Pattern-Based Approach for Designing Fail-operational Safety-Critical Embedded Systems

Dulcineia Penha, Gereon Weiss, Alexander Stante
Fraunhofer ESK
Hansastr. 32, Munich, Germany

Abstract— **To deal with fail-operational (FO) requirements in today's safety-critical networked embedded systems (SCNES), engineers have to resort to concepts such as redundancy, monitoring, and special shutdown procedures. Hardware-based redundancy approaches are not applicable to many embedded systems domains (e.g., automotive systems), because of prohibitive costs. In this scenario, adaptability concepts can be used to fulfill these FO requirements while enabling optimized resource utilization. However, the applicability of such concepts highly depends on the support for the engineering during system development. We propose an approach to cope with the challenges of fail-operational behavior of SCNES in which engineers are supported by design concepts for realizing safety, reliability, and adaptability requirements through the use of architectural patterns. The approach allows expressing FO concepts at the software architecture level. This lowers the effort for developing SCNES by utilizing generic patterns for general and reoccurring mechanisms.**

Keywords—reliability; adaptability; software architecture; architectural patterns; safety; networked embedded systems

I. INTRODUCTION

The increasingly dense amount of software deployed in today's safety critical networked embedded systems (SCNES) has been steadily pushing the borders on safety and reliability of such systems. Since SCNES permeate modern life, failures affecting these systems can cause severe damage to human beings or to the environment. Consequently, the number of safety and reliability requirements which SCNES have to fulfill is becoming larger. Therefore, defining, allocating, tracing, and enforcing these requirements are increasingly complex tasks. For example, in the automotive industry, novel driver assistance systems as well as the upcoming self-driving cars usually require fail-operational (FO) behavior, i.e., the system must remain functional even after experiencing failures.

To deal with such FO requirements, engineers have to build on well-tried reliability concepts such as redundancy, monitoring, and special shutdown procedures. However, these techniques alone are not always suitable. For example, in the automotive domain, redundancy usually adds prohibiting manufacturing costs due to additional hardware parts and tight integration of hardware and software [1]. Moreover, it is hard for engineers to obtain a clear picture of all software functions which are simultaneously executed, in order to optimize use of resources while realizing safety and reliability requirements.

In order to optimize the usage of resources in a resource-scarce embedded system, one can take advantage of unused hardware resources. This can be achieved either statically, by analyzing the system, and defining scheduling and degradation strategies; or dynamically, by reconfiguring the activation or deployment of software systems at runtime. The applicability of such an approach highly depends on the support of the methodology to include adaptability. Current design approaches for SCNES do not support the engineer in such a definition of adaptation concepts. A common technique in such design approaches is the utilization of patterns, which represent reusable design or best practice solutions to reoccurring problems in software design. In this paper, we argue that an approach to realize patterns using models is more suitable to integrate FO and adaptability concerns into modern development approaches (e.g., model-based design - MDD), in contrast to the classic definition of patterns assuming textual form (e.g., patterns catalogue [2]). The presented approach takes advantage of patterns and integrates these into the development process of SCNES. Architectural patterns are thereby defined as meta-model extensions on the software architectural level, where the system-wide FO and adaptive requirements can be managed best by a designer. Thereby engineers are supported in assuring that safety, reliability and adaptability requirements are considered at the architecture level, which is the first contribution of this paper. Furthermore, as a second contribution, the approach provides a generator which supports the conversion of the above mentioned requirements into artifacts (i.e., modeling elements), which either enrich the architecture or implement the modelled pattern. This is highlighted by the application of our approach to an automotive software system.

The remainder of the paper is organized as follows: Section II introduces architectural patterns and Section III describes the proposed approach and supporting development methodology. The feasibility of the approach is evaluated with case-studies of automotive systems in Section IV. Finally, we present an overview of related work in Section V and the conclusions and future work in Section VI.

The research leading to these results has received funding from the European Commission within the Seventh Framework Programme ([FP7/2007-2013] [FP7/2007-2011]) as part of the SafeAdapt project under grant agreement number 608945.

II. Architectural Patterns

In the following we discuss groundwork of architectural patterns with respect to fail-operational requirements in the context of SCNES. Safety patterns or safety-critical patterns [2] consider concepts to increase safety. In the same way, adaptation-oriented patterns [3] consider adaptation concepts to ensure FO requirements. Two fault-tolerant patterns are used as application examples of our approach and are presented in Section III and IV. They capture concepts of redundancy and graceful degradation for SCNES. In the following, we discuss redundancy and graceful degradation patterns.

There are different fault-tolerant patterns which explicitly implement redundancy. They are mostly based on providing multiple similar or equal copies of a given system or of individual system elements, which run in parallel. The *N-Version Programming (NVP)* [2] [4] is conceived based on software diversity; i.e., N ($N \geq 2$) functionally equivalent versions of the same software are implemented by different teams based on the same specification. Although NVP improves reliability of software systems [5], in the context of SCNES, it cannot guarantee that different versions, implemented by independent teams, will not be susceptible to the same failures [6]. Similarly to NVP, both the *Homogeneous Redundancy Pattern* (HoRP) and *Heterogeneous Redundancy Pattern* (HeRP) [7] provide a system with the ability to continue in the presence of a failure by replicating elements of the system. The *Triple-Modular Redundancy Pattern* (TMR) provides likewise multiple ($N=3$) components replicas, which run in parallel to protect the system against single point-of-failures. It is a very common solution for achieving redundancy and, together with the HoRP, has the advantage of low-design but high-recurring cost. However, unlike the NVP, both the TMR and the HoRP provide good support for failures but not for errors [7]. The *Active-Active Redundancy Pattern* (AARP) and the *Active-Passive Redundancy Pattern* (APRP) allow the software versions to be designed as active or passive copies. In this way, the AARP includes both versions being active and the APRP has active and passive versions [8]. In SCNES, these can be interpreted as hot- and cold-standby designs, respectively. The chosen design depends on the system's safety and timing-requirements.

