Science of Computer Programming 90 (2014) 161–179

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Employing early model-based safety evaluation to iteratively derive E/E architecture design

V. Rupanov [a,*,1], C. Buckl [c], L. Fiege [b], M. Armbruster [b], A. Knoll [a], G. Spiegelberg [b]

[a] Institut für Informatik, Technische Universität München, Boltzmannstr. 3, 85748 Garching bei München, Germany
[b] Corporate Technology, Siemens AG, Otto-Hahn-Ring 6, 81739 München, Germany
[c] fortiss GmbH, Guerickestr. 25, 80805 München, Germany

HIGHLIGHTS

- A model-based methodology for early evaluation of design decisions is proposed.
- An analysis technique for ISO 26262 safety metrics is presented.
- Essential metamodels to support the methodology have been developed.
- Safety mechanism selection is performed based on model-based analysis results.
- A tool prototype implementing the methodology demonstrated.

ARTICLE INFO

Article history:
Received 16 October 2012
Received in revised form 15 September 2013
Accepted 15 October 2013
Available online 4 November 2013

Keywords:
Automotive systems
Embedded systems
Model-driven engineering
Quantitative safety analysis
ISO 26262

ABSTRACT

ISO 26262 addresses development of safe in-vehicle functions by specifying methods potentially used in the design and development lifecycle. It does not indicate what is sufficient and leaves room for interpretation. Yet the architects of electric/electronic systems need design boundaries to make decisions during architecture evolutionary design without adding a risk of late changes. Correct selection of safety mechanisms from alternatives at early design stages is vital for time-to-market of critical systems. In this paper we present and discuss an iterative architecture design and refinement process that is centered around ISO 26262 requirements and model-based analysis of safety-related metrics. This process simplifies identification of the most sensitive parts of the architecture, selection of the best suitable safety mechanisms to reduce thereby failure rate on the system level and improve the metrics defined by the standard. To support the defined process we present the metamodels that can be integrated with existing DSL (domain-specific language) frameworks to extend them with information supporting further extraction of fault propagation behavior. We provide a framework for architecture model analysis and selection of safety mechanisms. We provide details on the model-based toolset that has been developed to support the proposed analysis and synthesis methods, and demonstrate its application to analysis of a steer-by-wire system model and selection of safety mechanisms for it.

* Corresponding author.
E-mail addresses: vladimir.rupanov@fortiss.org, rupanov@in.tum.de (V. Rupanov), buckl@fortiss.org (C. Buckl), ludger.fiege@siemens.com (L. Fiege), michael.armbruster@siemens.com (M. Armbruster), knoll@in.tum.de (A. Knoll), gernot.spiegelberg@siemens.com (G. Spiegelberg).
[1] Currently with fortiss GmbH.

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.10.005

1. Introduction

In today's cars, most of the functionality is implemented using a combination of hardware and software solutions. As more and more safety-critical functions heavily rely on software, safety of software systems becomes a hot topic. The international standard ISO 26262 [1, Part 1] addresses this topic by defining a design process and proposing safety mechanisms. It provides process guidelines for the whole lifecycle, but does not indicate what is sufficient and leaves room for interpretation [2]. An electrical/electronic (E/E) system architect has to account on numerous non-functional design aspects at the same time: real-time properties, safety, security, cost, etc., and needs guidance when making decisions during architecture evolution without adding a risk of late architecture changes. As late architectural changes are very expensive, it is extremely important to support early architectural decisions. An important part of these decisions covered by ISO 26262 is selection and configuration of safety measures.

To reduce the complexity of the E/E architectures, the car manufacturers intend to use generic hardware and software platforms executing mixed-criticality functions, such as AUTOSAR[2] in software domain. This enables software reuse, but introduces limitations to the system-level analysis for early assessment of functional and non-functional properties [3]. In particular, the platform design and configuration are separate from the system as a specific application, yet both need to be analyzed together for successful certification. Despite the Safety Element out of Context (SEooC) concept introduced in ISO 26262, analysis of quantitative properties of such systems remains a major challenge.

In this paper, we propose an approach for safety analysis and design guidance early in the design process. Although safety analysis has been applied in the automotive industry for decades, no common safety lifecycle was applied. ISO 26262 triggers a change in the development lifecycle that requires adaptation and alignment of numerous processes in E/E system development. The standardized lifecycle also acts as an enabling factor for extensive use of model-based tools for engineering support including automation of routine analysis steps, collection of data in a unified format, and reuse of that data in new developments. Another important trend is common acceptance of evolutionary design methodologies, first predicted by [4]. Evolutionary design needs strong support from the modeling side, which combines simplicity of domain specific models with possibility of changes or refinements during late design steps.

We propose to systematically evaluate the system with instantiated safety mechanisms in context of ISO 26262 requirements and assess design alternatives from different viewpoints (safety, performance, cost). The approach starts with a specification of the fault behavior of hardware component models. We relate safety mechanisms to the fault models and provide methods to evaluate achievable design metrics for software applications, running on the hardware platform. Presented design methodology supports evolution of E/E architecture (EEA) under a fixed guarantee that the target Automotive Safety Integrity Level (ASIL) can be reached. This is achieved by stepwise refinement and further combination of software component, hardware and safety mechanism models controlled by evaluation of ISO 26262 architectural metrics based on the combined model. The necessary steps for implementing our approach are discussed including the definition of metamodels, quantitative metrics to be calculated, and design and analysis workflows. Validating instantiation of these items in a tool is accomplished via an application example.

The rest of the paper is organized as follows. We discuss context and the range of design techniques and applications motivating our approach in Section 2. In Section 3 we provide details on our approach. First we gently introduce the intended design flow and describe the interaction of different concerns in a complex application. We provide details on system models and transformations required to perform analysis and rate an architecture. Instantiation of these ideas in a model-based tool is presented in Section 4 along with a validating application. The paper is concluded by a discussion of results and of related work in Section 5.

2. Better engineering today

In this section background information on model-driven development and safety-critical systems is provided, and an overview of state of the art in automotive industry is done. During the last 30 years, electronic systems have become widely used in cars, resulting in numerous advances in driving safety, comfort and controllability of vehicles. The use of computers in vehicle applications raises the question of adequate behavior of automotive systems, especially of those controlling or related to vital functions, such as braking, steering and longitudinal speed control. E/E systems already play a key role in implementation of assistance functions like electronic stability program (ESP) or electronic brakeforce distribution (EBD), and the likely introduction of Drive-by-Wire systems will lead to total reliance of driver safety on E/E systems [5].

Modern development methods, such as model-driven engineering, simplify design of systems through increased level of abstraction. However, the evaluation methods for architectures remain almost the same as with a manual approach. In [6] it has been mentioned that the most important direction, in which architecture modeling needs to be developed, is practicability of applications and automation of the modeling techniques.

[2] AUTOSAR: AUTomotive Open System ARchitecture, more details at: http://www.autosar.org.

2.1. Safety-critical systems

Safety is a dependability attribute of a system or an architecture, which reflects "absence of catastrophic consequences on the user(s) and the environment" [7]. The systems, which imply risks for their users, third parties or environment, should be developed with more care and satisfy a certain level of quality. Quality is seen important from both financial and reputational perspectives, as the manufacturer is liable for consequences of its products' use. To limit liability manufacturers and their suppliers need a stable safety process which is satisfying state of the art. Confirmation of such conformance is usually maintained through independent certification authorities based on safety-related standards.

Demonstration of fulfillment of state of the art is very important in the case of new system generations, which can be built on principles, different from the predecessors (e.g., for control systems without mechanical connection between user input and object under control). Safety standards are different for specific application domains, yet there is much in common: they are regularly updated to reflect the state of the art in safety engineering, to address new technological complexities and clarify application of safety-related techniques. This means that following a standard is one of the necessary requirements to fulfill the state of the art.

Some of the important safety-related standards include IEC 61508 [8], – a general standard covering electrical and electronic systems, ISO 26262 [1], – an extension and adaptation of IEC 61508 for automotive systems, and RTCA DO-178B/C [9], – the standards that represents state of the art in airborne system development.[3]

Absolute safety is hardly reachable, so safety-related standards introduce a notion of "unacceptable risk" [1, Part 1] into the safety definition, as well as more detailed requirements for the development of safety-critical systems. As it is hard to measure the exact risk levels and to provide very specific process guidelines, a criticality level is defined in every standard. For example, Safety Integrity Level (SIL) in range 0 to 4 represents general risk applied to Electrical/Electronic/Programmable Electronic (E/E/PE) safety-related systems as defined by IEC 61508 [8]. In ISO 26262, ASIL assignment to a function specifies which level of risk is acceptable for this specific function. Similarly, Design Assurance Level (DAL) in DO-178B/C is defined for failure condition range from A (catastrophic) to E (none).

To sum it up, any manufacturer or supplier has to follow safety standard guidelines. Automation of this routine work is beneficial to system architects and engineers.

2.2. Model-driven engineering

Model-driven engineering (MDE) is a software engineering methodology that focuses on creating and exploiting domain models (abstract representations of knowledge for a specific domain), or DSLs to engineer software. In one of the best known MDE approaches, Object Management Group (OMG) model-driven architecture (MDA), platform-independent models (e.g., state-machines) are defined using DSL's and further, given a platform model, transformed into platform-specific models (e.g., Java classes), which are then followed by code generation. This is common for most of MDE methodologies.

OMG Meta Object Facility (MOF) is the metamodeling structure originally used to define UML. It defines metamodel levels M0 through M3, where M3 is the meta-metamodel, used to build metamodels, called M2-models, which in turn are used to create M1-models, that is classes of runtime instances of the system (layer M0). A subset Essential MOF of MOF has been implemented in Eclipse Modeling Framework [11], an open source project that forms a foundation for many model-based tools.

In an MDE environment roles of model and metamodel developers are different, and not always can these be mapped to traditional software development process roles: application developers, system integrators, which typically work on model level, define application models, whereas architects define metamodels and platform models. These models can be used to generate other models with model-to-model transformations. Typically model-to-model transformations extend the model with additional information, inferred from the model itself (e.g., generate a schedule based on timing constraints), merge multiple models (e.g., software and hardware models that have been developed separately), or both. Another class of transformations are model-to-text transformations, that are used to produce source code or textual artifacts.

MDE has a number of advantages: the DSL's are defined specifically for the application area, and the level of abstraction stays higher than during traditional development. Model-driven safety assessments are gaining acceptance [12] because they allow automation of safety analysis. This engineering methodology needs, however, proper tooling support to be successful.

2.3. Safe ICT platform for mass production vehicles

The safety of a system is directly influenced by the EEA, which describes the subsystems, their boundaries and interfaces, and includes the allocation of functions to hardware and software elements. In context of systems with integrated architecture, EEA represents the functional structure of the system (functional concept), physical structure of the system (hardware components), and mapping between them.

