RWTH Aachen

Department of Computer Science
Technical Report

Empirical Evaluations of Safety-Critical Embedded Systems

Falk Salewski

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2008-18

RWTH Aachen · Department of Computer Science · Nov. 2008

Empirical Evaluations of Safety-Critical Embedded Systems

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Ingenieur
Dirk-Falk Salewski
aus
Siegen

Berichter: Professor Dr.-Ing. Stefan Kowalewski
Professor Dr.-Ing. Klaus Müller-Glaser

Tag der mündlichen Prüfung: 08.10.2008

Dirk-Falk Salewski
Lehrstuhl Informatik 11
salewski@embedded.rwth-aachen.de

Aachener Informatik Bericht AIB-2008-18

Herausgeber:	Fachgruppe Informatik RWTH Aachen University Ahornstr. 55 52074 Aachen GERMANY

ISSN 0935-3232

Abstract

Embedded systems based on different types of hardware platforms are nowadays increasingly used in safety-critical applications. These different hardware platforms lead to fundamental differences in design, particularly regarding the corresponding software. In this work, potential influences of hardware platforms on safety properties were gathered and open issues were identified. The most relevant of these open issues were evaluated for popular embedded hardware platforms (microcontroller, CPLD/FPGA). In detail, the impacts of hardware platform selection on *software diversity, encapsulation, reviewability, reusability* and the *development according to ISO26262* were chosen for investigation. Furthermore, the approach of *software diversity* was compared with a *fault removal approach.* The evaluation was realized in form of six experiments conducted for this work. During these evaluations, the following similarities and differences were observed for the considered hardware platforms. Despite the diversity between the hardware platforms, failures observed in the software versions, which were developed for these different platforms, contained high numbers of dependent (coincident) failures. Although failure dependency between two versions was reduced by the use of diverse hardware platforms, this effect was low. Most dependent failures were identified as implementation independent so that improvements of the software diversity by hardware diversity were limited. Thus, a comparison of software fault tolerance with a fault removal approach based on tests and reviews was conducted. As a result, different types of failures were mitigated by these alternative approaches. On the other hand, differences between microcontrollers and FPGAs were observed. First, certain advantages of FPGAs with respect to encapsulation and reuse of real-time functions could be demonstrated. Moreover, differences regarding the reviewability of software versions written for FPGAs and microcontrollers were observed. Finally, the development according to ISO26262 revealed only minor differences between the investigated hardware platforms but between the different safety concepts of *device supervision* and *function supervision.*

Acknowledgments

First of all, I would like to thank Prof. Dr. Stefan Kowalewski for giving me the opportunity to conduct this doctoral thesis at his chair at RWTH Aachen University. I especially appreciate his constant assistance, his helpful advices and valuable feedback, as well as the great degrees of freedom in determining the focus of my work. Furthermore, I thank Prof. Dr. Klaus Müller-Glaser for his interest in my work and for accepting the position of the second referee as well as Prof. Dr. Wolfgang Thomas, Prof. Dr. Horst Lichter, and Prof. Dr. Peter Rossmanith for their participation in the dissertation committee.

Many thanks go to all my colleagues at the Embedded Software Laboratory at RWTH Aachen University for providing a fruitful working atmosphere. Special thanks go to Dirk Wilking, who inducted me into the concepts of empirical evaluations and whose feedback was always very important to me. Moreover, Bastian Schlich helped me a lot with respect to conceptual and organizational aspects. I am also grateful for the support I gained from my student assistants. Especially the assistance of Ramona Dülks and Bodo Felger during the preparation of my experiments and their work on the development of the test environments was very useful. Further thanks go to Martin Lang and Thomas Gatterdam for their great work for the FAT project.

Further on, I want to thank all students who conducted their diploma theses under my supervision. Their work was a valuable feedback and in this respect my special thanks go to Eva Beckschulze, David Boymanns, Emilio Codina, Clemens Crämer, Ramona Dülks, Alexander Göres, Jianmin Li, Alexander Mehlkopp, Daniel Plugge, Thomas Siegbert, Sandra Theidel, Julian Wild, and Xiaoqiang Zhang. Further thanks go to all the students who participated in my experiments. Their development and implementation work was a very important part for the evaluations conducted for this thesis.

I also thank the Research Association of Automotive Technology (Forschungsvereinigung Automobiltechnik, FAT) for funding the project *Reliability for Automotive Embedded Systems*. Working for this project gave me valuable inputs for my research work and I especially thank all project members for the interesting discussions and their helpful feedback on topics of safety-critical automotive applications.

Moreover, many thanks go to Adam Taylor for his constant support concerning the application of FPGAs in safety-critical applications. It was really

valuable for me to profit from his industry perspective during my work.

Finally, I want to thank my friends and family for supporting me during this work. My special thanks go to my wonderful wife Verena Thaler for her constant support and her precious feedback concerning all presentation and writing issues.

Falk Salewski
Aachen, October 2008

Contents

List of Abbreviations

ASIL	Automotive Safety Integrity Level
BIST	Built In Self Test
BTN	push button
CAN	Controller Area Network
CMOS	Complementary Metal-Oxide Semiconductor
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DSP	Digital Signal Processor
DUT	Device Under Test
ECC	Error Correction Codes
ECU	Electronic Control Unit
EEPROM	Electrically Erasable Programmable Read-Only Memory
Exp.	Experiment
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HW	Hardware
I/O	Input/Output
JTAG	Joint Test Action Group
LB	Life Beat
LED	Light Emitting Diode
max.	maximum
MCU	Microcontroller Unit
min.	minimum
n.a.	not applicable
NVP	N-Version Programming
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
OTP	One Time Programmable
PC	Personal Computer
PLD	Programmable Logic Device
RAM	Random-Access Memory
SEL	Single Event Latch-up
SEU	Single Event Upset

SIL	Safety Integrity Level
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SW	Software
TB	Test Bench (set of test cases)
TID	Total Ionizing Dose
TMR	Triple Modular Redundancy
USB	Universal Serial Bus
VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

1. Introduction

1.1. Problem Definition and Objectives

The complexity of today's control applications is currently increasing. These applications range from autopilots in airplanes and brake assistants in modern cars to complex assembly lines in production automation. To enable flexible and efficient approaches, increased amounts of programmable electronics, whose behavior is substantially determined by software[1], are applied to implement these applications. A commonly used term for these programmable devices is *embedded systems*, as they are embedded into another system (e.g. car, airplane, assembly line).

According to the given examples of these systems, it is obvious that failures of these devices could lead to dangerous consequences (e.g. serious accident if brake assistant disables brake activity). In systems, in which the consequences of failures are not tolerable, measures are required to assure that the risk of such a failure is adequately low. Effective measures to reduce the risk of failure, as over-designing[2] (e.g. enforced steering column) and redundancy (e.g. doubling of pressure-relief valves) of critical components, are known in the area of mechanical components. However, problems might result from the costs of these additional measures and their additional weight could also be critical in several embedded applications. Moreover, if the concept of redundancy is applied, it has to be assured that the risk of both redundant units failing at the same time is acceptably low (e.g. two identical valves could both stop working if the temperature is above or below a certain threshold).

These techniques can be transferred partly to the hardware parts of embedded systems. While over-designing with respect to safety is not always suitable (e.g. improving the robustness of a semiconductor device by increasing its internal structure size typically reduces its performance), the concept of redundancy is generally applicable. As in the case of mechanic components, additional costs are often a problem of both approaches. To allow low cost devices that also meet the remaining requirements present in safety-critical embedded applications, the

[1]The term *software* includes all non-physical parts of a systems that determine its behavior, while all physical parts of an embedded system are referred to as hardware.

[2]The term *over-designing* describes measures of enforcing a component itself without adding redundancy.

use of highly integrated components would be desirable. However, applications with very high production volumes (e.g. consumer electronics, telecommunication, office applications), which drive the development of highly integrated components, are usually not safety-critical. Therefore, these devices do not include measures specific for safety-critical systems. As most of today's safety-critical systems are produced in comparatively low numbers (e.g. autopilot in an airplane), typically general purpose devices have to be applied for cost reasons, and safety requirements have to be targeted by additional measures. While the increasing amount of safety-critical applications (e.g. in modern cars) might change this situation in the future, this lack of built-in safety measures has to be considered in current applications.

On the other hand, these approaches are not applicable for software. While over-designing is not possible for software, a simple doubling of software can also not be considered as a solution. A fault in one software module would be present in the doubled version also, counteracting the idea of redundancy. The problem is that all software faults are introduced into the system at design time and therefore are *systematic*. Software can affect system safety in two ways: First, it can exhibit behavior that contributes to the system reaching a hazardous state. Second, it can fail to recognize or handle hardware faults that it is required to control [58]. Moreover, testing of software is notoriously difficult, as testing of every possible situation is usually not possible. Testing in embedded systems is further complicated by real-time properties present in most of these systems. Thus, it is not surprising that many engineers see the production of dependable software as the most important element in the creation of safety-critical systems [117].

Consequently, it is of major interest to avoid faults in the software of embedded systems. Additionally, approaches to tolerate the software faults that remained in the system upon completion are desirable. However, not many established results on influences regarding software reliability are available [79].

As further described in Section 2.1, many different hardware platforms are available in embedded systems. These platforms range from CPU based systems to programmable logic devices (PLDs) and differ with respect to their programming languages, design processes and their corresponding tools. According to limited resources and real-time requirements, present in most embedded systems, a close correlation between hardware and software is present in these systems. Therefore, it seems necessary to investigate the impact of hardware platform selection on faults in embedded software and fault handling[3] in general. The investigation of this impact is especially interesting, as decisions for a specific hardware platform these days usually consider only the handling of

[3]The term *fault handling* comprises *fault avoidance* and *fault tolerance* (see Section 3.2).

hardware faults or even neglect fault handling aspects.

Accordingly, the objective of this work is to identify impacts of hardware platform selection on the fault handling in embedded systems, especially in their software. Therefore, a suitable methodology of evaluation is required for these investigations. According to the large scope of potential hardware platforms, a selection of representative devices is required for the evaluations. Moreover, a methodology allowing the conduction of further evaluations in this field is desirable.

1.2. Contribution

To support the objective presented in the previous chapter, the main contributions of this work are presented in the following.

First, we identified fault handling as the major aspect for the comparison of embedded hardware platforms for their suitability for safety-critical systems. In this context, we developed a model of the impacts on the handling of hardware and software faults, which is presented in Chapter 3. Based on this model, a literature survey revealed several open issues, mostly with respect to the handling of software faults.

In our investigations, we compared the fault handling properties of different hardware platforms by applying empirical evaluations. For the investigation of the open issues identified, we performed 6 experiments with more than 100 participants resulting in 86 final (and several more intermediate) program versions, 48 review reports and 14 test reports. As a result, several differences with respect to fault handling could be shown for the considered hardware platforms. The results obtained are suitable to support the process of hardware selection in safety-critical embedded applications. Moreover, the methodology of evaluation presented in this work is suited to support the development of further experiment designs in the area of embedded systems. Additionally, the empirical data collected in our experiments was used for a case study of a software tool developed at our institute [111]. Finally, the obtained data is suitable for further empirical investigations.

1.3. Thesis Structure

The remaining thesis is organized as follows. Chapter 2 describes preliminaries, and is mostly a brief introduction into embedded systems and safety-critical systems. As impacts of hardware platforms in safety-critical systems are of interest for this work, safety-relevant impacts are evaluated in Section 3. In this section, which represents the analysis of related work, open issues are identified and selected aspects are chosen for later analysis.

The chapters 4 to 6 describe the empirical evaluations conducted for this thesis. The planning of the evaluations is described in Chapter 4. The different planning aspects for the conducted experiments are described jointly in this chapter, while illustrating the existent differences between the experiments. Next, the execution of each of the six experiments is described in Chapter 5 and the evaluation of the experiment outcomes can be found in Chapter 6. Finally, Chapter 7 represents an evaluation of educational and organizational aspects of the experiments conducted as well as a discussion of the significance of the obtained results.

1.4. Bibliographic Notes

Parts of this thesis are based on work presented in earlier publications. An early survey on fault handling in embedded systems was published in [93, 94]. The results of our first experiment investigating the diversity of software faults on different hardware platforms have been published in [106]. Moreover, an early identification of safety-relevant impacts of hardware platform selection was presented in [95]. The impacts of MCUs and FPGAs on the fault handling in safety-critical systems were partly published in [103]. Results of the second and third experiment (encapsulation, reviewability) can be found in [98] while parts of the results achieved by the forth experiment (test and review) were published in [96, 97]. The results of the fifth experiment (reusability) were presented in [101]. Moreover, the testing issues presented in 4.6.6 were described in [100] before. Finally, the application used for the first experiment was described in [105] in combination with educational aspects. A presentation of the application used in Experiment 2 can be found in [111]. Furthermore, the approach of systematic hardware platform selection presented in Chapter 8 was published in [99] with regard to educational aspects and was applied in [104] for a survey on the suitability of FPGAs for industrial applications.

2. Preliminaries

The preliminaries presented in this chapter include a brief introduction to embedded systems with a focus on the two hardware platforms investigated in this work. Moreover, a short introduction to safety-critical systems is given.

2.1. Embedded Systems

Several important properties of embedded systems have already been mentioned in the introduction of this thesis. First of all, embedded systems interact with another system in which they are embedded, the so called *embedding system*. Requirements for the embedded system have to be derived from requirements for the embedding system. These requirements include functional, but also non-functional requirements as dependability, modifiability, reusability, and marketability (see [99] for further details). Second, the interaction with the embedding system usually requires real-time properties of the embedded system. In this case, the correctness of an operation depends upon both, the correctness of the output values and the time at which they are released. These real-time requirements have to be considered, if functionalities (e.g. to avoid and tolerate faults) are introduced into the design.

Moreover, embedded systems can be subdivided into *product automation* and *production automation*. While in the first case, the embedded system itself is the product which is sold (e.g. mobile phone), the embedded system is used for production in the second case (e.g. control unit for a chemical process). Devices of these categories have different properties, mostly originating from differences in the production volume. In this work, the focus is on the first category of embedded systems, but several results could be transferred to the second category. An aspect typical for embedded systems of this first category are limited resources, e.g. regarding size, weight, power consumption, and costs. The cost aspect is usually resulting in limited computational power and comparable small memories. To fulfill the given requirements with the restricted resources, software for embedded systems is often developed closely to the hardware. In this manner, very short response times and computation cycles can be achieved with limited computational power. Thus, hardware abstraction and operating systems are only applied if necessary and if real-time requirements can be met this way.

If a function is implemented in the field of embedded systems, several different *hardware platforms*[1] are available. These platforms are ranging from CPU based systems as microcontrollers (MCUs) and digital signal processors (DSPs) to programmable logic devices (PLDs) as complex programmable logic devices (CPLDs) and Field Programmable Logic Arrays (FPGAs). The different platforms lead to differences in design and development activities. Since FPGAs and MCUs are compared as popular representatives of CPU and PLD based embedded systems for this work[2], these two devices will be introduced briefly in the following.

Microcontrollers and FPGAs

In principle, a given function could be implemented on both types of hardware platforms (MCUs and FPGAs) [119]. Depending on the functional and non-functional requirements, one hardware platform might be suited better than another one. We described the structure of both devices briefly in [99] while further information on the structure of FPGAs can be found in [70, 90]. Moreover, the publications [119] and [114] provide a good comparative introduction into CPU and PLD based systems. Therefore, only selected functional differences are presented below.

The implementation of a transfer function that does not change during run time is generally easy to implement on an FPGA. A fixed transfer function is present if, for example, required constants (offsets, sensors characteristics, etc.) are loaded into the module at start up. Moreover, changes in parameters of the transfer function can also be handled at run time. If it is required that calculations have to be performed with different transfer functions that depend on run time options, an MCU will be typically more suitable.

Otherwise, the parallel structure of FPGAs allows to implement concurrently operating modules. This aspect is especially suitable if several real-time functions have to be executed in parallel (e.g. signal generation or signal measurement). If sequential operations are required within an FPGA, they have to be implemented by using state machines or by integrating soft-core processors in the FPGA design. MCUs, on the other hand, usually include on-chip peripherals (e.g. different types of timer units) that also allow a concurrent execution of real-time tasks. However, the application of these hardware units is limited to

[1]The term *hardware platform* comprises all programmable electronic devices that are suitable for control tasks in embedded systems.

[2]A CPLD was used for the first experiment. However, FPGAs and CPLDs differ from the programmer's point of view only in the amount of logic that can be synthesized on them. Performance and power consumption issues might lead to further differences, but these differences could be neglected for this work.

those available on the chosen MCU, while the required function could be simply implemented in FPGAs. Finally, the advent of dual-core microcontrollers allows the introduction of a certain parallelism in MCUs.

Summarizing, there are differences in the development between MCUs and FPGAs, which could have an impact on functional and non-functional[3] properties of embedded systems.

2.2. Safety-Critical Systems

As mentioned before, an increasing number of embedded systems is applied in safety-critical applications. An application is considered as *safety-critical*, if faults in this application could lead to hazards (e.g. severe damage of people or the environment). All parts of these systems which could contribute to hazards are referred to as *safety-related* (see e.g. [46, 116]).

In safety-critical systems, specific requirements have to be considered. In this regard, it is important that safety is a system problem [58]. Thus, it has to made sure that the **combination** of hardware and software never leads to an unsafe state. This property is generally achieved by implementing a sufficient *safety function*. A safety function is responsible for the detection and the handling of all faults which could lead to unsafe states of the overall system. One form of fault handling is to shut down the system as soon as a critical fault is detected (*fail-silent system*). Another option is to put the actuators in a safe state and to try a form of fault recovery. This recovery could include a simple reset of the system (could mitigate transient hardware faults and some software faults) or a more fine grained recovery (e.g. to defined recovery point in the system). The disadvantage of these approaches is the interruption of the system's service during fault handling. This interruption might be not acceptable for safety-critical systems that require permanent service (e.g. drive-by-wire system). A combination of two fail-silent units to one *fail-operational system* is one solution to this problem (see e.g. [71]).

The reliability of the safety function is essential in safety-critical systems. First of all, it is important that all safety-relevant faults are detected and sufficiently handled to assure that safety requirements are not violated. Moreover, the fact that the safety function could perform a shut-down, a reset, or other recovery operations of the system, has an impact on the availability of the original application. Thus, the reliability aspect of functions implemented in safety-critical embedded systems is of great importance (and thus the major focus of this work).

[3]Note: *non-functional properties* do not represent the required functionality itself but additional *qualities* of the system (e.g. reliability, maintainability and testability, see also [99]).

For the development of safety-critical applications, several safety standards have been developed in the last decade. While certain standards are specific for a certain application domain, the standard IEC61508 [46] is comparatively application independent. Nevertheless, specialties of the process automation domain had an influence on the contents of this standard. The standard is based on integrity targets that are named safety integrity levels (SILs). These levels are described by one of four discrete bands (SIL1...SIL4) with SIL4 representing the highest target (most critical application). Next, an assessment of the design, the designer's organization and management, as well as the competence and training of operators and maintainers is carried out in order to determine if the equipment actually meets the target SIL in question [116]. Thus, all activities in the life cycle of a system have to be considered. In order to fill the roles of being both a template for the development of application specific standards, and being a standard in its own right, the IEC61508 necessarily leaves much to the discretion and interpretation of the user [116]. Therefore, no advice for hardware platform selection can be found in the IEC61508. However, a new annex is currently developed for the IEC61508, which targets new aspects of microelectronics (61508-2 Annex E: Special architecture requirements for ASICs with on-chip redundancy [47]).

According to the generic nature of the standard in combination with its background in process automation, difficulties might occur if the standard is applied in application domains that differ significantly from process automation. An example is the automotive domain. Among other aspects, the determination of safety integrity levels described in the IEC61508 is not suitable for this domain. Therefore, the specific safety standard ISO26262 [48] is currently developed for automotive applications. This standard is also based on integrity levels, the so called automotive safety integrity levels (ASIL), which range from ASIL A to ASIL D with the latter representing the highest target. Also the remaining contents of the current version of this standard (baseline 7 was considered for this work) is structured similarly to the IEC61508, but includes more details with respect to the implementation of safety-critical automotive applications. Nevertheless, recommendations for hardware platform selection cannot be found in this standard.

Finally, certain terms describing aspects of safety critical systems are used throughout this thesis. Beside the terms explicitly introduced in this and the previous section, the terms *fault, error, failure, reliability, safety, fault prevention, fault tolerance, fault removal, fault avoidance, development phase, use phase, random testing, deterministic testing, content failure, silent failure, timing failure, common mode failures*, and *fault activation reproducibility* used in this work are compliant with those described in [5].

3. Identification of Safety-Relevant Impacts

The research focus of this work is concretized in this chapter. Thus, design decisions that might affect the safety of an embedded system are presented (Section 3.1). In this context, fault handling is identified as an important aspect for safety-critical systems and is investigated in Section 3.2. Next, open issues in fault handling are identified (Section 3.3), whereof a selection was investigated in this work (Section 3.4).

We have published aspects of fault handling in embedded systems, which are presented in this chapter, in [93, 94, 95, 103].

3.1. Potential Safety-Related Impacts

Although safety requirements are of great importance in safety-critical systems, first design steps of these systems are often focused on functional requirements only. Accordingly, the safety standard IEC61508 allows the consideration of the system's safety after its functional development. The reason is that the IEC61508 has its background in process and production industries in which application and safety function are typically developed separately (see e.g. [42, 52]). The standard ISO26262 on the other hand propagates a planing and execution of all safety measures parallel to actual development activities. As described in Section 2.2, the ISO26262 is especially developed for the automotive domain. In this domain, application and safety function are often closely coupled according to requirements for low cost, low volume and low power consumption, which makes an early consideration of safety requirements necessary. Nevertheless, even in this case, several design decisions are typically made before the safety considerations. Thus, the design decisions made may lead to an insufficient design with respect to safety aspects. While this insufficiency might result in unnecessary overheads in development time or may require additional hardware measures increasing the overall costs of the system, safety requirements may even require changes of the complete system. Accordingly, it would be beneficial to know in advance how certain design decisions will affect the safety of the system.

For this reason, the influences of design decisions on the reliability of a function in embedded systems are considered in the following. A collection of some potential impacts is listed in the following:

- Design processes (none, V-model, life-cycle, etc.)
- Programming languages (Assembly, C, ADA, VHDL, etc.)
- Development style (code based ↔ model based design)
- Software architecture
- Operating system, hardware abstraction
- Hardware architecture (e.g. redundancy)
- Hardware platform

It is important to note that the selection of the hardware platform has an impact on most of the remaining aspects. As an example, a certain operating system or a specific programming languages might not be available for every hardware platform. This dependency supports the importance of hardware platform selection investigated in this work.

Moreover, the impacts of all these aspects listed above could be defined by their impact on the different techniques of fault handling (e.g. a specific programming language might allow especially efficient avoidance of software faults). Therefore, an analysis of fault handling techniques in embedded systems is presented in the following section.