Concepts of graceful degradation can be captured in design patterns as described in [9], where the *State Decrement Pattern (SDP)* and the *Minimum Subtrahend Pattern (MSP)* are presented. The SDP describes different aspects of a system and how its states can be lowered in the occurrence of an error. It contains an assessor which is responsible for deciding the new error-free state to be loaded after a failure occurs. The MSP is complementary to the SDP, describing one possible design for its assessor entity. Both patterns do not consider FO requirements, which are crucial for SCNES.

III. Integration of Architectural Patterns in the System's Design

Architectural patterns are proven pertinent for enabling reusability and documentation of recurring solutions for design problems. However, the definition of patterns usually assumes a textual form and does not integrate well into modern development approaches. With textual descriptions, the information stored, for instance, in data-bases is not explicit and cannot be straightaway transferred to software architectures. To take advantage of patterns and integrate these into the context of SCNES software development, this work proposes the definition of patterns as meta-model extensions on the architectural level. With our approach, information concerning the recurring design as well as the involved FO requirements is stored in patterns, at the same abstraction level of the software architecture. It allows associating the fail-operational-related strategies directly to the correspondent software components in the architecture, facilitating and automating its application.

A. Meta Patterns for Fail-Operational SCNES

Our representation of FO patterns at the software architecture level enables the realization of these patterns in the context of SCNES, where several non-functional properties and requirements permeate the different abstraction levels of the design process. In the automotive domain, for instance, the system's model-based design includes stages such as feature modeling, functional architecture, software design and hardware architecture, implementation, code generation, and deployment. The knowledge transfer between these several design phases is a challenge, especially if the information is not properly stored. Pattern descriptions, templates and catalogues, describe only informally the information, which cannot be easily transferred to later phases of the development process. Furthermore, they do not assist the engineer in earlier and more abstract phases of the design process (e.g., when modeling the functional architecture).

In order to overcome this independent representation of architectural patterns, our approach proposes to represent patterns as meta-models extensions which can be instantiated at the software architecture level, as shown in Fig. 1. For realizing patterns as meta-models, we define modeling levels as presented in Fig. 2. MP2 (Meta Pattern Level 2) is the main meta-model, or more precisely, meta-meta model. The meta-meta elements defined in MP2 are used to specify patterns in the MP1. In this way, MP2 allows MP1 to specify meta-models for each specific pattern, e.g., a meta-model for a redundancy pattern. Finally, MP1 is instantiated to model a specific occurrence of the pattern for a given system. These instantiated models constitute MP0.

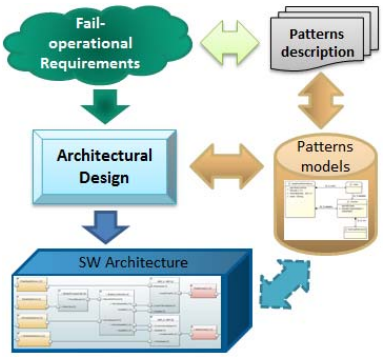

Fig. 1. Overview of architectural pattern definition

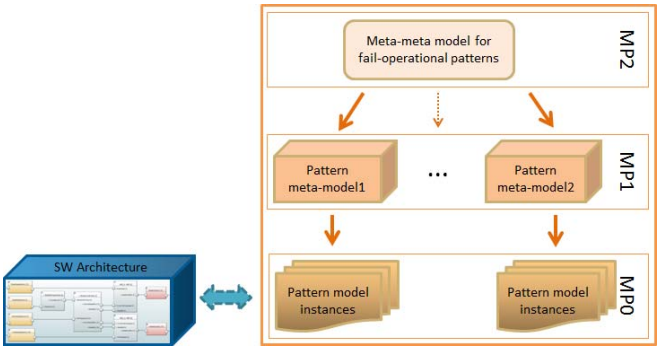

Fig. 2. Meta patterns levels

Pattern meta-models in MP1 are created based on MP2 and on existent patterns descriptions, with the necessary adjustments regarding FO requirements. In order to fulfill these requirements, the needed additional information is modeled through attributes in the pattern. As part of the pattern (meta-model) in MP1, this information becomes formalized and can be instantiated for specific problems/applications. Patterns are implemented in turn in MP0 as model instances of these meta-models to meet the requisites of specific requirements in the software architecture. The pattern elements in MP0 can directly reference software components in the architecture, enabling a tracing and coupling of the FO requirements with the architecture. Fig. 3 shows a small excerpt of our meta-meta model (MP2). A *fail-operational design* is composed by fail-operational *patterns*, which contain *pattern elements*. These pattern elements may contain, extend, or reference other pattern elements. Moreover, they reference respective external design artifacts in the software architecture. Patterns and pattern elements contain attributes (or constraints), which allow the pattern parameterization by defining its FO properties. The information represented in these levels enables the engineer to make use of arguments for safety, reliability and/or adaptability in the form of stored knowledge. Moreover, our approach enables tracing of information, requirements, and mechanisms, helping to avoid design failures. MP1 and MP0 are described in the following sections in more details.