The state of the art design process in industry focuses on optimizing safety at the function level. This approach was feasible, as most functions could be analyzed in an isolated fashion. With the trend towards more interconnected functions,

[3] A good overview of existing safety-related standards is provided in [10].

such as global energy mana gement in electric cars, the complete architecture must be analyzed to prove that the required safety level is achievable. The trend towards system safety is also increased by shortened development cycles. As functions have to be integrated into the car in shorter time intervals, E/E platforms providing generic mechanisms to reach safety goals are becoming more important [13]. The name "platform" comprises the following items of a full vehicle control system:

(a) a central (fault-tolerant) platform-computer with access to all sensors and actuators via network;
(b) software platform – an operating system extended by a middleware, running mainly on the central platform-computer. A software platform provides means to execute and transparently protect vehicle system functions; it also provides mode management functionality (including platform-reconfiguration due to faults, master-slave switchovers to ensure fail-operational behavior);
(c) communication and power distribution networks.

When talking about safety-related availability requirements, one has to distinguish between two categories of functions. For some functions it is possible to define a safe state. These functions can be designed in a way that any failure in a system leads to a safe state (which means passivation of the considered function), so they have "fail-safe", or "fail-passive" requirements. In the second category, functions do not have a defined safe state: these functions need to operate in order to maintain driver and occupant safety. "Fail-operational" behavior is necessary to meet quantitative safety requirements for these functions [5], and requires some degree of redundancy. Safety mechanisms (SMs), as defined by [1, Part 1], are redundancy measures implemented by an E/E system function or element to detect or control failures. The faults in hardware components result from wear-out, aggressive environment and manufacturing process variations.

An important task for an architecture designer is to provide a reasonable architecture sketch and its evolution into a safe and stable system, without limiting the software capabilities and undergoing deep changes. As the architecture evolves, the safety goals for a specific instantiation of the E/E platform could be violated, as the initially selected set of SMs does not provide sufficient coverage. This is why constraints in the evolution of the architecture are necessary. There is a huge range of implementations of SMs available on the market in the form of either hardware component features or middleware, but there are no tools to quantify the effect of selected mechanisms in advance and to compare these against objectives (e.g., target values for ISO 26262 architectural metrics). Our approach aims at methods and tools that support system architects in evaluating design choices.

The methods that engineers apply are usually dealing with consequences of faults; it is usually not possible to identify the source. Failure mode analyses in inductive (e.g., Failure Modes and Effects Analysis, FMEA) or deductive (e.g., Fault Tree Analysis, FTA) methods rely on partially or fully automated use of detailed knowledge on components and safety measures that are implemented in the system to reduce or eliminate the risk [14]. We propose a focused approach to model design trade-offs and limit effort spent for safety estimation when selecting SMs.

Although ISO 26262 defines concrete safety mechanisms, literature reports different problems [2], mainly regarding the correct selection of the right level of detail and traceability between FMEA and safety assessment. This leads to the problem, that sufficiency and adequacy of safety mechanisms are hard to predict for a concrete EEA design. We present a solution for the last problem by suggesting an approach for early safety evaluation to simplify the correct selection of safety mechanisms and avoiding changes late in the design process. It is important to note that this paper focuses on random faults in hardware and does not target the toleration of systematic faults in hardware and software. The latter problem is tackled by the suggested development process of the ISO 26262.

3. Modeling and analysis methodology

We address the challenge of adequate selection of safety mechanisms by combining advanced modeling techniques with a strict design process definition. To reduce time to market we apply and automate model-based safety analyses for automotive architectures while keeping the models at abstraction level that makes them feasible for industrial application.

3.1. Design process

This section in combination with the following two describes our approach for design-time quantitative safety analysis driven selection of safety mechanisms. It is based on quantitative evaluation of safety-related metrics in each design cycle. Structured models, which are described in the next section, enable the automation of safety assessment. The approach can be applied iteratively, which means that design boundaries are evaluated at each EEA evolution cycle, and changes in EEA are driven by results of this evaluation.

We want to provide guidelines for system architect in selecting hardware components and hardware and/or software implemented safety mechanisms to be realized in the EEA that targets a certain ASIL level. This is achieved by identifying boundaries of EEA design guaranteeing that each solution within these boundaries satisfies the safety requirements. Safety is only one design criterion (although, a primary architectural driver for critical systems); cost, performance, and other quality attributes are also important, and comparison of design alternatives with regard to other attributes needs to be supported. More clearly, the design process needs:

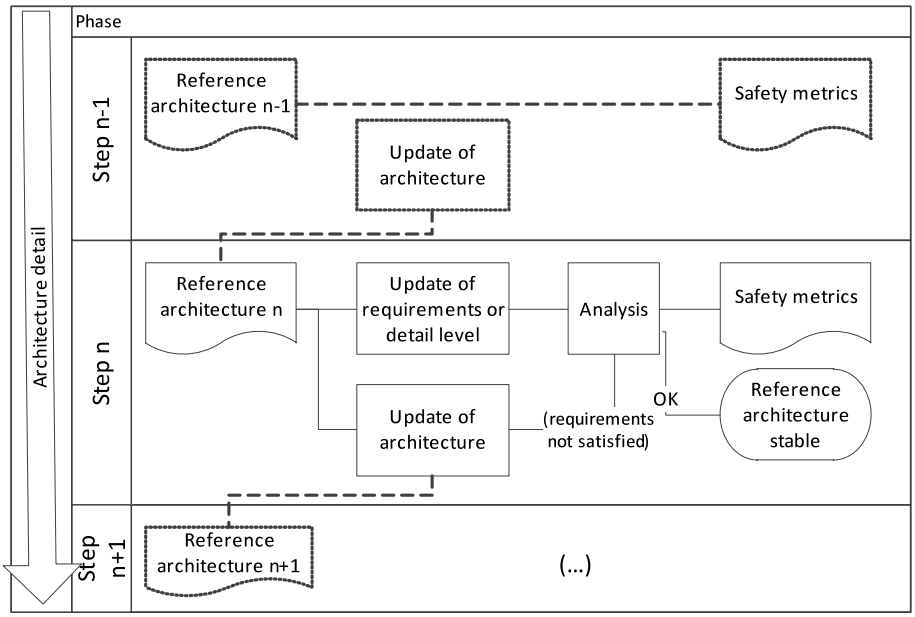

Fig. 1. Staged evolution of architecture triggered by changes in requirements or level of detail.

(a) support for EEA evolution: methods and models to enable different levels of detail;
(b) methods for early conservative quantitative evaluation of item safety according to the ISO 26262 standard;
(c) an approach to estimation of quality attributes (resources, cost);
(d) means of safety mechanism selection from possible alternatives and configuration of implementations based on resource allocation strategy.

The EEA design is a process of cyclic refinement of EEA in a domain-specific safety prediction framework (Fig. 1). Cycles represent levels of detail of EEA design that are passed throughout the design process. Every cycle of the process begins with model update: **a reference architecture** is created or updated. An initial EEA defines a set of computer nodes and middleware components – a platform, instantiated from a repository, an application model defined for a specific system. Combination of EEA artifacts is performed to result in a single model covering related concerns. Typical artifacts of the EEA draft are: structural block diagrams, functional network diagrams, and function deployment diagrams. These are already today developed in model-based environments, so the only necessary change could be an extension of the models to be full and consistent. Fig. 2 represents the details of the process from a single-iteration point of view. In parallel to EEA definition **identification of safety requirements** takes place. Requirements include safety goal (SG) definitions, target values for probability of safety goal violation target and architectural metrics. A SG typically defines a set of undesired states, or failure effects, on the system boundary. Detail level of those conditions determines the fault classification to be used in further analysis. Metric target values are directly derived from hazard and risk analysis of the functionality to be deployed on the EEA. Once determined, these requirements are often stable throughout the whole EEA design process. Update of safety requirements is often a result of changes in functional specification or correction of flaws in requirements.

Analysis of the EEA model to verify the design decisions is then performed. Depending on its results, the architecture can be accepted, selection of safety mechanisms can be updated based on provided coverage and requirements (this is performed by updating the combined model), or changes to EEA can be necessary (if no combination of safety mechanisms exists or the available resources are not sufficient). To limit the scope it is useful to split the design into smaller parts, providing certain budgets to each part. This approach is productive from the safety point of view, as conservative estimations hold even after the full design is evaluated. From the cost side it is more risky, as cost reduction is not efficient without re-estimation of the overall budget. A reasonable balance needs to be found, for example, by reassigning the budget based on slack of the parts.

The set of hazardous events that lead to failure effects in hardware is not stable over time. For example, integrated circuits become smaller and more sensitive to single event upsets (SEU), electromagnetic interference (EMI) and other environmental disturbances. Changes in the architecture often lead to re-estimation of the failure modes and effects. Detection mechanisms are to be adjusted to these changes as a result. Both the set of mechanisms and their parametrization over components might change over time. Our repository-based approach allows us to systematically collect a component model repository (from basic components down to concrete devices), keeping both failure modes/rates and corresponding detection mechanisms in a consistent manner and enabling reuse of data that has once been modeled or modified.

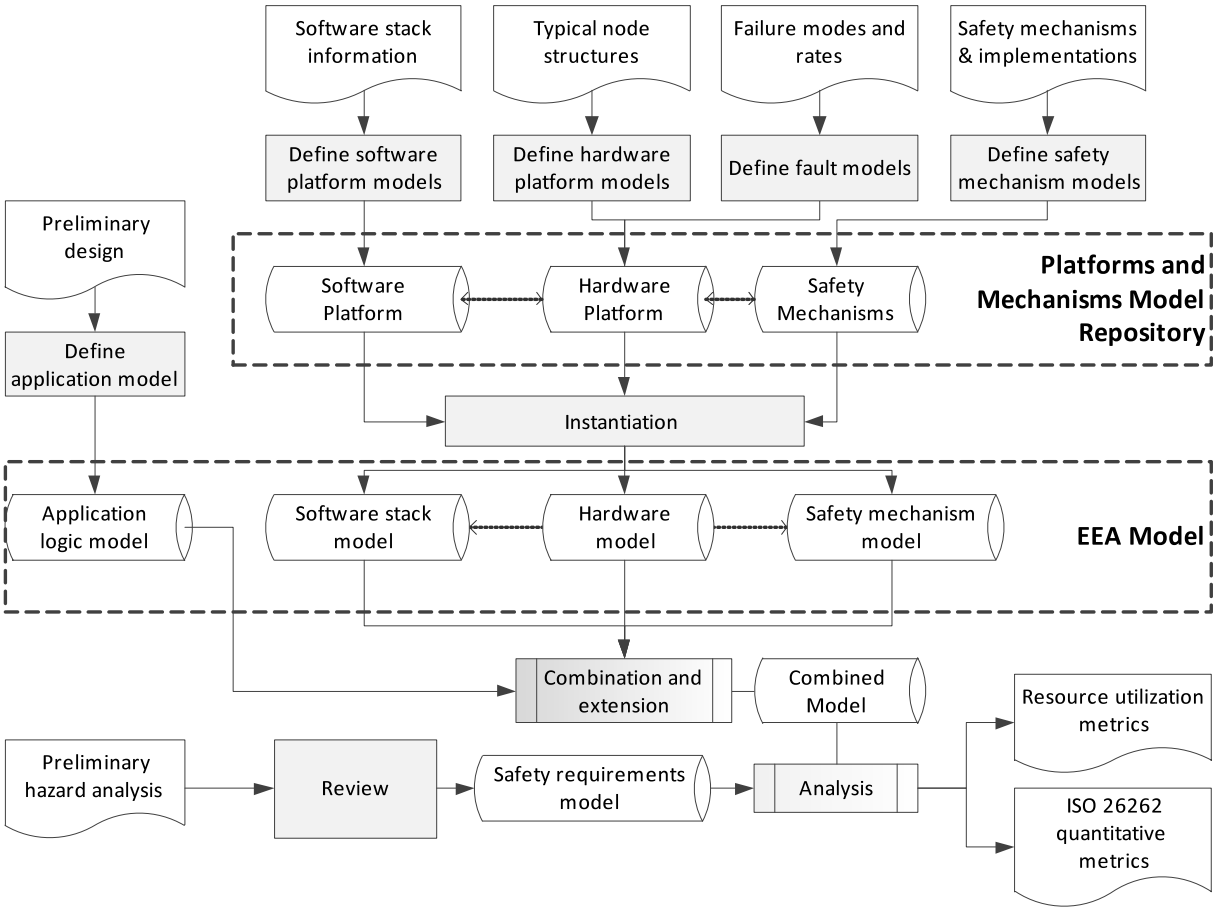

Fig. 2. A view on process inputs and artifacts from a single step perspective: gray process blocks require manual input from the architect, white are automated.

