Component-Wise Software Certification for Safety-Critical Embedded Devices

Detlef Streitferdt and Armin Zimmermann
Department of Computer Science and Automation
Technische Universität Ilmenau
98693 Ilmenau, Germany
{detlef.streitferdt|armin.zimmermann}@tu-ilmenau.de

Jörg Schaffner and Michael Kallenbach
Funkwerk Systems GmbH
Im Funkwerk 5
99625 Kölleda, Germany
{joerg.schaffner | michael.kallenbach}@funkwerk.com

***Abstract*—The development of industrial software systems increasingly requires safety certification. The certification process causes huge efforts and has to be repeatedly executed for changes of the systems. At the same time, many of the software architectures are developed in a component-based style. This paper presents the results of an industrial software development and certification project in the railway domain. It is a component-based development project that was designed to be certifiable in two steps with this new approach. First, the core of the system will be developed and certified. Secondly, new or adapted plug-in components will be certified individually. Thus, the re-certification effort is reduced to the certification effort of a single component.**

Keywords—Safety Certification; Component-based; Embedded Software

I. INTRODUCTION

Safety is an important requirement in many industrial domains. While "safety cases" gained popularity through the media, safety awareness of customers increased. As a result about one percent of the industrial market is safety related with a growth rate of 11% (Compound Annual Growth Rate) up until 2020[1]. Thus, safety is a selling point as well. Industrial products as well as manufacturing tools for their development address safety by using domain specific standards (e. g., ISO 26262 [1]for automotive or EN 50128 / 50129 [2][3] for the railway domain) which are originally derived from IEC 61508[4]. Compliance to a safety standard requires a certification process with additional efforts for the development and final certification of the product. Changes of certified products require a re-certification with efforts and costs for the overall project budget.This paper discusses the results of a successful certification along an industrial project of a software system with a component-based certification. Thus, components can be added to the final product without the complete re-certification.First, general requirements for a component-based as well as safety-enabled software architecture supporting the initial certification of the system will be presented. The important software architecture as enabler for the later certification of new or changed components without a re-certification of the whole system is discussed in the remainder of the section.

The next section gives a brief overview on the state-of-the-art in different domains with the focus on and a definition of *freedom of interference*. The third section elaborates on the resulting, general requirements of *freedom of interference*. In the fourth section, a new certification concept is presented, based on a software architecture and references to the parts of the EN 50128 / 50129 standard emphasized for a component-wise certification. Finally, the paper concludes with a short summary and outlook.

II. STATE OF THE ART

The component-based certification is currently a challenge in different domains such as automotive, avionic, railway, or automation [5]. Due to the continuously changing nature of software and its underlying execution environments certification becomes an expensive endeavor. This paper focuses on the software requirements in the EN 50128 [2] standard for systems in the railway domain. The corresponding hardware requirements are described in the EN 50129 [3] standard.

A good fraction of EN 50128 puts requirements on the development process and its methods to develop software safely. For a successful certification the compliance to these requirements has to be shown.On the one hand, the complexity of current software products and side effects that might occur due to software changes are the basic idea behind the requirement of a new certification of systems exposed to changes in the software. On the other hand, methods and tools proposed in EN 50128 are addressing the challenge by re-executing the tests and checks established for a given software project.The separation of safety and non-safety parts of systems is commonly used. Here, meta-models like OPENCOSS [5] help in specifying the safety compliance needs and also help in reducing the certification efforts. But still the re-certification effort has to be spent for any architectural changes.

EN 50128 proposes safety requirements for the system architecture without restrictions on the concrete structure, e.g., the architectural style. Thus, the software architecture is always part of the certification process. What if a component-based

[1] de.statista.com and www.marketsandmarkets.com

978-1-5386-2215-5/17/$31.00 ©2017 IEEE

software architecture would have well defined interfaces and an overall structure to provably support exchanging components without side effects? This idea is described as *freedom of interference*. The automotive safety standard ISO 26262 [1] in its annex D explicitly states requirements in three main areas, timing, memory and information exchange to address *freedom of interference*. Additionally, ISO 26262 requires the absence of the unwanted error propagation among components. Thus, for any component-based system the concept of *freedom of interference* specifically between the components and the core framework has to be stressed to certify systems in the safety domain.