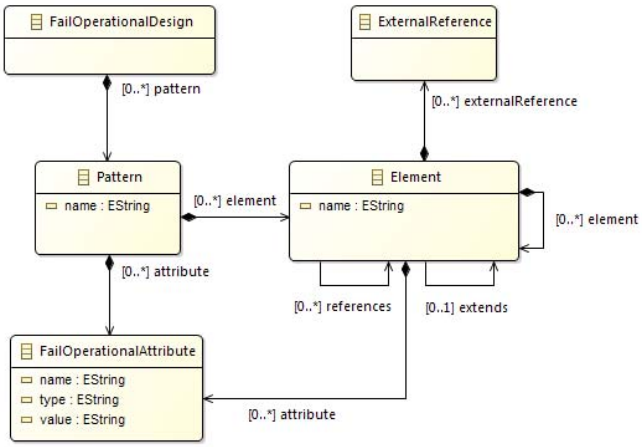

Fig. 3. Meta-meta-model for the realization fail-operational patterns (MP2)

B. *Patterns Parameterization*

As discussed previously, when realizing an architectural pattern several parameters are required in order to fulfil the pattern's purpose. This information is not necessarily listed in the pattern's description. In SCNES several safety-related and FO requirements must be parameterized [10]. For example, the maximum time interval in which a standby version has to take over carrying out the functionality in case of a failure (switchover time), the number of redundant versions, if they are hot-, warm- or cold-standby, which failures should trigger the degradation of a given system, together with several other properties regarding fail-operational requirements. In our approach, patterns are captured using the following structure (based on [11] and [2]): *Name, Purpose, Description – including FO requirements, Context, Problem, Preconditions, Solution, Structure, Implication, Implementation, Consequences, Constraints, Discussion, and Related Patterns*.

Two exemplary patterns (one with respect to redundancy and the other one regarding graceful degradation) and their fail-operational requirements have been investigated in order to evaluate the approach and are presented in the following sections. Safety, FO and adaptation related requirements are discussed in more detail for pattern-realization scenarios modeled at MP1 level. Necessary modifications were identified during the formalization process in order to express FO and adaptive aspects for designing SCNES and, therefore, adjustments have been introduced to the patterns.

C. *Redundancy*

Redundancy is a well-known concept for providing system availability. It allows backup elements to take control when main components malfunction. Furthermore, it is a commonly applied solution for FO scenarios like control applications of SCNES. For guaranteeing a reliable SCNES, besides software redundancy (with equal or different versions), the need of different redundant hardware resources and a counterbalanced optimization of resource usage must be taken into account.

1) Modeling Redundancy: As discussed in Section II, there are different patterns for enabling redundancy in the system. They contain either heterogeneous or homogeneous versions of the redundant function in order to tolerate common-cause and/or different failures. In current SCNES the redundant versions must not always be an independent functional equivalent software module implemented based on the same initial specification. For instance, in current driver assistance systems, where cars are not fully self-driving and the driver is still always available, an identical copy of the original software system is sufficient to implement standby techniques. Or even a simplified redundant version of the software function might be enough as proposed in [6]. In case a failure occurs, the vehicle could shut down the misfunctioning software and notify the driver. Moreover, the implementation and validation of SCNES according to safety standards such as the ISO 26262 [12] should prevent critical software bugs. In this scenario, not only heterogeneous redundancy (diverse implementation), but also homogeneous redundancy (equal implementation) patterns could be applied.

Furthermore, the number of versions or copies can vary according to the required safety and reliability levels, instead of being fixed to 3 copies (TMR). Additionally, the different versions can be active or passive as in the AARP and APRP, implementing hot-, warm- and cold-standby. This can be influenced, for instance, by the maximum allowed switchover time. In this way, the available individual redundancy patterns might not be the best choice for designing SCNES. Hence, when realizing the redundancy pattern according to our approach, the different configurations discussed above are considered, together with the necessary FO requirements. The *adaptive fail-operational redundancy pattern* (AFOR) supports both homogeneous and heterogeneous copies. Besides, the number of versions is flexible and each version can be either active or passive, according to the system's individual fail-operational requirements.

2) Realization of the Adaptive Fail-operational Redundancy pattern: As an example of the proposed approach, the AFOR pattern has been realized via meta-model extension. For this, a meta-model for the AFOR has been created in MP1, as shown in Fig. 4. Based on the aspects discussed above, AFOR is composed of *N* redundant versions and a voter. Versions denote the correspondent software component (SWC) in the software architecture via external references. Versions can be homogeneous or heterogeneous and are enhanced with a `StandbyClassification`, which can adopt the values hot-, warm- or cold-standby. For homogeneous redundancy, versions refer to the same SWC, while for heterogeneous redundancy, each version refers to its individual corresponding SWC. Regarding timing requirements with respect to failure recovery time, the AFOR pattern is enhanced with a `maxTakeOverTime`, meaning that the takeover process by a redundant version must happen within a given time interval. This value is obtained during safety analysis and can be validated using different timing analysis techniques, e.g. schedulability or worst-case-execution time analysis.

