Complex Systems Informatics and Modeling Quarterly
CSIMQ, Issue 4, October, 2015, Pages 66-91
Published online by RTU Press, https://csimq-journals.rtu.lv
http://dx.doi.org/10.7250/csimq.2015-4.05
ISSN: 2255-9922 online

Contract-Based Design of Embedded Systems Integrating Nominal Behavior and Safety

Bernhard Kaiser[1], Raphael Weber[2], Markus Oertel[3], Eckard Böde[2], Behrang Monajemi Nejad[1], Justyna Zander[1]

[1] Berner&Mattner Systemtechnik GmbH, Munich, Germany
[2] OFFIS - Institute for Information Technology, Oldenburg, Germany
[3] Carl von Ossietzky University of Oldenburg, Germany

{bernhard.kaiser|justyna.zander|behrang.monajemi}@berner-mattner.com, {raphael.weber|eckard.boede}@offis.de, markus.oertel@uni-oldenburg.de

Abstract. The distributed design process for safety-critical embedded systems has become an increasingly difficult challenge: Electronic Control Units (ECUs) in vehicles, for instance, participate in many vehicle functions, while each vehicle function, in turn, is spread across several ECUs. Many suppliers participate in systems design and many partial functions are reused from past projects, not always knowing the assumptions at the time of their development. In particular, efficient allocation of safety mechanisms and a sound safety case are difficult tasks for original equipment manufacturers (OEMs). Contract-based development has gained popularity as an approach for supporting distributed development by explicitly annotating assumptions and guarantees to components, but an integrated process covering specification of nominal behavior and safety has not been described so far. We present such an integrated development approach that encompasses the systematic breakdown of nominal system behavior using contracts, the consistent derivation of safety analysis by interpreting several types of contract violations as a specification for failure modes, and the subsequent integration of safety mechanisms that cover these failure modes through safety contracts. The approach equally fits hardware and software and is therefore applicable on the system level. We demonstrate it by an electric drive example. The extensibility of our approach towards Cyber Physical Systems, which compose themselves at runtime, is briefly outlined at the end of the article.
Keywords: Embedded systems, functional safety, contracts, component-based development, safety analysis.

1 Introduction

For a long time, embedded systems have been known as small, closed computing systems with a restricted purpose, deeply integrated in products, such as automobiles, industrial machinery, and consumer goods. Over the years, however, embedded systems have grown dramatically in computing power and in the number and complexity of functions they perform. In automobiles or industrial production systems, nowadays many automation functions are spread over a large number of interconnected processing nodes, and their development involves many different suppliers. Typical examples are advanced driver assistance functions as, for instance, automated parking or highway pilot. These functions involve many electronic control units (ECUs), such as

the engine, steering system, braking system, and ECUs for various types of sensors. Each of them, in turn, is involved in performing various functions at the same time: Automated Parking, Lane Keeping Assist, or Adaptive Cruise Control all access the camera set, influence steering and powertrain actuators, and possibly share common processing resources. The complexity of the vehicle architecture is also resembled by lower architectural levels, i.e., on the level of system, hardware, and software architecture within each ECU; because each of these ECUs itself consists of a complex network of components implemented in diverse technologies (e.g., analog and digital hardware circuitry, microcontrollers with a complex software running on them, FPGAs, sensors of different kinds, actuators, mechanical parts). Engineers from various disciplines and companies have to collaborate to get the function working correctly.

Due to their interaction with the physical world, most embedded systems in the automotive, industrial control, aerospace, or medical domains have also high reliability demands and are classified as safety-critical, which requires application of dedicated safety standards, such as ISO 26262 [1] or IEC 61508 [2]. These standards require, on top of a required process maturity and quality management, a systematic design flow, including tracing and justified breakdown of requirements, safety analysis in order to identify potential failures and their consequences, and the definition of safety mechanisms to mitigate the consequences of these failures. All of this must be subject to rigid verification and validation before the start of production for the system.

1.1 Current situation and problem statement

The V-model is today's most common development process model in many application fields of embedded systems, in particular, in the automotive domain. It is also the underlying model recommended in ISO 26262. According to this model, at first, the requirements derived for an embedded system must be correctly captured and then decomposed into allocable sub-requirements and assigned to components. This entails complex design decisions on how to distribute sub-tasks in an appropriate manner, involving, for instance, decomposition of signal accuracies or reaction time onto budgets for specific functional blocks of a controller chain. Usually, negotiation with potential suppliers is necessary about what is feasible. On the one hand, the inherent collaboration require formalisms that all parties understand (e.g., based on textual languages or graphical notations) that are, both sufficiently intuitive and sufficiently expressive to deal with all kinds of components and their various properties. On the other hand, it is desirable to aim at formalized notations wherever possible to avoid ambiguities and misunderstandings and in the ideal case to perform formal verification of the relevant correctness and safety properties.

Unlike suggested by the V-process, industrial practice today differs in several areas:

- **No strict top down process**: Often, a predecessor system already exists and shall be extended or adapted, or at least partially reused from previous projects (we could call this practice a *bottom-up process,* because it builds the system architecture out of existing building bricks and evaluates the achievable overall functions at the end). In any case, the development is an iterative and sometimes tentative process: it may run into a dead-end, requiring the architect to revise former decisions, resulting in an alternation of top-down and bottom-up phases. All of these introduce deep iteration cycles in the process where incompatibilities between components are only detected during the integration phases of the ascending branch of the V.
- **Re-Use of components or usage of commercial-of-the-shelf components**: There are cases, in which the engineer decides to use components originally designed for another system or a different purpose. In order to ensure compatibility, rigorous analysis is required. In many cases, however, specific details about the implementation may not be available due to trade secrets and intellectual property rights, which implies a black box view.

Contract-based development (see, e.g., [3]) appears as a suitable framework for addressing these issues. Contract-based development extends component-based development by stating assumptions and guarantees for any component on any level of the architectural hierarchy. Guarantees make explicit what behavior and service quality a component, which can be regarded as a black box, exhibits at its outer interface. Formalized assumptions state what the developer has been taken for granted when developing the component. Contracts can be used in a top-down and bottom-up process, and, depending on the language used for stating them, more or fewer checks for correctness of the refinement and composition of components can be automated.

Contracts have gained high acceptance as an intuitive formalism, and semi-formal languages have been proposed to make them applicable for engineers from different disciplines without formal background (see previous work [4]); assertion specification languages today can cover many relevant properties, such as timing or accuracy of values.

Contract-based development is based on the idealistic assumption that contracts verified during the design phase are always fulfilled during operation. Safety, however, needs to consider the impact of failures that will lead to a violation of contracts during operation. Hence, we identify the third problem:

- **Alignment between nominal behavior and safety aspects**: Functional safety is still handled quite distinctly from the development of the nominal function. The models used for safety analysis are often not aligned with the system models, and a consistent integration of system architecture, safety analysis and technical safety concept is often missing. This leads, in the best case, to inefficient processes, but, in the worst case, to safety risk due to inconsistencies or overlooked failure possibilities. Consistent integration of nominal function development and the safety process is still not achieved, as there is no formal passage from specification of the intended system to safety analysis models that identify situations where the system does not behave correctly and to safety-centered design extensions that assure a sufficient level of safety even in the presence of failures.

To boost integration and efficiency of the safety process, it would be greatly beneficial to provide a safety engineer with techniques that integrate well with the component-based design of the system and refer to the components, ports, signals, assumptions, and guarantees for formalizing failures. The safety-focused design process (i.e., Safety Analysis and creation of Functional and Technical Safety Concept acc. to ISO 26262) could in turn profit from using the same contract-based methodology as the development of the nominal function, but in this instance involving safety contracts, for which typical patterns already exist. Like the development process, the safety process is also tentative and iterative (note that the safety mechanisms themselves may also be subject to failure or even introduce new hazards) and continues in cycles until repeated analysis shows that the risk of remaining safety violations is sufficiently low, which is determined by some qualitative and quantitative criteria provided in the safety standards, such as ISO 26262 or IEC 61508. A safety-integrated development process will have to reflect all of these needs.

1.2 Contribution of this article

We propose a component-based approach addressing the challenges stated in Section 1.1 using contracts. Our special attention is on the integration of the safety aspect. Using contracts also in the safety domain is basically not new (see discussion below), but some aspects have not yet sufficiently been addressed.

In this paper, our contribution is three-fold:

- We summarize our recent work in the area of contract-based development of safety-critical embedded systems, mainly resulting from the SPES_XT research project [6]. In particular,

we refine and extend our approach by converting component contracts and interface contracts into each other. These contracts are integrated in a rigid procedure for refining system requirements and allocated to components. In this regard, this paper is an extension of our previous publication [7].

- We introduce an innovative approach that bridges the gap between the nominal function and the safety concept development of the system by automating parts of the safety analysis: we interpret different types of contract violations (i.e., failure to deliver the specified behavior for the nominal case) as formal definitions for failure types, which are then fed into a safety analysis based upon the Component Fault Tree (CFT) technique. This approach may save much of the work necessary for safety analysis by not only building upon the system architecture model created during the normal development, but also profiting from the definition of guarantees, examining the various possibilities for their violation based on a failure type system related to the type system of different kinds of signal flows.
- Finally, we combine these two steps into a novel design- and safety-process for complex systems to make it applicable to industry practitioners, covering (1) systematic decomposition of the requirements for the nominal function onto components, (2) systematic analysis of component failures and their consequences, and (3) systematic definition of safety requirements to cope with these failure consequences and decomposing and allocating these onto (safety) components in the same manner as in step (1).

1.3 Outline

The rest of the paper is organized as follows: in Section 2, we provide a basic overview on contracts-based design, specification languages, and related work. Section 3 introduces our approach for applying contracts and explains the differences and relationship between component and interface contracts. Section 4 illustrates how to apply them in a structured and modular development process. In Section 5, an example of an electric drive system is used to demonstrate the application of our approach. Section 6 concludes the paper and gives an outlook on our current research to extend our approach to the field of cyber-physical-systems, which assemble at runtime using insecure and unreliable communication links.

2 Fundamentals and related work

In this section, we give a short introduction to contracts and contract-based design along with some validation methods. Additionally, specification languages for contracts are discussed and component-oriented safety analysis techniques are briefly outlined. Since these topics are quite extensive we will also refer to some fundamentals and only shortly outline related work.