For EN 50128 *freedom of interference* has not been fully integrated yet. As part of the project work we defined the term for this domain.The central high level requirements for *freedom of interference*of safety-critical software deal with the component-based core architecture and the connecting interfaces to extension components. Errors in any of the components should be identifiable at runtime and they must not cause an unwanted error in another component. This refers to any effects of external or system-internal events onto the components of a system. The system and its components should be exposed only to desired effects, without unwanted or unforeseen interferences, which appear when executing any of the components. In safety-oriented software systems, the interaction between components should also and exclusively lead to desired behavior and influences. Finally, the execution of software components should result in repeatable results without exceeding the resource limits managed by an operating system. It also should not affect other components in their function. As a direct inherent requirement of a component-based system the components have to be interchangeable and pluggable to the underlying framework while ensuring the*freedom of interference*.

With this analysis and in cooperation with the Funkwerk GmbH our following definition of *freedom of interference* was developed for our project in the railway domain.

Definition 1: Freedom of interference for (safety-critical) software systems is the possibility of modifying a software system in a predefined way without breaking the previously defined features/requirements of the system. With regard to software systems a modification (e. g., extension, patch, etc.) that would change a software package “A” into “B” is considered to be *free of interference* with respect to a given property if the validity of the property for “A” also implies the validity of the same property for “B”.

This definition is needed in the remainder of this paper to specifically support the component-based certification and argue that even with a reduced certification effort the safety of the system is still in accordance to the safety standard.

III. Freedom of Interference

The extensibility of a system free of interferences requires that all the initial safety requirements must be valid for the extended system as well. Such requirements typically include the complete and correct functioning of the software, absence of faulty states or failures, and timely response to events. A set of general requirements reflecting the *freedom of interference* concept will be presented in this section.

The features (requirements) of the basic software framework (the core architecture) should not be jeopardized by any changed or new plug-in component. With respect to a particular property a changed or new plug-in component can be considered free of interferences if the fulfillment of this property by the basic framework always allows to conclude that this property is still valid in the presence of a changed or new plug-in component.

Thus, a possible approach is to provide such evidence/proof for each of the properties required in a specification. However, this is not feasible for the large scope of an industrial specification. The following sections analyze which conditions must be fulfilled in the light of changes for a software system with a desired and correct functionality.

A. Abstract Computational Model

The computer systems we address have a deterministic behavior if the underlying hardware is free of errors. This is also valid for smaller units such as processes as part of an operating system. Here, the physical execution unit is only partly available according to the time-sliced or parallel execution concept. All the other resources will have to be shared among the processes while using a safe and controlled access to these resources. With a set of given requirements and as long as enough resources are available on time such a virtual computer system would also operate in a deterministic manner.

Interferences are provably impossible, when changing a unit of execution (here: an operating system process) would not interfere with other units. Thus, this unit of execution can be considered free of interferences. The possibility to partition a system into such independent execution units is an important precondition to certify the feature *freedom of interference* for a given system.

B. Integrity

If two given systems are equipped with the same hardware and a defined starting state of the code/data in memory, then the proven and tested behavior of an execution unit would not differ from its behavior in the final production system. The behavior of such a unit is defined by the changes from one state to another, which happens deterministically. Thus, the same input data leads to the same output data. Proving hardware integrity is rather easy and related to the technical and manufacturing domain. This type of integrity cannot be used for software. Testing of devices, even of identical devices within an assembly line, had and has to be done to meet the safety requirements, despite the *freedom of interference* feature.

Software integrity of software systems running on devices or software units is technically achieved by correctly copying the software. Thus, the correct operation of e. g., a bootloader must be proven. For another level of safety checksums can be used to identify accidental changes or acts of sabotage by comparing the firmware against a checksum prior to starting the system. Overlapping data and program code has to be limited to hinder self-modifying code or unauthorized changes

of the program code. In addition, continuous memory monitoring techniques [6] should be used.

Data integrity is harder to prove. For such a proof the execution of code, operating on a given set of data entities (the internal state of a unit) with a given set of inputs, should always lead to a new and correct state. Accessing internal data entities is only allowed via well-defined interfaces and inside the allocated (by the operating system) memory area of the unit.

The separation of different units is typically realized at the operating system level. Given this, data faults or inconsistencies are only possible by incorrect input values, what is discussed in the next section.

C. Interaction and Cooperation

Partitioning larger software systems into units certainly requires the cooperation or interaction of these units. Such interactions can be messages or function calls and they are critical for our *freedom of interference* property for two reasons. First, the data integrity of a unit could be corrupted when receiving false data. Secondly, delays or blocking can occur when waiting for required data.