The design process that we propose allows rapid development of application models after some time has been invested into definition of models for platform and safety mechanisms. It also suggests iterative approach, as it contains phases of analysis and extension of architecture.

3.2. Structured description of EEA and safety mechanisms

As mentioned above, an adequate approach to modeling safety mechanisms and components allows partial automation of analysis activities while keeping the input of such models a straightforward process. In this section we present a suitable metamodel for representing the essential part of design artifacts.

Safety-critical systems development is a complex activity: it integrates numerous concerns in order to achieve safety. It is hardly possible to put all the concerns under one roof, and this has been valued as counter-productive by [15]. We try to define concerns separately and merge partial aspect-centered models to perform ongoing analysis and development of safety-critical systems.

At the top level, EEA is represented by a set of computing nodes, where functional networks responsible for execution of each item, and network or bus connections between those are defined. Network-related components can be treated as a separate node (e.g, as in [16]), and are at this level not in focus of our attention. To analyze the node-level fault behaviors and achieve flexibility in modeling safety mechanisms, the following requirements are to be considered:

1. Models of safety mechanisms have to be kept separate from component models, as a single safety mechanism can cover numerous failure modes of different components causing the same failure effect on the node level.
2. Flexible description of safety mechanisms should allow specialization with regard to particular attributes (such as specific algorithm, signature or array size) at later design cycles.
3. Semantic correctness (applicability of a specific mechanism to certain hardware component) has to be resolved.
4. Models need to provide sufficient information for bottom-up quantitative analysis up to node-level failure effects set, which acts as an interface layer to top-down analysis.

5. Architecture evolution must be enabled (by hierarchic approach to application modeling and decoupling of an application from a platform).

To satisfy these requirements, we developed a set of models representing different concerns and their relationships. The goal is to develop a minimal set of models that allow the user to represent all necessary concerns and which can be mapped to existing architecture description languages. One can break down the system description model parts into following categories:

1. functional components (representing the application);
2. software components and services, and related failure effect propagation models;
3. hardware components and corresponding fault models;
4. deployment-related model components: resources and communication;
5. safety mechanism models.

Out of the mentioned models, we do not concentrate on deployment models too much, as there is a lot of work going on in the community targeting effectiveness of using these.

3.2.1. Application components

The presented part of the metamodel is small, and contains only entities essential for modeling failure propagation relations. It can potentially be integrated with any system modeling framework. For example, in AADL[4] the corresponding concepts are modeled in "functional architecture" models (finer details on correspondence of our metamodel to existing languages can be found in Section 5.1). We explain the relations between entities in detail below.

To provide higher flexibility during system development, functional models are made hierarchical. **LogicComponents** represent members of the functional network, which correspond to software units to be present in the system. LCs communicate via **Ports**. Each logic component that communicates through ports has **Connector**s, each of which interfaces to a certain port, and is either a source or a sink. One source and unlimited number of sinks are allowed for ports. An LC also has **Pads** (by analogy with circuit board contacts, where smaller components are attached), entities connecting a Connector of a component to an internal Connector embedded into a Pad (Fig. 3). The reason for that is that direct communication of ports is just renaming, which is not implicitly supported in the metamodel. So pads perform port renaming while limiting possible use of this mechanism to hierarchic compositions.

3.2.2. Fault propagation

To model fault propagation at logic level, we need to assign fault behavior to system components, and fault propagation rules. For this reason we define the logic-level fault propagation metamodel. The essential part of the metamodel is presented in Fig. 4.

The main fault description entity on logic level is a **SwFailureEffect**. A Connector has associated corresponding SwFailureEffects which appear on the instance of the port throughout the system. A SwFailureEffect has a specific **FailureEffectClass**, which reflects the form of misbehavior. The set of FailureEffectClasses is defined by the user and can be both very generic and very detailed. The traditional classification into timing (early/late), sequence(commission/omission), and value (coarse/subtle) ([17]) might be sufficient in many cases (we use a subset of it in the further example).

In our analysis we use data segments as active fault propagation and transformation entities, which are LCs and CommComponents. **CommComponent**s are created in the system to represent LC communication replacing ports with exact communication semantics.

A set of **FailureEffectTransformationRule**s is associated with a **DataPathSegment** and defines the transformation of failure effects from input (sink) Connectors to selected output (source) in a LC, or from source to sink in a CC. We describe the semantics in some more detail when discussing analysis in Section 3.3.

3.2.3. Software and hardware platform

Modeling software platforms as a separate concern (Fig. 5) brings multiple benefits to decoupling of platform aspects from the application. **Service** abstraction separates an implementation of a service, which is a complex interaction of software components (e.g., I/O management or scheduling), from the purely functional requirement of **AppSwComponent**. The set of services is predefined and globally consistent, and corresponds in a typical setting to standard API (e.g., AUTOSAR RTE). **Resource** allocation to certain **SwComponent**s is managed via **ResourceRequirement**s. One hardware architecture can be supported by multiple software stacks with different resource requirements. In analysis phase these dependencies can be queried based on the actual set of services and software component **Dependency**s. Software dependencies are embedded into the stack model, so only service requirements from **AppSwComponent**s, which represent user application software, are coupled with these models.

[4] Architecture Analysis and Design Language, more details at: http://www.aadl.info.

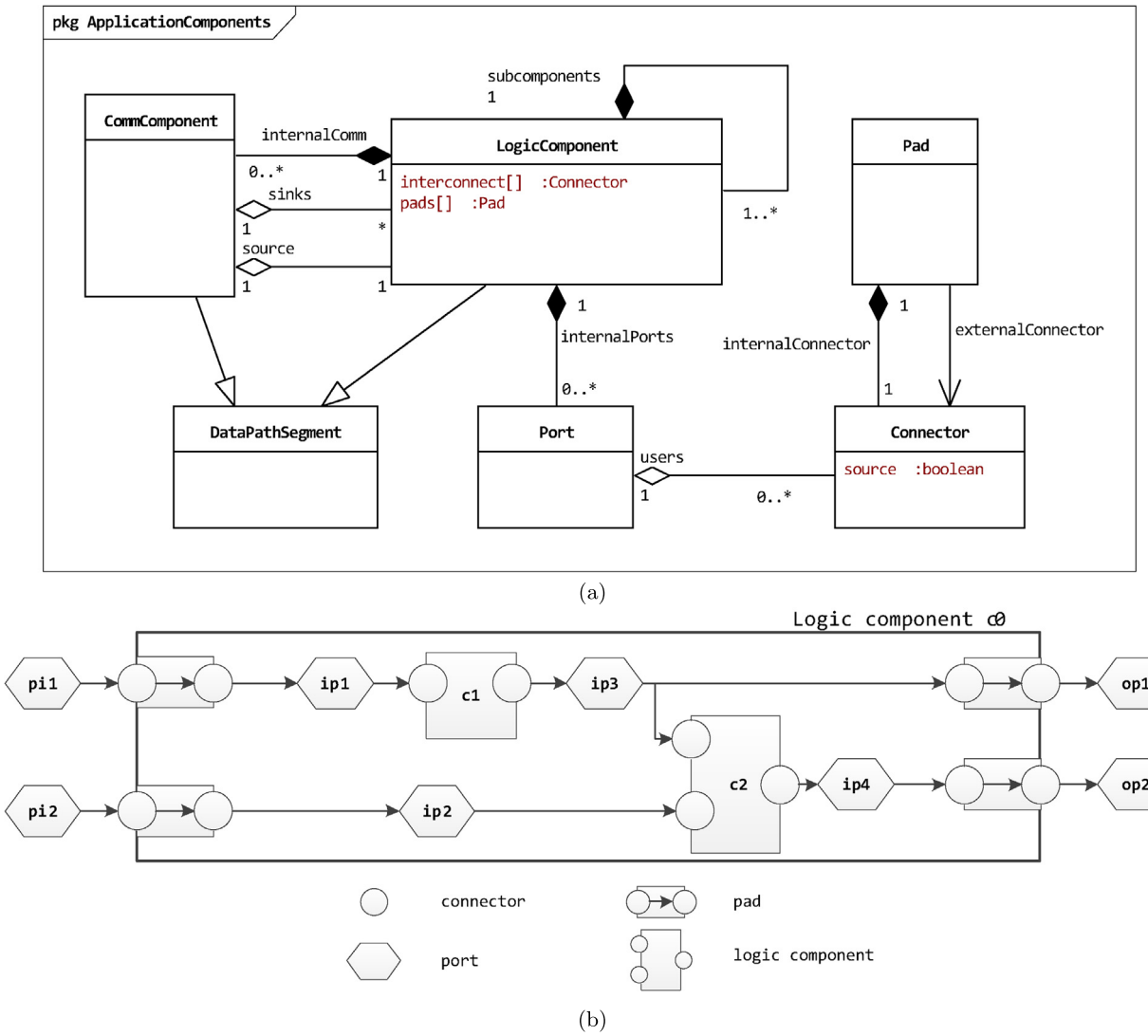

Fig. 3. Hierarchic description of application components: (a) metamodel, and (b) an example of logic component composition.

Hardware structure consists of computing **HwNode**s. A HwNode can be a member of one or multiple communication **Network**s and **PowerNetwork**s. A HwNode consists of **HwComponent**s. HwComponents provide Resources to SwComponents, and through this relation failure rates for corresponding **FailureMode**s are computed (Fig. 6).