D. Graceful Degradation

In SCNES, many functions cannot simply be shut down in case a failure occurs. For these systems, fault-tolerant design is required. One possibility for achieving fault-tolerance is the use of graceful degradation strategies, which allows the system to smoothly change into a decremented state. As an example we take a car with two-driving modes: *Comfort* and *Sport*.

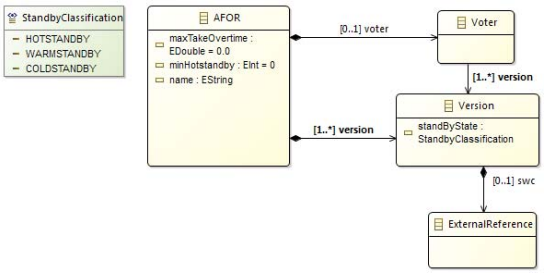

Fig. 4. Meta-model for the adaptive fail-operational redundancy (AFOR) pattern (MP1) [10]

These different modes affect, for instance, the throttle response. The same automobile is equipped with run-flat tires, which enable the vehicle to be driven even after a blowout, but with a limited speed (e.g., `80Km/h`). In the situation where the Sport Driving Mode is on and a blowout occurs while driving, the system's state must switch gracefully to the Comfort Driving Mode, notify the user, and block the Sport Driving Mode to be switched on again until the flat-tire is repaired.

1) Modeling Graceful Degradation: We assume that graceful degradation is represented as in [9], where a degradation step is a state decrement. As discussed in Section II, graceful degradation patterns describe the solution to situations where a given component of the system malfunctions and cannot be further executed/used within the system. Hence, they specify the different state decrements allowed within the system. They also describe which events or failures trigger the switch from current to the next state.

However, when applying a graceful degradation pattern to SCNES, additional properties must be considered in order to fulfill FO requirements. First, the *Fault Tolerant Time Interval* (*FTTI)* within which degradation has to occur must be specified. Moreover, it is necessary to determine which actions must be taken in order to transit from one state to another. In the automotive domain, for instance, the driver must be notified in case a given function has been deactivated. In the previous example, the *Sport Driving Mode* must be deactivated, the driver must be notified, and all functions related to the *Comfort (Driving) Mode* must be turned on, e.g., the *DSC* (*Dynamic Stability Control*), and so on. Based on our approach, when realizing a graceful degradation pattern for SCNES, the properties related to the degradation itself are modeled together with the necessary FO requirements. Therefore, the *fail-operational graceful degradation pattern* (FOGD) is inspired by the State Decrement Pattern with additional support for FO requirements of SCNES.

2) Realization of the Fail-operational Graceful Degradation pattern: The FOGD pattern is realized via meta-model extension in MP1 constituting a second example of our approach (Fig. 5). FOGD is composed by system state decrements and a handler. The handler is responsible for monitoring the system and transiting from the current state to another viable state due to occurred events.

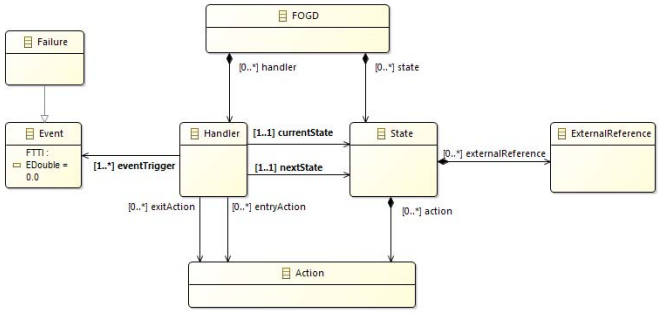

Fig. 5. Meta-model for the fail-operational graceful degradation (FOGD) pattern (MP1)

The FFTI within which the state decrement must occur is defined for each event. Moreover, a state requires actions in order to be put in operation. State entry and exit conditions are executed before transitions between states occur. Actions can be, for example, the re-wiring of input data, a notification in case of failure, or the shutdown of a given system. A state shall list the software components available and being executed at this state. The handler must specify handover mechanisms, which define how the actions are executed and which state is adopted next. The definition of state details and of the handler's behavior is beyond the scope of this paper.

IV. Case-Study Evaluation

In order to qualitatively evaluate our approach and show its applicability and advantages, the two patterns detailed in Section III are applied to representative case-studies from the automotive domain. To complete the evaluation, a prototype wizard for automotive systems has been implemented as Eclipse plugin. It supports the conversion of modeled information into artifacts which can either enrich existent software architectures or implement the modeled pattern. The generator creates software component instances based on a given pattern, according to the automotive standard AUTOSAR [13]. The transformation from the pattern model to the generator model supports interactions with the engineer in order to incorporate all information modeled in the pattern in a semi-automated way.