2.1 Contract-based design

Contracts have initially been proposed by Bertrand Meyer for verification of sequential software programs, using preconditions (that must hold at program entry), postconditions (that must hold at program exit) and invariants (that must hold all the time). Later, the idea of contracts has been transferred to component-based software and system development and called “contract-based design”. The paradigm of component-based system development defines systems as hierarchical compositions of components that exchange information, energy, and/or mass flow at their interfaces, also called ports. Contracts are assertions that allow formulating black-box specifications of components. They explicitly distinguish between assumptions about their operational environment and the guarantees they provide under the condition that these assumptions are fulfilled. Following the definition in [3], contracts explicitly handle pairs of

properties, respectively representing a system's or component's assumptions on the environment and the guarantees that this system or component promises, provided that these assumptions hold. The separation into assumptions and guarantees serves as a foundation for building a sound theory that allows the reasoning about the composition of systems in a formal way, provided that formal specification languages and proof systems are applied. But even if applied in a semi-formal way or using natural language, contract-based development can be beneficial for supporting a structured way of communicating the expectations of component manufacturers and system integration and still offer the possibility of verification by human experts [4]. In the past couple of years the research on contract-based systems engineering has increased dramatically (for an overview see [3]). The preconditions are interpreted as assumptions of a component on the signals provided at their input-ports and their operational environment; the post conditions are guarantees that the same component is able to fulfill. Accordingly, contracts are matching pairs consisting of an assumption (A) or a conjunction of several assumptions, and a guarantee (G) as provided in Figure 1. The assumption specifies how the context of the component, i.e., the environment from the point of view of the component, should behave. Only if the assumption holds, then the component will behave as guaranteed. This kind of specification allows replacing components by other ones with the same purpose and compatible interfaces, if they accept the same or weaker assumptions about the environment and provide the same or stronger guarantees towards the environment. A complete re-validation of the entire system after the exchange of some of its components is not necessary if the new components fit into the old contracts, and if they have been verified on their own to fulfill their guarantees, provided that the assumptions hold. This can reduce costs dramatically in case of later changes to the safety-critical systems requiring proper certification. The essential point is that the system decomposition can be verified with respect to contracts without the knowledge of the concrete implementation.

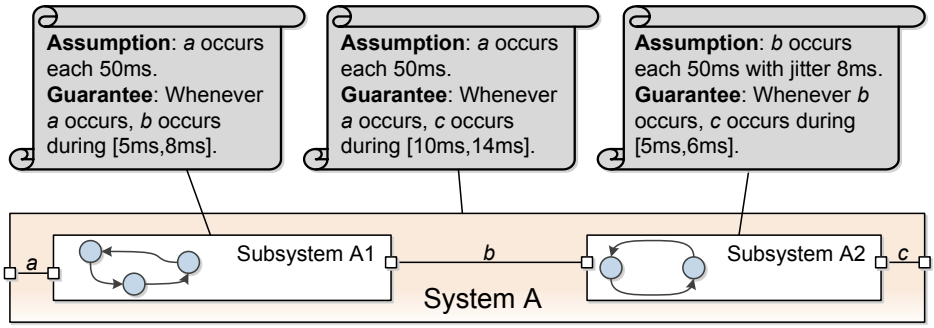

Figure 1. Example for a contract specification.

One advantage of using contracts is that they can help decrease the complexity of verifying the implementation against its specification. For example, consider the system in Figure 1, to which a contract is assigned. The system contract states that the system expects a triggering of the input port 'a' every 50 ms, and, when it is triggered, the system has to respond by sending an event on port 'c' within a specific time interval. The system is decomposed into two subsystems each with one contract, and some internal behavior modeled, for instance, by state machines. Assume that the functionality on subsystem A2 depends on the output of subsystem A1. Further, assume that the subsystems would *not* be annotated with contracts. Thus, to validate the contract of the overall system A, the composed behavior of both subsystems has to be computed, which generally leads to large state spaces. Using contracts for A1 and A2, we can omit the composition and validate the sub-contracts locally.

2.2 Formal underpinning and validation of contracts

Formally, contracts are defined as pairs of assumptions and guarantees: $C = (A, G)$. The refinement relation between two contracts C and C' is defined as follows: C'refines C, if $A \subseteq A'$ and $G' \subseteq G$. The validation of contracts works as follows: When a component is decomposed into a set of sub-components, we have to check whether the overall contract $C = (A, G)$ (which also will be called *global contract*) and all subcontracts $C_i = (A_i, G_i)$ (also called *local contracts*) for $i \in \{1, \dots, n\}$ are consistent. We check the following virtual integration condition:

(1) $i)\ A \wedge G_1 \wedge \dots \wedge G_n \Rightarrow A_1 \wedge \dots \wedge A_n$

(2) $ii)\ A \wedge G_1 \wedge \dots \wedge G_n \Rightarrow G.$

An in-depth discussion about virtual integration can be found in [8]. In [8] contracts were extended by so called *weak* assumptions. Weak assumptions are used to describe a set of possible environments in which the component guarantees different behaviors. This separation is only methodological, and does not affect the semantics of the definition of the original contracts: Let $C = (A_s, A_{w_1}, \dots, A_{w_n}, G_1, \dots, G_n)$ be a contract consisting of a strong assumption A_s, a set of weak assumptions A_{w_i}, and a set of corresponding guarantees G_i for $i \in \mathbb{N}$. Semantically, we map C to a standard contract of the form $C' = (A_s, G)$, where $G = (G_1' \wedge \dots \wedge G_n')$ $G_i' = (A_{w_i} \Rightarrow G_i)$. For timing and sporadic fault occurrence recognition the validation has already been done in [9]. For other aspects this may, however, be more difficult, which is one reason to also consider a methodological approach. We propose the methodological separation between component and interface contracts described in Section 3.

2.3 Related work regarding contract-based development

Contract-based design has shown many evidences of its applicability in industry. There have been proposals for the usage of contracts to specify real-time properties of continuous-valued controller structures and the control error of technical systems (e.g., [10]). Contracts have been applied to UML/SysML models as well as Simulink models (e.g., [11]). Contracts can be specified in a natural language, in a set of semi-formal languages (such as template-languages) or in formal languages. Natural language contracts are often accompanied by ambiguity, incompleteness, or inconsistency. Some proposals have been made with semi-formal languages (the syntax is defined and restricted, but verification has to be performed by human experts) to avoid these drawbacks of natural language while providing an understandable language for experts from different domains [4]. Text patterns, consisting of static text elements and attributes, also provide well-defined semantics so that a consistent interpretation of the system specification between all stakeholders can be ensured. To cope with the needs of the different aspects of a design, various sets of patterns have been defined [12], they build upon parametrized requirements patterns that have been known for a long time (e.g., [13], [14]). Many research contributions about contracts rely on formal languages, such as temporal logics [15] or IO-Automata [16]. Temporal logic is used in [17] for decomposing the system architecture with contracts. The framework automatically generates a set of proofs. Formal languages allow automatic verification of refinement and implementation of contracts, but they are often hard to understand for practitioners from the different disciplines involved and therefore it is difficult to promote them in industry. Moreover, depending on the language, their expressive power is more or less limited. For instance, many of them cannot deal with real time and with continuous values, which are required for describing mechatronic systems. A proposal that bridges this gap is the pattern-based Requirements Specification Language RSL [18] or the Contract Specification Language CSL from the SPEEDS project [19], which provide parametrized text patterns in a well-understandable language while providing formal semantics.

Contract-based design has been proposed also in the functional safety domain (e.g., [20],[21],[22] and many others). Safety ADD (see [23]) helps to define and verify the safety contracts for software components in a graphical editor. The algorithm traverses all assumptions and guarantees to make sure that they match. A tool for checking the refinement between contracts called *OCRA* (Othello Contracts Refinement Analysis) was presented in [8]. It provides means for checking the contracts specified in a pattern based language called *Othello,* which are later translated to a linear-time temporal logic for discrete and real-time constraints. The underlying engine allows reasoning whether contract refinement is correct. A full range of safety mechanisms, such as definitions of faults and failures, fault containment, safety mechanisms, handling the degradation modes and safe states at multiple abstraction levels, is proposed in [24]. The interface between the safety view and the functional design is highlighted as well. Multiple safety patterns are provided in LTL [15] notation.

2.4 Component-oriented safety analysis

As the aim of this paper is an integrated process for designing the nominal system function and safety aspects, we now draw attention to safety analysis. Here we can build upon a long series of research contributions aiming at integrating safety analysis techniques and component-oriented design of embedded systems. Safety analysis of various types is a mandatory activity in all relevant safety standards. It serves for the purpose of systematic identification of faults or failures that could lead to the violation of safety goals or safety requirements and for identifying their root causes (cf. [1], Part 9, Clause 8.4.9). There is a wide range of applicable techniques; most of them can be classified as either deductive (or "top-down"), a popular example being Fault Tree Analysis (FTA) that reversely searches potential reasons for given hazards, or inductive ("bottom-up"), a popular example being Failure Mode and Effects Analysis (FMEA) that identifies basic failure modes and searches for their potentially dangerous consequences.

Over the past decades, many proposals have been presented to better integrate systems design and safety assurance, and to make safety analysis techniques more modular and consistent with the system architecture. One of the first approaches building upon the assumption that failures propagate along the interfaces between components was Hazard and Operability Studies (HAZOP) [25], which originally came from chemical industries. Another example of an early component-based failure analysis techniques is Failure Propagation and Transformation Notation (FPTN) [26], which highlighted the fact that failures propagate along the signal paths from one component to another, which means that failures at component output ports can either be caused internally or propagated by failures at the input ports. Moreover, failures can also be transformed into other failure modes (e.g., a too low actual value at the input of a proportional-integral controller will result in a too high actuator command at its output) or even mitigated (no failure mode is observable at the output of a component, although there is a failure present at the input). This observation will be used when we propose our approach for a contract-based safety concept in Section 4.3. A component (e.g., a sensor) that cannot guarantee that a failure mode at its output never occurs, can still be used in a safety-critical system in combination with another component downstream in the signal flow, which is capable of detecting and mitigating the failure (e.g., by model-based diagnostics and the possibility to mark the output value as "invalid", if this leads to a safe state of the system).

Interface-focused FMEA (IF FMEA) as a part of Hierarchically Performed Hazard and Propagation Studies [27] is another analysis technique that investigates failures by port interfaces of components. In the same way as IF FMEA interprets the traditional FMEA technique onto components with interfaces, Component Fault Trees (CFTs) [28] extend traditional Fault Trees by introducing a concept of components, along with "failure ports", by which failures propagate from one component to another. The integration of CFTs with rich component models was further improved by [29]. Meanwhile, prototypical tools have been presented that generate CFT

structures from SysML [29] and from Simulink [30] system models. In [29], the failure keywords from HAZOP were used with the failure classification scheme from [31] to propose the unified hierarchical failure type system shown in Figure 2, which will serve as the contract-based failure classification in Section 4.2.

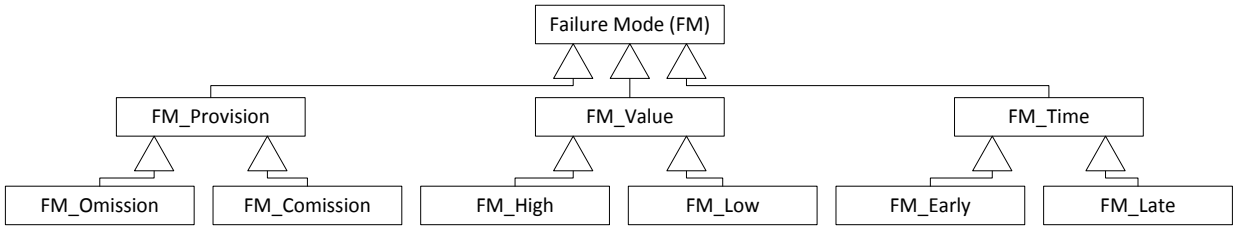

Figure 2. Failure Type System (adapted from [29])

[29] further proposes a sub-refinement of failure classes using attributes (e.g., to what extent is it too high?). If necessary, failure modes can be sub-classified, in particular, into safe and dangerous failures, e.g., "speed too high by more than 5% but not more than 20%" can be labelled "safely too high", "speed too high by more than 20%" can be labelled "dangerously too high".