The proof for *freedom of interference* with extensibility is difficult, since it is not sufficient to test static properties such as the integrity of program code. Instead, every possible behavior and all sets of input data would have to be tested, which is clearly impossible for complex, modern software systems. To keep the effort for the detection and implementation as low as possible, a software architecture supporting the feature *freedom of interference* should allow only a limited set of necessary interaction methods, which should enable testing the properties as shown below.

Testing the core software as complete unit with standard test cases is not sufficient without its variable components. Here, testing and validation methods have to be applied to show that the functionality and (required) properties of the core system remain unchanged for any set of extensions. Due to the huge number of test case permutations this can hardly be done. Thus, we propose a very detailed and tested specification of the desired and forbidden behavior. Testing is possible with a model-based statistical approach [8] and with automated test cases, e. g. generated by JUMBL[2].

Besides the definition of types or ranges of the data exchanged, the method *design by contract*[9] with its models should be used, e.g., assertions for pre- and postconditions which should be transformed into runtime tests. Behavior shall be modeled with UML interaction or sequence diagrams, which are also usable as test specification. Such models can be used to prove the correctness of future extensions (implemented as components).

Some components might be known or unavailable at the beginning of the development and the certification phase. Here, dummy components or test stubs for testing purposes need to be developed according to the protocol specifications of the interfaces. It has to be shown that any extension in correspondence with the specification will result in a correctly working basic system. Illegal extensions should be identified dynamically at runtime by the core system. This system should continue to work properly (e.g., default behavior as fallback). It is recommended to limit possible extensions and their possible behavior to an absolutely necessary minimum.

Errors should be caught and lead to messages, warnings, or a reset of an extension. Incorrect data can be detected partly via assertions and exceptions or software redundancy. Synchronous, asynchronous interactions, and waiting times for data should be enhanced with monitoring of the execution time and appropriate error handling. The above cases can be tested with test stubs using the boundary approach [10].

D. Resources

Resources are a bottleneck without parallel hardware. An execution unit requesting a resource could be delayed or, in the worst case, permanently blocked. To prove the correct integration of a new component into a running and tested system (core plus components), all required resources need to be analyzed regarding their timed availability as stated in the requirements. A detailed documentation of the required resources has to be part of the specification. The worst case scenarios for each component have to be positively tested with the available resources.

IV. Enhanced Certification Concept

The concept for a component-based certifiable system is twofold. First, we propose an architecture with a corresponding environment. This is the foundation for the second part, the adaptation of safety requirements.

A. Software Architecture

The final application is built on top of the operating system QNX Neutrino [11]. It manages the execution units (the tasks) which are using the resources such as CPU time or memory while competing for these resources. The first versions of the QNX operating system was available in the 80s, QNX Neutrino was built in 1995. For this version the concept of independent and controlled operating system partitions lead to many patents, the latest in 2016 [12] QNX partitions are an important concept and requirement for our *freedom of interference* feature.

The software architecture inFig. 1is developed using individual processes, which in turn form partitions. For these partitions, QNX Neutrino allows setting CPU usage resource limit. Each partition gets a guaranteed fraction of the CPU time. A maximum of eight partitions can be created. For checking and monitoring system resources within partitions a (1) memory monitoring via the Memory Management Unit (MMU) and (2) the load control implemented in the QNX scheduler are available.