The **HwFailureEffect**s are coarse effects that can be differentiated from others on the behavioral level. They provide links to FaultClasses to relate these effects to AppSwComponents. An example propagation path modeled system-wide for transient faults in memory looks as follows:

FailureMode: {single_bit_flip, odd_multi_bit_flip, even_multi_bit_flip} →
HwFailureEffect: {data_corr_transient} →
SwFailureEffect: {value_error, timing_error}

3.2.4. Safety mechanism models

Models of safety mechanisms are defining software components or hardware functions that can be embedded into the system design as a factor to limit the excessive failure rate that potentially violates the safety goal. We concentrate on modeling generic fault detection mechanisms, as these can be presented as a part of a platform and their selection can be automated easily.

A **DetectionMechanism** can have one or more implementations for a certain platform (Fig. 7). **HwDetectionComponentImpl**s denote components that are implemented in hardware (these are related to specific HwComponent). An example is a SRAM chip with built-in EDC logic. The **SwDetectionMechanismImpl** can be parametrized and is also a SwCompo-

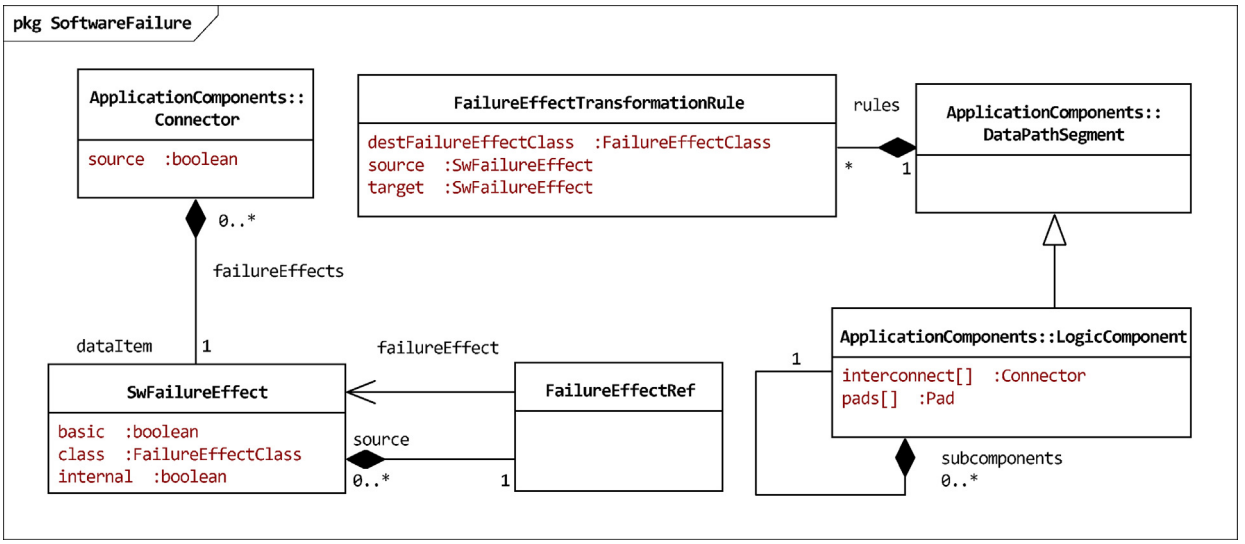

Fig. 4. Fault propagation at logic level.

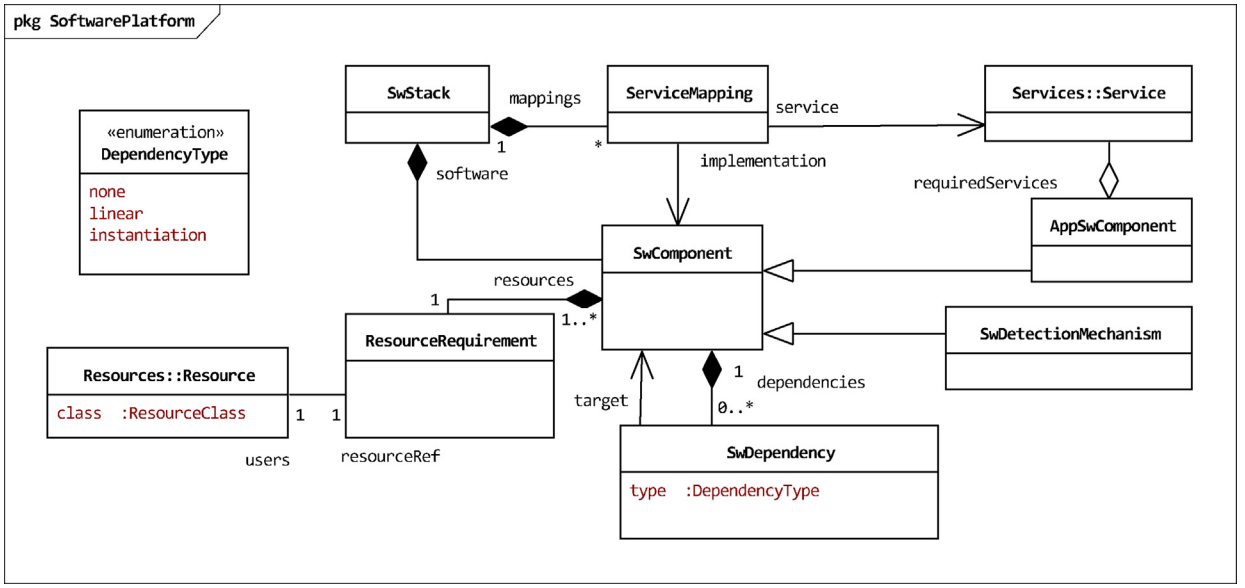

Fig. 5. Software platform metamodel.

nent, consuming certain resources for its execution. **DetectionCapability**s of a DetectionMechanismImpl define its coverage with relation to FailureMode that is covered. SwDetectionMechanismImpls are instantiated in the system model late at a HwNode, while they are unusual SwComponents and thus the same mechanism can be applied at different nodes where it protects different sets of SwComponents.

3.3. Conservative quantitative analysis

We describe in this subsection the approach to quantitative safety analysis that can be automated and makes use of the models defined above. We identify which safety requirements are defined, present an analysis framework to provide required metrics, and discuss how design qualities, other than safety, can be evaluated in this framework.

3.3.1. Standard requirements

The state-of-the-art requirements to EEA safety are defined by ISO 26262-5 [1, Part 5]. To simplify the diagnostic coverage assessment process, all the random hardware faults in ISO 26262 are classified into *single point*, *multiple point*, *safe* faults,

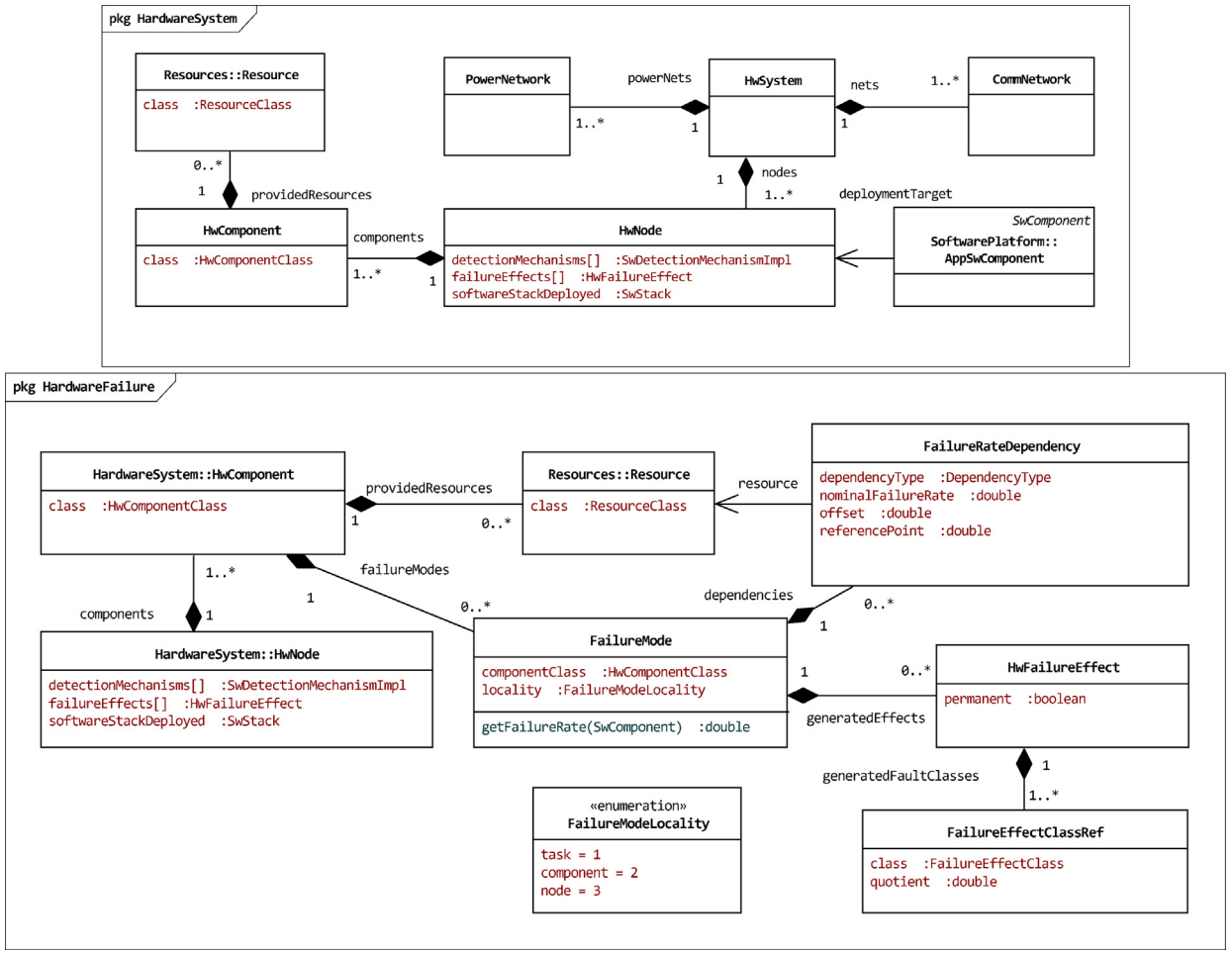

Fig. 6. Hardware system and failure modes metamodel.