3.2. Impacts on Fault Handling in Embedded Systems

Fault handling can be applied at different life-cycle stages of embedded systems. In the development phase, measures can be applied to *avoid* the occurrence of faults in the later system. This *fault avoidance* could be further divided into *fault prevention* on the one hand (e.g. by suitable design processes) and *fault removal* on the other hand (e.g. by testing and formal verification). Other measures try to *tolerate* faults, which have not been detected and removed during development, during the use-phase. This *fault tolerance* includes *error detection* and *handling of errors and faults* during the use-phase. Moreover, faults in embedded systems can occur as well in hardware as in software. Further on, fault handling techniques are also possible in hardware and/or software.

While several aspects have to be considered for an evaluation of fault handling measures, often only isolated aspects are considered in current publications. As an example, several publications consider different aspects of hardware fault handling (e.g. [9, 12, 51, 77, 124]), but do not consider aspects of software fault handling. Further on, comparisons between different hardware platforms as MCUs and FPGAs are rare. Similarly, recommendations given by safety standards generally focus on selected aspects. As an example, a rating of programming languages with respect to their suitability for safety critical systems can be found in the IEC61508. However, this rating is very general and limited to selected languages for microcontrollers only. Moreover, certain recommendations for software architecture aspects are given (e.g.: it is highly recommended for a SIL3 system not to use dynamic objects and to make only limited use of interrupts, pointers and recursion [46], Part 3, page 87). Further software architecture recommendations are given in the Annex C of Part 7 of this standard (e.g. the use of N-version programming). Unfortunately, the recommendations are usually very generic and some of them lack a well founded reasoning for their recommendation (e.g. for the use of N-version programming). Finally, the considered safety standards do not explicitly consider differences caused by diverse hardware platforms.

To organize the different aspects of fault handling in embedded systems, we propose a structured representation as depicted in Fig. 3.1. This figure consists of two major parts: fault handling in software (left part) and fault handling in hardware (right part). For both cases, impacts on fault avoidance (upper part) and fault tolerance (lower part) are considered. HW as well as SW could be influenced by the design process, available verification capabilities and the architecture chosen. Additionally, a factor influencing the fault handling in hardware is the device technology used for the actual device. Several fault handling techniques are known for software and hardware and several of these aspects depend on the hardware platform chosen.

Afterwards, these hardware dependent safety aspects are discussed for embedded systems. For clarity reasons, the comparison of hardware platforms will focus on MCUs and FPGAs, as popular representatives of CPU based systems and programmable logic devices (see Section 2.1). In the next subsection, the handling of hardware faults is surveyed followed by a subsection dealing with the handling of software faults.

3.2.1. Handling of Hardware Faults

This section gives a brief comparative overview on the handling of hardware faults in MCU and FPGA based embedded systems (aspects listed in the right part of Fig. 3.1).

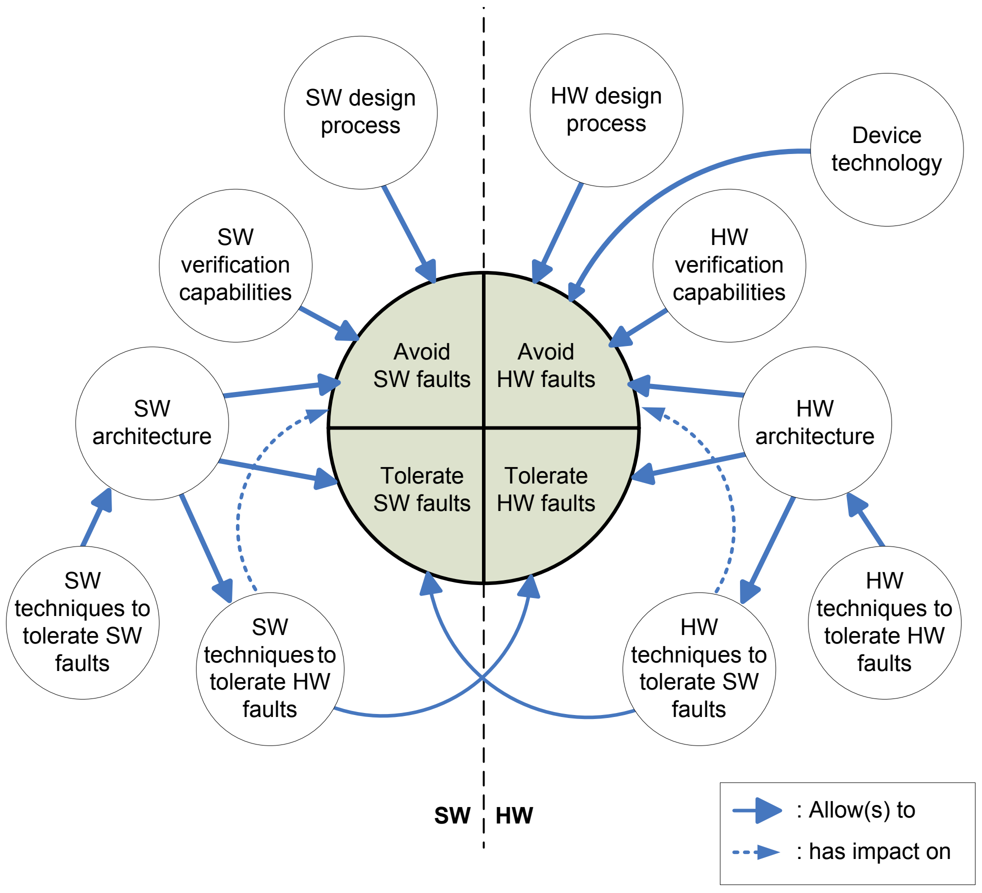

SW design process
HW design process
Device technology
SW verification capabilities
HW verification capabilities
Avoid SW faults
Avoid HW faults
SW architecture
HW architecture
Tolerate SW faults
Tolerate HW faults
SW techniques to tolerate SW faults
SW techniques to tolerate HW faults
HW techniques to tolerate SW faults
HW techniques to tolerate HW faults
SW
HW
: Allow(s) to
: has impact on

Figure 3.1.: Impacts on fault handling: HW vs. SW issues

Hardware Design Process

It is expected, that the hardware design process has a high impact on the quality of the resulting hardware. Especially measures to obtain certain levels of quality (e.g. testing, burn in) are of great importance for the avoidance of faults in the later device. Therefore, process requirements with respect to the hardware design process are present in safety standards as IEC61508 and ISO26262. However, safety standards do not distinguish between different types of hardware platforms. Moreover, impacts of the hardware design process are probably very similar for MCUs and FPGAs and are not considered any further in this work for this reason.

Device Technology

Publications dealing with hardware reliability often target specific problems in aerospace or space applications (e.g: SEU (Single Event Upset), SEL (Single Event Latch-up), TID (Total Ionizing Dose)). Although these effects are less critical at ground level, they are still existent [9, 12, 77]. Especially [77] gives a good overview on how memory cells could be affected at ground level. These effects have to be considered in safety-critical systems since a fault in the hardware of such a system could lead to catastrophic consequences. As a first approach, measures have been introduced to harden the silicon structure (e.g. alpha particles can be shielded by a thick polyimide layer prior to package), but shielding does not work for high energy cosmic radiation [9]. Furthermore, TID ratings can be improved through special circuit design techniques and shielding methods [124], and SEL has been all but eliminated for many FPGA devices through the incorporation of special current limiting circuitry [124]. The approaches presented have two drawbacks: First, devices including these measures are targeted primarily at space based applications and as such their cost is often prohibitive for use in many designs. Second, SEU cannot be avoided completely by device techniques and has to be mitigated by further measures in any CMOS device used in safety-critical applications.

One interesting question is whether certain device techniques are especially suitable to built reliable devices. As an example, increasing device density and larger dies of current devices make the devices less reliable [55]. Thus, "old" silicon techniques might be more robust than recent ones. However, designing with older devices can lead to a major problem in obtaining devices for repairs or new builds of the system. Moreover, there have been improvements in silicon techniques as described in [9]. Some sources of soft errors were detected in the last decades and according measures to avoid these are present in newer techniques only (e.g. removal of borophosphosilicate glass (BPSG) to avoid soft errors from low energy cosmic neutron interactions [9] or integration of specific shielding). Moreover, the authors of [51] indicate that (in case of FPGAs) it is not only the feature size, but also the supply voltages and the architecture itself which influence the hardware reliability. Thus, the use of improved techniques in combination with a large scale process might be an option to increase the robustness of the circuitry. Though, this concept again requires a production of specific devices, which is leading to increased costs.

Another aspect often discussed is the memory technique used for program and configuration memories. One time programmable (OTP) memories, as for example *anti fuse* technology, are very robust to soft errors [66]. However, they are less flexible and it has to be kept in mind that even devices, based on anti fuse program/configuration memories include SRAM for storage of dynamic data.

In-circuit reconfigurable memories allow higher flexibility than OTP devices. According to [124], the most promising in-circuit reconfigurable devices from the TID, SEL, and SEU point of view are those that are based upon FLASH or EEPROM. The most flexible but also most sensitive type are SRAM based memories. As already stated above, all devices include SRAM for storage of dynamic data. Accordingly, soft error mitigation techniques have to be applied on all devices and the only difference is the effort that has to be taken to check the program/configuration memory. Important factors here are how fast an error can be detected and how long it takes to mitigate the error. These issues will be handled in later parts of the section *Hardware & Software Architecture (Fault Tolerance)*.

Hardware Architecture (Fault Avoidance)

Techniques to avoid and to tolerate hardware faults could be applied at the hardware architecture level. In this section, architectural measures to avoid hardware faults are considered for MCUs and FPGAs, while fault tolerance techniques are discussed in the following section.

First of all, MCUs and FPGAs are affected differently by transient hardware failures. In case of MCUs, the program memory, the data memory, general and special purpose registers as well as internal buses could be corrupted. In case of FPGAs, the configuration memory, FlipFlops, internal buses/interconnections and, if available, the data memory could be corrupted. Changes in the configuration memory could fundamentally change the behavior of the FPGA. However, fundamental changes are also possible, if certain registers are affected in MCUs [121].

Finally, parts of the architecture, which are not directly involved in the application itself have to be considered. One of these parts are on-chip programming and debugging facilities as the Joint Test Action Group (JTAG) interface (IEEE standard 1149.1) [51]. This interface can be very useful during the development phase, but it is important to ensure that this interface remains inoperable during normal operation. Otherwise, noise at the JTAG pins appearing as a valid command could result in unpredictable operation of inputs and outputs [74].

While all aspects presented should be considered during hardware platform selection, no general differences in hardware fault avoidance in the architectures of FPGAs and MCUs could be found. Next, measures of hardware fault tolerance by hardware and software measures are considered.

Hardware & Software Architecture (Fault Tolerance)

The tolerance of hardware faults can be generally achieved by hardware and/or software measures (see Fig. 3.1). Depending on the type of hardware platform, different system components can be affected by hardware faults as described in the previous section.While an overview of several software based techniques for the tolerance of hardware faults on MCUs and FPGAs is given in [38], further fault tolerance aspects are considered in the following.

MCU: If the program or data memory of an MCU has a transient error, Error Correction Codes (ECC) can be used to mitigate the error. These codes can be implemented in software (no specific hardware needed, memory and computation time overhead, might increase complexity of the software). Moreover, an increasing number of MCUs with built-in hardware ECC for program and/or data memories is available (increased hardware costs, no software overhead). Mitigation of faults in CPU registers could be realized in software with time redundancy (redoing calculations and compare results). The protection of specific MCU registers as program and stack counter is more complicated. An implementation of this protection in a pure software approach leads to a high overhead (e.g. speed decrease (3 times) and memory overhead (4.5 times) [75]). Therefore, this approach is expected to complicate the software architecture and the verification of real-time requirements.

An approach using standard hardware has been proposed in [78]. This approach applies super-scalar processors and duplicate instructions for the mitigation of transient hardware faults. Further approaches propose the integration of specific hardware based reliability measures in processor based systems. Examples can be found in [120] (making registers SEU tolerant), in [12] (control flow checking), and in [13] (data checking). Eventually, an MCU for automotive applications which included BIST (Built In Self Test) was proposed in [7].

In addition, faults in the CPU could exist beyond those in registers (e.g. faults in CPU logic). These faults are often permanent and require specific measures to remedy them. While some of these faults could be detected with a standard watchdog (if the fault effects the function that triggers the watchdog), several faults might not be identified this way[1]. More sophisticated approaches to handle faults in the CPU are mostly based on two CPUs which compare their results. One approach with two CPUs could be an implementation with two discrete MCUs comparing their outputs. Because of the high hardware costs, this approach is considered as unattractive for most embedded applications. An alternative to the use of two discrete MCUs is the use of dual-core devices. Different approaches of fault handling with dual-core devices are known, which

[1]Maximum diagnostic coverage considered as achievable by a watch-dog with separate time base and time-window is considered as *medium* only in [48].

include a structure of two identical cores running in *lock step mode*. In this structure, both cores are receiving identical inputs and executing the same program concurrently, but the output lines are driven only by one of the cores. During each cycle, the output generated by the first core is compared to the second core's output. Every mismatch between the outputs indicates a transient or permanent failure in one of the two cores (see e.g. [107]). Furthermore, an appropriate error detection scheme to protect the input and output lines of each core is required in this approach. Otherwise, an approach with a heterogeneous dual-core device in combination with additional hardware measures as hardware CRC and memory protection units can be found in [18]. Further approaches for fault mitigation are known as the use of co-processors available in several MCU devices. However, these co-processors usually can monitor only specific functionalities of the main processor. A structure with duplicated processors and error checking can also be realized by utilizing the embedded cores available within some Xilinx FPGAs. These processors can be operated in lock step mode and error detection logic could be implemented within the FPGA [129].

Beside a supervision that is independent from the application as in the approaches mentioned above, the second core could also supervise the *application* running on the first core. A comparison of this approach with the application independent approach presented above, can be found in Section 6.6 as part of our experiment evaluations.

Moreover, specific *checker hardware* could be integrated in single-core MCUs, as proposed in [65]. This additional hardware includes checker for all critical components of the device including the CPU, the memory, internal buses, and internal peripherals. According to [65], the reduced overhead in the chip area is the advantage of this approach compared to the dual-core approaches described above.

Finally, modern MCUs offer multiple on-chip peripherals such as memory and communication controllers while not all of them will be used in each application. These unused peripherals must still be handled in accordance with the MCU data sheet to ensure that they cannot interrupt or otherwise influence the program execution. Thus, the components used to terminate these unused peripherals will contribute to the calculation of the failure rate of the device.

FPGA: One major advantage of FPGAs with respect to reliability is that the integration of hardware reliability measures is possible on **generic** hardware. However, FPGAs have the disadvantage that soft errors could not affect only their memories used during processing but the hardware structure itself. Anyway, the configuration of an FPGA can be checked in different ways, depending on the FPGA device used. One method is to use a built-in constantly running CRC check, which is executed in the background of the FPGA and allows a reconfiguration if an error is detected [1]. Another option is to read back the configuration data from the configured FPGA and to check the received CRC [130]. Again, reconfiguration will be possible, if a mismatch is detected. It is important to note that both approaches require additional hardware. In the case of the first approach, additional circuitry is required to detect faults and reconfigure the FPGA while in the second approach, a configuration controller is required to configure the device and then initiate the read back CRC checks. Further on, the addition of such a CRC can affect the design: The maximum operating frequency of the device could be reduced by integrating the CRC checks (see [103]).

In [55], an approach of dynamic fine grained reconfiguration is proposed to overcome the problem of faults in the configuration. Another approach to handle the problem is the self configuration checker proposed in [123]. Further on, TMR (Triple Modular Redundancy) allows to mitigate most hardware faults by triplicating the application (3x application logic + voter). Partial TMR, as proposed in [85] is an option to reduce the area overhead in some applications. An overview on fault tolerance methods, including TMR, in SRAM based FPGAs can be found in [50]. Furthermore, it has been proposed in [89] to use alternating logic to detect hardware faults. The success of TMR and other FPGA reliability measures can be validated with tools as those presented in [85, 125].

Finally, also the vulnerability of the memory holding the original configuration file has to be considered [36]. However, standard error detection and correction measures, as those presented in the MCU section above, could be applied.

An aspect, which makes FPGAs very attractive for safety-critical applications, is the possibility to integrate individual safety measures into the device. An example could be a safety monitor that is applied to check the outputs of the application for constellations, which could violate the safety requirements. As this safety monitor can be implemented in parallel logic in the FPGA, this additional function has no impact (e.g. side effects by common CPU) on the original application.

However, single points of failures have to be considered when designing redundant units (as application + monitor or TMR) on a single chip. Sources of single point failures could be input and output pins including clock and reset lines. To overcome this problem, these in- and outputs should be provided to each redundant unit via separate pins. A further source of single point faults could be a fault in one redundant unit, which results in a critical temperature increase (e.g. short circuit). This temperature increase might affect other units on the same chip or even the overall chip. This problem could be partly mitigated by placing the redundant units with sufficient distance on the chip. Finally, the most critical aspect for single point faults are the common power supply lines used by all circuitry on the FPGA. In some applications, a monitoring of the power supply might be sufficient (e.g. by an external Watchdog chip with brown out detection) while other applications would require an approach with two separated chips (e.g. two FPGAs).

Beside a combination of two fail-silent units (which typically include two nodes each) the TMR approach allows to built a fail-operational system. In a TMR system, reliability will decrease, if a defect affects the functionality of one module. Accordingly, a defect module has to be repaired as soon as possible. The authors of [36] state some advantages of using three discrete reprogrammable FPGAs in a TMR architecture. A fault in one of the FPGAs can be detected and, in case of a transient failure, corrected by reconfiguring the defect FPGA. In case of a permanent physical defect in a minor part of the FPGA, reconfiguration has to be performed avoiding the faulty area. A certain area overhead is necessary in order to provide alternative circuit blocks and to include the logic managing the reconfiguration. Nevertheless, this approach allows a degree of fault tolerance that is unique for reconfigurable logic devices as FPGAs. While the risk of the described permanent faults is comparatively low, their handling is very important in applications that require high availability but prohibit maintenance during operation (e.g. space applications).

Hardware Verification

Another important aspect in safety-critical systems is to demonstrate that the hardware is always acting as intended. One approach is the formal verification of the hardware built (with respect to specification) which is applicable for both MCU and FPGA in the same way. The most common approach is functional verification by testing. In case of larger MCUs and FPGAs, it is usually not possible to test the complete functionality. Therefore, coverage criteria have to be applied for the test processes in case of both devices.

Otherwise, devices, which have been used in other applications without failures for a certain time, can be considered as *proven in use*. This property can be sufficient for less critical systems [46]. However, the use of these devices can be considered as a drawback as they do not represent state of the art technologies. Moreover, it has to be assured that the devices are really identical (e.g. identical fabrication process) and not only functionally equivalent.

Summarizing, no significant differences with respect to hardware verification could be identified for MCUs and FPGAs.

3.2.2. Handling of Software Faults

Real-time requirements, hardware/software dependencies, and the increasing complexity of applications make developing reliable software for embedded systems a non trivial task. Additionally, the software measures for hardware fault mitigation, described in the previous section, are expected to increase the software complexity. Therefore, handling of software faults is an important issue, which involves several aspects (see left part of Fig. 3.1). As for hardware fault handling, these aspects will be evaluated in the following.

Design Process

The software design processes of MCUs and FPGAs differ in many aspects. Differences can be found in the programming languages, the development styles (sequential vs. parallel), and the tool chains for development and debugging.

Today, the most common programming languages used for MCUs are C/C++ and ADA. According to the safety standard IEC61508, the language ADA is preferred over the C language [46]. However, subsets of the C language are still allowed, as the subset developed for the automotive domain called MISRA-C [68].

FPGAs are currently programmed mostly in VHDL and Verilog. However, recent publications propose the use of languages developed for CPU based systems as ADA [43] or Esterel [40, 44]. The reason behind these approaches is that classical hardware description languages (HDL) have only limited abstraction capabilities and are therefore not suited for more complex designs. Nevertheless, programmable logic devices cannot be treated as CPU based systems, especially with respect to the hardware description language. While the HDL may look like a software program for CPU based systems, it describes the function to be implemented in digital logic. Thus, the HDL development process has to follow different rules and procedures than the software development for MCUs to ensure fault tolerance or avoidance. Further on, additional transformations (e.g. Esterel to VHDL) are needed if "MCU languages" will be applied, which have to be verified if applied in safety-critical applications.

A motivation for higher programming languages is to handle the increasing complexity of today's embedded systems. In this context, approaches of *model based design* gain importance. In embedded software design, the underlying hardware has to be considered in safety-critical applications, since many faults occur at this hardware/software boundary (especially timing problems). This aspect is one reason that makes the application of hardware abstractions complicated in safety-critical applications.

Furthermore, languages as Esterel allow formal verification of the hardware description. Nevertheless, formal verification is also possible for systems written in VHDL [14, 24, 76] and is applied in industry already.

The different development styles of FPGAs and MCUs might have an impact on the reuse of real-time components. As components are included into the design as logic elements in FPGAs, their side effects are limited to the interface of the component. In MCUs, on the other hand, side effects could occur according to the common CPU, common memories as well as the use of common on-chip peripherals. Moreover, the low hardware dependencies of a design described in a hardware description language benefits reuse activities. If care is taken during programming activities, a real-time software component could be reused on any FPGA device without modifications. Higher reuse of own and/or standardized commercial components could have a positive impact on fault avoidance (e.g. the component could be considered as *proven in use*), but also potential negative impacts have to be considered (e.g. the component does not fit in the new context and modifications introduce new faults). Furthermore, excellent reuse properties of the hardware description language VHDL are stated in [57].

Finally, further process requirements are given by relevant safety standards as [46] and [48], but these requirements are very generic and do not depend on the type of hardware platform.

Software Architecture

All faults in software are introduced into the system at development time. In other words: all software faults are systematic. Therefore, handling of software faults mostly focuses on fault avoidance. However, measures of fault tolerance are also possible in the software architecture as will be discussed later in this section.

The approaches for fault avoidance in the software architecture are similar to those described in the previous section. As an example, coding guidelines are suitable to improve the structure of the developed software. This approach is applicable for MCUs and FPGAs. Moreover, coding guidelines are essential for FPGA software development from the functional point of view (see e.g. [88]). The reason is that not every hardware description that passes the syntax check is resulting in the synthesis of correct logic structures.

Furthermore, the implementation of a specific application might be easier on one hardware platform than on another. The resulting software architecture would most probably benefit from a development on the hardware platform suited best. A brief comparison of the functional properties of MCUs and FPGAs can be found in Section 2.1. Therefore, the implementation of several concurrent real-time tasks might be easier to handle on FPGAs than on MCUs.

Otherwise, fault tolerance measures can be applied in the software architecture. Generic techniques known to handle software faults are information hiding, recovery blocks or defensive programming in general (see e.g. [46, 48]). An example for defensive programming is the check of all input values (range, plausibility) by each function, which is processing these values. It has to be noted that certain tools can perform checks, e.g. the range check, automatically as it is the case in typical VHDL compilers.

Moreover, fault tolerance could be archived by redundant software units. An example is the implementation of a monitor function, which checks the results of the original application. While this approach has been already described for the tolerance of hardware faults in Section 3.2.1, additional thoughts are required if it is applied for the tolerance of software faults. In this case, it has to be assured that the application and the monitor are sufficiently different. In other words, they are not allowed to fail at the same time. To achieve the required failure independence between the redundant software versions, approaches of N-version programming [4] and functional diversity [62] have been proposed. Nevertheless, limitations of approaches that implement the same function by different teams to achieve failure independence have been observed [28, 53]. Finally, if a monitoring function or another form of software redundancy is applied on a single FPGA, the different functions (e.g. application and monitor) can be implemented in different encapsulated blocks. Therefore, side effects, as present in a single MCU according to a common CPU and common memories are not present in this case.