A general issue when deriving failure propagation models from signal-flow-oriented component-based system models is that the system models in most cases contain cycles (e.g., closed loop control), but failure models, such as CFTs, are restricted to directed acyclic graphs. This problem and possible solutions have been discussed in [29] and [30].

3 Development process using interface- and component-contracts

This section explains how to capture, refine, and allocate requirements using contracts while creating the hierarchical system architecture. Our aim is addressing both the top-down process from requirements towards implementation by step-wise refinement (see Section 3.4) and the bottom-up process where a system is composed of pre-existing and pre-qualified components described by assumptions and guarantees.

The term "multi-aspect" contract refers to the possibility to make assertions about different system aspects, as for instance:

- Reactions on stimuli or conditions becoming true
- Timing aspects in terms of delays between stimuli and reactions
- Signal quality, e.g., accuracy of values w.r.t. a reference value in the physical world

We distinguish two flavors of contracts that have their specific advantages at different points of the development process: component contracts and interface contracts.

3.1 Interface contracts

The relationship between assumptions of a specific component to guarantees of its neighbors (i.e., predecessor and successor in terms of signal flow) can best be captured by an *interface contract*. Each assumption is linked to one specific input port and each guarantee is linked to one specific output port. The assertions to be expressed may be of the same type, and the language for assumptions and guarantees may also be the same. The guarantees of a signal producer component must imply the assumptions of the corresponding signal consumer component(s). Interface contracts are often made between neighbor components, which share the same super-component. However, if an input comes directly from an input port of the super-component or an output feeds directly an output port of the super-component, the stated assertions are propagated to the next higher level in the system hierarchy and matched with the super-component's

assumptions and guarantees. Unfulfilled contracts, i.e., assumptions that are not satisfied by corresponding guarantees can be detected and highlighted automatically, which has been demonstrated by a prototype tool in [4]. Figure 3 illustrates the concept of interface contracts.

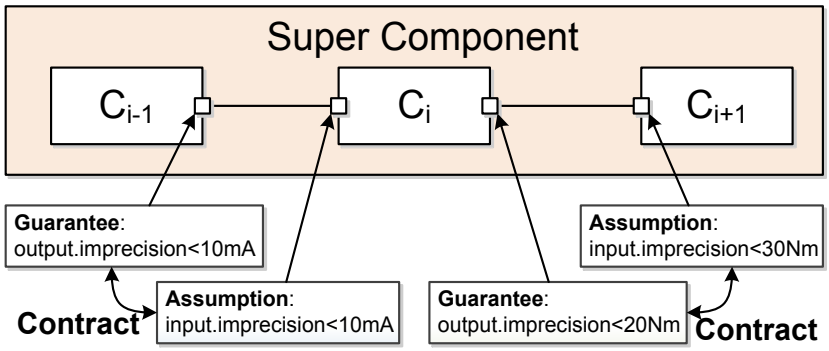

Figure 3. Signal quality example of interface contracts

Assumptions and guarantees for interface contracts are easy to derive for a system architect, if he has denoted assertions to each signal at certain "probing points". E.g., a signal representing a physical event (e.g., acceleration value getting higher than a specified limit) can be tagged with an accuracy or delay assertion. This assertion marks the weakest acceptable guarantee for the component that produces the signal at its output port, and the strongest assumption for the component that consumes the signal at its input ports. Specifying the meaning of such assertions requires sometimes referring to signal characteristics at a given point, as seen by an omniscient external observer (a role, which is actually taken by the system architect). In case of accuracy, for instance, this might be the absolute difference of the signal at that point with respect to some physical quantity in the physical world, in case of delay – the physical time elapsing between a condition change in the outside world and an event (e.g., voltage edge or software service call) generated at some output port of the component.

3.2 Component contracts

What is an advantage of interface contracts for the system architect as an omniscient observer, turns out to be a drawback for the component supplier: when the aim is to treat components as reusable entities out of context, it is not possible to reference to any part of the physical environment outside the integrated system. Instead, other types of requirements are preferred, for example, *"An event at the output shall be issued no later than x milliseconds after some condition noticed at the input"*. These are the types of guarantees the component may promise, under no assumptions, or under other assumptions regarding environmental conditions (e.g., environmental temperature, voltage supply, processor resources) or assumptions about input signals (e.g., events not occurring more often than with a certain frequency, values staying in certain ranges). To specify contracts in this style, we recommend the usage of component contracts. *Component contracts* are contracts between a system (e.g., a component) and its operational context. They allow assumptions and guarantees affecting the component as a whole (e.g., environmental conditions that do not relate to any specific input or output ports) and allow for relationships between behaviors at two or more input or output ports.

3.3 Passing from interface to component contracts

There exists a semantic relationship between component contracts and interface contracts. The meaning of an interface contract is that the component asserts to behave in a way, such that certain guarantees associated with provided signals at some of its output-ports are implied by the

fulfillment of a set of assumptions assigned to delivered signals to some of its input-ports. Hence, a component C_x has to fulfill the following specification:

(3) $$C_x\colon A_1(C_x.In_1) \wedge \ldots \wedge A_m(C_x.In_n) \Rightarrow G_i(C_x.Out_j)\ .$$

When integrating the component C_x into a system environment, the inputs and outputs of different components are connected to each other. For instance, let the input In_1 of the component C_x (we will use the notation $C_x.In_1$) be connected by a signal to the output Out_1 of the component C_w, and the output Out_1 of the component C_x to the input In_1 of C_y, then these components engage in interface contracts with each other:

(4) $$\text{Interface Contract 1: } G_1(C_w.Out_1) \wedge \ldots \wedge G_m(C_w.Out_1) \Rightarrow A_1(C_x.In_1) \text{ and the like for all } A_i \text{ of } C_x.In_1$$

(5) $$\text{Interface Contract 2: } G_1(C_x.Out_1) \wedge \ldots \wedge G_m(C_x.Out_1) \Rightarrow A_1(C_y.In_1) \text{ and the like for all } A_i \text{ of } C_y.In_1$$

Each interface contract is made between neighbor components (interface contract 1 between C_w and C_x, interface contract 2 between C_x and C_y), without involving the super component (or system).

Let us assume, components are constrained to negotiate only with their super-component, being unaware of their (future) neighbors. This is a typical case for a Component-out-of-Context supplier. How would the same set of statements be interpreted? The environment (i.e., super component, including all neighbor components supposed to be there) would guarantee to the candidate component C_x that all of its assumptions are met, whereas requiring from this component to fulfill all of its guarantees. The candidate component C_x, in turn, must assert that it assures all of its guarantees, provided that its assumptions hold. This would be equivalent to the following statement:

(6) $$C_x\colon true \Rightarrow \left[A_i(C_x.In_1) \wedge \ldots \wedge A_m(C_x.In_n) \Rightarrow G_i(C_x.Out_j)\right].$$

From C_x's point of view, this can be interpreted as part of a component contract, which would mean in natural language as: "I assume nothing specific and I guarantee that I will react on input signals with properties A_1,…, A_n at my inputs by providing output signals compliant with G_i at my output [within the defined time span, with the defined accuracy and so on]". Of course, in most practical cases the assumptions for any given technical component will not be empty (true), but contain general conditions for well-functioning of the component (e.g., supply voltage or temperature ranges for hardware components, and memory budget or correct scheduling for software components). The distinguishing feature, however, is that now the component engages in a contract with its respective environment and doesn't need to know its neighbors and that the assumptions and guarantees refer to the component as a whole and not to one single port (i.e., may state input-output-relationships).

3.4 Proposed top-down process

In the following, we propose a sequence of rigorous activities for the development process. We first address the top-down development process as described by the V model, i.e., recursive refinement and allocation of requirements for some technical system to the proposed system architecture. Then, we mention the aspects of verification of a correct decomposition, followed by a level allowing the technical design and implementation. This verification will include the verification that the components are compatible to each other at their interfaces, as well as the verification that the combination of the given components, in exactly the way defined by the

architecture, assures fulfillment of the requirements (guarantees) of the super-component, provided that the assumptions of the super-component are fulfilled. We will use interface contracts and component contracts during these process activities.

Starting Point: Let S be some system to be developed with given external interfaces, a given set of requirements regarding its behavior (this could be some signal to be provided continuously according to a control law with a required accuracy, or an event to be triggered with a specified maximum delay counted upon some external condition becoming true) and a given set of assumptions about its environment (e.g., some external values always being within certain limits, some external events occurring sporadically with a minimum time interval in between, etc.).

The task is to decompose the requirements, to propose a suitable architecture consisting of components and connections in between, and then to allocate the decomposed requirements onto the designated (sub-) components C_1 … C_n of the system, thereby verifying that the refinement is correct. This refinement continues recursively to subcomponents, sub-subcomponents etc., until a granularity of components is reached that allows to switch to a more technical viewpoint, where each components is associated with an implementation either as a hardware component or as a software component which is deployed to some hardware component.

For doing so, we propose to proceed as follows in Figure 4, which is based on a simplified and fictitious example of an airbag system:

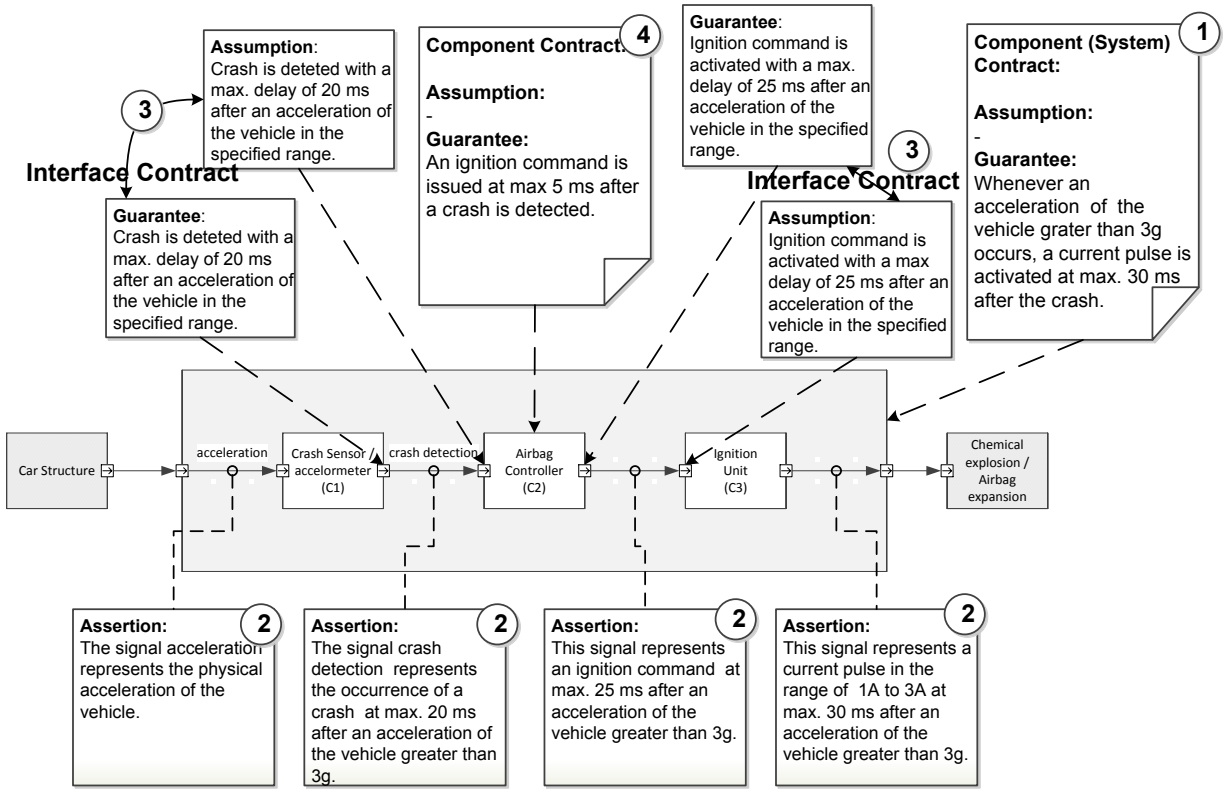

Figure 4. Use of contracts in the design of a Car Airbag System.