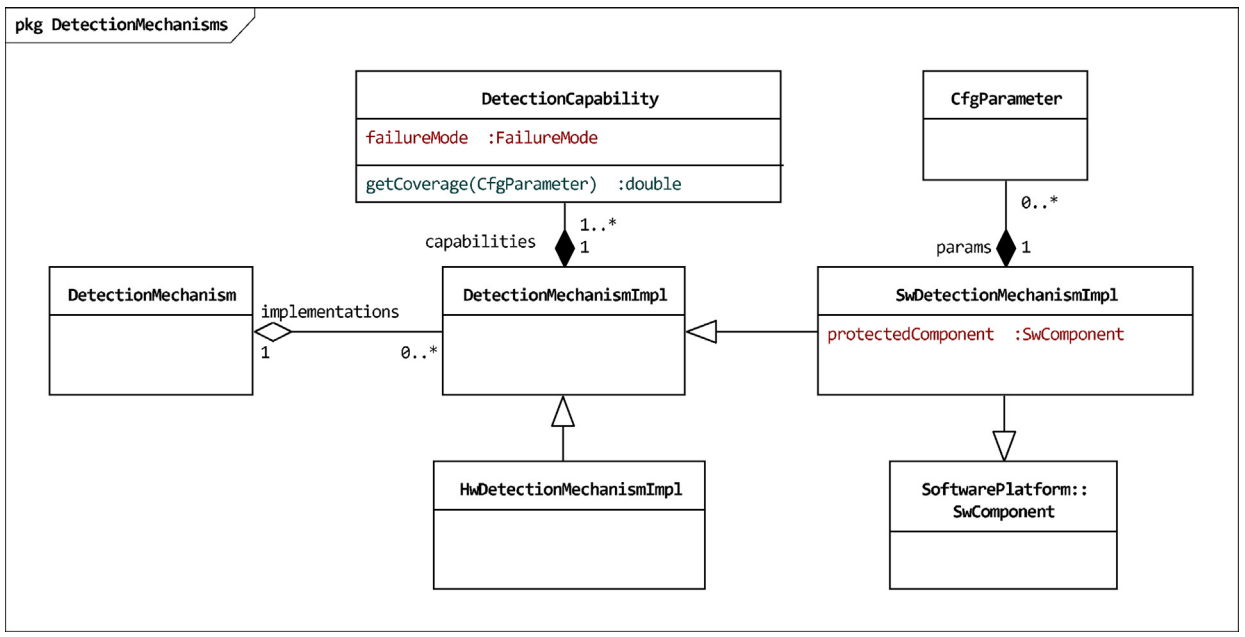

Fig. 7. Detection mechanisms metamodel.

Table 1
ASIL-specific target values for safety metrics [1, Part 5].

ASIL	*SPFM*	*LFM*	P_{VSG}
B	⩾90%	⩾60%	$<10^{-7}$ h^{-1}
C	⩾97%	⩾80%	$<10^{-7}$ h^{-1}
D	⩾99%	⩾90%	$<10^{-8}$ h^{-1}

and a set of architectural metrics is defined based on these classes. The most important categories are single-point faults, residual faults (not covered by any mechanism) and latent multiple point faults.[5]

$$SPFM = 1 - \frac{\sum(\lambda_{SPF} + \lambda_{RF})}{\sum\lambda} \tag{1}$$

$$LFM = 1 - \frac{\sum\lambda_{MPF\_latent}}{\sum(\lambda - \lambda_{SPF} - \lambda_{RF})} \tag{2}$$

λ_{SPF}, λ_{RF} and $\lambda_{MPF\,Latent}$ represent the failure rates of corresponding non-intersecting fault classes:

$$\lambda = \lambda_{SPF} + (\lambda_{detected} + \lambda_{RF}) + (\lambda_{MPF\_detected} + \lambda_{MPF\_perceived} + \lambda_{MPF\_latent}) \tag{3}$$

The higher the value of each of the architectural metrics, the more robust is the design. Target values for each metric along with target probability of violation of safety goal P_{VSG} are summarized in Table 1.

Another set of requirements are probabilistic metrics of hardware. Each ASIL level is assigned a quantitative target value enforced by ISO 26262. It can be based on analysis or field data for existing similar systems, or derived from standard recommendations (Table 1).

All faults in the system are assigned failure rate classes depending on how associated failure rate compares with the target for specified ASIL level:

$$\lambda_{class\,i} < \frac{P_{VSG\,max}}{10^{3-i}} \tag{4}$$

Based on failure rate classes, some ASIL-specific constraints are applied (for example, residual and single point faults are acceptable in an ASIL D system only if ranked as class 1).

We use the modeling schema proposed in the previous subsection to predict quantitative safety metrics required by ISO 26262 and to analyze a set of safety mechanisms for sufficiency when implementing a specific item.

It is important to understand that values resulting from quantitative analysis can be seen as the only driver of design in an ideal case: functionality of the item is stable and verified, and resource constraints are satisfied. In reality, this does not hold, so our framework needs to be aligned with the real processes of change management. The real goal of design is not making a system certifiable by analyzing it in the end of development, but designing it safe.

3.3.2. Analysis framework for ISO 26262

While existing generic analysis frameworks provide enough flexibility to define propagation rules between functional components, most of them utilize plain-structured models, which are hard to be built incrementally because of scalability issues. Hierarchic approaches work better in this case, as single components can be first defined as “black boxes”, and specialized internally afterwards. Therefore we define a specialized analysis framework that is inspired by component fault trees (CFT) [18] and fault propagation and transformation analysis (FPTA) [19]. In preceeding subsections we have given the reader basic ideas of the design process and models being used. In this subsection we strongly rely on the metamodels described above.

At the beginning of the analysis process all the input models are defined. In the Functional Component Model the sink-Connectors of the systems root component, are initialized with associated SwFailureEffects and corresponding failure rates. These effects will propagate through the system and reach its outputs.

Goal of analysis (safety goal) is defined as a set of safety requirements and an ASIL assignment. Each safety requirement is by its nature a negation of SwFailureEffects of given FailureEffectClass reaching systems boundary (certain Connector). An ASIL requirement implies the following constraints (Table 1):

$$\begin{aligned} &P_{VSG} < P_{VSG\,max} \\ &SPFM > SPFM_{min} \\ &LFM > LFM_{min} \end{aligned} \tag{5}$$

At the next stage initial structural model of the system is updated with system deployment details. At this point in the design process it is already decided where the LogicComponents are deployed. Based on this, Ports of the system are broken

[5] Scope of multiple point fault analysis is limited to order of two unless higher-order faults are shown relevant by the safety concept.

down into CommComponents, which can represent either internal communication on the node level, or network-based communication. After that the system under observation consists only of DataPathSegments. In addition each DataPathSegment is extended with a "virtual" internal Connector, which represents failure effects generated by the segment itself because of faults in underlying hardware.

Direct fault propagation. Data path fault propagation models need to be defined at the next stage. These are defined via simple rules, where multiple sources can be combined in one rule (this implies a conjunction of sources) and same target can have multiple rules (this implies a disjunction of input rules). A rule for a single FailureEffectClass, as a result, is presented as a logical formula in disjunctive normal form (DNF). As DNF is a canonical logic form, any existing software FTA results can be encoded by the rule. A rule might target a FailureEffectClass (FEC) or, if one has already been initialized manually, a specific SwFailureEffect (SwFE). An example of a full DNF rule construction could be built as following: if a function F1 depends on values from two different sensors $S1$ and $S2$, one rule for its output failure effect of class VALUE will be: F1.VALUE = S1.VALUE $\wedge$ S2.VALUE. Imagine we have a dependency on an additional input from a different function F2. The second rule will look as follows: F1.VALUE = F2.VALUE. When combined, these rules result in F1.VALUE = (S1.VALUE $\wedge$ S2.VALUE) $\vee$ F2.VALUE. We should note that defining rules is a very time-consuming task if performed manually.

Forward fault propagation model is a graph, consisting of vertexes corresponding to SwFailureEffects $\{\mathrm{SwFE}_i\}$ and edges corresponding to DataPathSegments $\{\mathrm{DPS}_j\}$. An activation of a vertex means creation or modification of the target SwFailureEffect using the rules that are defined at the input. All vertexes are activated, which leads to a cascade of propagation waves. After stabilization activation of propagation rules leads to absolutely no change in the graph.

Inverse fault propagation. A set of fault trees[6] for analysis is extracted from this model. The algorithm for fault tree extraction is very simple and is based on topological ordering of the SwFailureEffect dependency graph, followed by interpretation of rules in the reverse order – from the affected output to inputs, concatenating the rules in DNF form into a more complex expressions. The resulting fault tree lacks failure rates, but can already be utilized to identify bottlenecks in the design. The failure rates are calculated through utilization of the rest of the models.

Failure rate calculation. First, the software dependencies are analyzed. During the deployment transformation each LogicComponent is extended to AppSwComponent, for which required services need to be defined. It is then possible to select SwComponents which implement the service for the specific software stack (defined for a HwNode), and SwComponents, on which these depend. In this way platform models simplify deployment specification. After full lists of SwComponents (SwC) for each node are determined, for each pair $\{\mathrm{SwC}_i, \mathrm{FEC}_k\}$ hardware failure effects $\{\mathrm{HwFailureEffect}\}_{i,k}$, which can cause failure effects of class FailureEffectClass (FEC), are chosen to determine the set of relevant failure modes.

Next level of fault behavior is defined by FailureModes (FM) of HwComponents. Partial failure rate $\lambda(\mathrm{FM}, \mathrm{SwC}_i, \mathrm{FEC}_k)$ is determined through resource usage analysis (via utilizing FailureRateDependency). A FailureMode can be independent from resource usage (e.g., address decoder failures for RAM), or a dependency of the failure rate can be defined via a formula (e.g., probability of so-called "soft errors" depends linearly on the space occupied in the memory array).

Partial residual failure rate for one DetectionMechanism Impl (DMImpl) is calculated as:

$$\lambda_{\mathrm{residual}}(\mathrm{FM}, \mathrm{SwC}_i, \mathrm{FEC}_k) = \big(1 - \mathrm{DC}(\mathrm{DMImpl})\big) \times \lambda(\mathrm{FM}, \mathrm{SwC}_i, \mathrm{FEC}_k) \tag{6}$$

A sum of partial failure rates over all relevant failure modes through all HwComponents results in $\lambda_{\mathrm{residual}}(\mathrm{SwC}_i, \mathrm{FEC}_k)$, which is the failure rate for the SwComponent deployed to a specific node. Finally, for an AppSwComponent the internal failure rate is a sum of failure rates of all dependencies, and can be assigned to the SwFailureEffect of the corresponding class related to the "virtual" internal Connector.

Quantitative metrics. To perform ISO 26262 assessment we need to perform analysis of the same fault tree with different failure rate inputs:

- A_{nm} is an analysis of the fault tree utilizing values $\lambda(\mathrm{SwC}_i, \mathrm{FEC}_k)$;
- A_{s} is an analysis of the fault tree, where $\lambda_{\mathrm{residual}}(\mathrm{SwC}_i, \mathrm{FEC}_k)$ values are used.

An analysis of a fault tree provides probability of top-level event and minimal cutsets – conjunctions of failure effects that lead to the top-level event. The following failure rates are calculated from the fault tree analysis results:

- λ_{SPF} is the sum of individual failure rates, which contribute directly to undesired event (cutsets of rank 1 in A_{s});
- λ_{RF} is the probability of top-level event in A_{s};
- $\lambda_{\mathrm{MPF_{latent}}}$ is the sum of all cutset components of rank $r = 2$ from A_{s};
- λ_{total} is the probability of top-level event in A_{nm};
- P_{VSG} is the probability of top-level event in A_{s}.