Software Verification

Verification is especially important for safety-critical systems [58, 117] and has become the bottle neck of the design process [49]. The main problem for verification is the increasing complexity of the developed systems. Thus, it is not possible any more to consider all possible combinations during the tests. Coverage criteria have been introduced to find areas of the design which are not tested. The authors of [33] indicate that an increase in the code coverage is likely to increase reliability. While this work was conducted on MCU software, there is no reason to believe the results could not be extended to HDL code coverage (in [49] an introduction to different coverage metrics used in HDL design can be found).

In order to ease verification, Lala and Harper proposed in [56] to partition redundant elements into individual fault-containment regions. These should be provided with independent power and clocking sources and their interfaces should be electrically isolated. As already discussed before, partition of redundant elements could not be achieved with a single MCU, but to some instance on a single FPGA (electrical isolation between redundant elements and independent power sources are not possible in today's FPGAs).

According to [124], FPGAs offer a benefit over traditional MCU based approaches as execution is more deterministic on a programmable logic device than on an embedded microprocessor using techniques as interrupt processing and cache memory. The determinism eases verification of real-time systems, especially as frequent failures in hard real-time systems result from the inability of the system to deliver the required services by the required deadline under various workload conditions [56].

Additional techniques like assertion based verification and code coverage analysis demand modifications of the code. In case of CPU based systems, these modifications change the timing properties, which could be problematic for real-time systems. Otherwise, assertions usually do not affect the timing properties of applications implemented on FPGAs.

Moreover, simulation is a common technique to verify designs, which can be seen as testing on a model of the system. Additional simulation of the environment is possible with certain tools. In case of FPGA designs, the environment generally can be modeled in the same HDL as the design. While an advantage of simulation is the dispensability of the concrete hardware platform, the simulation times may be considerable long, especially in larger designs. Therefore, at least in later stages of the development, the physical devices are used for testing of MCU and FPGA software (requires reprogrammable configuration/program memories).

Another form of verification is to inspect the code, written by another person or team, manually. Different approaches of reviews are well known [30, 81] and applied in industry. Challenges of review are at least twofold. The first challenge is to understand the unknown code, while the second challenge is to reveal a high amount of the faults eventually present in this code. In case of real-time systems, both aspects are challenging, as the correct timing is essential. If several real-time functions are executed concurrently, a verification of their timing behavior is complex on MCUs. This complexity is even increased if restricted resources permit the use of a real-time operating system. FPGAs on the other hand can execute concurrent real-time functions in parallel logic which might ease the verification of their timing behavior.

Formal verification of the software is another important issue. Approaches are available for systems written for FPGAs [14, 24, 76] and for MCUs [110] used in embedded systems. However, a formal verification of real-time properties remains a challenge for model checking of MCU code.

Finally, formal equivalence checking is applied in FPGA designs to verify that the *synthesis* and *place and route* stages (which usually occur after functional verification has been completed) have not introduced or removed logic and therefore changed the function of the logic. The design tools will attempt to modify and optimize the logical structure described by the HDL to make it best fit into the logical structures available on the target architecture. It is therefore important that in devices intended for safety-critical applications these processes do not change the behavior of the logic (also in case of hardware errors). An example of undesired changes can be found in [87]. Further aspects of the application of formal equivalence checking are described in [11, 103].

3.3. Summary and Open Issues

The previous section identified that sophisticated methods of fault handling are present for MCUs and FPGAs. MCUs and FPGAs are affected differently by hardware faults, but in both cases faults could fundamentally change the behavior of the device. Fault mitigation in program/configuration and data memories can be applied on both platforms by hardware or software techniques. Mitigation of transient faults in CPU registers needs further measures. They could be based on either pure software (resulting in an immense runtime and memory overhead), or specific hardware on general purpose hardware (super scalar, dual core), or on hardware specialized for safety-critical applications. Transient faults in FPGAs can be mitigated by full or partial duplication of functionalities resulting in an area overhead. Furthermore, mitigation of permanent faults is possible in FPGAs if only minor parts of the device are affected and partial configuration is supported. Systematic hardware faults could be mitigated by appropriate design processes and hardware verification capabilities on MCUs as well as on FPGAs.

Moreover, MCUs and FPGAs differ with respect to the handling of software faults. These differences include the software design process, especially the languages used with their specific properties and software verification capabilities. In both cases it has to be assured that the later system is behaving as intended. In case of MCUs, potential side effects (interrupts, memory, etc.) complicate this verification while in the FPGA design the handling of interconnected parallel processes might be challenging. Architecture measures can be applied in MCUs as well as in FPGAs to tolerate software failures during run time. However, in FPGA design it might be beneficial that these measures usually do not affect the timing behavior of the original function.

Summarizing, MCUs and FPGAs seem both suitable for application in safety critical systems. However, the differences in fault handling might be of interest, if the suitability of both devices is compared. Advantages and disadvantages of the different properties are obvious in some cases (e.g. avoidance of hardware faults, most aspects of hardware fault tolerance), but remain unclear otherwise (e.g. avoidance and tolerance of software faults). The latter is typically the case, when *human factors* contribute to the fault handling.

These human factors can have major impacts on the quality of intellectual products, as for example software. However, many approaches involving human factors are based on best practice approaches. This procedure may be critical, as some approaches are based on wrong assumptions. One example in this context is the improvement of reliability by using redundant structures. This structure will only improve the reliability if the redundant units fail independently. When transferred to software design, one method to improve this independence recommended by the IEC61508 [46] is the approach of N-version programming (NVP). In this approach, which was first proposed in [4], software for redundant units is developed by different teams. However, Knight and Leveson could show with an empirical experiment that even independent development teams tend to make the same faults [53]. The example of N-version programming clarifies the importance of empirical studies in the area of safety-critical systems. However, despite their importance, relatively few empirical studies have been published that investigate issues relating to the dependability of software [79]. Reasons can be seen in the high effort required for these investigations. In addition, these studies published (e.g. [8, 31, 53, 69, 79]) deal with faults in software only and do not take HW/SW dependencies into consideration which are an important factor in embedded systems.

Differences between the fault handling of MCUs and FPGAs have been identified. However, the specific advantages and disadvantages of certain aspects, especially of those dependent on human factors, remained unclear. For an investigation, empirical evaluations were introduced as a suitable measure to clarify these open issues instead of simply following best practice approaches. As it is beyond the scope of this work to investigate all unclear differences in the fault handling of MCUs and FPGAs, selected impacts of hardware platform selection were considered in this work. These aspects are presented in the following section. Nevertheless, the approach of evaluation presented in the chapters 4 to 6 could be applied to investigate further differences by conducting the experiments needed.

3.4. Investigated Impacts

As described in the previous section, open issues have been identified during the comparison of the fault handling on MCUs and FPGAs. In this section, a selection of those open issues that were investigated in this thesis is presented.

3.4.1. Software Diversity

An approach for the handling of software faults has been presented above, namely N-version programming. However, limitations of this approach were identified by analytical arguments [61] and empirical evaluations [28, 53]. To mitigate this problem, it has been proposed to increase the diversity between the redundant versions by additional measures [61, 84]. Examples for this approach, which is also known as *forced diversity*, are the application of different programming languages in [6] and different programming environments in [64]. Both experiments reported a decrease of dependent failures[2]. However, the analysis in case of the first experiment was based on only 6 different software versions. Thus, the validity of the results might be questionable.

Nevertheless, the approach of *forced diversity* might be successful in the context of hardware platforms in embedded systems. As for example, differences between the software design for MCUs and FPGAs were observed (see Section 2.1 and 3.2.2). Therefore, the application of different hardware platforms as a factor to force the diversity in the N-version approach might be interesting. Thus, the first aspect that was investigated in this work was the gain of diversity in software faults by applying different hardware platforms (Experiment 1).

3.4.2. Encapsulation

Effective encapsulation of critical functions is a desirable property in safety-critical applications [46, 48, 56, 109]. Reasons stated are the need to reduce faults in case of modifications [46] (Part 3, page 45), to ease verification activities [56], and to allow the integration of functions with different levels of criticality on a single device [109]. In the context of fault handling presented in Section 3.2.2, differences with respect to encapsulation have been identified between MCUs and FPGAs, especially with respect to real-time functions. However, it remained unclear, if the advantages seen in the structure of FPGAs really has a positive impact on the encapsulation. Thus, the impact of hardware platforms on the encapsulation of software functions was investigated in a second experiment (Experiment 2).

3.4.3. Review

Inspection of the source code developed is recommended by safety standards [48] (Part 6, page 31), [46] (Part 3, page 91). The application of code inspection, which is also known as *review*, allows to reveal problems in the software (insufficient documentation, bad code structure, potential risks as division by zero, undesired overflows, etc.) and to identify inconsistencies with the specification. Reviews offer chances to identify problems with respect to functional requirements, but also with respect to non-functional requirements as maintainability and reliability. However, the result of each review process depends a lot on the personalities of the reviewers and their review performance is hard to quantify.

[2]A failure is considered as *dependent*, if the failure occurs in two versions at the same test input. Moreover, dependent failures are also known as *coincident* failures (see e.g. [28])

To overcome this problem, it is desirable to have more than one reviewer and to control the review process (see e.g. [30, 81]). Moreover, real-time properties are an additional challenge for the review process [80]. Especially the time, which is needed for the execution of sequential code, is hard to determine, not to speak of program parts which can be disrupted by interrupts at any time. Differences in the software structure of MCUs and FPGAs might have an impact on the review of real-time functions as described in Section 3.2.2. Hence, possible impacts of this hardware platform selection on the review of real-time software was investigated in another experiment (Experiment 3).

3.4.4. Test and Review vs. N-Version Programming

According to the limitations of N-version programming, this approach was compared with measures of fault avoidance[3] in [112] (experiment) and [83] (analytical investigation). It is reported in [83] that there is as yet no evidence that the choice between design diversity and other means of reliability improvement can be decided by general arguments. Similarly, no clear results could be obtained in [112].

The dependent failures, present in the majority of the versions created in previous experiments, seem to be found easily as soon as they have been identified once. Furthermore, we added ambiguous statements to the specification which had not yet been identified by any experiment participant. In order to investigate which of the faults present in previous experiments are found, this aspect was investigated in a forth experiment (Experiment 4). The idea was to compare the dependent failures in the versions created in the first experiment with the faults that remained in the versions of this experiment after a fault removal phase based on review and testing.

3.4.5. Reusability

The systematic reuse of software is seen as a potential, powerful method of improving the practice of software engineering in terms of amortizing software development efforts [54, 57]. Despite the simplicity of the idea, software reuse still has its problems and limitations [37, 67]. Successful software reuse in embedded systems generally has to face further challenges according to real-time requirements, restricted resources, and safety and reliability requirements. To meet these requirements in embedded systems, the obvious solution is often seen in a redesign of the complete system for each application [73]. With re-

[3]as introduced in Section 3.2, *fault avoidance* is based on fault prevention and fault removal during design time

spect to safety, reuse can be seen as a chance, if software components[4] that are already verified and *proven in use* can be reused. Otherwise, reuse of application software has also resulted in major accidents (see e.g. [59]). Safety standards as the IEC61508 [46] recommend to reuse software components: *Use of trusted/verified software modules and components (if available)*, but also request additional safety measures. In [60], Leveson lists certain aspects to assure safe reuse of software. According to this work, the design rationale should be included with all design decisions in the specification. Additionally, a documentation of all assumptions concerning the environmental conditions that are implicit in the software as well as a documentation of the hazard analysis should be included. These requirements go well with requirements present in recent safety standards as IEC61508 [46] and the working draft of the ISO26262 [48].

Publications on different aspects of reuse are known for both, MCUs (e.g.: [60, 73]) and FPGAs (e.g.: [57, 108]), but publications comparing their reuse properties could not be found. As mentioned above, real-time requirements can be seen as a specific challenge for an effective and safe reuse in embedded systems. The differences in the internal structures of MCUs and FPGAs are expected to have an impact on the reuse of real-time functions as described in Section 3.2.2. To investigate the effect of these differences, empirical evaluations were chosen. Thus, the aspect of reuse was investigated in a fifth experiment (Experiment 5) with a specific emphasis on impacts on software faults.

Additionally, the concrete factors of interest had to be determined. Important factors of reuse can be seen in *comprehensibility* and *independability* [73]. While the first aspect stresses the ease of understanding the component to be reused, the second aspect comprises the degree of a component's independence from its environment. Apparently, deficiencies in both aspects could lead to serious faults. Nevertheless, we focus on the second aspect, since we expect in this case major differences with respect to different embedded hardware platforms. Further on, the reuse process itself (how to built reusable components, how to organize these components, how to find these components) is not part of the evaluation. Instead, our work focuses on the effect of a given reuse scenario on faults in an exemplary system (Experiment 5). As recommended in [60], this reuse was applied on component level.

3.4.6. Development according to ISO26262

Finally, safety standards, as IEC61508 [46] and ISO26262 [48], require certain measures for the handling of software and hardware faults. In case of safety-

[4]A software component is the implementation of one or more functions in software. It is a logically separable part of the software and consists of one or more software components and/or software units.

critical systems with higher safety integrity levels (ASIL C or higher in case of ISO26262), a sufficient handling of random hardware faults has to be demonstrated (see [48], Part-5, page 12 and [46], Part-2, page 55). This handling of hardware faults requires certain redundancy as described in Section 3.2.1. Redundancy is either required in form of independent redundant units, both executing the safety critical application, or in form of a component performing sufficient diagnosis of the unit executing the safety-critical application. In both cases, detected faults have to be handled by recovery measures (retry, reset, etc.) or by a shut-down of the system (see also Section 2.2).

Single hardware-platforms or combinations of identical/different hardware platforms are possible to meet the requirements presented above. For this work, the following two hardware architectures were selected[5]:

First, a dual-core microcontroller was selected. In contrast to dual-core microcontrollers which are currently developed specifically for safety-critical applications, this device is a general purpose MCU with two independent cores (see Section 4.3.2 for details of this hardware platform). The idea was to determine how suitable this general purpose device is for the development of an ASIL C application.

Second, a standard single core microcontroller was selected in combination with a standard FPGA (details in Section 4.3.2). The combination of these two hardware platforms was investigated as second hardware platform with respect to its suitability for implementing an ASIL C application.

The term of suitability is very general and requires further concretion. First of all, the development effort of the two approaches is an interesting aspect, as reductions of this effort are desirable for several reasons (e.g. costs, time-to-market). Moreover, impacts of the approaches on further important properties as reliability and modifiability are of interest. Therefore, a sixth experiment was conducted to enable the corresponding investigations.

[5]Selection was conducted by project members of the FAT/AK31 project *Reliable Automotive Embedded Systems*, according to potential trends in automotive electronics.

4. Methodology of Evaluation

Empirical evaluations were identified in the previous chapter as a promising measure to gain well founded information regarding safety-related impacts. Therefore, the techniques of empirical evaluations will be introduced in the following while the planning of the empirical studies actually conducted for this work will be presented in the sections 4.2 - 4.7.

Descriptions of the different experiments can also be found in [93, 96, 97, 98, 101, 106]. Moreover, parts of the testing issues in 4.6.6 were presented in [100].

4.1. Empirical Evaluations

Empirical evaluations can be categorized into three major types: the questionnaire based survey, the case study (observation based data collection during one project) and the controlled experiment (multiple participants, focused on statistical inference) [126].

The challenges in case studies result from the development of the system itself. The implementation task must be representative (complex enough), but also easy enough to allow an execution of the study in reasonable time. In addition, the control of variables is not always possible in case studies. Thus, small and simplified case studies may not be good instruments to evaluate principles.

As with case studies, the task must be complex enough in experiments to allow representative results. Further on, experiments are generally driven by one or more hypotheses, which should be confirmed or refused by the experiment results. Therefore, the structure of these experiments usually consists of two treatment groups implementing the same task. The results of the two treatment groups are compared and used to test the hypotheses. While one variable (treatment) is different for the two treatment groups, all other variables must be kept constant to show the impact of this specific treatment on the results. Experiments, which investigate aspects influenced by human factors, have to mask out the differences of the individual developers. One approach is to use a statistically significant number of developers which complicates the conduction of these experiments.

For this work, the impact of different hardware platforms on the corresponding software and the overall system was investigated. The design of software is strongly influenced by human factors. Therefore, controlled experiments have been applied for all of our investigations. During these experiments, several challenges were identified and possible solutions developed. Some of these challenges resulted from the specific properties of embedded systems and mostly had an impact on testing issues (see sections 4.6.6 and 4.7.4).

In accordance with [126], the experiment process is subdivided for each experiment into the experiment definition (Section 4.2), the experiment planning (sections 4.3 - 4.7), the experiment operation (Section 5) and the evaluation of the experiment outcomes (Section 6). Although six independent experiments have been conducted for this work, these different experiments share certain aspects of their definition and planning phases. Therefore, common aspects will be presented in the following sections, while aspects specific for certain experiments will be discussed in Section 5. The evaluation of the results in Section 6 will be centered around the hypotheses which are put up in the following sections. In this context, each experiment can contribute to one or more hypotheses. An overview of the experiments is given in Fig. 4.1 while the contents of this figure will be described in the following sections.

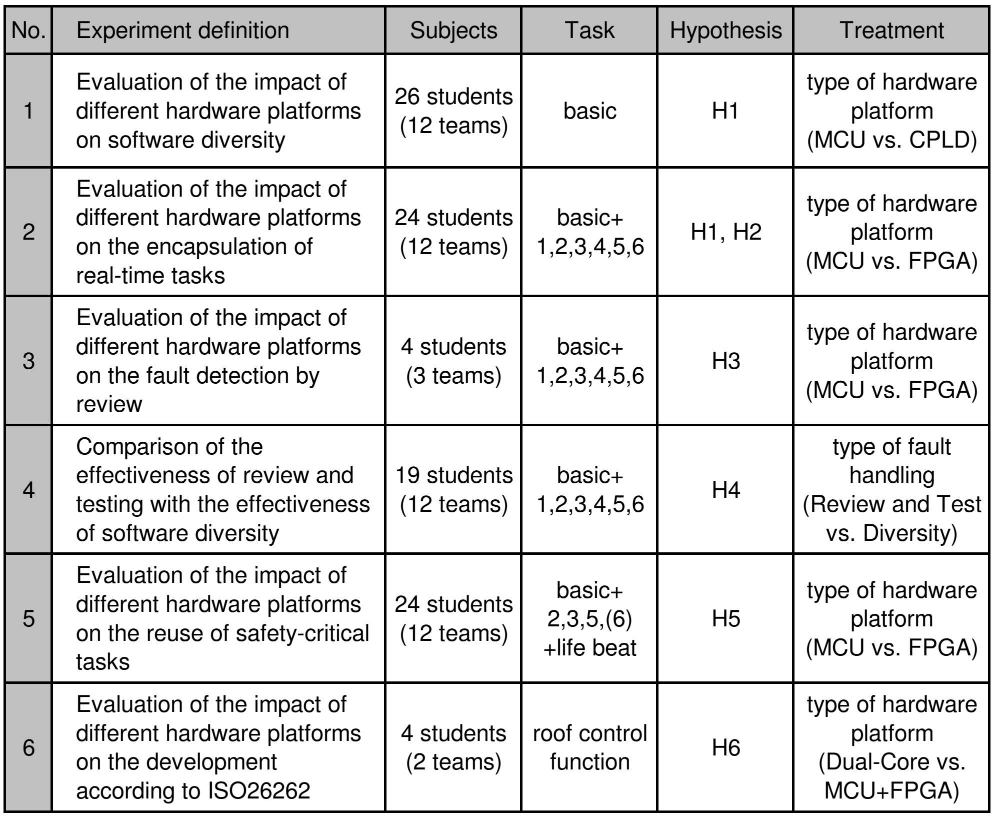
4.2. Definition of Experiments

The objective of the experiments presented is to investigate the impact of different hardware platforms on the aspects selected in Section 3.4. More precisely, the purpose of the experiments is to evaluate the impact of the application of two different hardware platforms on the effectiveness of

- diversity improvement of software faults (Experiment 1 and 2),
- encapsulation of real-time tasks (Experiment 2),
- fault detection by review (Experiment 3),
- safe reuse of real-time functions (Experiment 5),
- development according to ISO26262 (Experiment 6).

Moreover, Experiment 4 compares the effectiveness of review and testing with the effectiveness of software diversity determined in previous experiments.

The experiments 1, 2, 4 and 5 took place in lab courses with computer science students implementing a given application, while Experiment 3 was conducted using student assistants reviewing software versions developed in a previous experiment. Experiment 6 was conducted in the context of two diploma theses in computer science, which were supported by student assistants. Further details of the experiments can be found in the following sections.



No.	Experiment definition	Subjects	Task	Hypothesis	Treatment
1	Evaluation of the impact of different hardware platforms on software diversity	26 students (12 teams)	basic	H1	type of hardware platform (MCU vs. CPLD)
2	Evaluation of the impact of different hardware platforms on the encapsulation of real-time tasks	24 students (12 teams)	basic+ 1,2,3,4,5,6	H1, H2	type of hardware platform (MCU vs. FPGA)
3	Evaluation of the impact of different hardware platforms on the fault detection by review	4 students (3 teams)	basic+ 1,2,3,4,5,6	H3	type of hardware platform (MCU vs. FPGA)
4	Comparison of the effectiveness of review and testing with the effectiveness of software diversity	19 students (12 teams)	basic+ 1,2,3,4,5,6	H4	type of fault handling (Review and Test vs. Diversity)
5	Evaluation of the impact of different hardware platforms on the reuse of safety-critical tasks	24 students (12 teams)	basic+ 2,3,5,(6) +life beat	H5	type of hardware platform (MCU vs. FPGA)
6	Evaluation of the impact of different hardware platforms on the development according to ISO26262	4 students (2 teams)	roof control function	H6	type of hardware platform (Dual-Core vs. MCU+FPGA)

Figure 4.1.: Overview of the experiments conducted

4.3. Context Selection

Although the most general results can be achieved in an experiment if it is executed in a large, real software project with professional staff [126], the experiments presented in this work took place in an academic setting. The reasoning behind this decision are manifold. First of all, experiments involve risks for the project they are applied on. If an experiment takes place parallel to a real project, its conduction is resulting in high costs. Thus, experiments in the area of software implementation, which require high numbers of professional developers are possible only with the corresponding funding. These problems are less critical, if experiments are conducted at universities or other teaching organizations. This way, comparatively large numbers of participants are available through lab courses or project works. Furthermore, experiments in an educational context are probably easier to control. Otherwise, students might be considered as less experienced. The topic of students as experiment participants

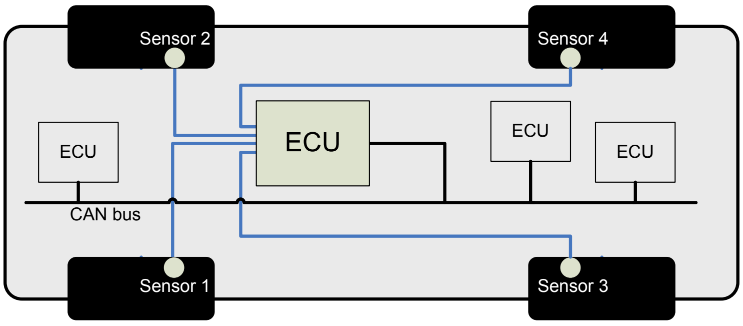


Figure 4.2.: Automotive four channel speed measurement application

is discussed in numerous works [19, 45, 115]. Accordingly, if different approaches (design processes, programming languages, hardware platforms, etc.) shall be compared, at least the differences in the previous knowledge will have to be recorded, to discuss potential impacts on the results after the experiment. This recording could be achieved by questionnaires or an initial test as proposed in [86]. Further on, an introductory course could be conducted to leverage the knowledge of the participants with respect to the approaches compared in the experiments. Furthermore, a suitable experiment design could overcome certain problems caused by different previous knowledge.