[2] jumbl.sourceforge.net

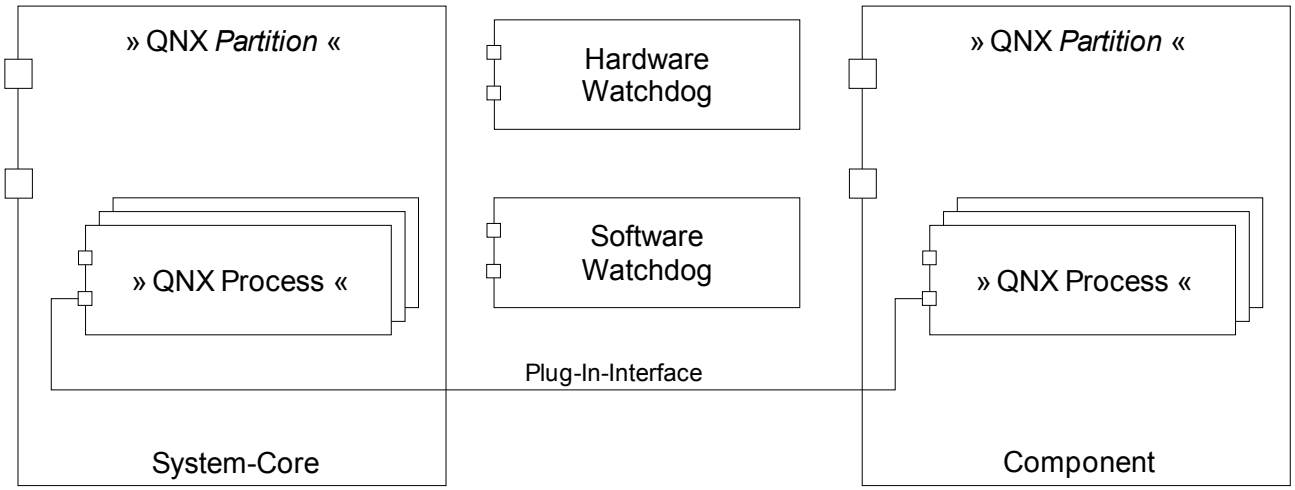

Fig. 1. Software Architecture Supporting Freedom of Interference

In Fig. 1the software architecture includes the processes for the system core, see the left side of the Fig. 2. These processes are part of a QNX partitions and can be limited in their CPU and memory usage. In the right part of Fig. 1an example of one component with its processes, again inside a QNX-partitions is shown. The one and only allowed interaction between the core and its components is the *Plug-In-Interface*. As discussed, all the requirements activities, the design activities and the implementation activities are serving this architecture. As long as the resource limits are known, tested and enforced in the running system, a new component or the changes for a component can be tested in advance and the certification of just this component is enough to reach a safe (according to EN 50128 / 50129) system. The middle part in Fig. 1holds the important watchdog functionality to continuously keep track of global constraints of the system in addition to the capabilities of QNX. With the hardware watchdog together with the software watchdog adding an improved structure, the independent runtime check of *freedom of interference* constraints is realized.

B. Safety Standard

EN 50128 / 50129 has to be addressed in total to accomplish a successful certification. With the *freedom of interference* requirements of section III and the software architecture of section IV-A the safety standard EN 50128 / 50129 is revisited in this section to reach the goal of a component-wise certification of the core system with additional components. The Plug-In-Interface of the system has to be very specific about timing, constraints and resource consumption. It is necessary to monitor and test the core as well as the components continuously with testing functions built into the system (see section 7.2.4.11 in [2]). As shown in section IV-A this can be accomplished with the presented software architecture.

In addition, the following parts of EN 50128 / 50129 in TABLE I. require special attention and emphasis for the *freedom of interference* requirement.

The emphasized points in TABLE I. have a strong correspondence to the development process used for a product with *freedom of interference*. As suggested in EN 50128 an adapted V-Model is shown in Fig. 2.

We propose an adaptation of the standard V-Model for the *freedom of interference* feature referenced by the five numbers in Fig. 2.

TABLE I. SAFETY CHECKS

50128 Reference [2]	**Explanation**
Table A.3, 3/4	Error detection for all interface usages while the system is running.
Table A.3, 5	Integration of pre- and post-conditions for the interfaces, especially for the interfaces between the components and the core of the system.
Table A.3, 6	A watchdog (at least hardware) is needed to continuously and reliably monitor constraints of the interfaces between the core and the components.
Table A.3, 12	An FMEA especially for the interfaces between the components and the core is required.
Table A.5, 9; Table A.6, 2; Table A.7, 1	Performance testing with the focus on constraints of the interfaces between core and the the components is needed.
Table A.8, 2	The dynamic analysis of the software is required especially to support the proof of the freedom of interference.
Table A.13, 1	Testing with the boundary analysis method should create and use a test component with the capability to create resource requests above and below any required limits.

Table A.13, 2	In addition to the above mentioned boundary analysis, the error guessing approach should be used to develop different error scenarios and also test them with the test component.
Table A.13, 3	Finally, equivalence classes should be developed for the complete interfaces between the core and the components.
Table A.18, 1	Avalanche tests should be integrated in the test components.
Table A.18, 2	Test cases for the correct timing of components and interfaces are needed. Response- and runtime-tests for the interaction between components and the core are needed.
Table A.18, 3	Performance tests will have to be integrated into continuous tests in the software. The system should be aware of, and report any performance issues at runtime.
Table A.21, 3	Test cases for the branch coverage for at least all the functionality of the interfaces between the components and the core are needed.
Table A.21, 5	Test cases for the path coverage for at least all the functionality of the interfaces between the components and the core are needed.