Single-point fault metric can be now calculated using the formula (Eq. (1)). Computation of latent fault metric is a more complex task, as "perceived" failure fraction is often very hard to estimate. Our approach results in a very conservative value, so additional manual review of the artifacts might be required.

[6] http://www.fault-tree.net.

3.3.3. Analysis of resource dependencies

Modeling resources of hardware components is one of important parts of the metamodel. On one hand, the resources (such as processor time or memory footprint) are of great interest on their own – they represent design quality attributes that need to be evaluated to prove that the system can perform correctly in a given deployment. On the other hand, failure rates designated to certain logic components depend on the pattern and proportion of resource access (i.e., usage profile). The resources have been initially defined (Fig. 6) with the goal to allow evaluation of failure rates, but can be also used to conservatively estimate the system's slack.

As the SwDetectionMechanismImpl is a SwComponent, the same resource requirements can be modeled for safety mechanisms. System architect can then perform selection of safety mechanism implementations, or even safety mechanisms per se, based not only on their coverage, but also on the computed resource balance.

Evaluation of resource utilization metrics is dependent on the metric selected. In the defined metamodel we do not limit the user in selection of metrics, as well as which properties can be defined as resources. Selection of good metrics and establishment of a link between those and external dedicated analysis tools (e.g., schedulability analysis) enables significant increase of architect's outlook.

4. Implementation of tool support for architecture safety evaluation

We have developed a modeling tool prototype that allows definition of platform models, safety mechanisms and application models, refinement and analysis of these models in accordance to the methodology. In this section we present the usage workflow for the tool prototype, details of implementation and validate the tool with a simple example. The experiences gathered during tool development are discussed in Section 5.2.

4.1. Motivation

At the stage of design or modification of an E/E system full structural and functional description of the target system is often not accessible because many functional components are not yet decomposed and/or implemented, and final details become available at integration stage. Yet one of the tasks of this stage is to evaluate different configurations of safety mechanisms, provided with the knowledge about possible/expected system software deployment, platform services and hardware fault models. This pays off: if the selection is correct (even if it is conservative), system architect can be sure that implementation of the design decisions leads to a predictable system design. Evaluation of safety can then be omitted at the later stages up to system level, provided that sufficient testing effort ensures correctness of implementation. In our case we have a model of the whole system created by the system architect. This model can be changed and updated depending on its maturity level, deployment of software functions to hardware might change, data from the field trials of components might be used to update/refine failure modes/effects and failure rates, but one point is still unchanged: analysis of the whole system is possible based on relatively compact user input.

4.2. Implementation

The modeling tool has been developed using Eclipse Modeling Framework (EMF) infrastructure. Structurally the tool consists of the following blocks:

1. metamodels defining the restricted set of entities that are allowed for use;
2. Java classes generated from the metamodel definition and extended by manually implemented methods;
3. graphical user interface (GUI) is generated from metamodels and allows tree-based editing of the models;
4. fault tree analysis (FTA) engine wrapper generates xml-files to initiate analysis of logic fault propagation models;
5. utilitary functions providing analysis and shortcutting functionality.

The metamodels have been defined as eCore models. The generated code has been extended by adding Java code to implement non-modeled aspects, such as analysis engine interface and model transformations. Metamodels have been implemented as cross-linked small models (to manage complexity), organized within a single class, which we called AnalysisRequest. In a different setting the metamodels are organized to form a repository, where software stacks, predefined hardware nodes and safety mechanisms can be stored in a cross-linked form.

Model storage and exchange is supported via XML Model Interchange (XMI) format provided by the EMF. As a format to generate fault trees we have chosen OpenPSA.[7] It is a joint initiative of multiple tool developers and agencies/institutions who apply probabilistic safety assessment methods. It is an open format, which supports modeling fault/event trees, Markov models, common cause failures, etc. Multiple tools implement it, so that analysis results can be easily checked with a completely different tool. We have built a wrapper around XFTA, a freeware tool by Antoine Rauzy [20]. This tool implements fault tree analysis with OpenPSA input format. The results are available as a text file which is formatted in an easy-to-parse way. This brings us flexibility in future to utilize any other tool supporting OpenPSA and providing command-line interface.

[7] www.open-psa.org.

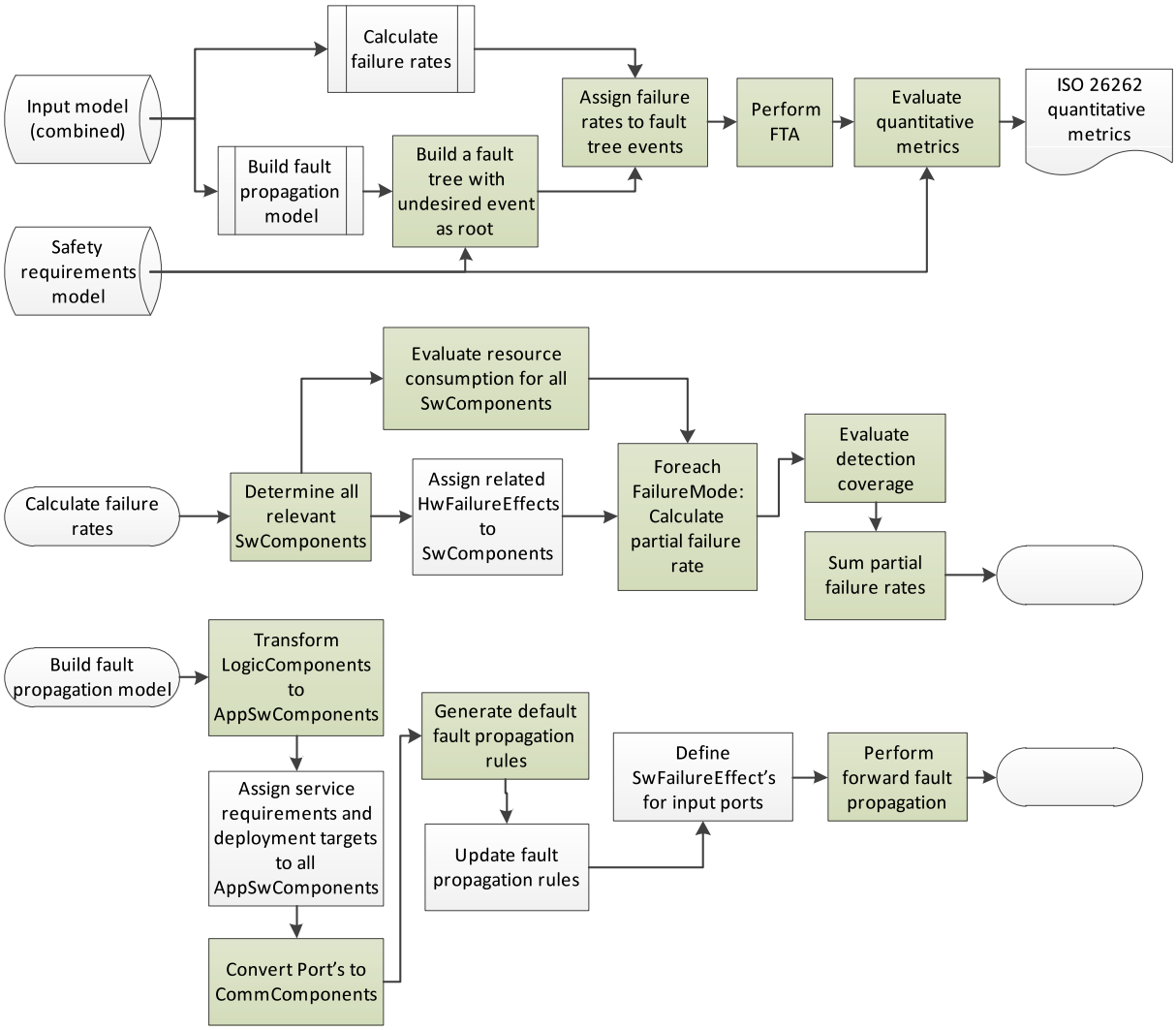

Fig. 8. Typical analysis workflow in the implemented tool.

4.3. Modeling and analysis of E/E architecture

The architecture of the tool follows the proposed analysis flow, but in addition to that we have made an attempt to support model developer in definition of routinely similar relations in the model.

The workflow in our tool differs during development of models for repositories and applications (Fig. 8). It is expected that first the platform models are defined by the architect, as well as models of safety mechanisms are provided. This is very straightforward, and only consistency of models needs to be monitored.

At the second stage the application model is defined in an AnalysisContext in form of LogicComponents and connecting ports. The LogicComponents can be further stepwise refined, their internal structure can be specified at later steps of design. In the final model of a system, a LogicComponent of the lowest hierarchy level has to correspond to an atomically-deployed software component.

Typically a predefined set of HwNode's exists as a part of platform model repository. These are copied by the application developer to an application-specific model from a respective RepositoryContext model, node names are changed. FailureEffectClasses are defined, which reflect the level of detail for fault propagation analysis. Detection mechanisms also need to be mapped manually. Input Connectors of the top-level LogicComponent are manually initialized by SwFailureEffects and corresponding failure rates. LogicComponents are automatically transformed by the tool into AppSwComponent entities (the latter are inherited from the former, see Fig. 5). Fault propagation rules in our tool are initialized by automatic generation of rules in the most conservative form: all the input SwFailureEffects reach each of the output SwFailureEffects. CommComponents are also generated automatically by the tool with a similar assumption. This needs to be corrected manually

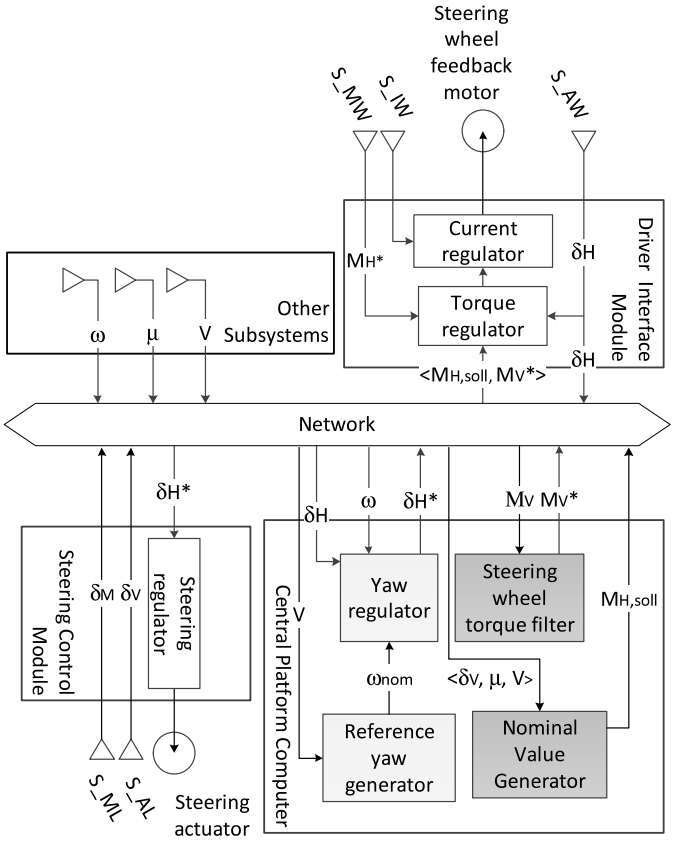

Fig. 9. Structural diagram of the example steer-by-wire system.

if different (less conservative) fault propagation take place. In the completing step, all HwFailureEffects are mapped to the corresponding SwFailureEffects.