The aforementioned aspects were applied during experiment design whenever applicable. While this application is described in the sections 4.6 and 4.7, further details of the context selection are given below.

4.3.1. Experiment Tasks

Experiment tasks often do not allow general experiment results according to their restricted complexity (forced by cost and time constraints). To allow a certain level of representativity with respect to embedded systems, properties typical for embedded systems were included in our experiment tasks. These properties were defined as the need to fulfill real-time requirements, to interact with a physical environment, and to consider memory and performance constraints. Moreover, the task had to be generic enough to allow an implementation on different hardware platforms.

Experiment Tasks for Experiments 1 - 5

The majority of the experiments presented in this work (experiments 1, 2, 3, 4, 5) are based on a common basic task, which is extended for each experiment to suit individual needs. This basic task represents an automotive measurement

Table 4.1.: Sensor data processing

Signal frequency f	**Speed value s**	**max. response**	**max. failure**
$f < 2Hz$	$s = 0$	2s	10%
$2Hz \leq f < 45Hz$	$s = f * 0.0436 + 1$	1s	10%
$45Hz \leq f < 550Hz$	$s = f * 0.0436 + 1$	0.2s	10%
$550 \leq f < 5733Hz$	$s = f * 0.0436 + 1$	0.1s	5%
$f \geq 5733Hz$	$s = 251$	0.1s	5%

application with a CAN bus interface, which is depicted in Fig. 4.2 and was originally targeted to an electronic control unit (ECU) on one of the institute's own prototyping vehicles.

The task includes the measurement of four independent speed signals in combination with a CAN bus communication, which is responsible for the transmission of the speed values measured. The application revealed several challenges: Concurrent measurement of four speed signals (frequency measurement of rectangularly shaped signals), sensor data processing within the limited resources of the given hardware platform and interfacing with the external CAN controller. All these tasks had strong real-time requirements. The sensor data processing had to be performed according to Tab. 4.1 in order to integrate the four speed values into the specific CAN message. Only 8 bit were available to store each speed value making this conversion necessary. Additionally, requirements for accuracy and response time were given, depending on the actual frequency of each individual sensor signal. It was assumed that during movement of the vehicle at high velocity shorter response times are required than during movement at low velocity. Accordingly, sensor signals with high frequency demanded short response times (<100ms) while sensor signals with low frequency demanded longer measurement intervals to achieve the required accuracy (maximum failure 10%). A strategy had to be developed to achieve an optimized compromize between short response times and high accuracy for different situations (all signals with high frequency, all with low frequency, different signal input, changes in signal input, etc.). This basic task was applied in the first experiment (Exp.1).

For further experiments (Experiment 2, 3, 4) this basic task was extended by additional tasks as depicted in Fig. 4.3. For these three experiments, the additional tasks were the following:

- **Additional task 1:** Flash a specified LED each time a CAN message is sent (max. delay between the action of sending the message and flashing the LED: 10ms)

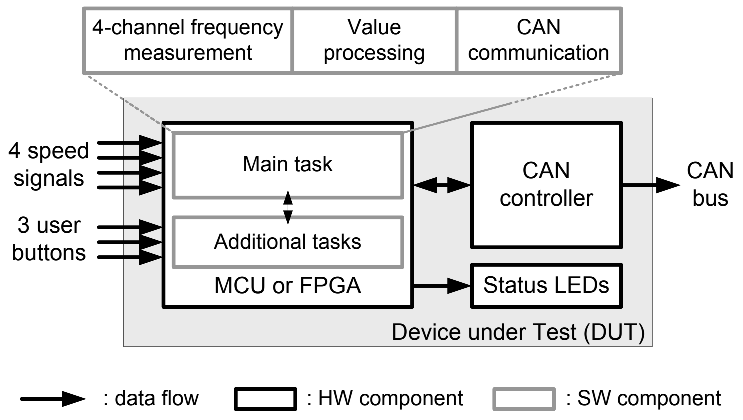


Figure 4.3.: Automotive application used for experiments

- **Additional task 2:** If a value sent is not up to date (because the corresponding measurement is not finished yet, e.g. in a situation in which one wheel is turning significantly slower than the remaining wheels), this situation will have to be marked in the 8th byte of the CAN message.
- **Additional task 3:** Display the content of the CAN controller status register after the last send request on 8 LEDs if the corresponding button (BTNA) is pressed. Otherwise, four of the LEDs shall be flashed in the frequency present at the input of the four measurement channels while the remaining four LEDs shall be flashed with 1/16 of these frequencies.
- **Additional task 4:** Send a test message every 100ms (±20ms) instead of the original speed message as long as the associated button (BTNB) is pressed. The test message includes a test counter, which has to be incremented with every message sent starting with 0 (the max. delay between the activation of the button and the sending of a test message is 500ms).
- **Additional task 5:** Send a message containing peak speed values as soon as the corresponding button (BTNC) is pressed (max. delay between the sending of a peak message and the activation of the button is 100ms). For the determination of the peak values, the interval between the last two activations of the button has to be considered.
- **Additional task 6:** Insert status information in the 5th byte of the CAN message. This status information includes the status of certain registers in the CAN controller and information concerning speed differences between the four wheels.

In another experiment (Experiment 5), a pre-designed software component representing a life beat functionality had to be reused. This software component is described in Section 4.7.1. To compensate this additional effort, additional tasks 1 and 4 were not implemented in this experiment and the amount of information included in the status (additional task 6) was reduced.

The use of a common basic task for these experiments allowed the use of a common test environment for the evaluation of these experiments and comparisons of the results obtained in the different experiments.

Experiment Task for Experiment 6

Finally, another automotive application was applied as experiment task in Experiment 6. The task originated from a real automotive application (open-topped roof control application), but was slightly reduced in complexity to suit the needs of the experiment. The interface description of the roof top control unit that had to be implemented is depicted in Fig. 4.4 in form of a context diagram.

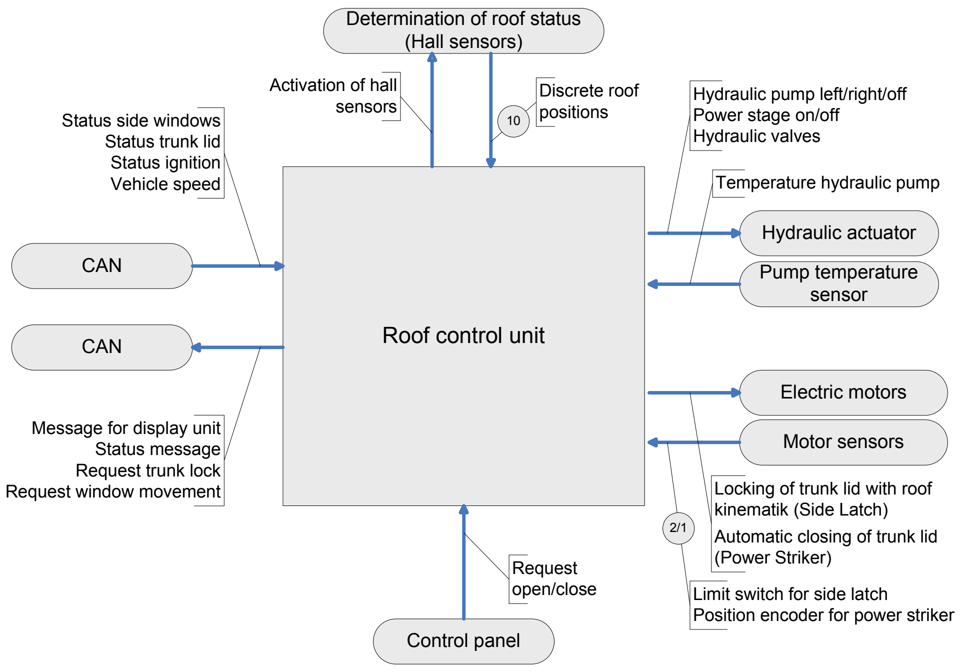


Figure 4.4.: Interface description of the roof control unit

According to a request from the user via a control panel, the roof had to be opened or closed. For this process, information from the sensors representing the status of the roof has to be evaluated and the corresponding actuators (electronic motors, hydraulic pump, hydraulic valves) have to be driven. Moreover, further information has to be retrieved via the CAN bus (e.g.: velocity of the vehicle, status of trunk and side windows) and additional actuators have to be actuated (e.g.: trunk lock and side windows). As a more detailed description of this application is not possible in this work for space reasons, the interested reader is referred to the system specification document developed for this experiment [102].

4.3.2. Selection of Hardware Platforms

Common embedded hardware platforms have been described in Section 2.1. As already mentioned in this context, the implementation of a task on a programmable logic device differs a lot from the implementation of the same task on a CPU based system. For the experiments 1-5, MCUs and FPGAs[1] have been selected as popular representatives of these two families of devices. As a microcontroller, an *Atmel ATmega16*[2] controller has been used while a *Xilinx Spartan3*[3] was used as an FPGA (respective a *Xilinx Coolrunner2*[4] as a CPLD in the first experiment). The development took place in the development environments provided by Atmel (AVR studio) and Xilinx (Integrated Software Environment (ISE)).

Certain real-time functionalities of embedded systems can be already simulated without the actual target hardware. However, usually neither a sufficient functionality of the embedded system's interface nor a simulation of external components (e.g. communication controllers or external memories) is available in this approach. Our experiences showed that working with a simulated embedded device makes the development for the participant more complicated. Therefore, the implementation was conducted with existing evaluation boards in all cases. Additionally, another printed circuit board was developed and produced to allow the CAN communication. This board included a CAN controller (Philips SJA1000, see [82] for data sheet) and an interface for the MCU development board as well as one for the CPLD/FPGA development boards.

For the last experiment, the selection of the hardware platforms was based on an investigation on potential future trends in automotive electronics, which has been conducted with partners from the automotive domain. As a result, a

[1]actually, a CPLD was used in the first experiment, but differences between CPLDs and FPGAs are low from the software developers point of view as described in Section 2.1

[2]Atmel ATmega16 @ 6MHz, data sheet: [2]

[3]Xilinx Spartan-3 XC3S200 @ 50MHz (typically reduced to 1MHz internally), data sheet: [128]

[4]Xilinx CoolrunnerII XC2C256 @ 1,8432MHz, data sheet: [127]

dual-core microcontroller was picked as first hardware platform, while a combination of MCU and FPGA was selected as second platform. Only few dual-core microcontrollers were available at the time of preparation of this experiment. The device chosen is a microcontroller from the company DUAL-CORE [5]. Although the device is a little overdimensioned for the experiment application (16-bit CPUs and smaller memories would have been sufficient), it is considered as useful for the experiment. The FPGA used for the second hardware platform was also a Xilinx Spartan-3 (details see above) while the MCU was a *Freescale HCS12*[6] microcontroller. The reason for not using the ATMEL ATmega16 applied before was the following. First, it was assumed that a 16bit microcontroller was required for the application selected for this experiment. Moreover, both hardware platforms were required to include an internal CAN controller, which was not the case in the ATmega16 controller.

Selection of Languages

According to [117], the choice of an appropriate programming language is of great significance in safety-critical systems in order to prevent faults in the software. In this regard, further discussions of the suitability of different programming languages can be found, e.g. in [23] and [41]. However, it has to be noted that not every language is available or even suitable for every hardware platform. Therefore, only suitable and available languages could be considered for this work.

Real-time applications executed on MCUs are nowadays most often implemented in the C language (especially in the automotive domain) while in case of strict resource constraints, the assembly language is still present in these systems. Otherwise, the amount of C code generated automatically from models is increasing (model based development). However, the code generated automatically comes usually with a certain overhead in size and execution time. Further on, drivers and other functions related closely to the target hardware still have to be programmed manually. Therefore, the C language has been chosen for the implementation on MCUs in case of all of our experiments. Another option, which is particularly suitable for safety-critical systems, would have been the language ADA. However, for the MCUs considered for the experiment, no suitable ADA compiler was available.

In case of the FPGAs, the two main languages are VHDL and Verilog while the use of the language SystemC is increasing. Moreover, model based development is also possible for FPGAs although current approaches mainly aim at sig-

[5]DualCore DCIC9907 @ 16MHz, internal operating frequency is derived by the DCIC9907-internal PLL, which is typically 128 MHz, data sheet: [26]

[6]Freescale MC9S12DP512 @ 16MHz, data sheet: [34]

nal processing tasks. Before our first experiment, the suitability of the different hardware description languages for safety-critical applications was investigated. In conclusion, advantages of using VHDL over Verilog were identified according to aspects as stronger typing, while no benefits of the language SystemC with respect to safety aspects could be identified. Therefore, VHDL has been selected for the software development for FPGAs. Moreover, an additional graphical representation (*schematics*) was applied, but only to join different VHDL blocks together.

Selection of Hardware Abstraction

The automotive application used in the first five experiments was characterized by limited resources, strong real-time requirements, and a medium complexity of the task. Therefore, no specific hardware abstraction, as a real-time operating system was applied in these experiments. Moreover, it has to be noted that hardware abstraction in form of an operating system will only be possible on FPGAs if processors are implemented within the FPGA. Furthermore, it is stated in [117] that even simple operating systems are quite complex so that the use of them is not acceptable in highly critical applications.

On the contrary, the real-time operating system OSEK was chosen for Experiment 6 according to the higher complexity of the automotive task applied in this experiment. However, no operating system was used for the implementation of the safety function.

4.3.3. Selection of Subjects

The selection of the experiment participants as well as their number has impacts on the results when generalizing [126]. As argued above, an academic setting was chosen, which limits the possibilities of selecting the participants.

In case of the experiments, which took place in lab courses (experiments 1, 2, 4, and 5, see Fig. 4.1), the options for selecting the participants were limited as the number of applicants exceeded the maximum number of lab course participants only by two or three students. In this case, a selection was made according to the highest previous knowledge. The number of participants in these experiments ranged from 19-26 students forming 12 teams (see Fig. 4.1 for details).

According to limited educational aspects, the review of software versions (Experiment 3) was conducted by student assistants. All student assistants, which were familiar with the corresponding programming languages and also present at the institute at the time of the experiment, were asked to participate. As a result, four student assistants participated in this experiment.

Finally, due to the higher complexity of the task in Experiment 6, a conduction of this experiment in the setting of a regular lab course was not possible. To overcome this problem, the task was implemented on the two different hardware platforms within the scope of two diploma theses. To reduce the implementation effort of this work, we employed two additional student assistants so that in total four students participated in this experiment forming two teams.

4.4. Setup of Hypotheses

Based on the experiment definition given in Section 4.2, the hypotheses had to be formalized, which is described in the following for the six aspects investigated.

H1: Impact of Diverse Hardware Platforms on the Diversity of Software Faults

The main idea behind these investigations was the assumption that the application of different hardware platforms is resulting in software with diverse failure behavior (see Section 3.4.1). The first hypothesis guiding this investigation is the following:

> **H1a:** Failures in the software versions written by different teams independently for different hardware platforms (CPLDs/FPGAs and MCUs) are not stochastically independent.

This hypothesis is tested by falsification of the opposite assumption called *null-hypothesis*. In this case, the null-hypothesis states that the probability of making independent failures on both platforms is calculated by the product of the single failure probabilities. The according formal expression is:

$$H_0 1a : p(F_{MCU} \cap F_{CPLD}) = p(F_{MCU}) \cdot p(F_{CPLD})$$

with p being the probability and F the failure event of the software implemented on CPLDs respectively MCUs.

However, it is expected that the number of dependent failures between different hardware platforms (heterogeneous pairs) is decreased in comparison to identical hardware platforms (homogeneous pairs). This aspect is expressed by the following sub hypothesis:

> **H1b:** The probability of dependent failures is higher in case of two identical hardware platforms than in case of two diverse hardware platforms.

The corresponding null-hypothesis is expressed by the following formula:

$$H_0 1b : Md[p(F_{pair_{heterogeneous}})] \geq Md[p(F_{pair_{homogeneous}})]$$

with Md being the median, p the probability and F the failure event of the corresponding pair of hardware platforms. A heterogeneous pair is comprised of two diverse hardware platforms while a homogeneous pair is comprised of two identical hardware platforms.

H2: Impact of Hardware Platforms on the Encapsulation of Real-Time Tasks

According to the parallel structure of FPGAs, it was assumed that the encapsulation of real-time software functions benefits of this structure (see Section 3.4.2). For this investigation, subfunctions were identified within the application and potential side effects were examined. It is assumed that in case of the FPGAs, changes in input signals fed into one subcomponent (a subcomponent implements a subfunction) lead to less changes in the timing of another subcomponent than in case of the MCUs. Major subfunctions identified in the experiment task are the signal measurement, the signal processing and the communication (see Fig. 4.3). This first aspect is expressed by the following hypothesis.

> **H2a:** In case of the FPGAs, changes in the frequencies fed into the devices tested (signal measurement and processing function) lead to less changes in the timing of the CAN communication (communication function) than in case of MCUs.

The corresponding null-hypothesis is expressed by the following formula:

$$H_0 2a : \frac{\sum_1^N |\Delta T_{FPGA}(f_L, f_H, V_f)|}{N} \geq \frac{\sum_1^N |\Delta T_{MCU}(f_L, f_H, V_f)|}{N}$$
$$\Delta T_x(f_L, f_H, V_f) = T(f_H, V_f) - T(f_L, V_f), x \in \{MCU, FPGA\}$$

with N being the number of versions considered for evaluation and ΔT the variation of the CAN communication interval T caused by changes in the input frequencies f. These changes are represented by one input constellation at the lower end of the input range specified ($f_L \approx 6\%$ of f_{max})[7] and one input constellation at the upper end ($f_H \approx 100\%$ of f_{max}). Finally, the final software versions V_f were used for this test[8].

[7]No upper limit for the input frequencies was specified. Therefore, the highest frequency, which was specified explicitly (6kHz), is considered as f_{max}.

[8]Note: final version = version including main task and all additional tasks; main version = version including the main task only.

Moreover, it was assumed that, in case of the FPGAs, the integration of additional tasks (see Section 4.3.1) would lead to less side effects with respect to existing tasks, namely the communication function as well as the signal measurement and processing functions. This assumption is expressed in the following sub hypotheses:

> **H2b:** In case of the FPGAs, the integration of the additional tasks lead to less changes in the *timing* of the CAN communication than in case of the MCUs.

The corresponding null-hypothesis is expressed by the following formula:

$$H_0 2b : \frac{\sum_1^N |\Delta T_{FPGA}(V_f, V_m, f_H)|}{N} \geq \frac{\sum_1^N |\Delta T_{MCU}(V_f, V_m, f_H)|}{N}$$

$$\Delta T_x(V_f, V_m, f_H) = T(V_{final_x}, f_H) - T(V_{main_x}, f_H), x \in \{MCU, FPGA\}$$

with N being the number of versions considered for evaluation and ΔT the variation of the CAN communication interval T caused by changes from the main version V_m to the final version V_f. Moreover, $f_H \approx f_{max}$ represents the input frequency determined for the test of this hypothesis.

The following hypothesis targets the integration of the additional tasks without considering any user interaction (via buttons) possible in case of the additional tasks.

> **H2c:** In case of the FPGAs, the integration of the additional tasks lead to less additional failures in the *contents* of the CAN messages than in case of the MCUs.

The corresponding null-hypothesis is expressed by the following formula:

$$H_0 2c : Md[\Delta\lambda_{FPGA}(V_f, V_m, f_{TB})] \geq Md[\Delta\lambda_{MCU}(V_f, V_m, f_{TB})]$$

$$\Delta\lambda_x(V_f, V_m, f_{TB}) = \lambda(V_{f_x}, f_{TB}) - \lambda(V_{m_x}, f_{TB}), x \in \{MCU, FPGA\}$$

with Md being the median and $\Delta\lambda$ the variation in the failure rate λ caused by the change from a main version V_m to a final version V_f, which includes the additional tasks. As input frequencies f_{TB}, the available test benches are applied.

The last hypothesis is expressing the impact of the external triggering of the execution of the additional tasks via three user buttons:

> **H2d:** In case of the FPGAs, the execution of those additional tasks activated via user buttons leads to less additional failures in the *contents* of the CAN messages than in case of the MCUs.

The null-hypothesis corresponding to this hypothesis is expressed by the following formula:

$$H_0 2d : N[F_{FPGA}(V_f, f_{TB_{button}})] \geq N[F_{MCU}(V_f, f_{TB_{button}})]$$

with N being the number of versions, which contain failures F in their output according to the activation of additional tasks. The activation is achieved by a special test bench TB_{button}. Of course, the final versions V_f were used for this hypothesis test as only these versions include the additional tasks.

H3: Impact of Hardware Platforms on Software Review

The review of real-time functions requiring fast response times is challenging on MCUs. Therefore, it is assumed that the action of review might profit from the parallel structure of FPGAs (see Section 3.4.3). First, it is assumed that the review on FPGAs reveals more timing faults than on MCUs. This aspect is expressed by the succeeding hypothesis:

> **H3a:** In case of the FPGA versions, review results show higher compliance with test results than in case of the MCU versions, with the tests performed on basis of the review results.

The corresponding null-hypothesis is expressed by the following formula:

$$H_0 3a : \sum_1^i \sum_1^N \sum_1^n \Psi_{FPGA}(R_i, V_{f_N}, S_n) \leq \sum_1^i \sum_1^N \sum_1^n \Psi_{MCU}(R_i, V_{f_N}, S_n)$$

$$n \in \{1, 2, 3\}, i \in \{1, 2, 3\}$$

with N being the number of considered final versions V_f and Ψ the number of review results that comply with the corresponding test results. In this context, each version is reviewed by different reviewers R_i regarding different review scenarios S_n. The number of review scenarios depended on the individual reviewer as discussed in Section 5.3.

Furthermore, it is expected that reviewers consider the review of software written for FPGAs as more pleasant as the review of the same functions implemented in MCU software. This expectation is expressed by the following hypothesis:

> **H3b:** In case of the FPGA versions, the reviewability is rated better (on average) than in case of the MCU versions.