1. The requirements engineer receives the requirements usually in some informal notation or natural language, checks them for completeness, consistency and understandability, clarifies remaining questions, and finally rephrases the requirements as a set of atomic assertions in a selected notation (RSL or CSL, for instance). These requirements become the guarantees that the whole system will have to fulfill. Similarly, the assumptions about the operational environment (e.g., value ranges of inputs, occurrence frequency of events) are clarified and captured in the same notation; these form the assumptions for the whole system (see No. 1 in Figure 4). These assumptions and guarantees constitute the first *component contract,* where the system to be developed is considered as the subject component and any operational context, under which it will have to operate, is considered

as its environment. If assumptions and requirements are not yet fully known, guesses must be made and revised in later design iterations.

2. The architect decomposes the system into components, specifying their interfaces (input and output ports) and use the mass/energy flows or signals that connect the components via their ports. To each component and each signal its respective purpose or meaning is assigned (which can be in a natural language description, such as, e.g., "detects a crash event by evaluating the acceleration sensor value"). All signals are listed in a signal dictionary. A structured way for deriving functional architectures in SysML and refining requirements from use cases is explained in [32]. In this process step, not only the qualitative functions (what has to be performed?) need to be decomposed, but also the quality properties (such as timing or accuracy), which often needs budgets (e.g., total reaction time) to be distributed among the components in a processing chain. This usually involves consultations and negotiations with domain experts or with the suppliers or designers of the individual components. The architect may structure this negotiation process by tentatively specifying assertions that are meant to hold at different "probing points" (signals) in the architecture (see No. 2 in Figure 4). The architect acts in this step as an "omniscient observer", i.e., any event or physical magnitude at any point of the system or even in the environment of the system may be referred to.
3. In the architecture, each signal is produced at exactly one output port of one component, and it is consumed by one component or by several components at some of their respective input ports. Thus, the assertions from the previous step can be interpreted as *interface contracts* as follows: The producer of a signal must guarantee its specified assertions (or something that is stronger than that), whereas the consumer of the signal may rely on the fulfillment of this assertion (or something weaker as that) as an assumption (see No. 3 in Figure 4). For example, if the assertion regarding the "crash detection" signal was "the maximum delay after a crash is 20 ms" and we know that this signal is produced by a component "Crash Sensor" and consumed by a component "Airbag Controller", then we can annotate the *assumption* "the maximum delay after a crash is 20 ms" to the respective input port of the "Airbag Controller" and the same statement as a (yet unconfirmed!) guarantee to the respective output port of the "Crash Sensor".
4. As mentioned in Section 3.2, interface contracts are benefiting for the system architect but not for the component suppliers. For instance, the architect of a car airbag system can claim that an ignition command at the input of the "Ignition Unit" is provided at maximum 25 ms microseconds after occurrence of a high acceleration of the vehicle. The manufacturer of the "Airbag Controller" that issues the ignition command, can, however, give no guarantee regarding this time, simply because he has no knowledge about the acceleration of the vehicle and the performance of other components in the overall system. He can guarantee, however, that an ignition command is issued at maximum 5 ms after the acceleration value at the output exceeds the limit defined as a crash condition, i.e., confirm a defined relationship between events at its input and output. This means that a transition from interface contracts to component contracts is made (see No. 4 in Figure 4). Hence, it is possible to specify sub-components as reusable standalone components, which can be bought from some external supplier.
5. The resulting component contracts for the components on the low level of the architectural hierarchy only consist of propositions that refer to the observable behavior at the outer interface of the component of interest and describe its input-output-behavior. Therefore, the guarantees that the component has to fulfill can be formulated as requirements to the component. In addition to the requirements, the assumptions granted to the component can also be delivered to the supplier to provide certainty about properties of the usage environment and allow him some optimization by excluding irrelevant environmental conditions.

Actually, there are two different engineering steps occurring in top-down development: refinement (passing from one level of hierarchy to the next lower level) and concretization (passing from the functional specification to a technical solution, by allocating functions to actual software functions or electrical or mechanical components). We assume that usually the refinement comes first and the allocation onto technical component takes place only on the lowest level of the hierarchy. However, we observed in many industrial projects that both aspects are often intertwined, which is not an obstacle for the application of our approach. An example of concretization and negation for a car braking system in the context of the AUTOSAR framework can be found in Section XII of [3].

3.5 Adaptation from a top-down to a bottom-up process

As mentioned in the introduction of this article, the top-down process from requirements to implementation as proposed by the V-model is by far not the only practical development process. Very often, components are reused from past projects or bought off-the-shelf, or components involving innovative technologies have been developed out-of-context by suppliers and are then proposed to various original equipment manufacturers (OEMs) to build solutions around. Many systems in automotive and other industries are, from the beginning, planned as extensions or variant projects on the basis of products that are already in the market. A flexible development process should be able to reflect all of these cases and allow bottom-up proceeding as well. The advantage of the proposed proceeding is that it works in both ways, since the transfer from interface contracts into component contracts can be reversed. Therefore, it is adaptable to any mix of top-down and bottom-up activities, including iterations of top-down and bottom-up phases. The steps introduced in Section 3.4 can be used in a rearranged sequence in order to handle different cases.

4 Integration of safety assurance with development

Safety could, at first glance, be considered as just another qualitative property of the system. For example, the Automotive Safety Integrity Level (ASIL) might be annotated as an attribute of a signal. But even if an ASIL-attribute somehow represents a suitable measure for safety, making a system safe involves much more, in particular, a systematic search for hazards that could arise from using the product, a systematic derivation of safety requirements to counteract these hazards, and their application to a next iteration of the design, in addition to the existing requirements regarding nominal behavior. While systematic faults in system design and software implementation might be reducible to a large extent by sound processes, failures of hardware parts will remain unavoidable. This forces us to change our focus to consider also failures in the system. A failure is defined as “a transition from correct service to incorrect service, i.e., to not implementing the system function” or the “termination of the ability of an element to perform a function as required” [1]. The system function is thereby defined as “what the system is intended to do and is described by the functional specification in terms of functionality and performance” [33]. This suggests that a failure is defined by the fact that the guarantees from Section 3 are, for some reason, no longer fulfilled. In such case, failures may be defined as contract violations.

Accordingly, the safety process entails an additional iteration loop in the system development process, considering failures and requiring additional safety features. These may manifest either in new safety requirements for the existing components, or in introducing some extra components because of safety issues (e.g., plausibility checks, redundant units). We suggest a modular top-down safety analysis applicable at the functional level, allowing early investigations and passing interfaces for component-based safety analysis down to the supplier, even before the technical solution has been defined.

The steps of this proceeding are in essence the ones prescribed by ISO 26262. They involve formulating safety goals, performing safety analysis and setting up a safety concept that defines mechanisms to enable the system to deal with failures at runtime. To benefit from the existing contracts for safety activities, we propose the following three-step approach:

1. Derivation of safety goals using contracts (see Section 4.1)
2. Contract-based modular safety analysis (see Section 4.2)
3. Definition of safety mechanisms by reusing the safety contracts (see Section 4.3)

4.1 Derivation of safety goals

During Hazard Analysis and Risk Assessment (HARA), all hazards of the system are searched and ranked by an (Automotive) Safety Integrity Level (SIL or ASIL, ranging from 1 to 4 or A to D, depending on the applicable standard). From the hazards, top-level safety requirements (i.e., safety goals) are derived. They typically state that the hazard shall be prevented. A safety goal comes with an annotation of a "safe state" and a fault tolerance time (FTT), i.e., a specification for how long can the hazard be tolerated without an accident occurring. Within this timeframe the safety mechanisms will have to react [1]. A sample safety goal for an electric drive system is "Overspeed of more than 20% shall be prevented [ASIL C]; Safe State = Motor Shutoff, FTT = 200 ms".

In many cases, safety requirements differ from those specifying the nominal customer or market requirements not in their style, but just in the values of their attributes. For instance, regarding customer satisfaction, the speed accuracy of some powertrain controller shall be in the range of +/- 5% (in order to avoid noise, vibration, or other discomfort), but only, if the accuracy gets worse than +/- 20% , the vehicle becomes unstable or uncontrollable, which constitutes a hazard. Hence, we might obtain two different guarantees for the same signal, e.g., the motor speed: first, without a safety attribute, to be within +/- 5% accuracy, and second, with a safety attribute (e.g., ASIL C) to be within +/- 20% accuracy. Leaving the range of +/- 5% is a failure regarding nominal behavior (a pure quality issue), but leaving the range of +/- 20% is a safety-critical failure. This gradation will later turn out to be helpful during safety analysis (where distinguishing safe from dangerous failures is required).

Most safety requirements follow a few patterns, for example: "It shall always be assured that <condition> is true", "It shall never occur that <event> occurs", or "Whenever <event A> occurs, the system shall react by <event B> within <time>". Requirements expressible in terms of these patterns can be stated as guarantees. According to most relevant standards, the terms "*always"*, "*never"* and "*whenever"* have to be interpreted in a way that still there remains a non-zero, but acceptably low probability that the safety goal is violated due to uncovered failures. Safety goals are usually weaker requirements than the requirements for the intended function. For example, instead of requiring that the motor speed shall always be within some accuracy limits, a safety goal might only require that over-speed above a certain limit shall be prevented. This means that something fundamental changes: we now have to allow weakening of guarantees. It might look as if adding safety makes the system worse, instead of better, but this paradox can be easily resolved: the system has been that bad before, but due to the assumption of idealistic components, throughout Section 3 we have simply ignored this. Taking the safety perspective, makes us accept that failures are unavoidable, which applies, in particular, to probabilistic failures of hardware parts through wear and tear (all parts *will* eventually fail). Inserting safety mechanisms into the system helps assuring the safety goal, even in presence of failures. To this end, safety analysis have to be carried out in order to learn what failures have the potential to violate any safety goal. A full refinement of safety requirements is only possible once there is a technical specification and requires assigning function blocks to technical components in hardware or software. After this step, auxiliary components appear in the architecture. For instance, a function block constituting a proportional-integral controller (PI-

controller) specified as a Simulink block may be compiled into an executable C language software function, which is then deployed to a microcontroller. The assumption in the component contract of the function block may have been empty ("true"), which means that nothing particular is assumed for the function block to exhibit the specified input-output correlation. Now when the same block is implemented as a software component, it will only work correctly under the *technical* assumption of a correct execution environment. This includes aspects, such as regular scheduling by the OS, sufficient memory space, no corruption of memory space by other components, and a correctly working CPU. The latter, in turn, depends on environmental conditions, such as clock frequency, power supply voltage accuracy, and environmental temperature. All of these aspects must be assumed in component contracts, and guarantees shall be given regarding their technical fulfillment. Violations of these guarantees shall be subjected to safety analysis and shall be treated in the safety concept.