A. Redundancy for Brake-By-Wire

This section presents a simplified model of a real car's Brake-By-Wire system (BBW) [14], which controls the brakes by electronic means and is composed of *BrakeTorqueCalculator*, *BrakeController* and two *ABS (Anti-lock Brake-Systems)* components. Fig. 6 (left side) shows the BBW software architecture modeled in EAST-ADL *Functional Design Architecture* [15]. Only the most relevant components and ports are shown for readability reasons. Because of the different possible hazardous events involved with the BBW, its safety classification is the highest of automotive systems with ASIL D [12]. On account of the high risks imposed by a failure on the BBW, a redundancy strategy is necessary. In this case study we focus on the redundancy of the ABS by applying homogeneous redundancy. The system requires that malfunctions in the ABS are detected within 10 milliseconds, and a deactivation of the function follows within 20 milliseconds. The FTTI is 50 milliseconds. Because of this low-latency timing requirements a hot standby redundancy is necessary, since the time required for recovering from a failure in any other scenario would lead to higher delays which would violate this requirement.

Our approach is applied to fulfill a specific selection of the fault-tolerant and safety requirements of the system. Hence, the meta-model in MP1 (Fig. 4) is instantiated for each of the ABS components in MP0. The configuration of the resulting pattern-instance for the ABS1 component of the BBW architecture is depicted in the excerpt on the right side of Fig. 6. As shown in this excerpt, the AFOR instance contains two hot-standby versions of ABS1 and a voter, according to the previously specified requirements. Each version refers to the individual redundant component in the software architecture, as represented by the arrow from the External Reference of the Version HOTSTANDBY to the ABS_1 SWC in Fig. 6. Since homogeneous redundancy is applied, both hot-standby versions refer to the same component ABS1 in the software architecture.

The information stored in the AFOR pattern for the BBW is used to generate the software architecture assisted by the wizard. The generated AUTOSAR architecture is shown in Fig. 7 and contains the modeled versions, as well as the voter.

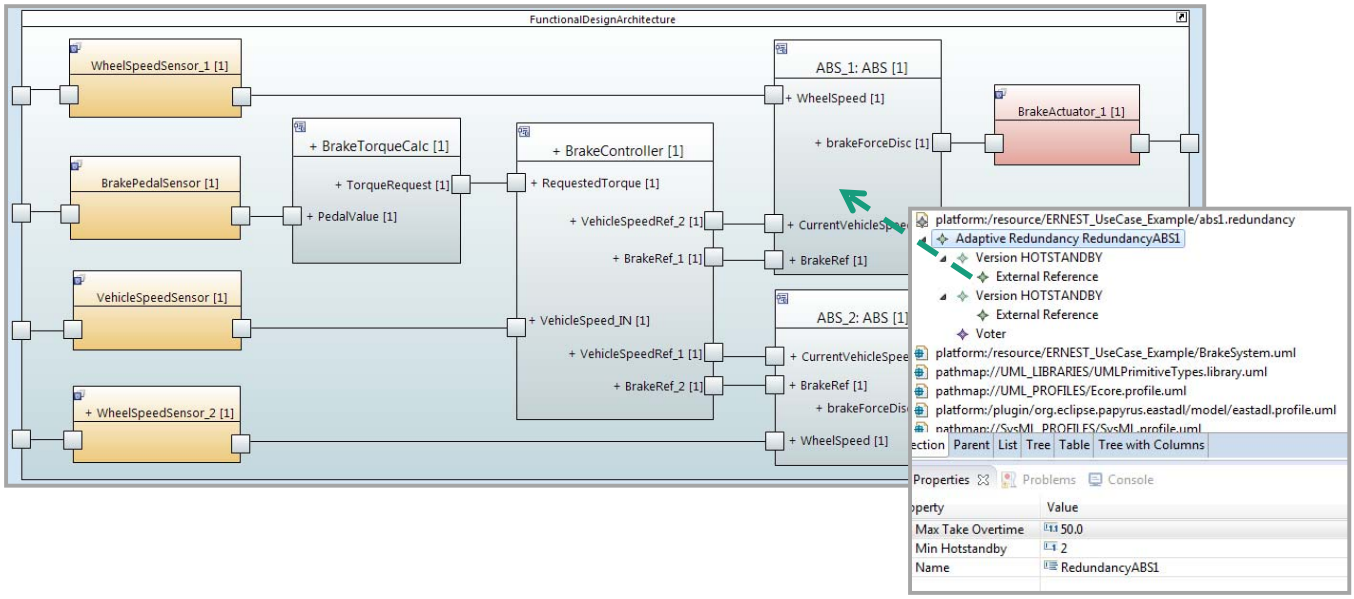

Fig. 6. High-level software architecture of the Brake-By-Wire System (left) and Realization of the AFOR pattern for ABS1 (MP0) (excerpt)

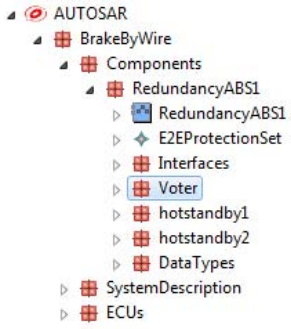

Fig. 7. Generated AUTOSAR Model for the redundancy of the ABS1

Since in this case-study both ABS1 and ABS2 are redundant, the same process has been applied to ABS2, with the pattern being instantiated and the versions referencing the ABS2 component in the software architecture. By applying our approach and realizing the AFOR pattern, the information regarding FO and adaptive aspects is clearly modeled allowing for traceability. Previously this was a manual step for engineers and therefore, it was, beneath the extra modeling effort, prone to human error.