The corresponding null-hypothesis is expressed by the following formula:

$$H_0 3b : \frac{\sum_1^i \sum_1^N \Re_{FPGA}(V_{f_N}, R_i)}{i \cdot N} \geq \frac{\sum_1^i \sum_1^N \Re_{MCU}(V_{f_N}, R_i)}{i \cdot N}$$
$$i \in \{1, 2, 3\}$$

with N being the number of considered final versions V_f and $\Re$ the normalized grades of reviewability given by each reviewer R_i. For the results of each reviewer, the grades for the reviewability ($\Re = 1$: good reviewability,...,$\Re = 5$: review not possible) are divided by the average grading of the corresponding reviewer for normalization.

H4: Software Diversity vs. Fault Removal

It has been discussed before that instead of applying N-version programming, the effort for implementing the second version could be spent on verifying the first version (see Section 3.4.4). It is assumed that this approach would discover dependent failures as those observed in the majority of the previous experiments 1 and 2. The failure which occurred most often, was the faulty behavior in case of fast changes in the input values. Therefore, it is expected that the corresponding faults will be also present in the majority of the versions developed in this experiment. The identification of the faults should be investigated, which is expressed in the following hypothesis:

> **H4a:** The failure according to fast changes in the input frequencies is identified by at least 3/4 of the experiment teams by review or testing

The corresponding null-hypothesis is expressed by the following inequation:

$$H_0 4a : \frac{N(V_f(F_{detected}))}{N(V_f(F_{detected})) + N(V_f(F_{undetected}))} < \frac{3}{4}$$
$$F_{detected} = F_{detected_{Test}} + F_{detected_{Review}} + F_{detected_{Test \wedge Review}}$$

with N being the number of final versions V_f and F the failure according to fast changes in the input frequencies, which could be either *detected* or *undetected* for each version. Moreover, fast changes are quantified as a difference in two consecutive frequency values of more than 200Hz.

The second aspect investigated refers to known problems in the specification. Four statements in the specification have been identified as ambiguous in previous implementations. As these statements are a source of dependent failures, it is of interest in how many cases these ambiguous statements are identified. This aspect is expressed by the following hypothesis:

H4b: The four known ambiguous statements in the specification are identified by at least 3/4 of the teams during their review and test activities. In this context, not all four statements have to be detected by one team.

The corresponding null-hypothesis is expressed by the following inequation:

$$H_04b : \frac{N(V_f(AS_{i_{detected}}))}{N(V_f(AS_{i_{detected}})) + N(V_f(AS_{i_{undetected}}))} < \frac{3}{4}, i \in \{1, 2, 3, 4\}$$

with N being the number of final versions V_f and AS_i the four known ambiguous statements in the specification. A description of these four statements can be found in Fig. 5.5 in Section 5.4.

Finally, it is expected that a fault removal approach based of review and testing is leading to less failures than the NVP approach investigated in the experiments 1 and 2. This assumption is described by the following hypothesis:

H4c: If a comparable effort is put into the verification and correction than into the second version of an NVP approach, the first approach leads to less failures. The verification in this case is based on review and testing

The corresponding null-hypothesis is expressed by the following formula:

$$H_04c : Md[\lambda_{NVP}(V_f, f_{TB})] \leq Md[\lambda_{V\&C}(V_f, f_{TB})]$$

with Md being the median and λ the failure rate of the corresponding approach. The NVP approach is based on two final versions V_f developed independently. If both versions fail, this approach will be considered as failed. The approach of verification and correction ($V\&C$) is based on a single final version V_f, which has been reviewed, tested and corrected. The input values f_{TB} used for this test are based on available test benches (TB).

H5: Impact of Hardware Platforms on the Reuse of Real-Time Functions

The parallel nature of FPGAs is seen as a potential benefit for the reuse of software, especially in the case of real-time functions (see Section 3.4.5). It is assumed that reused software components suffer from less side effects than in case of MCUs. This assumption is expressed by the following two hypotheses:

H5a: In case of the FPGA versions, the execution of the functions implemented beside the reused function on the same device, lead to less *content* failures in the reused function (LB component) than in case of the MCU versions.

The corresponding null-hypothesis is expressed by the following formula:

$$H_0 5a : N[F_{LB_{FPGA}}(V_f, f_M, f_{LB})] \geq N[F_{LB_{MCU}}(V_f, f_M, f_{LB})]$$

with N being the number of versions in which failures F_{LB} are observed in the content of the life beat function's output. For this test, the final versions V_f are applied. As this test represents a stress test, values up to three times higher than the input value explicitly specified are used as test input f_M for the four channel speed measurement function. Moreover, the input for the life beat function is $f_{LB} = \{0Hz, 10Hz\}$.

> **H5b:** In case of the FPGA versions, the execution of the functions implemented beside the reused function on the same device, lead to less *timing* failures in the reused function (LB component) than in case of the MCU versions.

The corresponding null-hypothesis is expressed by the following formula:

$$H_0 5b : \frac{\sum_1^N |\Delta T_{LB_{FPGA}}(V_f, f_M, f_{LB})|}{N} \geq \frac{\sum_1^N |\Delta T_{LB_{MCU}}(V_f, f_M, f_{LB})|}{N}$$

with N being the number of considered final versions V_f and ΔT the variation of the LB send interval T caused by integration of the LB component in the new context. Moreover, $f_M \approx 50\%$ of f_{max} represents the input frequency determined as input for the measurement channels for the test of this hypothesis. Additionally, the input for the life beat function is $f_{LB} = 0Hz$ in this case.

On the other hand, it was expected that especially MCU versions suffer from side effects with the integrated LB component. This aspect is expressed in the following hypothesis:

> **H5c:** In case of the FPGA versions, the inclusion of the function to be reused (LB component), leads to less failures in the remaining functions implemented on the device than in case of the MCU versions.

The corresponding null-hypothesis is expressed by the following formula:

$$H_0 5c : N[F_{context_{FPGA}}(V_f, f_M, f_{LB})] \geq N[F_{context_{MCU}}(V_f, f_M, f_{LB})]$$

with N being the number of versions in which failures $F_{context}$ are observed in functions embedding the reused life beat function. For this test, final version V_f are applied. Further on, the test frequency $f_M \approx 50\%$ of f_{max} represents the input frequency for the four measurement channels. Finally, the input frequency for the LB function f_{LB} ranges from 0Hz to 100kHz.

H6: Impact of Hardware Platforms on the Development according to ISO26262

In this investigation, two types of hardware platforms were examined with respect to safety and reliability issues. It had been expected that the two platforms differ with respect to these aspects as described in Section 3.4.6. First, it has been assumed that the effort of developing a system according to the requirements of the ISO26262 for an ASILC system differs for the two hardware platforms[9]. This assumption is expressed by this hypothesis:

> **H6a:** The effort (development time and occurring problems) for developing the given safety-critical application classified as ASIL C according to ISO26262 is not equal on both hardware platforms.

The corresponding null-hypothesis is expressed by the following two formulas:

$$H_0 6a : (T_{HW_1} \approx T_{HW_2}) \wedge (P_{HW_1} \approx P_{HW_2}), P = \sum_{1}^{N} C_N$$

with T being the development time for implementing the application according to ISO26262 on the corresponding hardware platforms. Moreover, P are the development problems identified on the two hardware platforms. The development problems are defined as the sum of the criticality C of all N problems, which occur during the development for this hardware platform. In this context, the criticality of each problem is rated from 0 (uncritical) to 5 (very critical). Finally, the notation of $\approx$ allows a difference of 10% of the larger value.

Moreover, differences in the resulting architectures with respect to reliability and modifiability are expected. As no concrete expectations were present at the time of hypothesis definition, no formal hypotheses were formulated for these issues. Nevertheless, these aspects are evaluated in Section 6.6.

4.5. Variable Selection

For the variable selection, it has to be differentiated between independent variables (those variables, which can be controlled and changed in the experiment) and dependent variables (the measurement of those variables determines the effect of the treatment) [126].

[9] The requirements considered for our investigations excluded process requirements (e.g. management and documentation activities), as no differences are expected between the two platforms in this field.

The most important independent variable in the experiments 1, 2, 3, 5, and 6 was the type of hardware platform which was applied in the implementation. In the remaining experiment (Experiment 4), the most important independent variable is the type of software fault handling (review and testing in contrast to NVP) applied.

The dependent variable in the majority of the experiments is the number and the type of failures in the developed systems (experiments 1, 2, 4, and 5) and is derived directly from the corresponding hypotheses.

In the third experiment, several dependent variables were selected. A first variable selected is the time needed for the review as it is comparatively easy to measure and might be used to determine the review effort. Moreover, the reviewability, which represents another dependent variable, had to be graded by each reviewer. The third dependent variable in this experiment is the number of faults identified correctly by review.

Furthermore, the success of the review and test activities in Experiment 4 was not only evaluated by the remaining faults in the system. Therefore, further dependent variables in this experiment are the type of faults identified by review and testing respectively.

The most important dependent variable in Experiment 6 is the development effort for implementing the specification following the requirements of the safety standard ISO26262. This variable was measured by the time needed for implementation and problems that occurred during development. Moreover, certain properties of the resulting implementations were investigated, which is described in Section 5.6.

Further factors in the experiments should be kept constant in each experiment and can be seen as further independent variables. They have to be controlled, if it cannot be guaranteed that they do not influence the dependent variables. This control could be achieved by keeping these variables constant (e.g. the time for implementing the experiment task) or by using statistical measures (e.g. previous knowledge of participant is controlled by a high number of randomly picked developers). During our experiments, we determined specific challenges and solutions with respect to the control of variables, which are presented in Section 4.7.2.

4.6. Experiment Designs

The design of the experiment has got a huge impact on the conduction of the experiment as well as on the quality of the experiment results. Therefore, several aspects have to be considered during experiment design. The designs of the experiments we conducted for this work are described in the following sections.

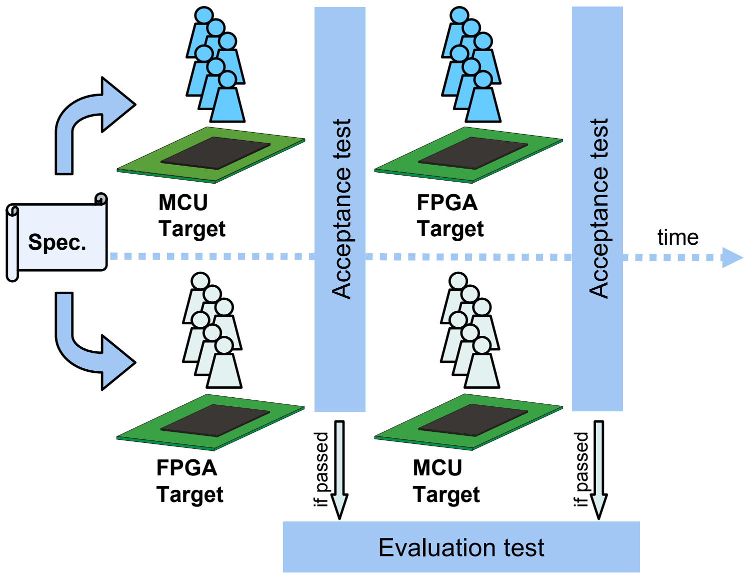


Figure 4.5.: Design of Experiment 1 and 2

4.6.1. Experiments 1 and 2

The experiments 1 and 2 took place in lab courses, as this setting allowed experiments with comparatively high numbers of participants. For these experiments, we applied a *paired comparison design* (see [126]). In this design, each subject (development teams in our case) applies both treatments on the same object (specification of experiment task) to improve the precision of the experiment. To reduce the impact of the order in which the subjects apply the treatments, the order is assigned randomly.

For the experiments 1 and 2, the resulting experiment design is depicted in Fig. 4.5. Both experiments took place in lab courses with 12 teams of development teams. We established two treatment groups and the teams of both groups received a common specification of the experiment task described in 4.3.1. Teams in the first treatment group had to implement this specification on a MCU target hardware platform while teams of the second group had to implement the same specification on an CPLD/FPGA[10] target. After the implementation was finished, all teams had to pass an acceptance test, which was applied to guarantee a certain minimum level of quality of all versions. Further details of this test are discussed in Section 4.6.6. If a team did not pass this acceptance test, it had the chance to correct the version and try another accep-

[10]Experiment 1: CPLD, Experiment 2: FPGA

tance test. Only those versions that passed this acceptance test were considered for the later evaluation. After this first half of the experiment, the hardware platforms were exchanged. The teams, which had programmed the MCU target before, now had to program the FPGA and vice versa. At the end, the same acceptance test had to be passed and again only those versions, which passed this test, were considered for evaluation.

According to this experiment design, each participant had to apply both approaches in an arbitrary order. This structure allowed to consider the different skills and previous knowledge of the participants. However, when applying the *paired comparison design*, potential learning effects from the first implementation on the second one have to be considered. Therefore, versions implemented in the first half of the experiment were compared with those implemented in the second half in order to investigate potential learning effects.

4.6.2. Experiment 3

In case of this experiment, we took data (software versions developed on different hardware platforms) from the second experiment and examined their reviewability. As the process of reviewing takes some time, it was not possible to conduct this experiment in the lab courses used for the development of the original software. Therefore, student assistants familiar with both hardware platforms conducted the review of 6 MCU and 6 FPGA versions each. These versions were picked randomly from a total of 22 versions.

Table 4.2.: Design of Experiment 3

Subjects	**Review Objects (in given order)**
Reviewer 1	M-2, F-2, F-5, M-5, M-6, F-6, F-7, M-7, M-8, F-8, F-9, M-9
Reviewer 2	F-9, M-9, M-8, F-8, F-7, M-7, M-6, F-6, F-5, M-5, M-2, F-2
Reviewer 3	F-9, M-9, M-8, F-8, F-7, M-7, M-6, F-6, F-5, M-5, M-2, F-2

Note: M-2 = MCU version 2, F-5 = FPGA version 5

As this investigation is based on outcomes of a previous experiment, it is not suited for a classical experiment design as the one applied in the experiment described above. The experiment design applied for this investigation is described in Tab.4.2. Accordingly, the three reviewers[11] had to review the 12 versions in the given order. The different orders were chosen to decrease potential impacts of this factor.

[11]for organizational reasons, the third review had to be split on two different student assistants.

Moreover, the reviewers were guided by a given review report form and a given review instruction (see Section 4.7.2). In the review report form, the following aspects had to be filled out by each reviewer:

- requirements tested + results
- requirements not tested + reasons
- scenarios identified, which could lead to problems + results
- quality of version tested (range 1..5, reviewer's subjective opinion)
- final remarks concerning implementation and specification
- grading of reviewability (range 1..5, reviewer's subjective opinion)

The review instruction only covered the main ideas of the review process, as for example that the reviewers were not allowed to execute or modify the inspected code. The review technique applied did not include any formal aspects as e.g. proposed by Fangan [30]. However, the aim of this review was to evaluate the code and not to find as much as possible faults. Therefore, this approach is considered as suitable for this experiment.

4.6.3. Experiment 4

Guided by hypothesis H4, the effect of review and test activities was investigated in this experiment, which also took place in a lab course. Therefore, a design was chosen as depicted in Fig. 4.6.

Each team had to program the same task as used in Experiment 2. However, only the microcontroller hardware was used for implementation, which took part in the first half of the lab course. The second half of the lab course was used for review and testing, which was organized as follows: Randomly, each team was assigned a version for review and another version for testing. The versions were anonymized to avoid interaction between the implementation and the verification teams and it was assured that no team checked their own version. Half of the teams started with review while the other half started with testing in order to mask out effects of execution order and to reduce the number of test equipment needed for the experiment. After three weeks (three appointments with 3 hours each) the teams changed from testing to review and vice versa. In the last two weeks, all teams had the chance to improve their own version on basis of the review and test reports. In the following, the test and review process used in the experiment will be presented briefly.

For testing, all six test teams were equipped with an own automated test environment, similar to the one used for evaluation (see Section 4.7.4). Moreover,

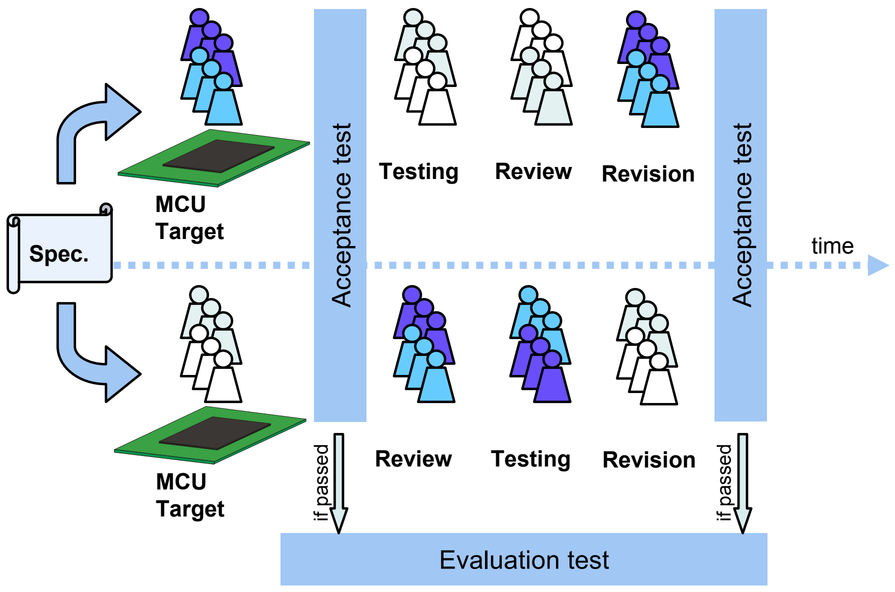


Figure 4.6.: Design of Experiment 4

each team received a documentation for the test environment, the machine code for testing, and an empty test report form. The following aspects had to be filled out in every test report:

- requirements tested + results
- requirements not tested + reasons
- scenarios identified, which could lead to problems + results
- quality of version tested (range 1..5, reviewer's subjective opinion)
- final remarks concerning implementation and specification

This given form of the report helped the students during test case creation and eased the later analysis of all test reports for the evaluation.

As in the case of testing, every review team received an empty review report form, a short review instruction and the source code to review. Additionally they received the corresponding documentation, which should help them to understand the code if necessary. The review report form was identical to the one used in Experiment 3. Therefore, it covered the same aspects as the test report and an additional grading of the reviewability.

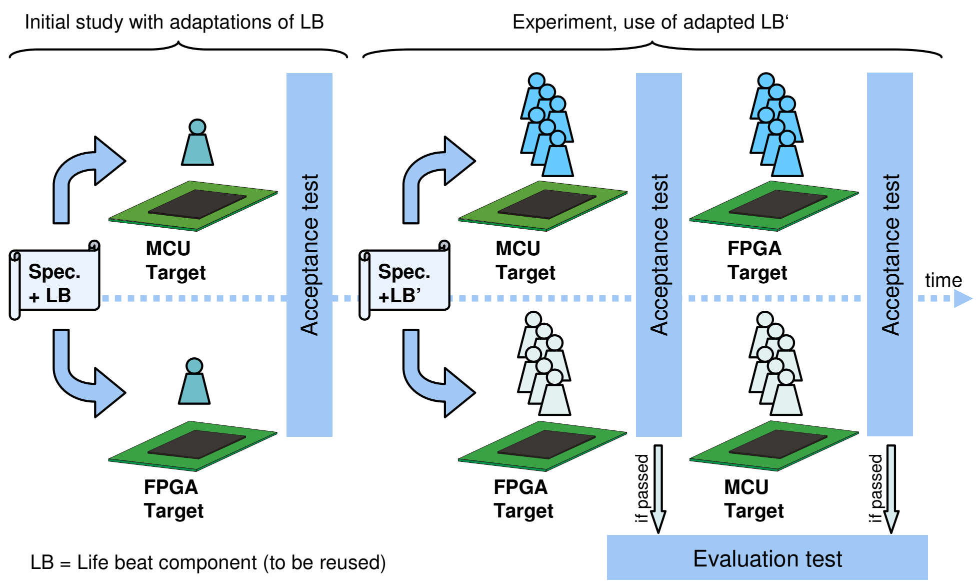


Figure 4.7.: Design of Experiment 5

Moreover, since the same experiment task was applied in experiment two and three, a comparison between the effects of N-version programming and those of review and testing is possible.

4.6.4. Experiment 5

In accordance with the experiments 1 and 2, a common specification was implemented on an MCU target platform as well as on an FPGA target. However, a specific software component, originally designed for another application, was reused in this experiment in case of both platforms.

The design of the evaluation is based on two steps. In a first step, we conducted an initial study in which the aforementioned software component was reused by a student assistant in the new application to investigate whether this reuse is feasible (see left part of Fig. 4.7). In this step, the software component could be adapted if needed while all modifications were documented.

In a second step, this adapted software component was provided to the participants of a lab course. During this lab course, the component was reused (without further adaptation) to allow an evaluation of the reuse on the two different hardware platforms (see right part of Fig. 4.7). As introduced in Section 4.6.1, we also applied a *paired comparison design* in this experiment (both treatments were applied to all teams in random order) to improve the precision of the experiment.

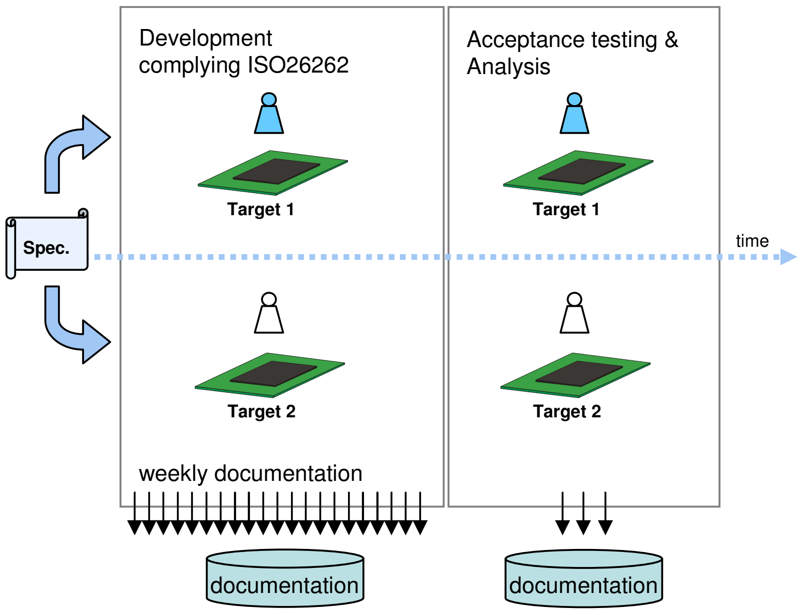


Figure 4.8.: Experiment design

4.6.5. Experiment 6

The investigations with regard to Experiment 6 required the implementation of a comparatively complex automotive application. Moreover, requirements by the safety standard ISO26262 [48] had to be considered. Therefore, this experiment required development time and previous knowledge, which could not be expected from students participating in a lab course. As mentioned previously, this experiment took place in the context of two diploma theses resulting in a simplified experiment design depicted in Fig. 4.8.

As determined in the variable selection (see 4.5), the dependent variable in this experiment was the development effort. Therefore, design problems and the development progress were documented in weekly meetings as can be seen in the left part of Fig. 4.8. Next, the student assistants tested the implemented versions by using acceptance criteria given in the specification. The discovered failures were documented and the corresponding development teams had the chance to improve their implementations. Moreover, the developed versions were analyzed with respect to further properties described in Section 4.4 and then tested with the test environment developed for this purpose (right part of Fig. 4.8).