4.2 Modular safety analysis

Now, we can push the integration of safety analysis with development activities one step further by exploiting the existing contracts for the formal definition of functional failure modes. When there is a guarantee that the output voltage of a power supply, for instance, shall always be between 10 V and 14 V, then a value of 9.9 V is obviously "too low" and a value of 14.1 V is "too high"; these indicate the two types of *value failures* according to the typology in Figure 5. Similarly, if an event is guaranteed to occur within a defined time interval, it can occur "too early" or "too late", indicating the two possible *timing failures*. If an event is guaranteed to occur under certain circumstances, but it does not, this is seen as an *omission failure*. If an event occurs, although under the given conditions there is an assertion that it shall not occur, this is called a *commission failure*.

Matching these signal types and their corresponding types of contract patterns with the potential failure modes (cf. Figure 5) propagated at component ports assures type compatibility and restricts the set of potential failures to be considered. Eliminating irrelevant failure modes saves effort and increases consistency and understandability. Moreover, a formal definition of failure modes is given by the boundaries specified by the assertions for correct nominal behavior. Some safety standards require putting safe and dangerous failures into relation with each other and calculating a "Safe Failure Fraction" or similar metrics. The distinction is made by the different attributes for failure modes as proposed by [29] (see Section 2). Dangerous failures are those that constitute a violation of a guarantee derived from a Safety Goal as described in Section 4.1. They require countermeasures in terms of safety mechanisms, whereas safe failures usually do not require any safety mechanism in the safety concept.

Of course, the presented failure type hierarchy is a generic one and might be adapted or extended; this in no way diminishes the applicability of contracts for failure definition, as the following examples illustrate. Let a contract assertion define the maximum harmonic distortion or the maximum noise in terms of a power density over frequency for a continuous signal. Then, new failure modes, such as "distorted" or "noisy" may be defined. Let an event be constrained by a contract to occur periodically with a maximum jitter; then, "uneven" may be an applicable failure mode. Let an event be constrained by a maximum number of occurrences during a time interval; then, "too often" may be appropriate.

Passing from a component-based system architecture to a CFT failure analysis according to the procedures developed by [29] and [30] is explained schematically in Figure 5. By matching failure modes to signals according to the matching types, CFT Frames are generated for each component with all applicable failure modes (e.g., too high, too low for continuous signals), and automatically linked according to the signal links. They have to be filled manually with the internal failure model at a later time (right part of the Figure 5; the logical gates are for demonstration only and have no meaning).

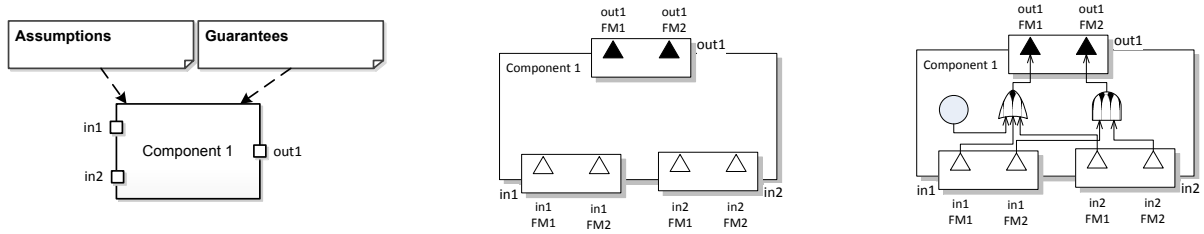

Figure 5. Transition from Component Model to CFT Frame, and insertion of CFT Logic

Failures do not occur in functional models, as these represent an idealistic mathematic model of the system behavior. They do occur in the technical implementation of the components as hardware or software. Therefore, a safety analysis is not complete (and cannot provide quantitative estimates for failure probabilities) unless performed on the technical architecture. On the lowest technical level, we recommend to abandon the signal-flow oriented way of thinking and to perform traditional techniques, such as FMEA on hardware design level or appropriate software analysis, where required (most safety analysis methods do not require probabilistic software failure modeling). The resulting failure consequences have to be linked to the failure ports of the lowest-level components, and only then the CFT can be evaluated qualitatively and quantitatively for each of the hazards as their top events. However, as a means of "frontloading" the safety process, component-based failure analysis, such as CFT or IF FMEA can be useful on functional architectures as well. They help examine the consequences of hypothetical failure modes and proposing countermeasures, before the technical design is complete. In our integrated process, we suggest using CFTs for the early failure analysis. The behavior of the fault propagation, which makes up the CFT structure, can be estimated early in the development process by fault injection simulation into an idealized behavioral model of the components or inserted manually. The internal failure modes inside the components must first be left as placeholders and be substituted later by the component supplier, using technical safety analysis ,methods, e.g., FMEA.

4.3 Definition of safety mechanisms

Once the failure modes have been identified, the last remaining step of the safety part of the process is defining appropriate technical countermeasures, called safety mechanisms, in order to ensure at least an acceptable level of safety in presence of failures. This is performed during the creation of the safety concept. The simplest solution is excluding certain failure modes by assumption: "Failure mode x does not occur". These assumptions have to hold with respect to the technical solutions. Because of the chosen implementation, it may happen that some of the theoretically existing failure modes do not occur. As "never" translates to "with sufficiently low probability" (e.g., 10^{-8} per hour) in safety engineering, it may also be possible to choose a component with an extremely low failure rate (measured in the unit FIT – "failure in time", which corresponds to one failure in 10^{-9} hours), such that the safety goal can be reached without any additional mechanisms. In most cases, mechanisms have to be inserted to diagnose the fault and to react on it appropriately, e.g., by switching to a redundant channel or by cutting the power supply to shut down the system safely. At the end, it has to be shown in the safety case that all failures that can occur have been sufficiently (i.e., with sufficient ASIL and sufficient Diagnostic Coverage) covered by safety mechanisms and are capable of entering and maintaining a safe state. As stated in Section 4.1, we are by now ready to accept that the overall system changes to a different operation mode as the normally desired one, in order to assure the avoidance of hazards. In the case of fail-operational systems (e.g., steer-by-wire) this alternative mode will even fulfill the same functional requirements than the nominal mode (with reduced level of redundancy though), at the cost of expensive hardware redundancy. In many cases, however, the alternative mode in case of failure (which is a rare case) will provide only a reduced performance

(e.g., worse accuracy leading to a reduction in comfort, a reduced set of available functions, etc.), or in extreme cases consist in a complete shut-off ("safe state") of the system (which must be a fail-safe system in that case). The latter, obviously, represents a *massive* violation of the specification of the nominal function. Hence, we can conclude that a top-level requirement for the nominal behavior, stated, e.g., in the form:

> *"The system shall always assure that the actual motor speed equals the target speed with a tolerance of +/- 5% at max."*

becomes weaker during the safety process:

> *"The system shall always assure that either the actual motor speed equals the target speed with a tolerance of +/- 5% at max,* ***or*** *the motor supply is shut-off within 100 ms."*

Note that the OR conjunction is not the usual symmetric one, but rather a kind of "or else": the first option is clearly the preferred one, the second option is only accepted as a last resort in case of failure, so an alternative representation could be:

> *"1. The system shall always assure that the actual motor speed equals the target speed with a tolerance of +/- 5% at max.*
> *2. If due to some failure the above requirement can no longer be complied with, the system shall shut off the motor supply within 100 ms"*

Violating the first partial requirement is a non-conformance with the nominal function specification and might be a reason for customer complaint. The quality management might want to set a probabilistic target for this violation to occur. Nevertheless, in this case the second partial requirement still holds, so safety is still assured. Violation of the second partial requirement would constitute a hazard, which is worse than just a quality issue.

However, diagnostic measures are also technical systems that are not perfect, and in rare cases there might be more than one failure present at one time. Hence, with some bad luck even the combination of both parts may still be violated, but this occurs with a *very* low probability. This is accepted according to the safety standards, because perfect safety is considered as unreachable. Most safety standards, such as ISO 26262 or IEC 61508, provide probabilistic target values for safety goal violations. These depend on the SIL or ASIL and can be as low as 10^{-8} safety goal violations due to hardware failures per hour.

Defining proper safety architectures and finding appropriate safety mechanisms is a creative process that cannot be automated. It is based on experience and can be supported by heuristics (e.g., placing diagnostics components close to the sensors and shutoff circuits close to the actuators is a useful rule of thumb), by architectural patterns (some standard patterns, such as 2-out-of-3 are addressed in [1] and [2]), or one can refer to proven-in-use standard architectures (such as the VDA E-Gas-Concept in the automotive industry). As explained in [34], the designer has several possibilities: It can be claimed with an assumption that the failure does never occur (i.e., with sufficiently low probability, systematic failures prevented up to the given ASIL), or the component causing the failure can be equipped with an internal mechanism, such that the failure does no longer propagate to its output ports, or the propagation is accepted and the mechanism is placed into another component, downstream the signal flow. The latter is a frequent pattern in embedded systems design: as it is often not possible to make a simple sensor sufficiently safe, a diagnostics function is placed into the component that consumes the sensor value. As this component is typically realized as software component running on a microcontroller, it is cheaper to implement the mechanism in software than enhancing the sensor. The simplest diagnostics mechanisms are range checks that issue an alarm for out-of-range values. This is already sufficient to detect some very common failure modes like too high out of

range, and too low out of range, which may technically be caused by frequent problems, such as cable brake, supply voltage brake down or short circuits to ground. Of course, there is a wide choice of available diagnostics mechanisms, including model-based diagnostics, comparison to redundant sensors or majority voters. They add a new output port "invalid", which signalizes to the consumers of the sensor value that the measurement value is no longer conforming to the specification (guarantees) for its nominal function. Failure detection alone does not assure the safe state. The "invalid" signal must be connected to existing blocks (e.g., operation mode state machine) or additional safety-specific blocks (e.g., failure manager), which are responsible for forcing the safe state (e.g., by cutting the power supply to the power electronics part of the actuator). An extract of a motor drive system with one of its sensors, the current sensor, is shown in Figure 6. On the left side, the system is shown before safety mechanisms have been inserted.