Analysis of the models follows the process described in Section 3.3. It is supported via transformation of the fault tree model into OpenPSA format, feeding the generated file to XFTA and parsing of the results. SPFM and LFM parts are computed in our tool using the cutsets generated by XFTA. After the analysis is completed, corresponding report-objects are generated. In case of need, template-based text file generation can be executed at this stage.

4.4. Evaluation: a steer-by-wire system

To evaluate the tool, we have analyzed an advanced use case. In the following subsection we demonstrate our modeling approach from a platform definition to implementation details, analysis generation and safety mechanism definition. We analyze only the safety metrics, leaving resource utilization and cost evaluation out of scope, but it should be clear, that the combined model contains enough data to support such properties in an optimization process.

Our example system is a steer-by-wire system (Fig. 9). It features a number of multi-level control loops, which makes it hard to analyze fully using traditional techniques like FTA. The steer-by-wire system functionality and initial architectural layout are inspired by [16] and [21]: the steering-related functions are allocated at the nodes of an ICT with centralized concept. The driver input controls and steering rack are parts that need to be spatially separated, so such a system in centralized form is deployed onto at least three different electronic control units (ECUs).

Safety requirements and fault classes. This function has no safe state: safety is maintained through maintaining "operational" mode of this function. The safety goal is "no uncontrolled steering should occur". For most usage scenarios (except parking and low speed driving) probability of exposure factor is assumed to be E4, controllability factor – C3 and severity factor – S3 (all maximal values possible). The resulting ASIL level is D. ASIL D requirements according to ISO 26262 (metric values and probability of safety goal violation) are:

$$\text{SPFM} > 99\%; \qquad \text{LFM} > 90\%; \qquad P_{\text{VSG}} < 10^{-8}\ \text{h}^{-1}$$

The failure effect classes that we use for this model are: {TIME, VALUE}; we can assign different fractions of CPU failure modes to such failure effects. Models of safety requirements are set up in a straightforward manner: the safety goal is defined as a combination of two requirements: no incorrect timing at the steering rack actuator and no incorrect value. So during analysis two different trees are built for this safety goal.

Table 2
Coverage of safety mechanisms [1, Part 5, Annex D].

Mechanism	FailureMode	Coverage, %	Notes
SEC/DED	SingleBitTransient	100	Hamming (72, 64)
	MultipleBitTransient	98	
Monitoring with CRC	SingleBitTransient	100	Parameters:
	OddBitTransient	100	CRC-32;
	EvenBitTransient	90	BlockSIze = 8K
Block	SingleBitTransient	100	
Replication	OddBitTransient	100	
	EvenBitTransient	100	
	Permanent	99	
MARCH B	Permanent	90	Period dependent
MSCAN	Permanent	48	Period dependent

Table 3
Failure rate computation for a 32-bit microprocessor based on MIL HDBK-217. Model formula used: $\lambda_p = (C1 \times \pi_T + C2 \times \pi_E) \times \pi_Q \times \pi_L$.

Parameter	Value	Remarks
$C1$	0.56	Complexity failure rate for 32 Bit Microcontroller
π_T	0.35	Temperature Factor
$C2$	0.0425	Package failure rate for 100 Pin LPFQ package
π_E	4.0	Environment Factor for Ground Mobile
π_Q	10	Quality factor assumed
πL	1.0	Learning Factor: mature product
λ_p	3.66	Failures/10^6 hours
λ_{FIT}	3660	FIT

Table 4
Results of analysis for different safety configurations.

Safety mechanisms applied	Failure rate, FIT[a]
None	4025.69
CRC	3890.366
CRC + March B	3863.85
SEC/DED + Galpat	3869.34
CRC + March B + CPU test	1490

[a] 1 FIT – 1 Failure in Time, is equivalent to 10^{-9} h^{-1}.

We have set up one *platform model* for all hardware nodes, where a 32-bit microcontroller with an external memory chip have been used as main computational component. Failure rates for components have been approximated using the failure rate prediction formulas and manufacturer data from open sources. Some handbooks, like [22] An example calculation for the MCU is presented in Table 3. A simple software platform model with multiple dependent component layers has been defined.

The *application model* is modeled straightforwardly with LogicComponents and Ports, following the defined metamodel. We have defined models for a range of *safety mechanisms*, in particular – memory protection mechanisms (see Table 2), both hardware (e.g., single-error-correcting/double-error-detecting code, SEC/DED) and software-based (e.g., memory tests). We have analyzed the system with different mechanism allocations (uniformly assigned to all SwComponents). The *results* of automated analysis of a system model completely match the results of a manual validation through part count method (as we did not use complex failure effect transformation rules in the logic model). Some of the results are presented in Table 4. Our initial suggestions have been fully confirmed: to reach ASIL D, a single-channel system needs to be protected by a higher number of safety mechanisms. As a reference to compare against, a CPU test has been defined and included into the system model. As we can see, the CPU contributes significantly more into the total failure rate.

We would like also to note that a strict resource modeling scheme will, once established, bring significant benefit to selection of safety mechanism implementations. For example, when using a CRC code to protect data from transient faults, important parameters for estimating diagnostic coverage are: monitored block size and specific generator polynomial. From the perspective of resource overhead these parameters play a minor role, – an important decision is selection of the mechanism implementation. CRC can be implemented in a straightforward bit-by-bit polynomial division, or, alternatively, using a lookup table. The former algorithm takes more time and cpu resources, but is very simple, so code and data memory requirements are very low; the latter uses more memory (code or data), but consumes less CPU time. Depending on the available resources on the node, selection in this case might be different. Analysis of such a trade-off is possible using the presented models, but current freedom of resource definition makes it almost impossible.

5. Results and discussion

In this section we make a compact review of existing work in the area, share our outlook on MDE, and conclude the paper with a short summary.

5.1. Overview of related work

The rapid growth of the number and importance of E/E systems resulted in tailoring of the IEC 61508 [8] standard for automotive domain and development of ISO 26262 safety standard [1]. It completes the existing homologation regulations (e.g., FMVSS[8] or ECE norms[9]) by introducing additional constraints on the way how the components of E/E systems are developed or reused, integrated and verified.

In many cases ISO 26262 is seen only as a collection of process practices, which are important for developing a dependable product in limited time [23]. This leads to full or partial avoidance of quantitative measures. In [24] the authors present a methodology with the goal of integration of architecture and failure net modeling, allocation of safety mechanisms to architectural elements, and traceability to requirements and test coverage. The discussion, however, concentrates around systematic requirements tracking (DOORS), reflection of requirements to system architecture (SysML) and tracing them down to analysis tools. The authors of another engineering support method [25] take an artifact-centric point of view and concentrate on the modeling approach of all aspects of E/E architectures, which is implemented in an "PREEvision" toolset. Selection of right safety mechanisms is, however, not supported sufficiently in the mentioned tools.

There are a number of developing model-based dependability analysis techniques that are aiming at specification and automated analysis of EEA dependability. FTOS [26] is a tool for synthesis of fault-tolerant real-time systems. It provides a system engineering approach which allows (under an assumption of model correctness) generation of source code for real-time systems. FTOS allows modeling of failure modes of components and their probabilistic behavior, but does not provide quantitative evaluation of achieved system safety.

An approach to selection of safety mechanisms has been proposed in [27]. It is similar to our approach in that a "library of diagnostic techniques" is used to deliver safety mechanisms for IEC 61508 compliance. The evaluation of alternatives during selection stage is performed inside the library and is not specified in detail that allows comparison with our approach.

Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS, [14]) is a top-down methodology to perform automated safety analysis of component-level hierarchical models using information from interface-focused FMEA. Relation to top-level safety goals is maintained through manually performed functional failure analysis. Optimization of dependability is addressed by HiP-HOPS in [28], but the evolutionary concept is missing: a separate model is required for early design stages. Often detailed fault propagation logic is not available for components of automotive systems, which renders the approach less useful than in military and aerospace domains.

Other related approaches to failure logic analysis build on component fault trees (CFT, [18]), which wrap the failure behavior into graph-based modularized fault trees, and on Fault Propagation and Transformation Calculus (FPTC, [29]), where a clear notation is used to describe the local failure logic of the components. These methods are not intended to provide full engineering support and can be used in combination with our design process.

As we neither set definition of a full-blown modeling language as a goal or did it, it is important to keep an eye on the metamodel landscape to find similarities and contributing ideas. In the modeling area the most relevant standards are AADL[10] [30] – a wide-spread architecture description language, MARTE[11] – a UML profile for modeling real-time systems, and EAST-ADL[12] – a domain-specific language for development of automotive electronic systems. AADL is the most relevant standard: it comes from the avionics domain, and also follows the principle of separation of concerns (software, hardware, application). A quick analysis has shown that even though one-to-one mapping of metamodels is not possible, due to similar principles our models can be transformed to AADL models and vice versa. MARTE as a real-time profile concentrates on precise timing behavior modeling. Due to its generic nature MARTE supports modeling of other quality attributes including failure behavior. EAST-ADL provides multiple levels of vehicle description, where "Design Level" models are in many points similar or intersecting with our concepts (e.g., hardware components are put in a hierarchy using prototypes). We see MARTE and EAST-ADL as perspective metamodels that are suitable for integration of our approach.

Our method is based on an application-specific safety prediction framework. Similar to generic framework presented in [31], it includes all the elements required (encapsulated evaluation models, operational/usage profiles, composition algorithm and evaluation algorithm) but is adapted to automotive system domain and its specifics. The key differences are practical: concentration on a single domain with the goal to optimally select and configure safety mechanisms.

In summary, system synthesis oriented tools currently lack the safety constraints and decision support for early modeling steps. Our design methodology can provide significant benefits for the evolution-time analysis and optimization of E/E architectures, especially for selection of safety mechanisms.

[8] Federal Motor Vehicle Safety Standards is a series of regulations issued by National Highway Traffic Safety Administration.
[9] Regulations issued by United Nations Economic Commission for Europe.
[10] Architecture Analysis & Design Language (SAE AS5506A), http://www.aadl.info.
[11] Modeling and Analysis of Real-Time and Embedded Systems, http://omgmarte.org.
[12] Electronics Architecture and Software Technology – Architecture Description Language, http://www.east-adl.info.