B. *A Graceful Degradation Scenario for the Speed Control System*

This section introduces the application of the FOGD pattern to an automotive scenario with an automatic speed control system. The high-level architecture for this scenario, with the most relevant components and ports, is presented in Fig. 8. The speed control system contains the *Dynamic Cruise Control (DCC)* and the *Active Cruise Control (ACC)* [16]. The DCC automatically controls a vehicle's throttle and brakes in order to maintain a constant speed determined by the driver. The ACC, on the other hand, controls the throttle and the brakes in order to maintain a steady speed while keeping a safe distance from vehicles ahead. For this, the ACC depends on radar input.

The speed control system should degrade gracefully in the case of a failure – either of the radar system or of the ACC itself. Therefore, the full-featured state of this function includes a working radar and ACC system, which safely adjusts the vehicle's speed. In this scenario, if a failure of the radar system occurs, the degradation of the system proceeds as follows: the ACC is shut down, the driver is notified, and the system falls back into a simple DCC mode. The DCC function adopts a reset state and the driver has to confirm the reactivation of the DCC, while the set speed is already provided.

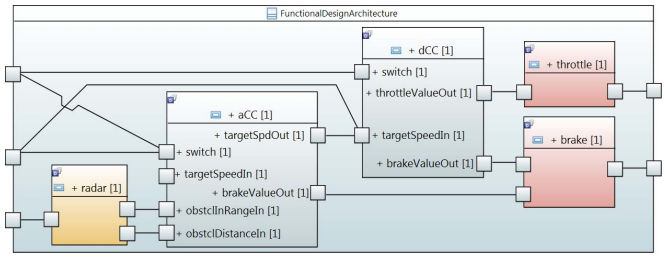

Fig. 8. High-level software architecture for a degradable speed control system

The FOGD pattern has been applied to design the graceful degradation scenario and fulfill its FO requirements. The procedures for applying FOGD to this scenario follows the same steps described above for AFOR and are omitted for the sake of brevity. Two states have been instantiated: the full-featured state provides all SWC and the degraded state includes only the DCC. The degraded state has the following entry actions: *Deactivate user switch of ACC; Activate user switch for DCC; Deactivate input targetSpeed of ACC; Replace input targetSpeed from DCC – input comes from user and not from ACC*. And the previous state has the following exit actions: *Remove ACC from schedule list; Error notifications*.

On the basis of this parameterized pattern, the wizard generates the correspondent AUTOSAR software architecture for each Degradation State with their corresponding SWCs (indicated as external references in the pattern model), as well as the events which trigger state changes. Further investigation is needed on how to generate automatically the transitions between states. One promising approach is to use the already existing mode-management of AUTOSAR. Generation of handlers and actions are being investigated to allow the integration at the OS and scheduler levels.

C. *Discussions*

Evaluation and analysis of classic architectural design can be performed based on metrics such as coupling and cohesion or by using, for instance, scenarios, simulation, mathematical modeling, experience-based reasoning, empiricism. [17] This is a hard aspect for software methods in general. However, the aggregation of safety, adaptive and FO requirements in architectural designs takes such metrics out of focus and demands a trade-off between metrics and requirements. Our approach proposes the enhancement of software architectures by modeling patterns at the architecture level, aiming at consolidating proven in using concepts (such as TMR and degradation) within patterns and supporting engineers in developing the systems. By modeling the FO behavior orthogonal to the architecture, its impact on these metrics should be minimal. Within the scope of this work, the generated fail-operational software architecture solely focuses on the design level. Unfortunately, an extensive industrial evaluation has been unfeasible due to costs and effort necessary to integrate a whole platform. Thus, an evaluation on development costs without a hardware architecture and operation system would have no value in this stage. Future work is considered in exploiting our approach with respect to cost optimization.

With present state-of-the-art, the modeling of FO and adaptive concerns in the software architecture is realized based on the engineer's individual experience and expert knowledge on how the system should and could be modeled. Most existent architectural patterns do not reflect the non-functional aspects of novel architectures, as required by SCNES, and contemplate mostly the structural modeling. Moreover, design patterns as textual descriptions and patterns catalogues do not assist the realization of these patterns during the design. The employment of parameterized patterns on meta-model level enables reuse and adoption for different applications.

Furthermore, the integration of FO and adaptive patterns to the modeling process improves the design by structurally guiding the engineer and by allowing the automatic generation of artifacts. The patterns can be used by engineers and tools to explicitly model aspects focusing on concerns related to adaptation and FO concepts. Moreover, the approach integrates well into modern model-driven development methods and allows machine readable specifications to be used for further analysis or engineering steps. Thereby, FO patterns as model-instances become valuable assets in the model-based development process of SCNES, since they enrich the architecture, are machine readable, enable the automatic generation of more detailed software architectures, as well as they allow traceability from requirements. Finally, by keeping non-functional concerns orthogonal to the software architecture our approach supports different levels of abstraction.