The low number of subjects (one development team for each treatment) can have impacts on the validity of the results, which is discussed in Section 6.6.4.

4.6.6. Testing Issues

To assure a certain minimum quality of the versions created in the experiments, acceptance tests were applied in the experiments 1, 2, 4, 5 and 6. Our experiences with these acceptance tests show a certain dilemma. If the acceptance test is too weak, the later evaluation might be impossible because some versions simply do not work as intended. If the acceptance test is too strict on the other hand, the evaluation test might be somehow useless as new faults may not be found during evaluation. Further on, the determination of the acceptance test can have unintended effects on the experiment results (with one acceptance test one might receive a different result than with another acceptance test). Acceptance tests are commonly used, but follow different approaches. An acceptance test generated randomly was used in [53] and it is stated in [112]: "The acceptance test was not, and was not intended to be, a basis for quality assessment of the code, but rather was a test of whether all major portions of the code were present in some operable form". Otherwise, a test case representing one of four functional profiles of the actual application was used in [64] in a two-step acceptance test and failures identified in this acceptance tests were used in the later evaluation. Based on our experiences, we propose the usage of a comparatively strict acceptance test to guarantee a sufficient quality of the experiment outcomes and to use the information of all failed acceptance tests for the later evaluation (as applied in [64]) to increase the independence of the evaluation results from the acceptance test used. This approach, which is depicted in Fig. 4.9 in form of a flowchart, requires to document date and time as well as the reason of failing in case of each failed acceptance test. As this approach has been developed during our work, it was applied in case of Experiment 5 and 6 only (details in Section 5.5 and 5.6).

Furthermore, *design for testability* could be an important issue in the experiment design. As soon as the overall function fails, it often would be desirable to determine, which of the subfunctions work correctly and which do not. One way would be to evaluate each version manually and to include debug commands in the code. An approach that turned out to be effective in our experiments, was to include test functionality within the actual specification used in the experiment (e.g.: write intermediate results to the output if a certain test variable is set to true). While this approach implies a knowledge of the subfunctions to be tested at the time the specification is written, it simplifies the testing of these subfunctions in the later evaluation. Moreover, this approach is especially useful in the field of embedded systems, in which the access to internal values at run-time is limited or not possible. Further testing issues typical for embedded systems can be found in Section 4.7.4 describing the test environments.

Further on, testing is important during the development as part of debugging

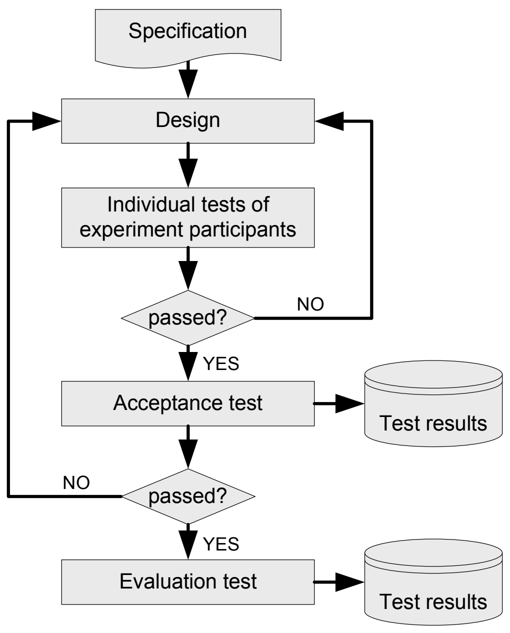


Figure 4.9.: Test flow

and verification activities. These activities require in some cases a simulation of the embedded system's environment (embedding system). The simulation could be achieved by comparatively simple measures in the experiments 1, 2, 4, and 5 (details in Section 5), while a simulation environment had to be developed for Experiment 6. Finally, debugging facilities could ease the development process and should be provided to the experiment participants for this reason.

4.7. Instrumentation

In accordance with [126], instruments for an experiment include objects, measurements and guidelines. These instruments, which are described in the following sections, had to be developed before the experiment execution. Moreover, the measurement of variables required dedicated measurement equipment in case of most of our experiments. These environments are described in Section 4.7.4 followed by the corresponding test case generation described in Section 4.7.5.

4.7.1. Experiment Objects

The experiment task specification represents the *experiment object* in all experiments, which involve implementation activities (experiments 1, 2, 4, 5 and 6). This task description was given in written form to the students and is described briefly in Section 4.3.1. Moreover, the experiment object in case of Experiment 3 is the code developed in a previous experiment. In the same manner, code developed in the first half of Experiment 4 is also an experiment object for the investigations performed in the second half of this experiment.

Beside the task specification, a further experiment object in Experiment 5 is the software component, which had to be reused. This component was developed by a student assistant for both hardware platforms with another context in mind. The original application combined a safety-critical life beat (LB) function, which is continuously checking the liveness of another device, with a function checking the status of several buttons (BTNs) as depicted in Fig. 4.10. In detail, the LB function had to check for a correct length of the high and low phase of the given LB signal. A correct LB signal was defined by a frequency of 10Hz and a length of the high phase equal to the length of the low phase. Both functions use an external CAN controller for communicating their status via the CAN bus. The initialization of the CAN controller and the send requests for CAN messages is conducted outside the LB-function. Since the LB functionality is considered as safety-critical, the function can send their own CAN messages if no CAN messages have been sent for a given time. To assure the applicability of the reuse in the new context (see Fig. 4.11), the reuse was tested by a student assistant in a second step in which also minor modifications were applied to both components (details in Section 5.5.1).

4.7.2. Measurements

Several measurements were applied for the experiments conducted for this thesis. These include measurements to control variables, to measure variables and to determine the statistical relevance of the measured results.

Control of Variables

With regard to the control of the variables, all participants should receive the complete information concerning their task in the experiment in a written specification. This procedure is important to avoid influences by the experiment supervisor by answering individual questions. Moreover, the implementation of the task by a test team prior to the actual experiment is suitable to verify the completeness and comprehensibility of this common specification. However, it turned out that different participants may have different problems with the

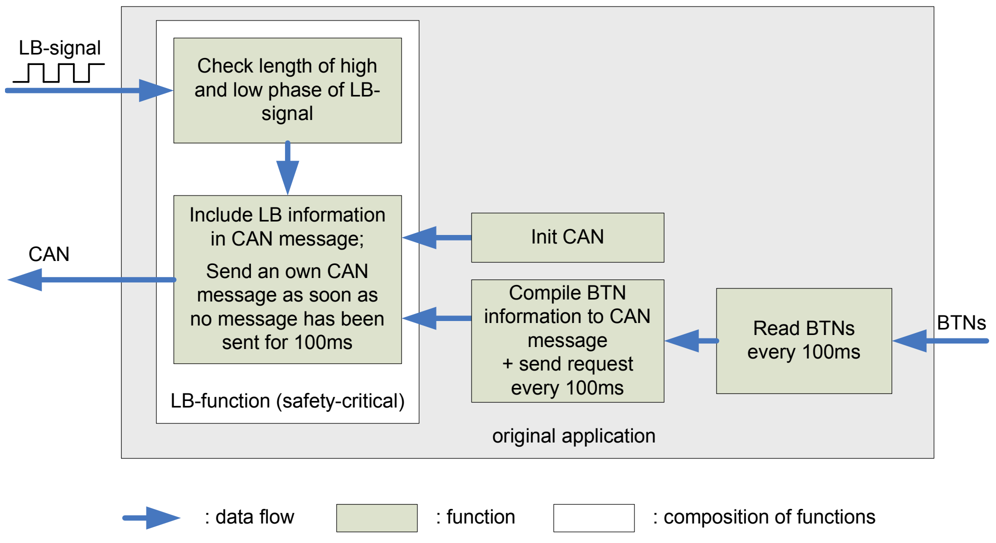

LB-signal
CAN
Check length of high and low phase of LB-signal
Include LB information in CAN message;
Send an own CAN message as soon as no message has been sent for 100ms
LB-function (safety-critical)
Init CAN
Compile BTN information to CAN message + send request every 100ms
Read BTNs every 100ms
BTNs
original application
: data flow
: function
: composition of functions

Figure 4.10.: Original context of the life beat funktion (LB function)

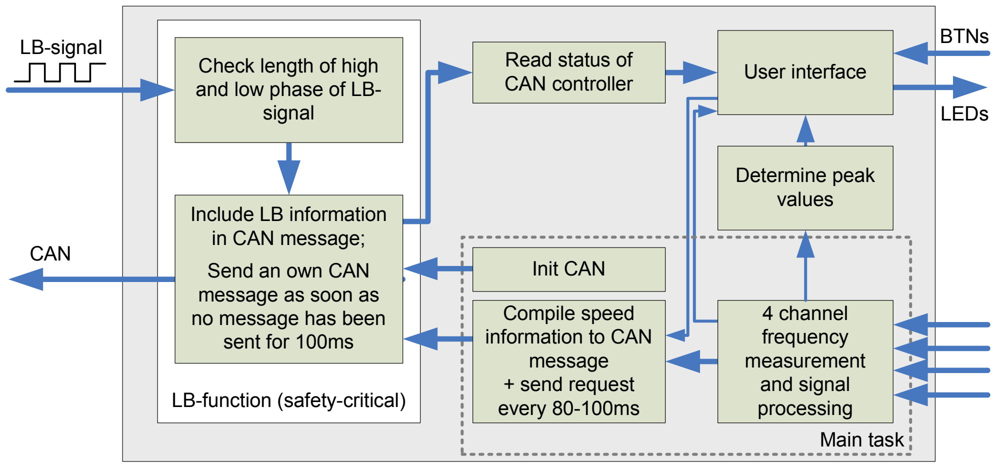

LB-signal
CAN
Check length of high and low phase of LB-signal
Include LB information in CAN message;
Send an own CAN message as soon as no message has been sent for 100ms
LB-function (safety-critical)
Read status of CAN controller
User interface
BTNs
LEDs
Determine peak values
Init CAN
Compile speed information to CAN message + send request every 80-100ms
4 channel frequency measurement and signal processing
Main task

Figure 4.11.: Application in which the LB component had to be reused

same specification, which makes it necessary to provide additional information during the experiment. In this case, this information should also be provided in a written form (e.g. via email, as applied in [53]) to all participants. In our experiments, we also chose to give additional information via email to all experiment participants. Moreover, it turned out to be extremely useful to apply the same task in several of our experiments and to improve the specification for every run to avoid problems that appeared in the previous experiment.

Further on, a certain control of the experiment participants is also needed. This control includes several aspects. First, the *development effort* performed by the experiment participants will have to be controlled, if this variable is not the dependent variable. This effort is usually measured by the *development time*. The obvious approach is to give a fixed time to fulfill the experiment task (e.g. development is limited to the times of a lab course). However, time is a variable that is hard to keep constant, as some participants continue their work at home, even if they are explicitly asked not to do so. Even without explicit implementations outside the actual experiment, it makes a difference if participants think outside the experiment hours how to go on and how to solve a problem or if they start thinking how to go on each time they enter the lab. Therefore, questionnaires were conducted at the end of experiments, which took place in lab courses to gain information concerning development activities outside the actual experiment hours. Moreover, the time needed for reviews was measured in Experiment 3. Since reviews had to be conducted as a whole and a maximum time for review of three hours was given, no impacts on the variable time are expected in this case. Finally, the development effort was not only determined by measuring the time in Experiment 6, but on basis of weekly meetings in which problems and challenges, which occurred during implementation, were discussed.

As reliability aspects are a major concern for the evaluations in this work, a certain minimum quality of the experiment outcomes has to be guaranteed. As it cannot be expected that all experiment participants put similar effort in the verification of their implementation, acceptance criteria were applied in all of our experiments, which involved development activities (see Section 4.6).

Moreover, students had to conduct certain actions during our experiments in order to allow a later evaluation. In the majority of experiments conducted for this work, the participants had to save the current state of their work as soon as one of a set of subtasks was implemented successfully. These activities had to be controlled very strictly, as the participants could understand them wrong or simply forget them. The same is true for questionnaires. While multiple choice questionnaires could be checked for soundness and completeness, most other types of questionnaires cannot be checked automatically and require manual checks for this reason. However, a tool that checks and indicates incomplete parts of the questionnaires automatically could also lead to participants checking the boxes arbitrarily to finish the questionnaire quickly.

Another important aspect is to assure that all participants develop their implementation of the specification independently. Beside the fact that we asked the participants to develop their versions independently, we partly used a plagiarism finder tool to assure no suspicious similarities were present in the different versions (see Section 6.1). However, we experienced no problems with plagiarism in our experiments. The reason for this result is seen in the fact that we did not give the students hard deadlines and that we supported them in case of problems whenever possible[12] in order to release the pressure of delivering a final version.

Another important variable that has to be controlled is the previous knowledge of the participants. The previous knowledge was tried to leverage by an introductory course described in the following section. Moreover, differences in the previous knowledge were determined by questionnaires described later in this section.

Introductory Course

The students participating in our experiments had certain experiences in embedded systems. However, experiences with the hardware platforms applied in the experiment differed between the participants. Therefore, an introductory course was used in all experiments, which were conducted in lab courses, with the intention to leverage the knowledge of the participants. The main idea was to make the students familiar with the development environments provided for the experiments. The introductory course took place on two days and was held prior to the actual lab course. An introduction to microcontroller programming was given on the first day and an introduction to PLD[13] programming on the second day. To emphasize the differences and similarities between the two platforms, a common introductory example was used in both cases. Moreover, the introductory example was chosen to be different enough from the later experiment task, but still allowed to clarify challenges and possible solutions of the implementation on these two hardware platforms.

Variable Measurement

In our evaluations, the quality (in terms of low numbers of failures) of the experiment outcomes is important. While being an acceptance criteria as already mentioned above, it is also a major part of the evaluation in the experiments conducted for this work.

In case of the acceptance criteria, testing could be applied to verify if the experiment outcomes had fulfilled the acceptance criteria. As testing cannot guarantee the absence of failure, the acceptance criteria have to be formulated in

[12]It is important to note that only implementation specific support (e.g. where to find information about interrupt priorities) and general suggestions (e.g. try to test each module on its own before you test the overall system) are allowed to avoid undesired influences by the experiment supervisor

[13]PLD: CPLD in Experiment 1, FPGA in Experiment 2,4,5

a way that unambiguous test cases can be generated easily. For our experiments, the acceptance criteria were directly formulated as test cases.

The actual variable to be measured in the majority of our experiments (Experiment 1,2,4,5) is the number of faults in the versions developed in the experiments. Beside reviews/inspections or formal verification, testing is the most important means for evaluating software with respect to correctness [46, 81]. While in case of acceptance tests, a defined set of test scenarios could be used, the evaluation of functional correctness with respect to reliability is more challenging. But although testing can never guarantee the absence of failures, the number of failures found by extensive testing can still be used as an indicator for the system's reliability [32, 81]. Analyzing the results of an experiment includes the testing of all the different outcomes created with equal test cases. Therefore, black box testing should be applied since this technique is independent from the individual subject tested. Since the significance of the results gained by black box testing is increasing with the amount of test cases a high number of different test inputs is desirable.

Further dependent variables, as the reviewability in Experiment 3, were based on a grading of the reviewers/developers. These variables were collected by using forms and questionnaires. This measurement revealed only challenges in the development of the corresponding forms/questionaires. These report forms were provided for the documentation of test and review activities (Experiment 4, only review in Experiment 3) and were applied for several reasons. First, they allowed an easier comparison of the different reports. Second, the structure of the reports was intended to give a certain guidance for the test and review processes. Finally, the given structure of the reports allowed a minimum control of the test and review activities as described in Section 5.4.

Finally, the development effort had to be measured in Experiment 6. As stated in the setup of the hypothesis in Section 4.4, this variable was determined by measuring the development time and documenting problems, which occurred during development. Of course, these measures are not very exact. However, as only a general effect should be evaluated in this experiment, this approach is considered as sufficient.

In all experiments that took place in lab courses, two different questionnaires were used. A first questionnaire was conducted prior to the experiment to determine the previous knowledge and the expectations of the participants. The second questionnaire was conducted at the end of each experiment to receive feedback and to gain information about the implementation work (e.g. how much work was done at home). The results of these questionnaires are used during the validity determination of the experiment results in Section 6.1. Moreover, certain aspects are used for the discussion of the educational aspects of these experiments in Section 7.3.

For additional information with respect to the differences of FPGAs and MCUs, the students had to fill out a feedback sheet during Experiment 5. The idea behind this form was that students document any difference they experience between the two platforms during the experiment. These information are analyzed in Section 6.7.

Further on, the test forms used for the acceptance tests were an important measure to assure that all relevant aspects were tested during the acceptance tests, especially in case of the additional tasks. Finally, the test environments applied for acceptance and evaluation tests are described in Section 4.7.4.

Determination of Statistical Relevance

The data used for successful hypothesis tests has to be evaluated with respect to statistical significance. The reasoning behind this measure is to determine the probability that the observed results are not dependent on the treatment. In the first experiment, the significance was determined within the method applied for the test of hypothesis H_01a, namely the method of *resampling* (*bootstrap method* to be exact). This approach is following an intuitive approach of drawing samples from the set of observed results. The important aspect is that the drawn sample is put back to the set (re-sampling) each time. While this approach is further described in Section 5.1.4, additional details can be found in literature, e.g. in [16].

For the remaining successful hypothesis tests, the significance of the difference between the results observed in the two treatment groups was also determined by using the resampling method. In all cases, the following approach was chosen. Given is the set of all N observed values (both treatment groups). From this set N samples are drawn with replacement and the difference is calculated between the mean[14] of the first half of this sample and the mean of the second half of the sample. The value calculated is documented. Then this procedure (take N samples and compare the two means) is replicated 100000 times. Finally, it is determined how often the difference observed between the two treatment groups or even a larger difference occurred in the 100000 random comparisons. If this value is sufficiently low[15], the observed difference is considered as significant.

4.7.3. Guidelines

In all experiments, one or more guidelines were provided to the participants in order to structure and/or control the actions within the experiment.

[14]Alternative: median (depends on hypothesis)
[15]An accepted threshold for this value is 0.05 [16]

In the experiments, in which intermediate versions were of interest (especially Experiment 2), storing of the correct versions was essential. Therefore, guidelines determining this process were provided in these experiments.

For the experiments including test and review activities (Experiment 3 and 4), brief guidelines for the test and review process were given to the experiment participants. Further guidance for these processes was given by the test and review report forms used in these experiments (see Section 4.6). Otherwise, a guideline for the integration of the component to be reused had to be provided in Experiment 5.

Furthermore, a composition of typical implementation issues was provided for both targets in case of the experiments 1, 2, 4 and 5. They included the basics of the VHDL language and specialties of the C language for embedded systems. Moreover, certain coding guidelines were used in these experiments. These guidelines restricted, for example, the use of in-line assembly statements and existing VHDL components. In Experiment 6, the coding standards according to *MISRA-C* [68] had to be followed for the C language and the recommendations given in *The VHDL Golden Reference Guide* [25] had to be followed for VHDL.

Finally, guidelines for the proper use of the provided hardware platforms and measurement equipment were needed to avoid damage to the material and the participants. Parts of the guidelines for the individual experiments differed as miscellaneous devices were used.

All guidelines targeting the implementation itself were included in the experiment task description. Experiment 6 is an exception as the specification and the coding guidelines were separate documents in this case.

4.7.4. Setup of Test Environments

Embedded systems realize functions in interaction with their environment. In contrast to mere software functions, these functions have to be analyzed via dedicated hardware/software interfaces. Additionally, embedded systems often have to fulfill real-time requirements. According to these requirements, it is usually of great importance to analyze the system's real-time behavior. For the task chosen for the experiments described in this work it has to be checked if the response on inputs (reset, speed signals, and certain experiment specific signals) generates outputs according to the specified timing behavior. In order to analyze this real-time behavior, specific measurement equipment is necessary.

The aspects presented above in combination with the need for high numbers of test cases identified in Section 4.7.2 make automated test environments necessary. For the majority of embedded applications, these test environments have to be designed for each individual application. The testing for correct real-time behavior demands real-time properties of the corresponding parts in the test environment, which complicates the design and verification of the test environment. Another advantage of automatic testing is that it usually is less error prone than manual testing [32].

For four of our experiments, a similar experiment task was applied. Therefore, a common test environment could be developed for the evaluation of these experiments. However, it is required that this test environment is flexible enough to target the variations in the four experiments. For the acceptance tests conducted in these experiments, a semi automatic test environment was developed that is described in the following section, while the automatic test environment developed for the evaluation is described afterwards.

While the embedding system could be simulated by simple measures in case of the experiments mentioned above, the complexity of the embedding system in Experiment 6 (several sensors and actuators interconnected via the roof mechanic, further system components connected via CAN bus) required the development of a sophisticated simulation environment. As this environment included the same interface as required for test activities, a combined test- and simulation environment was developed, which is described at the end of this section.

Semi Automatic Test Environment (experiments 2,4,5)

While the acceptance test was executed manually in the first experiment conducted (Experiment 1), we applied a semi automated acceptance test in the following experiments. This test approach allowed to generate a defined selection of physical test signals in real-time. The reasoning behind this change was to assure equal, reproducible test conditions in the acceptance test and to speed up the process of acceptance testing.

Since physical signals (four speed signals) had to be generated in real-time for these acceptance tests, we implemented the semi automated acceptance test on an FPGA development board. With this board, different test scenarios could be selected by an array of switches. The correctness of the outputs of the *device under test* (DUT) could be checked on a CAN monitor. The results of this monitor were displayed via video projector making the results visible for all lab course participants. The check was performed manually by comparing the values displayed with a given table of expected results. This approach allowed fast and reproducible acceptance tests.

Advantages of this approach are the small form factor of the test hardware (just one FPGA development board in combination with the CAN monitor already present) as well as the easy handling of the test board within the given setting. The drawbacks of this approach are that the timing of the DUT outputs cannot be verified by this manual approach (only large violations of the maximum response time can be detected this way) and that the number of test cases is very limited. While this approach is sufficient for the acceptance tests, a more sophisticated test environment is required for the evaluation tests, which is presented in the following section.

Automatic Test Environment (experiments 1,2,4,5)

All versions, which passed the acceptance test, were considered for evaluation testing. As in case of the acceptance test, physical signals had to be generated in real-time. While only selected input scenarios were generated for the acceptance test, in this case, all possible input scenarios must have been able to be generated to allow a sufficient test coverage. Therefore, an extended, flexible approach of signal generation was realized. Unlike the semi automated test used for acceptance tests, a completely automatic evaluation test environment is desirable for several reasons. First of all, it allows faster correctness evaluation of the outputs of the device under test (DUT) and therefore faster test runs. The increased evaluation speed is important for the evaluation tests as high numbers of test scenarios should be tested in sufficient time. Secondly, the correct timing of the DUT outputs cannot be verified manually with acceptable effort. Thus, the test environment had to be extended by a component providing the test scenarios for automatic test runs. Moreover, a component is required that manages the test process in a way, that test scenarios are fed to the DUT at defined instances in time and that records and time stamps all DUT outputs. Finally, a component storing and evaluating the test results is needed.