Extension 1 in Fig. 2. The following requirements and add-ons need to be considered in the*requirements phase*.

Req-00 – For all the core and extension components requirements within the system requirement phase the complete resource limits (CPU time, memory consumption, bandwidth, latencies/deadlines) shall be specified.

Additional Review: Check the completeness of the resource limits for all specified requirements.

Req-01 – For the scenarios in Req-00 test cases for the resource limits shall be developed. To test for *freedom of interference* all test cases shall be executed with a pass. The resource limits shall be tested at development time as well as continuously at execution time (see Req-03 in the next section).

Additional Test Coverage: The test cases in the validation phase (at development time) referring to the resource limits regarding the Plug-In-Interface shall include path- and condition coverage of 100%.

Req-02 – All changes shall have an impact analysis to identify the affected development artifacts. Changes limited to an extension component can be included by checking the affected extension component only.

Extension 2 in Fig. 2. The following requirements and add-ons need to be considered in the *design phase*.

Req-03 – A monitoring component should test the known resource limits continuously. This monitoring component should be executed in a high priority and isolated environment, to ensure it is executing properly. The end of the execution of the monitoring component should be analyzed and managed (e. g., by a hardware-watchdog). The time to notify the user an erroneous monitoring component should be specified as requirement.

Additional Logging: Exceeded resource limits should be logged. The required history depth should be included in the requirements.

Additional Test Result: For individually exceeded resource limits as well as for chains (time-dependent events) of exceeded resource limits the counter measures should be documented and implemented.

Req-04 – The Plug-In-Interface shall be monitored to detect critical states and react upon it. This interface shall be controlled by means of stress tests up to beyond the limits. In any case, the system shall have a predictable behavior.

Additional Two-Sided Test Cases: Monitoring of resource limits shall be present in all components.

Extension 3 in Fig. 2. The following requirements and add-ons need to be considered in the *component design- and test phase*.

Req-05 – The behavior of all components shall be documented completely with regards to the expected and required resources. Within the requirements phase these limits shall be reviewed. Within the modeling phase these limits shall be tested automatically, tool supported.

Additional Component Tests: The test cases of the system requirements phase shall be refined to be able to trace each violated resource limit down to the component level (e. g., using logging, see Req-03).

Req-06 – Testing and reviewing extension components shall be done independently (see 7.7.4.3 in [2]). Extension components shall be tested only together with the basic system (see 7.3.4.7 b [2]).

Extension 4 in Fig. 2. The following requirements and add-ons need to be considered in the *implementation phase*.

Req-07 – A test environment shall be available to be able to test independently.

Req-08 – The software architecture as shown in Fig. 1is the foundation for the *freedom of interference*. Only valid source code shall be included into extension components.

Additional Code-Review: Resources can only be used via operating system functions or via the defined Plug-In-Interface. Via a code-review such issues including the coding style shall be checked.

Req-09 – Constraints (e. g., one component cannot be used in the presence of another component) between extension components shall be documented and tested. All hardware requirements shall be documented for each component. All the components shall test these constraints and requirements at runtime.

Extension 5 in Fig. 2. The following requirements and add-ons need to be considered in the *certification phase*.