To conceive our approach, different architectural patterns have been investigated regarding safety, reliability and adaptability requirements of SCNES. While formalizing these patterns, adjustments are required in order to enhance the patterns to support the design of FO systems. AFOR and FOGD are exemplary patterns and a proof of concept for our approach, which also comprises other patterns modelled based on FO requirements. The discussion and realization of other patterns, e.g. Diversity Pattern, monitoring and detection patterns such as Watchdog and Heartbeat, were not included in this paper for the sake of brevity. How to capture FO parameters for patterns and model error recovery is currently under investigation, as are patterns reflecting hardware aspects of FO systems (e.g. Independence, Hardware-Diversity).

It is important to remark that our approach still needs the support of expert engineers for providing and analyzing the requirements and strategies modeled in the patterns realization. For instance, timing experts have to fix the timing parameters of the patterns. Moreover, the software architecture and the FO requirements modeled via our approach still need to be validated, just as with any other design approach. However, generating architectures with FO behavior should require less effort because of the automatic utilization of the FO and adaptive patterns and the therein captured expert knowledge. Finally, the adaptation behavior is currently not explicitly modeled in the patterns. Promising solutions are Activity Diagrams, State Machines, Sequence Diagrams, among others, which can be integrated in our approach to model the behavior of the architectural patterns.

V. Related-Work

In literature, several authors propose pattern catalogues [2] [3] [11] [18], which partially consider safety and adaptation-oriented concepts. Unlike these works, we do not attempt to provide another patterns catalogue; rather our approach takes advantage of patterns and integrates these into the development process of SCNES. Previous work presented new safety [2] and adaptation-related patterns [3] [9], as well as modifications of existent patterns considering safety [7] or adaptability aspects [19]. In [2], a catalogue with hardware and software safety-patterns is introduced. The work also supports the decision to select a suitable pattern to solve a particular problem. Other attempts model adaptability through design and architectural patterns: design patterns which capture adaptability concepts and facilitate the reuse of adaptation expertise [3]. The patterns presented in [3] are classified as monitoring, decision-making, and reconfiguration patterns. Monitoring patterns aim at observing system and environmental conditions to identify when reconfiguration should be applied. The decision-making patterns and the reconfiguration patterns attempt to support the design of the decision process and of the adaptive system's behavior, respectively. Unlike our approach, the patterns from [3] are not parameterized to meet FO requirements of SCNES.

An adaptive variant of the n-version pattern [4] is presented in [19], which propose the use of weight factors for versions. These allow modeling and managing different quality levels of versions, instead of considering identical version reliability. Software reconfiguration patterns to support dynamic software reconfiguration in product lines are presented in [18]. Although the patterns are proposed for self-adaptive component-based software architectures and support transitioning between system states, they only focus on software evolution, e.g. architectural updates, and are not suitable for FO systems. Moreover, the software reconfiguration patterns are behavior-oriented, while our approach focuses more on the structural design and on the fail-operational nature of SCNES. However, our approach could benefit from the concepts and patterns presented in [18] when the software architecture is enriched with the corresponding reconfiguration behavior. According to our approach, the patterns and catalogues discussed in the previous paragraphs can be parameterized and formalized in order to support the engineer during the design by allowing the explicit modeling of individual pattern concepts.

In [21], the authors propose an approach and framework, which supports the definition of patterns via UML Profiles. Their approach is equivalent to our MP2 level, which supports the definition of patterns. However, it would have to be extended in order to contemplate safety, fail-operational, and adaptive concerns, since it reflects only composition, compositional relations, and stereotype applications. Moreover, despite the fact that our approach considers fail-operational and adaptive concerns in the modeling, it also has the advantage of representing these concepts independently. The project KARYON [22] addresses the importance of defining abstract architectural patterns and of determining well-defined abstraction levels in the design and development processes. The authors propose an architectural solution which aims to support the hybrid nature of safety-related cooperative systems. Unlike our approach, the KARYON architectural solution focuses on the development of multiple levels of services, supporting the modeling of components with heterogeneous properties concerning reliability and timeliness. Their solution considers adaptation at the level of a service to meet safety requirements, defining a pattern where adaptation is managed by a safety-kernel. In this way, our approach for realizing architectural patterns which support the design of fail-operational and adaptive systems could be integrated with the KARYON architectural solution.

In the automotive domain, the standardization of practices through the ISO 26262 safety standard has being changing the way how safety-critical automotive systems are designed and

developed. The approach presented in [23] deals with the automatic realization of software safety requirements based on the ISO26262, allowing these requirements to be implemented automatically, mostly in the form of software safety mechanisms. However, it does not consider the system's software architecture. In this way, our approach could leverage the results presented by [23], in order to orchestrate the realization of requirements exploiting our approach for realization of architectural patterns.

VI. Conclusion

To incorporate redundancy or degradation in one function of a system might be a simple task. However, to deal with FO behavior, safety and reliability requirements, hardware cost optimization, etc. when designing and integrating several distributed functionalities within a system is a comprehensive task that requires safety expertise from engineers. Supporting the engineers eases the cumbersome and error-prone manual task of fulfilling these requirements. We have shown that FO and adaptive patterns employed as model-instances are valuable assets in the development process of SCNES. They enrich the architecture, are machine readable and allow automatic generation approaches. Our approach cannot only provide proven in use building blocks (through patterns), but also improves traceability towards requirements and reduces development efforts, e.g. while structuring the safety argumentation (ISO Safety Case). Finally, our approach can profit from other existent approaches and provides a vital element to systems engineering. It makes the connection from the higher level requirements and safety goals (ISO or IEC) to their lower level realization.