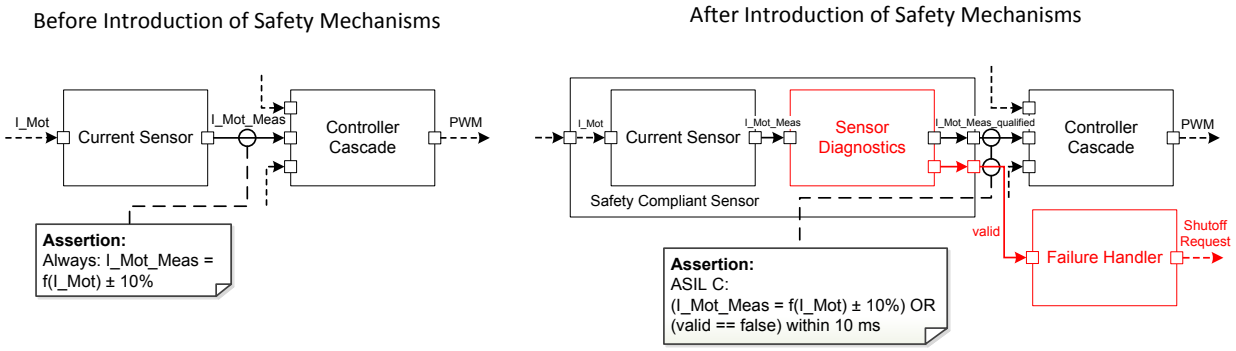

Figure 6. Current Sensor without (left) and with (right) safety mechanism

One sample assertion about nominal behavior is depicted – it determines the guarantee of the current sensor and the assumption of the controller cascade by defining interface contracts. On the right side, the same system is shown after safety analysis, with an extra block "Sensor Diagnostics" inserted after the creation of the safety concept. Now, the combined subsystem of current sensor and sensor diagnostics guarantees with respect to its output ports: "[With ASIL C it is guaranteed that] Either the measurement value for the current corresponds to the actual physical current, transformed by some sensor function, with an accuracy of 10%, or the 'valid' output will be set to false within 10 ms after occurrence of the failure". Note that the first part of the requirement was the original guarantee for the nominal function of the sensor. The ASIL is a qualifier for the guarantee and indicates the level of trust that can be placed into the implementation of the safety mechanism.

Safety mechanisms themselves are complex systems and may be subject to failures as well. Therefore, the proposed safety process will often require several iterations, each time integrating the new safety mechanisms into the architecture and repeating the safety analysis considering the consequences of its failures. This repeats iteratively until some stop criterion is reached. For example, some standards allow to stop when all single and double point failures have been analyzed; sometimes an analysis of triple, quadruple, or higher order failures is required (cf. [1], Part 5, Note 1 of Clause 7.4.3.2).

Since we augmented the existing assertions for the nominal behavior by the safe alternative behavior for the failure case, we obtain augmented contracts involving safety. We call them safety contracts. As the Safety Goal (see the sample in Section 4.1) as the top-level requirement can be stated as just another guarantee the system shall exhibit in its operational requirement, we can now jump back to the procedure, explained in Section 3 and refine the safety contracts into the components of the augmented architecture, thereby verifying that the safety concept is complete and adequate.

As safety standards usually require taking failures in the environment into consideration, a top-down safety process must also analyze situations where the assumptions about the environment are broken. It must be evaluated whether the assumption that the failure mode does

not occur can still be maintained (e.g., the assumption that in a vehicle, the environmental temperature is always below 140 °C can be justified by experience and existing standards), or whether additional safety mechanisms even for the violation of assumptions about the environment must be integrated. If a designer of an off-the-shelf component requires an input signal from a sensor, and the sensor is labelled with a guarantee that, up to integrity of ASIL C, the current measurement is either inside the tolerance range, or labelled as invalid, this holds as an assumption for designing this component. Now, partitioning of work between several suppliers, designing and selling pre-qualified components is possible (called "Safety Elements out of Context" in ISO 26262 language).

4.4 Validation of safety mechanisms

As discussed in the previous section, safety mechanisms are also prone to failures and might therefore deliver incorrect results. Hence, an iterative analysis process has been suggested to ensure that the safety concept sufficiently covers the detection and mitigation needs required by the applicable safety standard. As a second analysis activity it needs to be ensured that the safety mechanisms are correctly realized by a functional model or an implementation as discussed in [35]. The contract specifying the safety mechanism defines the context, in which the safety mechanisms shall operate correctly, and how the result shall look like. The Plausibility Check in Figure 6 detects an invalid current signal and sets the valid signal to false. The assumption for this behavior is the absence of internal faults of the plausibility check. Hence, the contract will informally look like:

> *Assumption: "Multiple-Point Failures do not occur" or, alternatively: "Safety mechanisms do not fail".*
> *Guarantee: "The output signal is either correct or the invalid signal makes the output signal recognizable as faulty."*

From the assumption in our example (which could be derived from some process standard) it follows that it should be considered that, at the same time, the signal is faulty due to some sensor failure *and* the Sensor Diagnostics Component from Figure 6 is defective. Of course, many standards in reality require considering at least dual-point failures and would not accept the claim that safety mechanisms never fail; some standards, however, allow neglecting failure sets of higher order than two, or failure combinations with sufficiently low probability.

A technique to verify fault tolerance mechanisms is fault-injection. It is defined by [36] as "the deliberate introduction of faults into a system." In order to execute a fault injection, failure hypotheses are needed, which are given by our formalization of failure modes. The different relevant failure modes are introduced at the input ports of some component in order to verify that the safety mechanism reacts appropriately.

5 Application example: electric vehicle drive

In SPES_XT project [6] an adaptive cruise control (ACC) system proposed by Daimler [37] was used to evaluate our approach. The ACC model includes an electric traction drive (E-Drive) consisting of a 3-phased electric machine and the corresponding ECU. A consistent development and safety assurance process was examined. Both, vehicle-level hazards and technical failure modes within the ECU were examined. A safety concept including ACC and E-Drive [34] was created. The ACC and the electric drive control were both modeled in MATLAB/Simulink, from which code was automatically generated. The safety mechanisms were also developed using Simulink or plain C code, and the whole software was loaded onto an evaluation board, mounted

on top of a 1:8 scaled model car. Most parts of the overall safety approach, as described in this article, have been tried out on this platform in a series of master theses.

Since the 3-phased motor with field-oriented control has a complex controller structure and failure propagation mechanisms, a stationary DC motor drive system (which we have also physically implemented in lab scale) was used at its place to evaluate the contract-based approach. The purpose is the same, but the equations of a DC motor (at least approximately) are quite simple: torque = current * motor_constant; emf (i.e., the voltage induced due to the rotation) = rotational_speed * motor_constant; and voltage = emf + current * motor_resistance. The sample DC drive system consists of a small DC motor, a power electronics board, a microcontroller with some peripheries on a low-cost evaluation board, an operator panel with ON and OFF push button, and a rotary knob with a potentiometer to adjust the target speed. On the power board, a MOSFET transistor (*Metal-Oxide Semiconductor Field-Effect Transistor*) switches the battery voltage to the motor armature circuit according to the pulse width modulated (PWM) signal generated by the microcontroller at one of its timer outputs. The purpose of a cascade control is to adjust the rotational speed of the motor to the target speed commanded by the rotary knob with a specified accuracy. The finding a suitable control algorithm and a state machine for the primitive operation modes (ON and OFF) is explained in the remainder of the paper.

The first step (see Section 3.4) is to write down the requirements in template language and to create a preliminary system architecture as presented in Figure 7. This architecture already indicates what sensors are required. The design includes motor current sensor, speed sensor, supply voltage sensor, and their connection to analog-to-digital converters (ADCs), timer outputs, and general purpose outputs of the microcontroller.

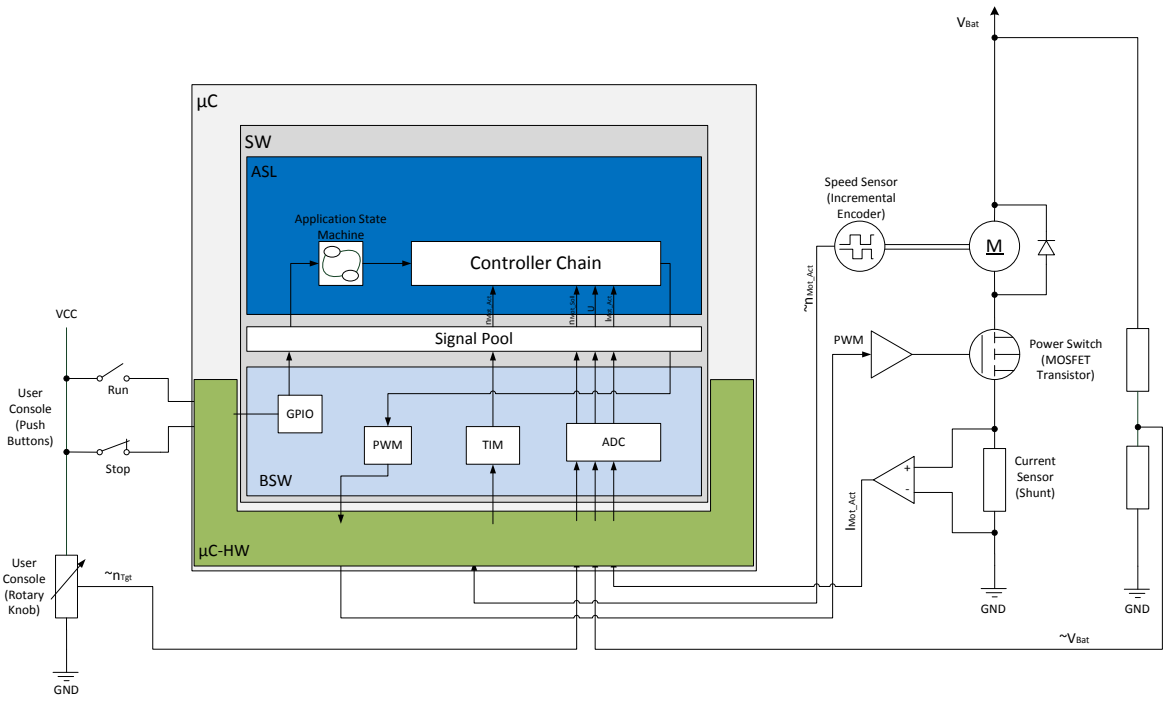

Figure 7. Technical overview of a simple DC Drive system

We then proceed by deriving a set of assumptions and guarantees from the requirements in natural language, where we aim at using a restricted set of templates as a preparation for the later contract formalization. In addition, we formalized a few assumptions on the operational environment. An excerpt of this work is shown in Table 1.

Table 1. A selection of requirements and assumptions for the DC Drive system

ID	Guarantee/Assumption
G01	When the system is in RUN state, the system shall generate torque at the motor shaft to achieve and maintain the rotational speed as specified.
G02	When the system is in STOP state, the system shall not generate any torque at the motor shaft.
G03	When system is in STOP state and the RUN button is pressed, the system shall react by changing to RUN state.
G04	When system is in RUN state and the STOP button is pressed, the system shall react by changing to STOP state.
G05	When the system is in RUN state and the STOP button is pressed, the motor current shall, after 100 ms at maximum, enter the range [0 mA; 1 mA] and stay within this range until the RUN button is pressed.
G06	When the system has been in RUN state for at least 1s and the external input speed knob position has not changed for at least 1s and the load torque is zero and the voltage supply is at its nominal level, the actual speed of the motor shall remain in the range [target speed – *speed accuracy*; target speed + *speed accuracy*] (where *speed accuracy* is 10 rpm).
A01	The supply voltage is always in the range [10V; 14V].
A02	The rotary knob position is always in the range [0 rpm; 1000 rpm].

