface is type correct with the input interface type of component B. This is how PVS would prove that the contract is fulfilled.

- Property
  A property is simply anything of the above.

- Rule
  A rule is a relation which may be specified as a tuple:
  - Example: (<property>, <action>) – if the property does not hold then perform the action.

  In KARYON we may specify the action instead as a Level-Of-Service:
  - (<property>, <Level-of-Service) or vice versa (Level-of-service, <property)

  As long as the property holds we may stay in the Level-of-Service otherwise we may look for another rule where the property holds and select that Level-of-Service.

- Policy
  A policy is one or several rules (i.e. a list of rules) that are meaningful to group and handle together.
  - Example: [(Level_of_service_A, <property A), (Level_of_service_B, <property B), (Level_of_service_C, <property C)]

The above definitions are also in compliance with the definitions of a coverage model called modified condition/decision coverage (MC/DC). It is a requirement that this coverage model is used for the most critical (Level A) software according to the standard DO-178B [39] . As our properties can be used for specifying any safety requirements and is suitable to be monitored during simulation they can be directly used in such a coverage measurement.

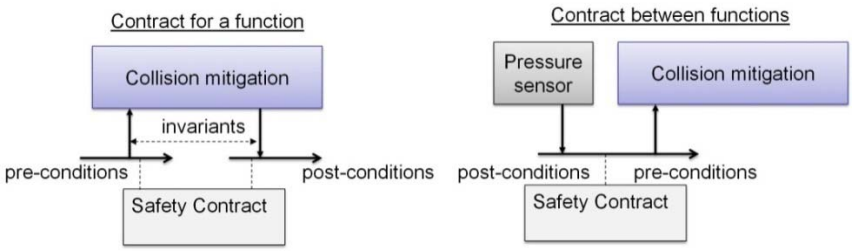

Figure 31: Variants of contracts.

We are explicitly a bit vague here about the exact notation and semantics of a rule because we would like the KARYON architecture to be sound and solid but still be somewhat flexible to allow for many different instantiations. The type system follows that principle. It is very simple in its base but still rigid with well-defined systematic fault models for the base and then allow for more application dependent type definitions. A concrete example of how to use the type system is described in next chapter using XML.

9. Safety rules definition

In this section we provide a brief description of the specific solution that was adopted to define safety rules within the KARYON Safety Kernel. A preliminary definition of the Safety Kernel was provided in deliverable D4.2. The Safety Kernel can be described as a part of a KARYON system that encompasses a set of components responsible for managing the Level of Service of the different cooperative functionalities in order to achieve functional safety objectives. In this way, it can be seen as a black-box providing a well-defined interface to the functional components that are necessary to implement the desired functionality. It receives inputs from these components concerned with the validity of information and with the timeliness of certain operations, and provides the necessary data to change their behaviour (performance level) whenever this is necessary to enforce a Level of Service change.

Internally, the Safety Kernel includes a safety manager, which is responsible for assessing the received validity information and the collected timeliness information against a set of predefined, static safety rules. For each level of service of each cooperative functionality there is an associated set of safety rules. The only exception is that the lowest level of service does not have any associated safety rules, because it must be guaranteed during the design phase that the lowest level of service always matches the available integrity. If the safety rules associated to a certain level of service are satisfied, then the functionality can be provided with this level of service. The safety manager is also able to evaluate the level of performance of individual functional components that is suitable to enforce the desired level of service for all the considered functionalities.

To perform its tasks, the Safety Kernel must hence incorporate all the safety rules that have been defined in design time. This has been called the safety rules database, but the specific format and solution for storing these rules can assume different shapes.

In the development of the Safety Kernel prototype that will be used in the KARYON vehicular demonstration, we defined a solution in which the Safety Kernel uses a configuration file in XML format. This allows for additional flexibility during testing phases, given that the file can be loaded during bootstrap of the system, and can be even changed in run time (which is clearly not a feature one will need or want in a production system).