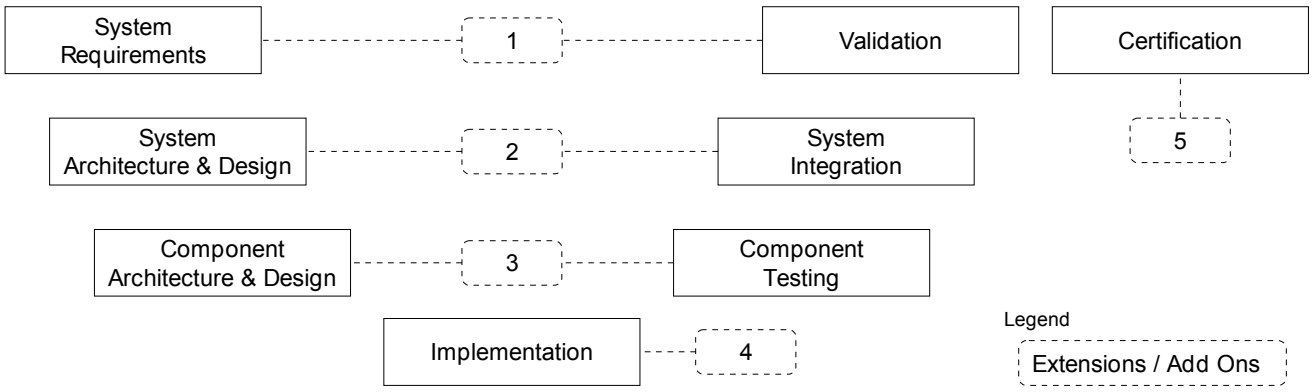

Fig. 2. Development Process Extension

Req-10 – The integration test of new or adapted extension components shall be done in addition to the single test. It shall be independent (see 7.7.4.3 in [2]).

The development of a component-based software system with the capability to be certified in two steps, starts with the certification of the core. The reduced effort for the certification of each extension component is included in Req-10. Here, the complete test of the core system and the new or adapted component will be available to a reviewer. This review together with a thorough review of the requirements listed here in this paper in the light of the new or adapted component is the input for the reduced re-certification of a system by certifying only an extension component.

V. Conclusion

This paper presented the safety certification of component-based software systems in the railway domain. Requirements for such safety relevant systems have been discussed in section III. The new certification concept is based on a specialized software architecture presented in IV-A and the QNX operating system. The stringent use of the methods and tools defined in the safety standard EN 50128 together with additional test cases in the design phase as well as built-in tests in the specialized software architecture are included in a new development process supporting the component-based certification of SIL0 applications.

The three elements of the approach described in this paper, the software architecture, the operating system, and the enhanced development process lead to a successful certification of the core system of an industrial project in the railway domain. With this system we are able to host several, yet unknown plug-ins.

The results are very promising for future efforts to certify systems with the built-in possibility to add components with the re-certification of only these components. An on-going and future task is the analysis of other domains towards their capabilities for a component-based certification. The similarities of the software design phases for different domains analyzed in [13] will be used to elaborate on this approach in other domains as well.

References

[1] ISO, 26262: Road vehicles-Functional safety, International Standard ISO/FDIS Std., 2011.

[2] DIN, DIN EN 50128:2012-03; Railway applications - Communication,signalling and processing systems - Software for railway control and protection systems, Beuth Verlag Std., 2012.

[3] BS EN 50129:2003-05-07; Railway applications. Communication,signalling and processing systems. Safety related electronic systems forsignalling, Beuth Verlag Std., 2003.

[4] ISO 61508-4; Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbar elektronischer Systeme Teil 4: Begriffe und Abkürzungen, 2002.

[5] D. Cofer, J. Hatcliff, M. Huhn, and M. Lawford, "Software certification:Methods and Tools (Dagstuhl seminar 13051)," in Dagstuhl Reports, vol. 3. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013,Conference Proceedings.

[6] J. L. de la Vara, A. Ruiz, K. Attwood, H. Espinoza, R. K.Panesar Walawege, A. Lopez, I. del Rio, and T. Kelly, "Model-Based Specification of Safety Compliance Needs for CriticalSystems: A Holistic Generic Metamodel," Information and SoftwareTechnology, vol. 72, pp. 16–30, 2016.

[7] J. L. Hennessy and D. A. Patterson, Computer Architecture: A QuantitativeApproach. Morgan Kaufmann, 2011.

[8] G. H. Walton, J. H. Poore, and C. J. Trammell, "Statistical testing ofsoftware based on a usage model", Software: Practice and Experience, vol. 25, no. 1, pp. 97–108, 1995.

[9] B. Meyer, Object-oriented software construction. Prentice hall NewYork, 1988, vol. 2.

[10] L. Copeland, A practitioner's guide to software test design. ArtechHouse, 2004.

[11] R. Krten, Getting started with QNX Neutrino 2: a guide for realtimeprogrammers. PARSE Software Devices, 1999.

[12] D. Dodge, A. Danko, S. Marineau-Mes, P. Van Der Veen, C. Burgess,T. Fletcher, and B. Stecher, "Adaptive partitioning for operatingsystem," Jun. 7 2016, uS Patent 9,361,156.

[13] F. Drogoul, S. Kinnersly, A. Roelen, and B. Kirwan,"Safety in Design - Can One Industry Learn from Another?" Safety Science, vol. 45, no. 1-2, pp. 129–153, 2007.