The next step includes refining the functional system architecture and assigning meanings to all components and signals as illustrated in
Figure 8. The safety-related blocks in the figure are depicted in red (shaded) and were not part of the initial version, before safety analysis had been performed.

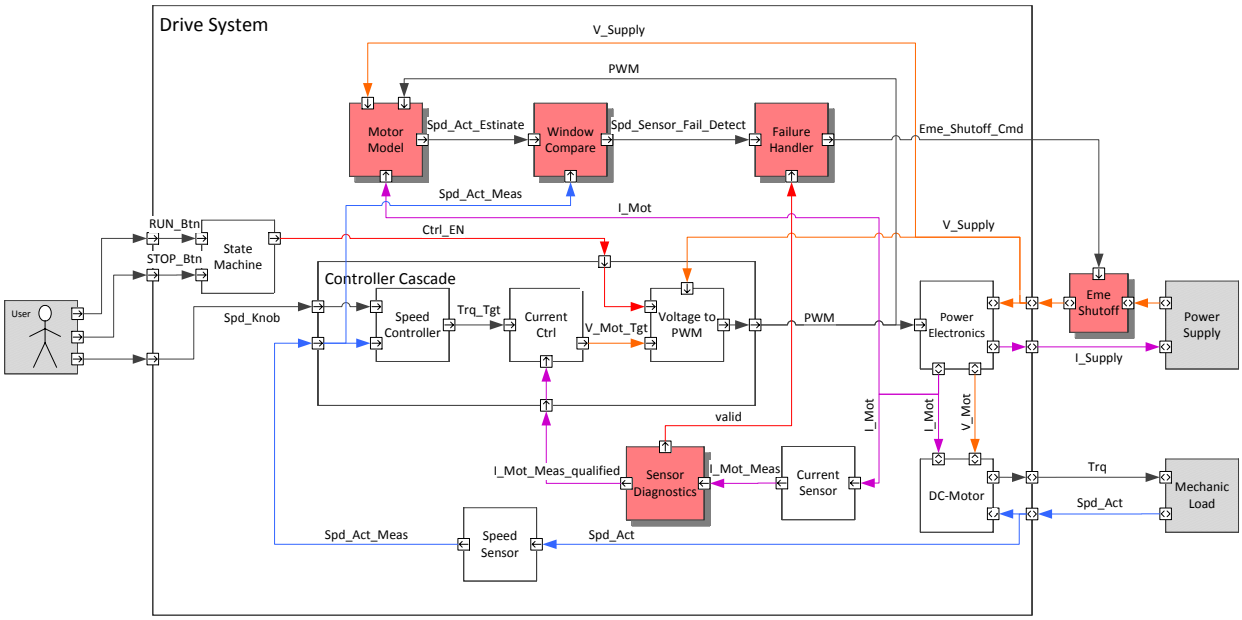

Figure 8. Functional system architecture

The description of each component and signal is described in natural language and can be represented in tabular notation. Furthermore, all signals are classified by types (continuous, discrete, event), which helps assigning compatible assertions and afterwards interface contracts to them. Examples are given in Table 2 and Table 3.

Table 2. Component names

Component Name	Description
Current Controller	Calculates setpoint for motor voltage from current target and actual current applying PI control law
Current Sensor	Measures motor current and transforms it into an analog signal according to the formula i_mot_meas = a * i_mot + b
State Machine	Determines the current operation state of the drive system according to the related requirements, listening to signals representing RUN pressed and STOP pressed events, producing a discrete output signal ENABLE that is true whenever state is RUN and false else

Table 3. Signal names

Signal Name	Description	Type
I_mot_meas	Measurement value from current sensor, represents the motor current, transformed according to the formula i_mot_meas = a * i_mot + b	continuous
I_mot	The physical current through the DC motor, measured at the mounting point of the current sensor	continuous
Stop_Btn	Event that occurs whenever the STOP button on the operator terminal is pressed	event
Ctrl_EN	Enable signal to switch on/off the output of the controller chain	discrete

The description of each function block and its ports is the starting point for deriving the specification. Now the specification of the entire system has to be decomposed into the blocks, as described in Section 3.4. For instance, the imprecision, which is an additive disturbance that applies to continuous signals, has to be estimated and stated by assertions at each point of the signal flow, such that at the output of the system, the desired imprecision is met (see Figure 3). Similarly, the delay between events and their executions (STOP button pressed, for instance) over the processing chain (capturing the event, processing it in a state machine, forcing the controller chain to set current to zero, reaction of the power part and motor inductance) are also stated by assertions (see Figure 4). Note that making assumptions about imprecision or delay always means anticipating the technical solution, because pure mathematical function blocks do not cause imprecision and a mathematical state machine reacts without any delay on events. For blocks that are intended to be implemented by hardware (such as a current sensor or an operational amplifier), the expectable performance can be found in the corresponding data sheets or derived from the circuit diagram. For blocks that are intended to be implemented by software, the determining factors for delay and accuracy are the scheduling frequency, the worst case execution time of an algorithm, and the data types and calculation precision. Assertions can state, for instance, that *the enabling signal for the controller chain goes to false within 20 ms after the STOP button has been pressed, or that the output signal I_mot_meas at the Current Sensor output deviates by not more than 10% from the mathematically correct value relating to the real physical current*.

Next, the assertions annotated to the signals are transformed into guarantees at the output ports of the components that produce these signals (e.g., the Current Sensor shall guarantee that the continuous signal I_cur_meas at its output stays in the specified accuracy range). These guarantees may then be used as assumptions by the consuming components. This establishes the interface contracts between respective neighbor components, which are subsequently turned into component contracts. Consequently, a specification for each of the components is provided, which can be further refined, or passed to a supplier, or concretized by a technical implementation.

Now let us pass to the safety-related process steps, as described in Section 4. We consider two typical hazards, which might be taken from the HARA of the electric vehicle drive (the assigned ASILs are only for the example) (See Table 4).

Table 4. Some Sample Hazards of the E-Drive System

ID	Hazard	Safety Goal	ASIL	Safe State	FTT
H01	Overspeed	Speed Deviation of more than 100 rpm above target speed shall be prevented	C	Motor Off	200 ms
H02	Unintended Start	Applying motor current without having the RUN button pressed shall be prevented	C	Motor Off	100 ms

As indicated in Section 4.1, the decomposition of the safety goals is performed similarly to the functional requirements above, but as a next step we have to identify potential failures that could impair the safety goals (i.e., cause the hazards). The definition of a safe state indicates how to react on the related failures, and the Fault Tolerance Time specifies the guarantee on the delay of this reaction. According the procedure described in Section 4.2, we transform the system architecture into a CFT structure. This is shown in Figure 9.

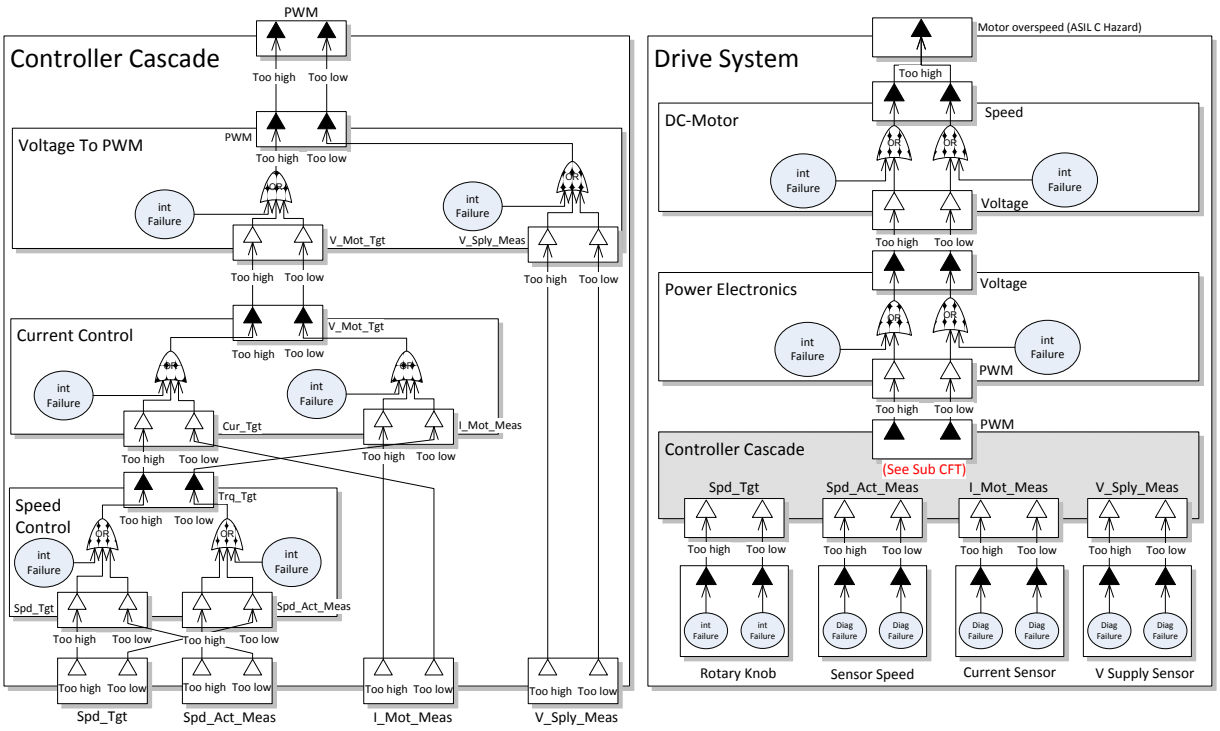

Figure 9. Component Fault Tree for DC Drive system (controller part)

The left part depicts the subcomponent of the black box "Controller Cascade" in the right part. The correspondence of components and ports is visible in the architecture and in the CFT, with the failure ports (triangle) shown inside the signal ports (rectangles). The structure has been generated automatically and the candidate failure modes have also been proposed automatically ("too high" and "too low" in the example, as we are dealing with continuous signals). The inner Fault Tree structure has been created manually, as it requires knowledge about the failure propagation (e.g., "too low" for the actual current at the input of the current controller I_mot_meas is related into "too high" for the motor voltage target value at the output) and the inner failure modes of each component.

After inserting the safety mechanism, a new iteration of the safety analysis is performed. Now we find that no more single point faults can cause the "Overspeed" hazard H01. Only if two failures coincide (i.e., the sensor is defective and the safety mechanism fails to detect this), the safety goal would be violated. If the safety standard permits the assumption that not more than one failure occurs at a time, the system can be considered sufficiently safe. In practical cases, a probabilistic estimation of failure rate and diagnostic coverage would have to be evaluated using

quantitative fault tree analysis, putting the failure rates of the actual hardware parts into the fault tree equations.

At this point, a new failure consequence can now occur: The current measurement value can have the new failure mode “unavailable”, in case that the raw value from the sensor is corrupt and this is detected by the safety mechanism. Typically, such a failure leads to an undesired unavailability of the drive system. This violates the specification for the nominal function, but not the safety goal, and is therefore acceptable from safety point of view. The general product management, however, has to check, under which circumstances and with which probability the unavailability of the drive system can be tolerated without violating market demands. Sometimes, market demands may even lead to more strict requirements than safety demands.