The basic structure of the XML configuration file is the following:

```
1  <?xml version="1.0"?>
2  <config>
3     <!-- System definition -->
4     <system>
5        ...
6     </system>
7     <!-- Unit definition -->
8     <unit id="0">
9        ...
10    </unit>
11    ...
12    <!-- Unit definition -->
13    <unit id="3">
14       ...
15    </unit>
16    ...
17 </config>
```

The *system* section is optional and used to set up some overall attributes of the Safety Kernel while the *unit* sections allow configuring the calculation units. Each unit has an *ID* and refers to an input, output or any locally calculated value (a *local LoS* for instance).

9.1 System definition

This section contains three optional Safety Kernel attributes: *FAILURE*, *SUCCESS* and *PERIOD*. The *FAILURE* and *SUCCESS* attributes are provided to allow the definition of a tolerance degree on possible timing failures of some components (those that are above the hybridization line, as defined in previous KARYON deliverables), and to set a minimum number of consecutive observation of timely behaviour in order to "trust" the observed components (which must also be a component above the hybridization line).

```
<system>
    <failure>2</failure>
    <success>3</success>
    <period>200</period>
</system>
```

The *PERIOD* attribute defines the Timing Failure Detection (TFD) period and, consequently, the output sending frequency. This period cannot be too small, to avoid having a Safety Kernel requiring too many processing resources. In design time it is necessary to verify that this period is feasible, never leading to overload situations. On the other hand, it must be small enough so that the Safety Kernel can quickly react to changes regarding the observed integrity data (validity and timeliness).

9.2 Unit definition

A basic unit definition contains three optional parts. Firstly, the definition of the unit attributes. Secondly, the safety rules. And finally, the multi-component sources. Multi-component sources can be defined to deal with redundancy patterns, which occur when some function is implemented by redundant components, and at least one of them is above the hybridization line. Each unit is associated with an *ID*. For input and output units, this *ID* is used for the identification of information exchanged with the Safety Kernel.

```
<unit id="2">
    <!-- Unit attributes -->
    <mode>update</mode>
    ...
    <!-- Rule definition -->
    <rule level="4">
        ...
    </rule>
    ...
    <!-- Multi-component definition -->
    <from id="4" level="2"\>
    ...
</unit>
```

There is a particular unit type corresponding to the input components above the hybridization line. These components may not be timely and hence are required to periodically send information to the Safety Kernel (ultimately sending a *HEARBEAT*) to prove they are behaving timely.

The different kinds of information that can be sent to the Safety Kernel are defined in deliverable D4.2. In brief, they include *HEARTBEAT* or *VALIDITY* data, but also *LEVEL* (Level of Service) data, which is sent by the Cooperative LoS Evaluator component (which is not implemented within the Safety Kernel due to the fact that it involves, even if indirectly, the need to communicate through a wireless network). Finally, and although it would be possible to deal with redundant components (multi-component pattern) outside the Safety Kernel, the interface also supports *DATA* to be sent and received, so that the Safety Kernel manages itself the selection of the appropriate data to be forwarded from redundant components.

9.2.1 Potentially non-timely components

For components above the hybridization line, which may become untimely, a *TIMEOUT* attribute defines the maximum period of time, in milliseconds, between two consecutive interactions. If not specified, the timeout is equal to 0 by default and the component is considered to be below the hybridization line (thus not requiring timeliness monitoring).

The configured timeout should be set equal to the interaction period plus the considered maximum jitter in the execution of these interactions.

```
1 <unit id="0">
2     <timeout>400</timeout>
3     <failure>2</failure>
4     <success>3</success>
5 </unit>
```

The meaning of *FAILURE* and *SUCCESS* is similar to the attributes defined in the system section except that they have priority over the previous ones and are only available for the associated unit.