The structure of the test environment we developed for the evaluation of the different experiment outcomes is depicted in Fig. 4.12. The test environment was built on basis of the following requirements, driven by the need to determine the real-time behavior of the DUT and the idea to provide a flexible and easy to handle test environment. The test environment must be able to:

1. read test cases from text files,
2. generate test signals with values and timing given in test cases,
3. record and time stamp the responses of the DUT on test signals,
4. store test results (test case, time stamp and test response) in a text file, and to
5. evaluate test results (compare test results with specified behavior).

Further on, it should be possible, to check for the correct behavior of the test environment

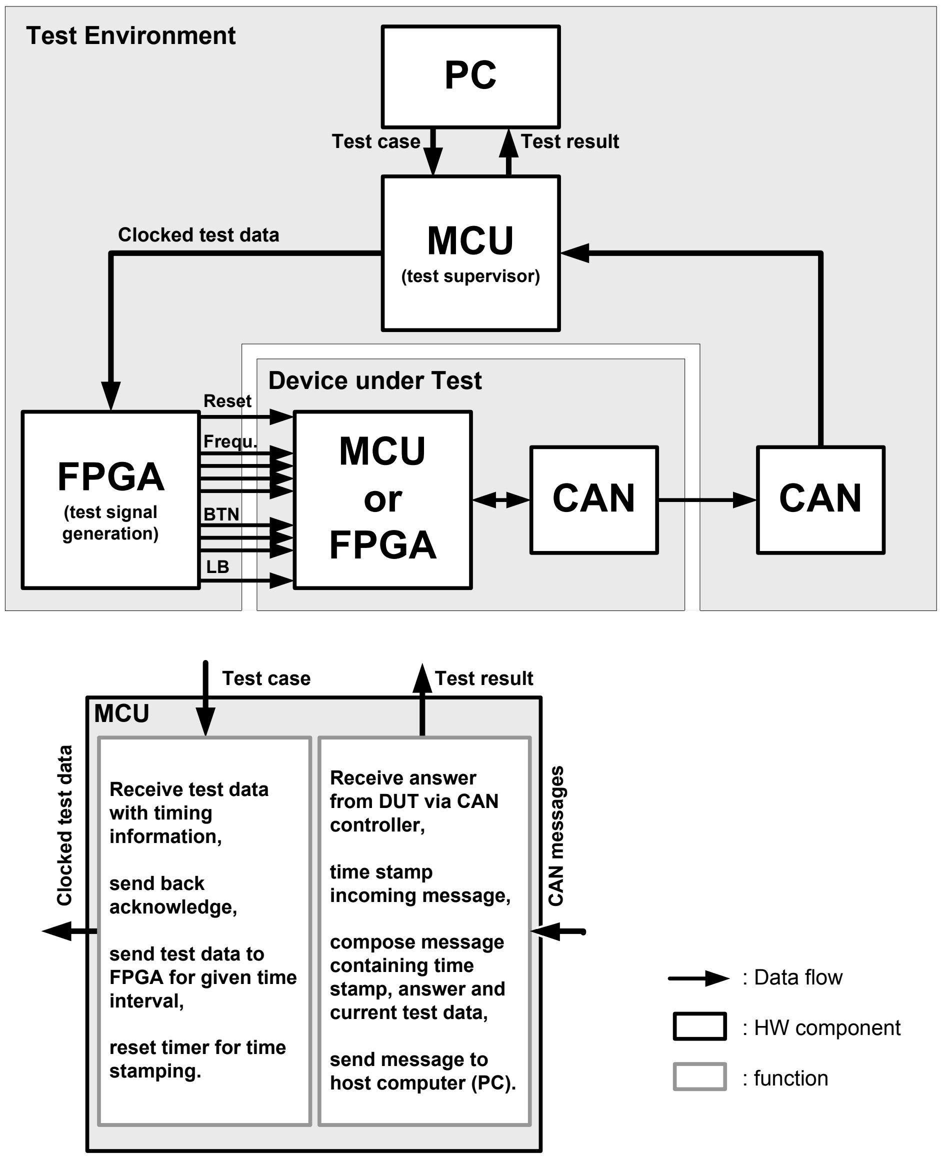

Test Environment
PC
Test case
Test result
Clocked test data
MCU
(test supervisor)
Device under Test
Reset
Frequ.
BTN
LB
FPGA
(test signal generation)
MCU or FPGA
CAN
CAN
Test case
Test result
MCU
Clocked test data
CAN messages
Receive test data with timing information,
send back acknowledge,
send test data to FPGA for given time interval,
reset timer for time stamping.
Receive answer from DUT via CAN controller,
time stamp incoming message,
compose message containing time stamp, answer and current test data,
send message to host computer (PC).
: Data flow
: HW component
: function

Figure 4.12.: Automatic test environment used for evaluation

Based on these requirements, we developed the test environment presented in Fig.4.12, which in principle works as follows: Test cases are read from a text file by the host computer (PC) and are sent to a microcontroller (MCU) via serial connection. Each message sent to the MCU contains one set of inputs for the DUT and a value for the period of time this set of inputs should be fed into the

Table 4.3.: Signals to be generated and processed by the test environment

Experiment	**Test signals**	**Output**
Experiment 1	4x frequency + reset	CAN
Experiment 2	4x frequency + reset + 3x button	CAN
Experiment 4	4x frequency + reset + 3x button	CAN
Experiment 5	4x frequency + reset + 2x button + life beat	CAN

DUT. The test data is then sent to the FPGA for test signal generation. The second task of the MCU is to time stamp all messages coming from the DUT and to send them, together with the current test case present at the inputs of the DUT, to the host computer, in which they are stored in another text file for later off-line analysis. Accordingly, the MCU acts as the test supervisor in this approach, which is depicted in more detail in the lower part of Fig. 4.12.

The separation of measurement and evaluation activities was chosen for two reasons: Firstly, it simplifies the analysis since no real-time requirements have to be faced during the analysis process. Secondly, it simplifies the verification of the test environment, since measurement and evaluation are well separated.

The failures considered during analysis are: timing failures (e.g. check if a message arrived in time), silent failures (e.g. no message arrived for a certain test input), and content failures (e.g. a message arrived in time but contained one or more wrong values). While the test included only the test of the speed measurement functionality in case of Experiment 1, additional aspects had to be considered for Experiment 2, 4, and 5 (see Tab. 4.3). These aspects included a correct reaction of the DUT on certain user inputs via buttons and the additional life beat function.

The design of the evaluation test environment presented above allowed automatic test runs with given test scenarios. To assure correct evaluation, the behavior of the test environment had to be verified. In this context, the strict separation between the actual test run and the later off-line evaluation turned out to be very useful as this structure allowed a separate evaluation of these two parts. Both parts were examined by conducting several identical test runs under constant conditions and by comparing test and analysis results with results achieved by other measures (oscilloscope, CAN monitor, manual calculation, etc.). During the verification of the test runs, problems with the reproducibility of the test results *(fault activation reproducibility)* occurred. The reason behind this problem originate in the real-time properties of the application [118]. According to these real-time properties, not only the combination of inputs fed into the DUT is of importance, but also the time they occur. Therefore, the internal state of the DUT and thus its outputs could depend on previous inputs. To overcome this problem, a defined reset of the DUT was conducted at the start of each test run, which could enhance the reproducibility of the test results. To handle incidental, remaining variations, certain test runs were conducted twice and potential differences were considered in the evaluation.

Moreover, real-time properties have to be considered in the process of recording the corresponding response data from the DUT. The task chosen for the experiments required measurement intervals up to 2 seconds, depending on the input (low input values required a long measurement interval). This aspect led to particular long test runs in case of several test cases (e.g. $> 2h$ for 20000 lines of test data). Another challenge were DUTs with high rates of sending CAN messages. Some versions of the first experiment, in which we applied this test environment, updated their outputs (CAN messages) faster than expected. Hence, the test environment had to be adapted in order to cope with this speed and amount of data. In order to reduce the amount of data produced with every test run we specified a minimum interval of sending CAN messages in the following experiments. In the majority of our experiments, the output format of the DUT was a CAN message, which allowed a comparatively easy evaluation of the results. Outputs based on binary values seem easier to evaluate on the first view, but only if the bit width is low. Accordingly, the complexity of the output data will be limited if binary outputs are used. Additionally, a sophisticated communication interface as the CAN communication benefits from built in error detection and correction codes. Finally, all recorded results had to be evaluated. Beside every input combination, the timing of these input combinations had to be considered during evaluation. For the evaluation of the timing, the time stamps added during the test run were analyzed. To ease the evaluation (and its later verification), the evaluation was divided into independent sections (e.g. determine those messages that arrived in the considered time interval in a first step and analyze the correctness of the content of these message in a second step) and intermediate results were listed beside the final result.

The modifiability of the test environment is another aspect that turned out to be very important. As described above, the generation of all physical test signals was realized with an FPGA. This approach allows great flexibility since new signals can be added to the test environment without influencing already existent signals (parallel structure of the FPGA). This concept has shown to be very useful as additional signals had to be added for the experiments 2, 4 and 5 (see Tab. 4.3).

An overview of the testing issues we identified in embedded systems can be found in Fig. 4.13. These issues show that testing is an important aspect in the evaluation in embedded real-time systems. If empirical evaluations are conducted in the field of embedded systems, the design of the experiment has a major impact on the testing of the experiment outcomes. The so called *design*

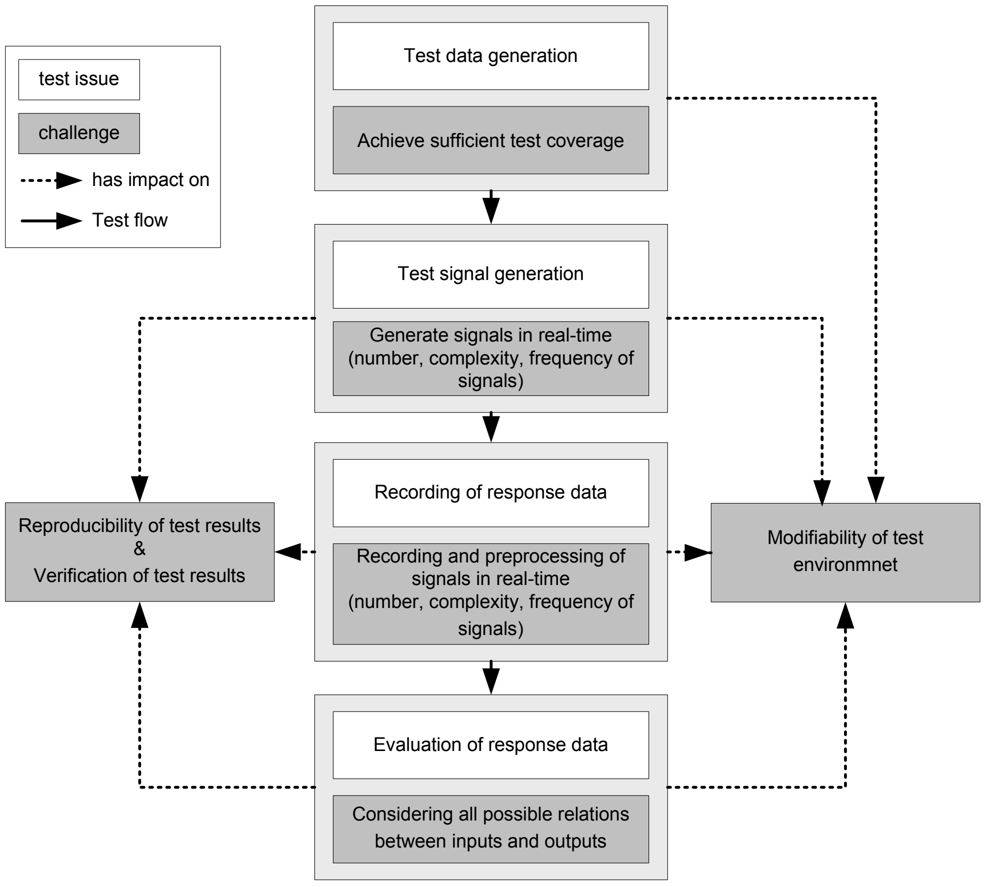

test issue
challenge
has impact on
Test flow
Test data generation
Achieve sufficient test coverage
Test signal generation
Generate signals in real-time (number, complexity, frequency of signals)
Recording of response data
Recording and preprocessing of signals in real-time (number, complexity, frequency of signals)
Evaluation of response data
Considering all possible relations between inputs and outputs
Reproducibility of test results & Verification of test results
Modifiability of test environmnet

Figure 4.13.: Testing issues in embedded systems and their challenges

for testability [122] can significantly ease the overall evaluation process and therefore should be considered in the experiment design.

Finally, a simplified version of this test environment was applied for the testing activities of the experiment participants in Experiment 4. While the approach with a combination of an MCU and an FPGA allowed an intuitive[16] and modular structure, an approach requiring less hardware resources and allowing a smaller layout was needed for this experiment. Therefore, the approach was completely integrated into a single FPGA while the external CAN controller remained. The structure allowed to reduce the test environment on a single evaluation board with a CAN controller, which could be provided in a sufficient

[16]signal generation was suited best for the FPGA while control and communication aspects could be implemented in a straight forward manner in the MCU

number to the experiment participants. Further details regarding both types of the test environment can be found in the corresponding technical report [91].

Automatic Test and Simulation Environment (Experiment 6)

The test and simulation environment applied for Experiment 6 was built for two purposes. First, a roof control had to be implemented in this experiment. As no physical roof could be provided for this experiment, the function of the corresponding sensors and actuators had to be simulated. Second, the applications developed during this experiment had to be evaluated with respect to their failure behavior.

The corresponding test environment was based on an FPGA for signal generation and signal measurement. As part of the measurement activities, the FPGA also received all CAN messages from the DUT and extended all relevant signals measured with time stamps. The simulation of the roof itself was implemented on a personal computer with the tools Matlab and Simulink[17]. Moreover, most check and test activities were implemented with these tools. The communication between the FPGA and the personal computer was realized via a serial connection as in the previous test environment.

Further details of the test and simulation environment are present in [27].

4.7.5. Test Case Generation

For the evaluation tests conducted on the data collected in the experiments 1,2,4, and 5 and partly on those collected in Experiment 6, sufficient test cases had to be generated. As all experiment outcomes had to be treated equally (see Section 4.5), black box testing was applied since test data generated following this approach is independent from the different versions tested.

Test Cases for Experiments 1, 2, 4, 5

The test cases for these experiments were based on *random testing* and *deterministic testing*. In the latter case, the test cases were determined by a specific driving scenario (a tool was developed to generate test data based on a given velocity profile) and by manual selection of specific input constellations, which were considered as especially critical. In the case of random testing, the test cases were determined randomly (even probability distribution) by a another tool developed for this purpose. An overview of the test cases used in the evaluation of the different experiments can be found in Tab. 4.4. The test benches in this table are not consecutively numbered, as certain test benches used turned

[17]www.mathworks.de

out to be unsuitable and were not used for the evaluations presented in this paper. However, the numbers remained to allow an easy comparison with other documents (e.g. test reports). Moreover, a graphical representation of certain test benches can be found in Fig. 4.14 and further details on the tools developed for test case generation can be found in [91].

Table 4.4.: Test benches used for evaluation testing

Test bench	**Exp.**	**Type of test cases**	**# test cases**
TB1	1,2,4,5	moderate and extreme values selected manually, partly individual values (applied for initial checks)	156
TB2	1,2,4,5	values from 0 to 7 kHz	~2100
TB3	1,2	random values from 0 to 3kHz, individual values	~20000
TB4	1,2,4,5	velocity profile including values from 0 to 6kHz	~10000
TB8	1,2,4	random values from 0 to 6kHz	~15000
TB9	2,4	random values from 0 to 6kHz, individual values, difference between two consecutive values limited to 200Hz	~15000
TB12	5	increasing values up to 100kHz and constant LB signal	42
TB13	5	increasing values up to 100kHz + LB signal switched on/off	75
TB4b	2,4,5	based on parts of TB4, extended by signals for buttons (file is named TB4button)	~12000

Since the maximum allowed response time to certain input combinations was up to 2 seconds, the test runs for the 76 final versions[18] considered in this thesis needed more time than might have been expected by the number of test input combinations. Moreover, the measurement took place not only for the final implementation, but also for intermediate versions (main version) in some experiments increasing the measurement effort.

As soon as testing is applied for evaluation, coverage criteria are an important issue to estimate the test coverage. As an example, test coverage metrics have been applied in [63] to characterize the different experiment outcomes. However, no correlation between test coverage and identified faults could be shown in this

[18]exp.1:20+exp.2:20+exp.4:12+exp.5:24

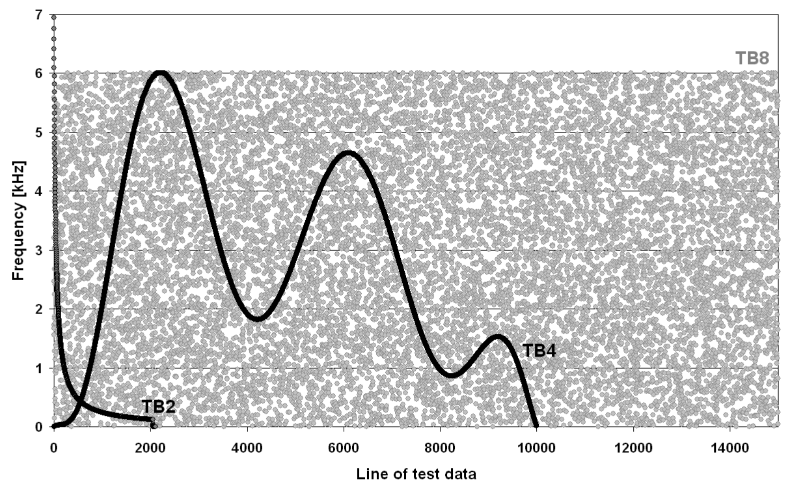


Figure 4.14.: Test cases used for evaluation testing

work. A problem present for the application of coverage metrics in this work is that no common test metric could be found for MCUs and FPGAs. Moreover, real-time properties have an impact on usual test coverage metrics [29] and therefore require sophisticated coverage models. Since the aim of our experiments is not to show that the different versions implement the specification correctly, but to identify differences in failure behavior of the various versions, no coverage criteria were applied in these evaluations.

Test Cases for Experiment 6

The test cases used for acceptance testing in Experiment 6 were taken from the acceptance criteria that were part of the specification. As the evaluation of the development effort was the focus of this experiment, this acceptance tests represented the major part of the test activities in this experiment. Moreover, tests were applied in this experiment to show the efficiency of the safety measures applied in the different architectures. Therefore, fault injection was the major concern of this testing. Faults were injected in all components that were covered by safety measures. According to the chosen safety concept (see Section 5.6.2), faults were included in the program and data memories, in CPU registers, in I/O registers as well as in CAN registers. Moreover, variations of the power and clock supplies were simulated.

5. Operation of Empirical Studies

This chapter describes the operational phase of the experiments, in which the different treatments were applied to the subjects. Therefore, the operation of each experiment is described in an own section. Moreover, the hypothesis tests are included in each section while further analyses of the results are present in Chapter 6.

We published parts of the description of the operational phase in [96, 97, 98, 101, 106].

5.1. Software Diversity by Diverse Hardware (Exp.1)

Investigations in this first experiment were driven by the hypothesis H1 (software diversity by different hardware platforms) stated in Section 4.4. Therefore, two different hardware platforms were applied to implement a common application as described in Section 4.6.1. Further details on the microcontrollers and CPLDs used as hardware platforms in this experiment are given in Section 4.3.2.

5.1.1. Preparation

Participants

This experiment took place in a lab course students could choose in their main study period. The successful participation in a certain number of lab courses and/or seminars is binding for each student. This requirement was an inducement for participating in our lab course and thus in the experiment. Accordingly, 26 students participated in this experiment with all of them being computer science students in their 5th semester or higher.

As mentioned before, an introductory course was given prior to the experiment mandatory for all participants. This introductory course, which is described briefly in Section 4.7.2, allowed an introduction into both hardware platforms and therefore to leverage the initial knowledge of the participants.

Materials

Prior to the experiment, all materials needed, namely the specification of the experiment task, the experiment guidelines (see Section 4.7.3 for details), and the initial and final questionnaires, were prepared. The specification of the experiment task, which has been described in Section 4.3.1 as well as the experiment guidelines were provided to the students in paper form. The support for the programming languages applied was given in electronic form (text files) and the initial and final questionnaires were conducted electronically with a tool developed for this purpose. Moreover, the test cases for the acceptance test were generated randomly and sheets for the acceptance test were created and printed. Finally, all equipment necessary (development boards, debugging hardware, frequency generators, CAN bus monitor, required cables) was collected and prepared for its use in the experiment.

5.1.2. Execution

The experiment took place in the lab course mentioned above with 14 weekly appointments of three hours each. Prior to the experiment, the students had to fill out the initial questionnaire while the final questionnaire had to be filled out after the completion of the experiment. The experiment was executed according to the experiment design described in Section 4.6.1. The actual data collection was achieved by a tool storing all program versions compiled. However, only the final version of each team was evaluated in this experiment. Moreover, anonymization of the experiment results, which is desirable to reduce external influences, was achieved by assigning random numbers to the 12 development teams (see also Section 7.3).

The acceptance test was conducted manually and was based on the test cases generated randomly. If a team did not pass the acceptance test, the team had the chance to improve the implementation. In the end, only versions that passed the acceptance test were considered for later evaluation.

By applying the acceptance tests as precondition for the later evaluation, it was assured that the treatments were applied in correct order and that only accepted versions were stored for later analysis. In this experiment, 10 teams passed the acceptance test with both versions (MCU and CPLD implementation) resulting in 20 versions considered for evaluation.

5.1.3. Variable Measurement

All experiment versions, which passed the acceptance test were tested. Therefore, the automatic test environment described in Section 4.7.4 was applied and each version was tested with different test benches. For the evaluation tests in this experiment, four test benches were applied. These test benches (TB2, TB3, TB4, and TB8) are introduced in Section 4.7.5 and stress different aspects of the application. The corresponding results were stored for later analysis.

A listing of the median and the arithmetic mean of the failure rates obtained by testing can be found in Tab. 5.1 for both hardware platforms. Moreover, a listing of the failures observed for each individual version can be found in Appendix A. The comparatively high numbers of failures in most CPLD versions in case of test bench 3 were surprising. Later analysis identified crosstalk between the cables carrying the sensor signals. This effect occurred only if different frequencies were present on the four signal lines (as in TB3). Moreover, the problem occurred only in CPLD versions, although the effect of crosstalk was present in both cases. Later improvements in the test process (shorter cables or filter elements for the four lines carrying the speed signals) could mitigate this problem. Then, the test run of TB3 was repeated with a shortened version of this test bench for CPLD versions and the results can be found in brackets in Tab. 5.1. Results of the remaining test benches were not affected by the effect described. Further interpretation of the measurement results can be found in the succeeding section and in Section 6.2.1.