6 Conclusions and outlook

We have proposed a contract-based development approach for the development of safety-critical embedded systems in various industry domains. It facilitates the system specification, structural composition, and technical architecture design. In contrast to previous articles on the topic, we have shown how the development of the nominal behavior and the safety architecture can be integrated by referring to contract violations as a formal definition for failure modes. We also presented how to interpret the safety concept as an extension of the contract-based specification with safety mechanisms, leading to guarantees regarding the non-occurrence of hazards or the appropriate reaction to failure situations. Unlike the traditional V-model approach, our approach can be applied top-down or bottom-up, which fits to the current situation where systems also reuse components. The approach was illustrated step-by-step using an automotive electrical drive example. We have applied all parts of our approach to example systems in the context of the SPES XT research project and partially implemented the controllers and safety mechanisms in a small-scale model car. The approach appeared easy to learn and some parts have already been integrated in consulting projects with German carmakers.

Our next steps include lifting the described approach to the development of Cyber Physical Systems (CPS). This next generation of embedded systems is expected to mark the transition to interconnected and self-organizing embedded systems that together provide even more complex functionality. Besides openness and dynamic reconfiguration at runtime CPS are often characterized by properties, such as tight integration of physical and virtual world, context-aware or highly automated planning of their behavior, and close interaction of humans and machines with changing or cooperative control (see [5]).

In the context of CPS, even new design challenges will arise, e.g., the lack of a responsible manufacturer of the entire system-of-systems who could be held responsible for the behavior of the whole, and correspondingly also the lack of a requirements engineer, an architect or a safety manager. On a technical level, unstable communication, security issues, and lack of a common system state, time base, and agreed upon world model lead to a drastic increase of complexity.

An example for a CPS we are currently working on is a cooperative adaptive cruise control (CACC). A CACC is an extension of today’s adaptive cruise controls, which enables vehicles to connect with each other to form a temporal convoy, to negotiate and agree on synchronized setpoint values for speed and distance (including warning against hazards or brake maneuvers), which are then locally controlled. As the V-model process of the current edition of ISO 26262 requires an item definition in the beginning, it does not scale up to systems-of-systems with dynamically changing members, because the item on the highest level (the convoy) emerges at a runtime and has no fixed system boundary. In this case, contracts might be composed of global assumptions (e.g., regarding traffic rules and road layout) and safety goals demanded by legal authorities could be translated into verifiable guarantees (e.g., minimum distance between vehicles, no side crashes when changing lanes etc.). The negotiation of assumptions and guarantees would shift from development time to runtime and would, thus, have to be automated.

Solutions for contract negotiations at the instant of joining a convoy and runtime checking of contracts fulfillment before allowing certain operation modes (e.g., very close following) need to be developed, considering security aspects as well. Furthermore, the operational context is also dynamically variable for vehicles in traffic scenarios (e.g., number and quality of lanes on the highway). Hence, parameterized scenario set definitions are needed to replace today's situation catalogues used for hazard analysis. Contract-based development seems a promising approach to deal with these new challenges. Our presented formulation of contracts make them suitable for an automated negotiation between partial systems connecting at runtime, and, hence, useful to address the above issues.

References

[1] ISO, ISO 26262 Road vehicles - Functional safety. International Organization for Standardization, 2011.

[2] IEC61508, IEC61508: Functional safety of electrical/electronic/programmable electronic safety related systems. International Electrotechnical Commission, 1999.

[3] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. Larsen, "Contracts for Systems Design," Research Report INRIA, 8147, 2012.

[4] B. Kaiser, S. Sonski, S. Buono, H. Petersen, and J. Zander, "Lightweight Contracts for Safety-Critical Automotive Systems," in Tagungsband 13. Workshop Automotive Software Engineering. INFORMATIK 2015 - 29.9.2015, Cottbus, 2015.

[5] M. Broy and E. Geisberger, "agendaCPS: Integrierte Forschungsagenda Cyber-Physical Systems," acatech, 2012.

[6] "Software Platform Embedded Systems 'XT', SPES_XT; http://spes2020.informatik.tu-muenchen.de/spes_xt-home.html."2015.

[7] P. Battram, B. Kaiser, and R. Weber, "A Modular Safety Assurance Method considering Multi–Aspect Contracts during Cyber Physical System Design," in 1st International Workshop on Requirements Engineering for Self–Adaptive and Cyber–Physical Systems (RESACS), 2015, pp. 185–197.

[8] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand, "Using contract-based component specifications for virtual integration testing and architecture design," in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011, pp. 1-6. Available: http://dx.doi.org/10.1109/DATE.2011.5763167

[9] T. Gezgin, R. Weber, and M. Girod, "A Refinement Checking Technique for Contract–Based Architecture Designs," OFFIS Technical Report (SPES 2020 Project), 2011.

[10] S. Aboubekr, G. Delaval, R. Pissard-Gibollet, É. Rutten, and D. Simon, "Automatic Generation of Discrete Handlers of Real-Time Continuous Control Tasks," in Proc. of the 18th IFAC World Congress, 2011, vol. 18.

[11] P. Boström, "Contract-Based Verification of Simulink Models," in Formal Methods and Software Engineering, , vol. 6991, LNCS, 2011, pp. 291–306. Available: http://dx.doi.org/10.1007/978-3-642-24559-6_21

[12] A. Baumgart, E. Böde, M. Büker, W. Damm, G. Ehmen, T. Gezgin, S. Henkler, H. Hungar, B. Josko, M. Oertel, and others, "Architecture modeling," OFFIS eV, Oldenburg, Technical Report, 2011.

[13] E. Hull, K. Jackson, and J. Dick, Requirements Engineering. Springer, 2004.

[14] C. Kühl, "Formalisierung von Requirements durch Nutzung von Templates," Master Thesis Freie Universität Berlin, 2010.

[15] A. Pnueli, "The temporal logic of programs," in Foundations of Computer Science, 1977., 18th Annual Symposium on, 1977, pp. 46–57. Available: http://dx.doi.org/10.1109/sfcs.1977.32

[16] N. A. Lynch and M. R. Tuttle, "An Introduction to I/O Automata," CWI Quarterly, vol. 2, no. 3, pp. 219–246, 1989.

[17] M. et al. Bozzano, "ESACS: An Integrated Methodology for Design and Safety Analysis of Complex Systems," in Proc. of ESREL 2003, 2003, pp. 237–245.

[18] P. Reinkemeier, I. Stierand, P. Rehkop, and S. Henkler, “A pattern-based requirement specification language: Mapping automotive specific timing requirements,” in Software Engineering (Workshops), 2011, vol. 184, pp. 99–108.

[19] SPEEDS, “D.2.5.4 Contract Specification Language (CSL). Available at: http://speeds.eu.com/downloads/D_2_5_4_RE_Contract_Specification_Language.pdf (retrieved on 03-Aug-2015),” 2008.

[20] I. Bate, R. Hawkins, and J. McDermid, “A Contract-based Approach to Designing Safe Systems,” in Proc. of the 8th Australian Workshop on Safety Critical Systems and Software, vol. 33, 2003, pp. 25–36.

[21] J. Fenn, R. Hawkins, P. Williams, and T. Kelly, “Safety case composition using contracts-refinements based on feedback from an industrial case study,” in The Safety of Systems, Springer, 2007, pp. 133–146.

[22] A. Baumgart, P. Reinkemeier, A. Rettberg, I. Stierand, E. Thaden, and R. Weber, “A Model-Based Design Methodology with Contracts to Enhance the Development Process of Safety-Critical Systems,” in Software Technologies for Embedded and Ubiquitous Systems, vol. 6399, LNCS, 2010, pp. 59–70. Available: http://dx.doi.org/10.1007/978-3-642-16256-5_8

[23] F. Warg, B. Vedder, M. Skoglund, and A. Soderberg, “Safety ADD: A Tool for Safety-Contract Based Design,” in Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium, 2014, pp. 527–529. Available: http://dx.doi.org/10.1109/issrew.2014.18

[24] M. Oertel, A. Mahdi, E. Böde, and A. Rettberg, “Contract–based Safety: Specification and Application Guidelines,” in Proc. of the 1st International Workshop on Emerging Ideas and Trends in Engineering of Cyber–Physical Systems (EITEC 2014), 2014.

[25] IEC, “Hazard and operability studies (HAZOP studies) - Aplication guide,” BS IEC 61882:2001, 2001.

[26] P. Fenelon, J. McDermid, D. Pumfrey, and M. Nicholson, “Towards Integrated Safety Analysis and Design,” Newsletter ACM SIGAPP Applied Computing Review - Special issue on saftey-critical software, vol. 2(1), pp. 21-32, March 1994. Available: http://dx.doi.org/10.1145/381766.381770

[27] Y. Papadopoulos and J. McDermid, “Hierarchically Performed Hazard Origin and Propagation Studies,” SAFECOMP ’99, 18th Int. Conf. on Computer Safety, Reliability and Security, vol. 1698, LNCS, pp. 139–152, 1999. Available: http://dx.doi.org/10.1007/3-540-48249-0_13

[28] B. Kaiser, P. Liggesmeyer, and O. Mäckel, “A New Component Concept for Fault Trees,” in Proc. of the 8th Australian Workshop on Safety Critical Systems and Software, vol. 33, 2003, pp. 37–46.

[29] D. Domis, “Integrating fault tree analysis and component-oriented model-based design of embedded systems,” Verl. Dr. Hut, 2011.

[30] V. Ramich, “Teilautomatische Erstellung von Component-Fault-Trees aus Simulink-Modellen,” Master Thesis Universität Kassel, 2014.

[31] A. Bondavalli and L. Simoncini, “Failure Classification with Respect to Detection,” in Predictably Dependable Computing Systems, Task B, Vol. 2, 1990. Available: http://dx.doi.org/10.1109/FTDCS.1990.138293

[32] J. G. Lamm and T. Weilkiens, “Functional Architectures in SysML (‘Funktionale Architekturen in SysML’ English translation by J. Lamm),” Tag des Systems Engineering 2010, pp. 109–118, 2010.

[33] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” in IEEE Transactions on Dependable and Secure Computing, 2004. Available: http://dx.doi.org/10.1109/TDSC.2004.2

[34] F. Feyerherd, “Methodische Erstellung eines Sicherheitskonzeptes für einen elektrischen Antrieb,” Master Thesis Hochschule für Technik und Wirtschaft Berlin, 2014.

[35] J. Zander-Nowicka, I. Schieferdecker, and A. Marrero Perez, “Automotive Validation Functions for On-Line Test Evaluation of Hybrid Real-Time Systems,” in Autotestcon, 2006 IEEE, 2006, pp. 799–805. Available: http://dx.doi.org/10.1109/AUTEST.2006.283767

[36] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, “Fault injection for dependability validation: a methodology and some applications,” IEEE Transactions on Software Engineering, vol. 16, no. 2, pp. 166–182, Feb. 1990. Available: http://dx.doi.org/10.1109/32.44380

[37] F. Houdek, U. Löwen, and C. Wehrstedt, “Advanced Model-Based Engineering of Embedded Systems,” K. Pohl, H. Dämbkes, H. Hönninger, and M. Broy, Eds. To be published by Springer.