9.2.2 Sending mode

For every unit which refers to a component above the hybridization line, the Safety Kernel will update the unit's required performance level. By default, this level is not sent to the interface as an output. The *MODE* attribute allows to force the sending. There are three available modes:

- < *mode* > *regular* < */mode* > : The level is periodically sent to the interface. The sending period is the same as the Safety Kernel execution period.
- < *mode* > *update* < */mode* > : The level is only sent to the interface if it is different from the previous value (e.g., when the level of performance needs to be degraded).
- < *mode* > *silent* < */mode* > : Nothing is sent to the interface (default setting).

9.2.3 Safety Rule definition

A safety rule is identified by a unit ID and a level. This level value will be the unit's level of service or performance level if the rule is evaluated to true. The rules are evaluated from the highest to the lowest LoS but it's not necessary to declare them in this order as the rules are sorted by the safety manager.

As soon as a rule is evaluated to true, the corresponding level is set to the unit. The subsequent rules with lower levels are not evaluated. If neither of the rules is true, the unit level is set to 0 (lowest level of service). A rule contains an evaluation tree with four different types of nodes: *test*, *validity*, *level* and *value*.

- **TEST node**: This node runs a Boolean test between different sub-node operands. Each operand might be a *test* node or a *terminal* node. The different *test* operations that are recognized are the following:
  - *and*: All sub-nodes must be evaluated to true. This operator is used as default one between the first-level nodes.
  - *or*: At least one sub-node must be evaluated to true.
  - *sup*: True if first operand is greater than second one.
  - *supe*: True if first operand is greater or equal than second one.
  - *inf* : True if first operand is lower than second one.
  - *infe*: True if first operand is lower of equal than second one.

  - *equal*: True if first operand is equal to second one.
  - *diff* : True if first operand is different from second one.

- **VALIDITY node**: Evaluation of this node returns the latest data validity timely received by this unit as *VALIDITY* data.
- **LEVEL node**: Evaluation of this node returns the current level of the unit. This value can come from three sources:
  1. The latest level of service timely received by this unit through as *LEVEL* data.
  2. The latest rule evaluated to true for this unit. In this case, the level can be either a level of service or a performance level.
  3. The level of service of the latest selected output from a multi-component function.
- **VALUE node**: Evaluation returns the static value of the node.

To illustrate the rule creation, we show below a basic LoS rule of some functionality. The level of service of the functionality is 1 if the validity of component 0 is greater than 50. Otherwise the level of service is 0 (default value).

```
1 <unit id="1">
2    <rule level="1">
3       <test type=" sup ">
4          <validity id="0"\>
5          <value>50</value>
6       </test>
7    </rule>
8 </unit>
```

9.2.4 Multi-component definition

A multi-component definition is a list of component sources introduced by the *FROM* attribute. Each source has an associated performance level and is supposed to periodically send to the Safety Kernel some output data. Among all sources which sent their data in a timely way, the Safety Kernel will select the one with the highest performance level and send it back to the interface.

The source with the lowest performance level must be a component below the hybridization line. In the example below, this component is identified by id=3.

```
1 <system>
2    <period>200</period>
3 </system>
4 <unit id="2">
5    <from id="3" level="0"\>
6    <from id="4" level="1"\>
7    <mode>regular</mode>
8 </unit>
9 <unit id="4">
10    <timeout>200</timeout>
11 </unit>
```

In the example, the multi-component function is composed by two sources: component 3 with performance level 0 and component 4 with performance level 1. Component 4 may not be timely and must send its result with a maximum period of 200ms. The Safety Kernel will forward the selected data to the interface (*ID* 2) every 200ms (Safety Kernel's period).

Every 200ms, the performance level of component 2 will be updated with the performance level of the selected source (0 or 1). As the *REGULAR* mode is enabled for component 2, this performance level will be periodically sent to the interface (*ID* 2).