The selection of patterns presented in this paper can be extended to support different levels of abstraction and also take into account allocation, hardware resources, and dynamic aspects. In future, we want to further investigate the propagation of information stored in the patterns fully automatically to later stages of the development, e.g. at runtime. Moreover, the support for the adaptation behavior at runtime is being investigated.

Acknowledgment

A special thank you goes to those who contributed to this paper: Ludwig John for collaborating with the investigation of patterns and the implementation of the generator, during his internship, and Raphael Trindade for his valuables comments.

References

[1] H.Seebach, et al., "Designing self-healing in automotive systems," Proceedings of the 7th International Conference on Autonomic and Trusted Computing (ATC'2010), Xi'an, China, LNCS, Springer Berlin Heidelberg, vol. 6407, pp.47-61, October 2010.

[2] A. Armoush, "Design patterns for safety-critical embedded systems," Ph.D. Thesis, Faculty of Mathematics, Computer Science and Natural Sciences, RWTH Aachen University, Aachen, Germany, June 2010.

[3] A. J. Ramirez, B. H. C. Cheng, "Design patterns for developing dynamically adaptive systems," Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'10), Cape Town, South Africa, ACM New York, USA, pp. 49-58, May 2010.

[4] A. Avizienis, "The N-version approach to fault-tolerant software," IEEE Transactions on Software Engineering, vol. 11, No. 12, IEEE Press Piscataway, NJ, USA, pp. 1491–1501, December 1985.

[5] X. Cai, M. R. Lyu, M. A. Vouk, "An experimental evaluation on reliability features of n-version programming," Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering , Chicago, Illinois, IEEE, pp. 161-170, November 2005.

[6] L. Sha, "Using simplicity to control complexity," IEEE Software, vol. 18, Issue 4, IEEE Computer Society Press Los Alamitos, CA, USA, pp. 20-28, July 2001.

[7] B. P. Douglass, "Real-time design patterns: robust scalable architecture for real-time systems," ISBN: 0201699567, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[8] K. S. Ahluwalia, A. Jain, "High availability design patterns," Proceedings of the 2006 Conference on Pattern Languages of Programs (PLoP'06), Portland Oregon, USA, ACM New York, USA, pp. 24:1--24:11, October 2006.

[9] T. Saridakis, "Design patterns for graceful degradation," Transactions on Pattern Languages of Programming I, Lecture Notes in Computer Science, vol. 5770, Springer Berlin Heidelberg, pp. 67-93, 2009.

[10] D. Penha, G. Weiss, "Parameterization of fail-operational architectural patterns", Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC'15), Salamanca, Spain, ACM New York, USA, vol. 1, pp. 471-473, 2015.

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design patterns: element of reusable object-oriented software," ISBN 0-201-63361-2, Addison-Wesley Professional, USA, November 1994.

[12] ISO 26262: Road vehicles – Functional safety: https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-1:v1:en – last access June 2015.

[13] AUTOSAR - AUTomotive Open System ARchitecture: http://www.autosar.org/ – last access June 2015.

[14] TIMMO2USE - Timing Model – Tools, algorithms, languages, methodology, USE cases: https://itea3.org/project/timmo-2-use.html – last access June 2015.

[15] EAST-ADL - An Architecture Description Language for Automotive Software-Intensive Systems: http://www.east-adl.info/ - last access June 2015.

[16] BMW Technology Guide - Cruise Control: Dynamic Cruise Control and Active Cruise Control: http://www.bmw.com/com/en/insights/technology/technology_guide/articles/cruise_control.html – last access June 2015.

[17] J. Bosch, P. Molin, "Software Architecture Design: Evaluation and Transformation", Proceeding of the IEEE Conference of Engineering of Computer-Based Systems. ECBS, Nashville, pp. 4-10, March 1999.

[18] H. Gomma, M. M. Hussein, "Software reconfiguration patterns for dynamic evolution of software architectures," Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004), Oslo, Norway, IEEE Computer Society Washington, DC, USA, pp. 79-88, June 2004.

[19] K. E. Grosspietsch, "An adaptive approach for n-version systems," Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'2003), Nice, France, IEEE Computer Society, pp. 215.1, April 2003.

[20] S. Sialhir, "Guide to applying the UML," ISBN-10 1468492934, 1st. ed., Springer, October 2002.

[21] N.C. Debnath, A. Garis, D. Riesco, G. Montejano, "Defining patterns using UML profiles," Proceedings of IEEE Internatinal Conference on Computer Systems and Applications, IEEE, pp.1147-1150, March 2006.

[22] A. Casimiro, et. al., "A kernel-based architecture for safe cooperative vehicular functions," Proceedings of the 2014 Symposium on Industrial Embedded Systems, Pisa, Italy, IEEE Industrial Electronics Society, pp. 228-237, June 2014.

[23] R. F. Trindade, L. Bulwahn, C. Ainhauser, „Automatically generated safety mechanisms from semi-formal software safety requirements", Proceedings of the 33rd International Conference SAFECOMP 2014, Computer Safety, Reliability, and Security LNCS, vol. 8666, Springer International Publishing, Switzerland, pp. 278-293, September 2014.