5.2. Experience with model-based engineering

In this subsection we share some experiences on the setup of model-based engineering tool. First of all, domain modeling is very important, even if you do not use those model-based techniques. It allows critical analysis of subsystem interfaces and better understanding of the context in large. After this step model transformations can be much easier defined, and implementation is already obvious for the developer.

In the modeling area we have not gone beyond state of the art. The aim of the tool development was demonstration of practical feasibility and maturity of our approach. The metamodels that we developed have the goal of demonstrating that we are working with generic models, not safety-concentrated idiosyncratic language, and that these models can be extended in different directions. For example, necessary aspects for schedulability analysis can be introduced, and deployment can be generated, or at least validated, automatically.

We realize that standard widely used metamodels exist (e.g., AADL), where the same aspects are represented with, perhaps, same or lower level of concern decoupling. It does limit some of the features (like repository-based approach), but is fully compatible with the rest (analysis framework and safety mechanism selection). In the nearest future we plan to map our metamodel to AADL and implement selection of safety mechanisms without "yet another metamodel" in place. We admit that the approach to utilize the resource models is far from perfection, and will address this in our AADL application.

The implementation of model-based methodology has shown some inflexibility in our setting. We see this as a consequence of our approach to model transformations. One reason is the use of endogenous model transformations on the whole modeling and analysis path. Using exogenous transformations at some point seems useful to limit the volume of information contained in a model, and to exclude the possible inconsistence.

Another issue with model-based approach is that modeling flexibility and extension points need to be encoded in the metamodel itself (or be implemented outside the modeling infrastructure). An alternative approach is to use multilevel metamodeling (e.g., one presented in [32]), which allows both architect and application developer work at the model level, while the architecture model is transformed into a metamodel, in terms of which the application is defined.

To conclude, apart from demonstration of feasibility of our approach, development of the model-based tool has proved the maturity of MDE technologies, and the short learning curve.

5.3. Conclusion and outlook

In this paper we presented a method to evaluate design choices early in development process and iteratively add architectural detail until a specific safe and cost-effective E/E architecture is derived. We proposed model repositories to collect an important part of domain knowledge (coverage of safety mechanisms and associated resource trade-offs), so design decisions can become more deterministic. At the same time instances of repository models can be used to perform safety evaluation following a well-defined workflow. As an effect, evidence and arguments on safety-essential attributes, such as coverage and fault models, are continuously accumulated within design infrastructure, which makes reuse of this data, and even of the whole reference architecture, possible.

To provide maximum benefit, the presented approach has to be applied in a model-based development process. In such a setting requirement-driven iterative process provides high traceability from input requirements to implementation and quantitative argumentation necessary for ISO 26262 certification. Our preliminary evaluation with a tool prototype shows that iterative evolution allows concentration at sufficiently high modeling level (thus, reducing the required level of expertise) to make design decisions. Exact fault models are used (if necessary and available at all) only at late design steps, when related information is available. We have not used design quality attributes (particularly, resource-related constraints) in the tool prototype, so complete set of metrics remains an open point. Applicability of the approach at the larger scale (as well as issues of integration with real processes in the industry) is very important and will be considered in the nearest future.

Further refinement and formalization of our methodology heads in multiple directions. We are working on support for SysML models as input. SysML models are already used to represent automotive architectures: they have been proven suitable for continuous data streams [24], and are often used in a tight combination with MARTE profile, which allows coordination of safety mechanism selection with real-time behavioral models of the E/E architecture. Another perspective extension is related to AADL Error Annex, where uses a state-based formalism to define fault models. Taking into account existence of analysis methods for AADL fault models, and the fact that AADL is a widely used cross-industrial standard, integration of our approach with AADL models has a good chance to result in a solid standard-supported toolchain.

Development and integration of engineering support tools remains an important prerequisite of a qualitative change in the E/E architectures. Our methodology has the potential to speed up preliminary safety analyses for automotive systems, assisting safety architect in generating design decisions that allow rapid adaptation to the ISO 26262 development lifecycle.

Role of the funding source

The work presented in this paper has been partially funded by Siemens AG.

References

[1] ISO 26262:2011, Road vehicles – Functional safety, International Organization for Standardization (ISO), TC 22/SC 3, 2010.
[2] T. Dittel, H.-J. Aryus, How to "survive" a safety case according to ISO 26262, in: E. Schoitsch (Ed.), SAFECOMP 2010, in: Lecture Notes in Computer Science, vol. 6351, Springer, 2010, pp. 97–111.
[3] H. Heinecke, W. Damm, B. Josko, A. Metzner, H. Kopetz, A.L. Sangiovanni-Vincentelli, M. Di Natale, Software components for reliable automotive systems, in: K. Preas (Ed.), Design, Automation and Test in Europe 2008, IEEE, 2008, pp. 549–554.
[4] M. Fowler, Is design dead?, in: G. Succi, M. Marchesi (Eds.), Extreme Programming Examined: Proceedings XP 2000, Addison-Wessley, 2001, pp. 3–18.
[5] R. Isermann, R. Schwarz, S. Stolzl, Fault-tolerant drive-by-wire systems, IEEE Control Syst. 22 (5) (2002) 64–81.
[6] A.W. Brown, J.A. McDermid, The art and science of software architecture, in: F. Oquendo (Ed.), Proceedings ECSA 2007, in: Lecture Notes in Computer Science, vol. 4758, Springer-Verlag, Berlin/Heidelberg, 2007, pp. 237–256.
[7] A. Avizienis, J.-C. Laprie, B. Randell, C.E. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Trans. Dependable Secure Comput. 1 (1) (2004) 11–33.
[8] IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems, International Electrotechnical Commission (IEC), TC 65/SC 65A, 2010.
[9] RTCA/DO-178C Software Considerations in Airborne Systems, Equipment Certification, RTCA, Inc., 2011.
[10] J. Börcsök, Funktionale Sicherheit: Grundzüge sicherheitstechnischer Systeme, Hüthig, 2006.
[11] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, Emf: Eclipse Modeling Framework 2.0, 2nd ed., Addison-Wesley Professional, 2008.
[12] O. Lisagor, T. Kelly, R. Niu, Model-based safety assessment: review of the discipline and its challenges, in: B. Xu, M. Zhao, T. Zhao (Eds.), Proceedings 9th International Conference on Reliability, Maintainability and Safety (ICRMS), IEEE, Guiyang, China, June 2011.
[13] C. Buckl, A. Camek, G. Kainz, C. Simon, L. Mercep, H. Staehle, A. Knoll, The software car: Building ICT architectures for future electric vehicles, in: Proceedings of the First IEEE International Electric Vehicle Conference (IEVC), IEEE, 2012.
[14] Y. Papadopoulos, J. McDermid, R. Sasse, G. Heiner, Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure, Reliab. Eng. Syst. Saf. 71 (3) (2001) 229–247.
[15] E.W. Dijkstra, On the role of scientific thought, in: Selected Writings on Computing: A Personal Perspective, Springer-Verlag, New York, NY, USA, 1982, pp. 60–66.
[16] R. Reichel, M. Armbruster, X-by-Wire platform – concept and design, Autom.tech. 59 (9) (2011) 583–596.
[17] P. Fenelon, J.A. McDermid, M. Nicolson, D.J. Pumfrey, Towards integrated safety analysis and design, in: H. Berghel, Paul Chung (Eds.), ACM SIGAPP Appl. Comput. Rev. 2 (1) (March 1994) 21–32 (Special issue on safety-critical software).
[18] B. Kaiser, P. Liggesmeyer, O. Mäckel, A new component concept for fault trees, in: P. Lindsay, T. Cant (Eds.), Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software, vol. 33, Darlinghurst, Australia, 2003, pp. 37–46.
[19] X. Ge, R.F. Paige, J. McDermid, Probabilistic failure propagation and transformation analysis, in: B. Buth, G. Rabe, T. Seyfarth (Eds.), Proceedings 28th International Conference on Computer Safety, Reliability and Security (SAFECOMP), in: Lecture Notes in Computer Science, vol. 5775, Springer-Verlag, Hamburg, Germany, 2009, pp. 215–227.
[20] A. Rauzy, Anatomy of an efficient fault tree assessment engine, in: R. Virolainen (Ed.), Proceedings of International Joint Conference PSAM'11/ESREL'12, June 2012.
[21] E. Dilger, P. Ahner, et al., US Patent 6 219 604, Steer-by-wire steering system for motorized vehicles, Robert Bosch GmbH, 1999.
[22] The Reliability Information Analysis Center (RIAC), FMD97: Failure mode/mechanism distribution, RAC, 1997.
[23] P. Löw, R. Pabst, E. Petry, Normiert auf die Strasse, iX kompakt 1 (2011) 136–138.
[24] B. Kaiser, V. Klaas, S. Schulz, C. Herbst, P. Lascych, Integrating system modelling with safety activities, in: E. Schoitsch (Ed.), International Conference on Computer Safety, Reliability and Security, in: Lecture Notes in Computer Science, vol. 6351, Springer, 2010, pp. 452–465.
[25] M. Hillenbrand, M. Heinz, N. Adler, K.D. Müller-Glaser, J. Matheis, C. Reichmann, ISO/DIS 26262 in the context of electric and electronic architecture modeling, in: H. Giese (Ed.), Proceedings ISARCS 2010, in: Lecture Notes in Computer Science, vol. 6150, Springer, 2010, pp. 179–192.
[26] C. Buckl, D. Sojer, A. Knoll FTOS, Model-driven development of fault-tolerant automation systems, in: Proceedings 15th IEEE International Conference on Emerging Technologies and Factory Automation, IEEE, 2010, pp. 1–8.
[27] D. Sojer, D. Knoll, C. Buckl, Synthesis of diagnostic techniques based on an IEC 61508-aware metamodel, in: Proceedings of the 6th Symposium on Industrial Embedded Systems (SIES 2011) Work-in-Progress Session, IEEE, June 2011, pp. 59–62.
[28] M. Adachi, Y. Papadopoulos, S. Sharvia, D. Parker, T. Tohdo, An approach to optimization of fault tolerant architectures using HiP-HOPS, Softw. Pract. Exp. 41 (11) (2011) 1303–1327.
[29] M. Wallace, Modular architectural representation and analysis of fault propagation and transformation, Electron. Notes Theor. Comput. Sci. 141 (3) (2005) 53–71.
[30] L. Grunske, J. Han, A comparative study into architecture-based safety evaluation methodologies using AADL's error annex and failure propagation models, in: Proceedings 11th High Assurance Systems Engineering Symposium, IEEE Computer Society, 2007, pp. 283–290.
[31] L. Grunske, Early quality prediction of component-based systems – a generic framework, J. Syst. Softw. 80 (5) (2007) 678–686.
[32] G. Kainz, C. Buckl, A. Knoll, A generic approach simplifying model-to-model transformation chains, in: R. France, J. Kazmeier, R. Breu, C. Atkinson (Eds.), Proceedings Model Driven Engineering Languages and Systems (MODELS 2012), in: Lecture Notes in Computer Science, vol. 7590, Springer, Berlin/Heidelberg, 2012.