10. Conclusions and Next Steps

Deploying a co-operative system is most likely to be introduced in steps and to maintain public acceptance. We therefore suggest the creation of a methodology for systematically specifying and managing the concerns of such system. We have discussed the need for:

- An information-model that allows a systematic and formal definition of context-awareness data in each individual system participating in the system.
- An information-model for a standardized parameterization and structuring of a wide range of concerns in the System-of-system context.
- A methodology for enabling a systematic approach to preliminary hazard analysis for system safety.

The proposed abstract type system will make it possible to define safety properties that may apply both for expressing safety requirements (what is needed) in design time and for expressing evidence in safety argumentation (what is provided) in run time. It has been designed to be able to support the above information-model for describing the complexity of a co-operative system.

The abstract type system has attributes to describe:

- Failure to avoid
- Safety Integrity Level telling how sure it is that the failure is avoided
- Reference to signal for which the failure is applicable.

When collecting evidence in run time that safety requirements are fulfilled, such information needs to be able to able to specified in a safety rule and stored in a rule data storage so that the safety manager can access the rules and monitor the safety properties specified by the rules.

The type system is very small in the beginning but has the needed feature to be able to be used for XML, Haskell, PVS and the C programming language and to build a hyper graph as the fundamental data storage unit in a graph database.

The KARYON architecture specifies 4 data storage units: The Service Data, the Quality Data, the safety rule data storage and the environment model.

The KARYON architecture instantiated in the ITS Station context will benefit from having the same data storage for all those needs but this may not be the best option taken in another instantiation context.

The KARYON architecture is flexible enough to allow for an almost immediate use but also to allow for adjustment and enhancement for problems that cannot so easily bee foreseen at this stage.

References

[1] SAE ARP4754, AEROSPACE RECOMMENDED PRACTICE, Guidelines for Development of Civil Aircraft and Systems, Revision A, 2010.

[2] ISO 26262, Road vehicles – Functional safety, Part 1-9, 2011.

[3] SAE ARP4761, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems, 1996.

[4] RTCA DO-178B, Software Considerations in Airborne Systems and Equipment Certification.

[5] RTCA DO-254, Design Assurance Guidance for Airborne Electronic Hardware.

[6] R. Johansson, *et al.*, "A road-map for enabling system analysis of AUTOSAR-based systems," Proceedings of the 1st Workshop on Critical Automotive applications: Robustness & Safety (CARS), Valencia, Spain, 2010.

[7] D. Chen, *et al.*, "Integrated Fault Modelling for Safety-Critical Automotive Embedded Systems," *Springer IE&I elektrotechnik und informationstechnik,* vol. 128, pp. 196-202, 2011.

[8] A. Avizienis, *et al.*, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on,* vol. 1, pp. 11-33, 2004.

[9] P. Cuenot, *et al.*, "Engineering Support for Automotive Embedded Systems - Beyond AUTOSAR," *ATZautotechnology* 2009.

[10] Bate, I., Kelly, T.P. (2003). Architectural Considerations in the Certification of Modular Systems. Special Issue from SAFECOMP 2002 of the Journal of Reliability Engineering and System Safety.

[11] Di Natale, M, Sangiovanni-Vincentelli, A (2010). Moving from Federated to Integrated Architectures in automotive: The Role of Standards, Methods and Tools. Proceedings of the IEEE Vol 98 No. 4 April 2010.

[12] Östberg, K, Johansson R, Use of Quality Metrics for Functional Safety in Systems of Cooperative Vehicles, Proceedings of Workshop on Architecting Safety in Collaborative Mobile Systems (ASCoMS), 2012.

[13] Bergenhem, C, Johansson, R, Lönn, H, A Novel Modelling Pattern for Establishing Failure Models, Proceedings of 31st International Conference, SAFECOMP, 2012.

[14] ETSI EN 302 665, Intelligent Transport Systems (ITS): Communications Architecture

[15] OSEK/VDX Communication Version 3.0.3, July 20, 2004

[16] 1076-2008 – IEEE Standard VHDL Language Reference Manual. 2009

[17] Rushby, John. "Formalism in safety cases." *Making Systems Safer*. Springer London, 2010. 3-17.

[18] Owre, Sam, John M. Rushby, and Natarajan Shankar. "PVS: A prototype verification system." *Automated Deduction—CADE-11*. Springer Berlin Heidelberg, 1992. 748-752.

[19] D2.5 - Definition of failure modes and failure semantics, KARYON FP7-288195

[20] ETSI TS 101 539-3 V1.1.1, Longitudinal Collision Risk Warning (LCRW) application requirements specification, 2013-11

[21] ETSI TR 102 863 V1.1.1, Local Dynamic Map (LDM). 2011-06

[22] D2.4 - First report on the integration of environment models in the architecture, KARYON FP7-288195

[23] Östberg, Kenneth, and Magnus Bengtsson. "Run time safety analysis for automotive systems in an open and adaptive environment." Proceedings of Workshop ASCoMS (Architecting Safety in Collaborative Mobile Systems) of the 32nd International Conference on Computer Safety, Reliability and Security. 2013.

[24] Cuenot, Philippe, et al. "Managing complexity of automotive electronics using the East-ADL." *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*. IEEE, 2007.

[25] Kelly, Steven, Kalle Lyytinen, and Matti Rossi. "Metaedit+ a fully configurable multi-user and multi-tool case and came environment." *Advanced Information Systems Engineering*. Springer Berlin Heidelberg, 1996.

[26] Burrough, Peter A. "MCDONNELL. 1998. Principles of geographical information systems."

[27] Angles, Renzo, and Claudio Gutierrez. "Survey of graph database models." *ACM Computing Surveys (CSUR)* 40.1 (2008): 1.

[28] Vicknair, Chad, et al. "A comparison of a graph database and a relational database: a data provenance perspective." *Proceedings of the 48th annual Southeast regional conference*. ACM, 2010.

[29] Klein, Gerwin, et al. "seL4: Formal verification of an OS kernel." *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009.

[30] Shi, Jianqi, et al. "ORIENTAIS: Formal verified OSEK/VDX real-time operating system." *Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on*. IEEE, 2012.

[31] Rushby, John. "Kernels for safety." *Safe and Secure Computing Systems* (1989): 210-220

[32] Rushby, John. "Logic and epistemology in safety cases." *Computer Safety, Reliability, and Security* (2013): 1-7.

[33] Pierce, Benjamin C. *Types and programming languages*. MIT press, 2002.

[34] Randell, Brian. "System structure for software fault tolerance." *Programming Methodology* (1978): 362-387.

[35] Specification of Watchdog Manager V2.2.0 R4.0 Rev 3

[36] Becker, Jürgen, Thilo Pionteck, and Manfred Glesner. "Adaptive systems-on-chip: architectures, technologies and applications." *Integrated Circuits and Systems Design, 2001, 14th Symposium on.*. IEEE, 2001.

[37] Jusufovic, Marko, and Matthias Nilsson. "Wireless Security in Road Vehicles-Improving Security in the SIGYN System." (2009).

[38] Meyer, Bertrand. "Applying'design by contract'." *Computer* 25.10 (1992): 40-51.

[39] RTCA DO-178B/C, Software Considerations in Airborne Systems and Equipment Certification

[40] Falcone, Ylies, Jean-Claude Fernandez, and Laurent Mounier. "Runtime verification of safety-progress properties." *Runtime Verification*. Springer Berlin Heidelberg, 2009.

[41] Vichare, Abhijat. "The Conceptual Structure of GCC." (2004).

[42] McKusick, Marshall K., et al. "A fast file system for UNIX." *ACM Transactions on Computer Systems (TOCS)* 2.3 (1984): 181-197.

[43] Adl-Tabatabai, Ali-Reza, et al. "Fast, effective code generation in a just-in-time Java compiler." *ACM SIGPlAN Notices*. Vol. 33. No. 5. ACM, 1998.