Table 5.1.: Mean and median values of the fraction of failures made on different hardware platforms

	TB2	TB3	TB4	TB8
Mean MCU	2.98%	9.41%	2.54%	6.19%
Mean CPLD	6.05%	88.04% (20.82%)	2.32%	2.04%
Median MCU	1.00%	2.16%	0.18%	4.96%
Median CPLD	4.69%	98.60% (3.32%)	1.92%	1.00%

5.1.4. Hypothesis Testing

The contents of Tab. 5.1 show that MCU versions as well as CPLD versions lead to faults in case of all test benches. Moreover, all except one CPLD versions show very high failure rates in case of TB3. As mentioned in the previous section, these problems resulted from crosstalk between the four measurement channels. While these disturbances showed no effect in MCU versions, CPLD versions tended to interpret these disturbances as valid signals. This aspect could not be discovered with the acceptance test, as only equal values were used in this test (only a single frequency generator was available at this time).

While the problem described is considered in the analysis in Section 6.2.1, only test benches based on equal values for all four measurement channels were considered for hypothesis testing. In the end, the evaluations were performed on TB8 only, as it was generated randomly and is the most independent test bench for this reason.

Potential learning effects and thus impacts of the initial implementations on the second implementations had to be considered (see section 4.6.1). For the relevant test bench (TB8), only small variations in the failure rate could be observed. Moreover, a consideration of the mean and median values showed no clear trend: In case of the mean values, the initial MCU versions had lower failure rates than the versions developed in the second half of the experiment (5.17%→7.2%). On the other hand, the CPLD versions of the second half had better mean values (2.8%→1.3%). However, this effect was the other way round when median values were considered (MCU: 8.17%→4.69%, CPLD: 0.7%→1.6%). Therefore, we see no hint for the mentioned learning effects. Nevertheless, only inital versions were considered for the test of the hypothesis H1a as an independent development is a precondition for this hypothesis test. On the other hand, all versions were considered in case of hypothesis H1b as this hypothesis did not aim at showing the independece between the developed versions.

Test of Hypothesis H1a

In contrast to the NVP experiment described in [53], in which a homogeneous group of software versions existed, the software versions in this experiment differ with respect to the hardware platform they have been developed for. This aspect required a hypothesis test that allowed to consider different treatments. Therefore, the approach of *resampling*, as described in [106], has been applied to test the hypothesis in this experiment.

The aspect which has to be tested for the hypothesis H1a is the probability of dependent failures in MCUs and CPLDs. Moreover, it will have to be determined if the probability measured differs from the probability one would achieve in the case both versions fail independently. This aspect has been investigated by resampling. Therefore, for each test case, a random pair of one MCU and one CPLD version was picked and it was determined if both versions fail for this test case. For this evaluation, an output was considered as false as soon as any of the failure types introduced in Section 4.7.5 had occurred. Additionally, it was simulated on basis of the independent failure model and of the average failure fraction of both versions, whether they would fail for this test case. The independent failure model is expressed by the following equation:

$$p(F_{MCU} \cap F_{CPLD}) = p(F_{MCU}) \cdot p(F_{CPLD})$$

representing the null-hypothesis $H_0 1a$ introduced in Section 4.4. Both results were stored for later comparison. This process was performed on all test cases present in the chosen test bench and in a last step, the sum of both results was subtracted. The result was indicating, if the number of observed dependent failures was higher, equal or lower than the number of dependent failures simulated on basis of the independent failure model. In order to increase the precision of this approach, the process steps described above have been repeated 100000 times.

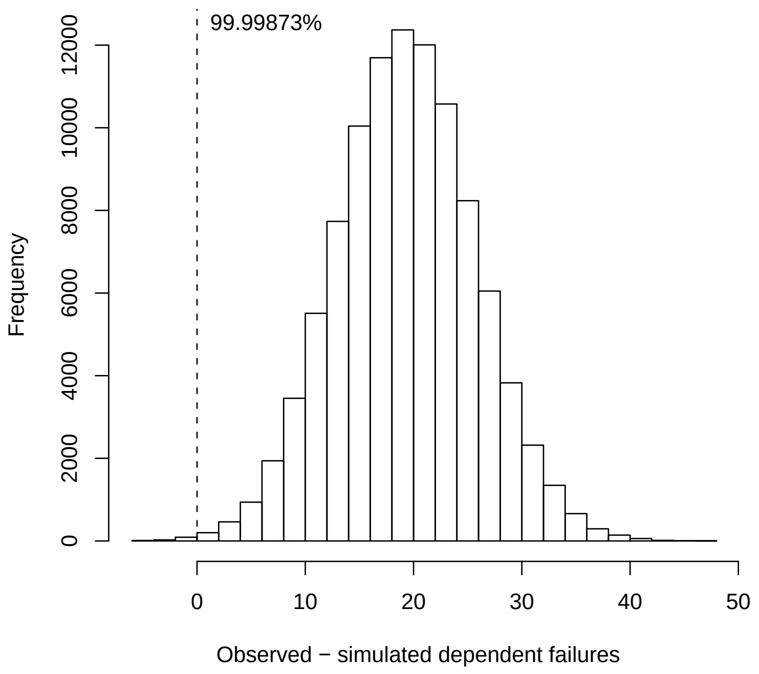


Figure 5.1.: Results of the resampling process

The simulation results achieved by the resampling process described above are shown in the histogram of Fig. 5.1. The difference of observed and simulated failures is indicated on the x-axis while the occurrence is indicated on the y-axis. For example, the x-value of 7 indicates that there are 7 additional observed failures in comparison to the number of simulated failures, which occurred in 2000 comparisons during the resampling process.

According to these results, the dependent failures observed during the experiment are higher than the failures simulated on the basis of the independent failure model. In fact, the same number of dependent failures or less is only simulated in 127 of the 100000 iterations of the resampling process. Thus, the model of independence appears to be unlikely compared to the observed failures. Therefore, the null-hypothesis of independent failure behavior $H_0 1a$ was rejected and $H_1 a$ could be accepted.

Test of Hypothesis H1b

While even software versions implemented on diverse hardware platforms evidently do not lead to independent failures, it has been assumed that they result in an increased failure independence as stated in hypothesis H1b. The analysis was realized with a computer program written for this purpose. This program compares all possible combinations of versions for all test cases applied. Each time a failure is observed in two compared versions, the corresponding test case is marked and the number of observed dependent failures is increased. Again, all failure types introduced in Section 4.7.5 were considered. In the following, a result is considered as wrong, as soon as a fault of one of the three categories appears in both versions.

For the analysis, four categories of combinations are considered: The first category contains pairwise combinations of all CPLD versions (45 combinations), the second pairwise combinations of all MCU versions (45 combinations) and the third pairwise combinations of one CPLD and one MCU version, each created by a different team (90 combinations). The last category contains combinations of CPLD and MCU versions created by one team (10 combinations). The results are depicted in Fig. 5.2 for the four categories of combinations in form of box plots[1]. Outliers are depicted in form of circles (mild outliers) and of stars (extreme outliers). On basis of this graphical representation, the number of dependent failures varies more in case of MCU pairs than in the other three categories of combinations. Neglecting the outliers, the MCU/CPLD pair led to better result than the first two categories of combinations. This result could be regarded as a hint for improved failure diversity. However, a closer look on the outliers reveals that they represent more than 16% of the results. Some of these outliers also represent numbers of dependent failures which exceed the number of dependent failures existent in CPLD pairs. Accordingly, not every combination of diverse platforms does automatically lead to less failure dependency. Nevertheless, the median[2] of the observed failure rates is lower for heterogeneous pairs than the median of both homogeneous pairs.

For the evaluation of the significance of the observed effect, the resampling method was applied. Samples were drawn from the set of observed values for 100000 times as described in Section 4.7.2. In this simulation, the observed difference or a larger difference between the median values of the two treatment groups (heterogeneous vs. homogeneous pairs) was reached only in 0.04% of the simulated results. This result states a high significance of the observed difference between homogeneous and heterogeneous pairs. Moreover, another resampling approach was applied in [106] comparing the individual failure be-

[1]An introduction to box plots can be found e.g. in [15], page 40.

[2]The median is represented by bold horizontal lines in the box plots in Fig. 5.2.

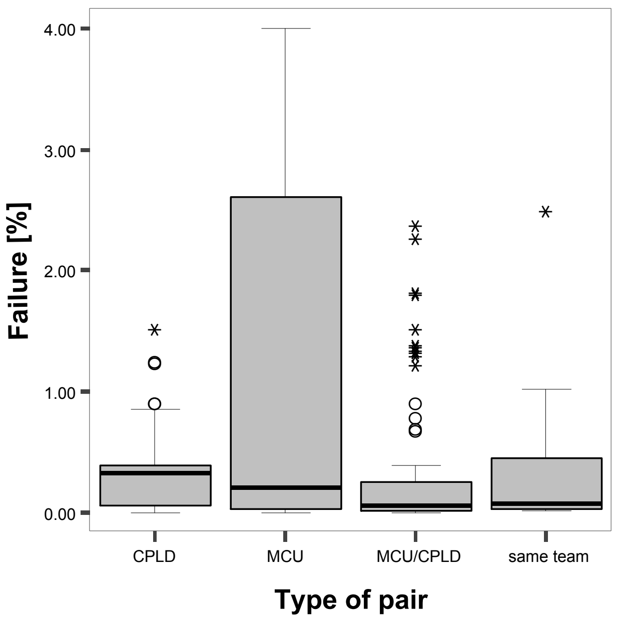


Figure 5.2.: Fraction of dependent failures in pairwise combinations

havior of homogeneous and heterogeneous pairs. The result obtained by this alternative resampling approach supports our findings that the application of heterogeneous pairs improves the independence of software failures over homogeneous pairs.

Additionally, it has to be noted that the median of the heterogeneous pair is also better than both homogeneous pairs in case of test bench 4. Nevertheless, the effect is lower than in case of TB8. Finally, it has to be noted that is was possible to create a test bench which lead to opposite results. With this test bench (TB2), the median of the MCU pairs was slightly better than the median of the mixed pair. As this test bench was comparatively small and minor changes in the results could have changed the overall result, this test bench was not considered for the hypothesis testing.

According to the results presented, the null-hypothesis $H_0 1b$, which states that the median of dependent failures is higher or equal in heterogeneous pairs compared to homogeneous pairs, could be rejected. Therefore, hypothesis $H1b$ was accepted, which indicates improvements in software failure diversity by diverse hardware platforms.

A further analysis of the results gained by the two hypothesis tests is presented in Section 6.2.

5.2. Encapsulation on Different Hardware Platforms (Exp.2)

Hypothesis H2 (impact of hardware platforms on encapsulation, see Section 4.4) was driving the investigations in this second experiment. For this reason, two different hardware platforms were applied to implement a common application as in the previous experiment (see Section 4.6.1 for the design of this experiment).

While the task of the first experiment included only the *main task*, the task of this experiment was extended by 6 additional tasks (see 4.3.1) in order to increase the complexity of the application and to provide functionalities that might be useful to encapsulate. The additional functionalities exceeded the complexity which could be implemented in the CPLD device applied in the first experiment. Thus, this device was replaced by a larger device, namely a Xilinx Spartan 3 FPGA, while the second hardware platforms remained an Atmel ATmega16 MCU (see Section 4.3.2 for details of hardware platforms).

According to the independent implementation of a common specification on different hardware platforms, the results achieved in this experiment could also be used to support the aspects investigated in the first experiment.

5.2.1. Preparation

Initial Study

An initial study was conducted with one student assistant for two reasons: First, the quality of the specification, now including the additional tasks, should be verified this way. As a result, minor changes were applied to the specification to improve their clarity. Second, the feasibility of an implementation of this extended task in the given context on both hardware platforms should be assured. As the initial implementation did not reveal any major problems, the task was used for the following experiment.

Participants

As in the first experiment, this experiment took place in a lab course, which students could choose in their main study period. Therefore, the same concepts were applied including anonymization of the experiment outcomes and the conduction of a mandatory introductory course prior to the actual experiment. In this lab course, 24 students (computer science, 5th semester or higher) applied for this experiment and all of them were selected.

Materials

The materials for this second experiment are mostly identical with those of the first experiment. However, the specification document had to be extended as described in Section 4.3.1.

Instead of a manual acceptance test, a semi automated test environment was used for the conduction of the acceptance tests in this experiment (see 4.7.4). The reason for this change was to reduce the time needed for each acceptance test and to increase the reproducibility of the tests. Further on, the testing with four different inputs, which could not be achieved before, was possible with this test environment. After all, the test cases were generated manually, representing the minimum quality required as discussed in Section 4.6.6 and sheets for this acceptance test were created and printed.

Beside the equipment provided in the first experiment, we also provided multimeters. These measurement devices allowed to measure[3] the frequencies generated by the frequency generators, which were applied for the testing of the frequency measurement functions. Finally, the semi automatic test environment had to be prepared for the acceptance tests.

5.2.2. Execution

The experiment took place in the lab course characterized above with 14 weekly appointments of three hours each. As in the previous experiment, the students had to fill out the initial questionnaire before the experiment start while the final questionnaire had to be filled out after the completion of the experiment. Only minor modifications were applied to these questionnaires according to the adapted application and partly according to problems identified during the evaluation of the questionnaires in the first experiment.

The acceptance test was conducted twice for each implementation in this experiment. A first acceptance test took place after the completion of the main task (same task as in the first experiment) while a second acceptance test was conducted after the implementation of all additional tasks. In the end, only those versions that passed both acceptance tests were considered for later evaluation.

The actual data collection was achieved by a tool storing all program versions compiled. Additionally, the students were asked to store their versions after the first and the second acceptance test and after the successful implementation of each additional task. The reason for the additional storing of versions is to provide intermediate versions with a known functionality. Since the intermediate versions had to be stored by the students, this procedure had to be controlled.

[3]Note: Truth tables were used for frequency determination in the first experiment.

This control could be achieved by storing these versions on net drives, which could be read by the experiment supervisor.

5.2.3. Variable Measurement

In this experiment, 11 MCU and 11 FPGA versions passed the first acceptance test. The main task tested with this first acceptance test was identical with the task implemented in the first experiment. However, the acceptance test applied was modified as described in Section 4.7.4. As in the previous experiment, the test benches TB2, TB3, TB4, and TB8 were used for evaluation testing.

Moreover, all versions which included the additional tasks (final versions) and passed the second acceptance test were tested. In this case, 10 teams passed the acceptance test with both versions and were considered for evaluation. For this second evaluation test, the test environment had to be adapted to allow a testing of the additional tasks. In this case, additional test cases were used to test the additional tasks (TB4b). For further investigations of the problem of processing fast changes in the input channels, a test case was applied instead of TB3, which represented only limited changes between two consecutive input values (TB9).

A summary of the results can be found in Tab. 5.2 for the *main versions* and in Tab. 5.3 for the *final versions*. As in the previous experiment, the failure rates of each individual version can be found in the Appendix A.

Table 5.2.: Mean and median values of the fraction of failures made on different hardware platforms (22 main versions)

	TB2	TB3	TB4	TB8	TB9
Mean MCU	1.15%	27.90%	0.09%	34.07%	9.53%
Mean FPGA	4.56%	23.71%	2.59%	36.84%	8.83%
Median MCU	0.8%	18.5%	0.03%	10.1%	5.1%
Median FPGA	1.9%	20.1%	0.6%	31.2%	4.0%

Table 5.3.: Mean and median values of the fraction of failures made on different hardware platforms (20 final versions)

	TB2	TB4	TB8	TB9	TB4b
Mean MCU	0.95%	1.80%	25.31%	8.26%	27.86%
Mean FPGA	4.64%	0.99%	35.59%	10.17%	12.68%
Median MCU	0.81%	0.04%	8.75%	2.92%	28.36%
Median FPGA	1.50%	0.36%	29.60%	5.58%	7.89%

Derivations of these results from those achieved in the first experiment can be observed. In the main and final versions, the measurement accuracy of constant and slowly changing input values seems to be improved in this second experiment (TB2 and TB4). But fast changes in the input values seem to lead to more failures than in the first experiment (TB3, TB8 and TB9). The reason for this derivation can be seen in the changes in the acceptance test. As the values could be tested more strictly in the second experiment, more effort was put into achieving sufficient accuracy by the students. Moreover, multimeters for measuring the input frequencies were provided in this experiment the first time. These multimeters allowed better testing of the measurement function by the students. However, the additional effort in the measurement function was resulting in longer or more complex measurements, which was often a drawback on measurement speed. Nevertheless, the types of faults identified in these systems are similar as presented in Section 6.2.1.

Moreover, specific test runs were conducted to test specific aspects formulated in the hypotheses driving this evaluation (see Section 4.4). These test runs will be described in the context of the corresponding hypothesis in the following section.

5.2.4. Hypothesis Testing

In this section, the differences between hardware platforms with respect to encapsulation of real-time tasks are investigated. Therefore, the different hypotheses formulated in Section 4.4 are tested in the following.

Test of Hypothesis H2a

The main task contained three subtasks that are depicted in Fig 4.3. The first subtask is a speed measurement, which requires the counting of four input signals for a certain time period while the time period has to be adapted to the input frequency. Second, a processing of the measured values is required to transform the measurement result in the format needed in the CAN message. Finally, the third subtask is the CAN communication, which is necessary for sending the measured and processed values via the CAN bus. In this section, the influence of the first two subtasks on the third subtask is investigated to test hypothesis H2a (see Section 4.4). Thus, the input frequency fed into the DUTs (and thus the execution of subtasks one and two) has been modified and changes in the CAN communication (subtask 3) have been recorded. Two different frequencies (6% and 100% of the max. value specified) were used as input and the time between two consecutive message on the CAN bus was recorded in both cases. The results can be found in the 2nd column of Tab. 5.4.

Table 5.4.: Experiment results regarding encapsulation

Version	Impact of input frequency on the timing of the CAN messages	Impact of user button 1 on the timing of the CAN messages	Impact of user buttons on the values in the CAN messages
MCU 2	yes (+1.0%)	no	no
MCU 3	no	no	yes (button3)
MCU 4	no	no	yes (button1)
MCU 5	yes (-44.4%)	no	no
MCU 6	yes (+2.4%)	yes (stopped)	yes (button1)
MCU 7	yes (+9.8%)	yes (+5.5%)	no
MCU 8	no	no	no
MCU 9	no	no	no
MCU 10	no	no	no
MCU 11	yes (+3.7%)	yes (+1.2%)	no
FPGA 2	no	no	no
FPGA 3	no	no	no
FPGA 4	no	no	no
FPGA 5	no	no	no
FPGA 6	no	no	no
FPGA 7	no	no	no
FPGA 8	no	no	no
FPGA 9	no	no	no
FPGA 10	no	no	no
FPGA 11	no	no	no

According to these results, the timing of the CAN-bus communication was not affected by changes in the input frequencies in case of the FPGA versions. In case of the MCUs, the CAN communication was slowed down with increasing input frequency in case of 4 of the 10 versions (see 2nd column of Tab. 5.4, raise stated in brackets), while the decrease in version MCU5 was intended (longer measurement interval for low frequency input ⇒ was not considered for hypothesis testing). The other four versions changed their timing because of longer program execution times according to the higher input values. Thus, an impact of the speed measurement functionality on the timing behavior of the CAN communication cannot be neglected for MCU versions. The corresponding test of the null-hypothesis is conducted as follows (all values in [ms]):

$$H_0 2a : \frac{\sum_1^N |\Delta T_{FPGA}(f_L, f_H, V_f)|}{N} \geq \frac{\sum_1^N |\Delta T_{MCU}(f_L, f_H, V_f)|}{N}$$

$$\Rightarrow \frac{\sum\{0,0,0,0,0,0,0,0,0,0\}}{10} \geq \frac{\sum\{1,0,0,0,2,8,0,0,0,3\}}{10}$$

$$\Rightarrow 0 \geq 1.4 \Rightarrow \mathit{disagreement}$$

Accordingly, the null-hypothesis $H_0 2a$ could be refused and the hypothesis $H2a$ was accepted.

The significance of the difference between MCU and FPGA versions was determined by using the resampling method on the observed values (see Section 4.7.2). In this simulation, a difference equal or larger than the observed difference occurred in only 5.10% of all random samples. While a value of ≤5% would have stated the significance of the observed difference, the calculated value indicates a limitation of the significance (see Section 7.1 for further discussion).

Test of Hypothesis H2b

For the test of hypothesis H_2b (less changes in the *timing* of the CAN communication in the FPGA versions than in the MCU versions according to the integration of additional tasks, see Section 4.4), the timing of the CAN communication of the *main versions* was compared with the timing of those versions including the additional tasks (*final versions*). In case of 5 MCU versions, the inclusion of the additional tasks led to changes in the timing of the CAN communication (increase of time between two consecutive CAN-messages from 0.5ms to 20ms as displayed in the equation below) while the CAN communication was not affected by the additional tasks in any FPGA version. As depicted in the third column of Tab. 5.4, this difference was even increased by pressing user button 1 (changes behavior from displaying the CAN status to displaying the input frequencies on LEDs) in three MCU versions. In one of these versions, the execution was stopped completely in this case. These results are leading to the following hypothesis test (all values in [ms]):

$$H_0 2b : \frac{\sum_1^N |\Delta T_{FPGA}(V_f, V_m, f_H)|}{N} \geq \frac{\sum_1^N |\Delta T_{MCU}(V_f, V_m, f_H)|}{N}$$

$$\Rightarrow \frac{\sum\{0,0,0,0,0,0,0,0,0,0\}}{10} \geq \frac{\sum\{0,0,1,20,0.5,2,0.5,0,0,0\}}{10}$$

$$\Rightarrow 0 \geq 2.4 \Rightarrow \mathit{disagreement}$$

Therefore, $H_0 2b$ could be rejected leading to the acceptance of the corresponding hypothesis H_2b. Again, the significance of the observed difference was evaluated by using the resampling method. In this case, the determined value is 6.14% representing a value larger than the accpeted threshold on 5%. While the result is still considered as valuable, the observed difference is not statistically significant (see Section 7.1).

Test of Hypothesis H2c

As in the previous hypothesis testing, versions implementing the main task only (main versions) were compared with versions including the additional tasks for the test of hypothesis H_2c. However, instead of impacts on the timing of the CAN messages, *content failures* in these messages were considered in this case. Nevertheless, those tasks activated by user buttons were not activated for the evaluation of this hypothesis but will be considered in the test of hypothesis $H2d$. In [98], we applied a manual comparison of the 20 versions based on the results of four different test benches. According to the formalized null hypothesis H_02c used for this work, the analysis itself had to be formalized. Therefore, the difference of the failure rates of the main and final versions was calculated for all versions and four test benches. Next, the average increase of the failure rates was calculated, which is depicted in the last column of Tab. 5.5. Positive values indicate an increase of the failure rate while negative values represent a decrease and therefore an improvement of the final versions.

It had been expected that the final versions lead to more failures according to their increased complexity. However, five of the final MCU versions (No. 2, 4, 6, 10, 11) showed fewer failures than the corresponding main versions while only two demonstrated more failures (No. 7 and 9). It has to be noted, that for this evaluation only values > 1 were considered as well as cases with no disagreement between the results of the four test benches. As a reason, we presume constant improvement of the code that represents the main function during implementation of the additional tasks. However, it has to be mentioned that one of the final MCU versions (MCU6) stopped all CAN activity if user button 1 was pressed and frequencies higher than ~33% of the maximum frequency specified were fed into the DUT, while the main version was working well for specified inputs. Regarding the FPGA versions, only one of the final versions showed significantly fewer failures (No. 11), while two revealed more failures (No. 2 and 5) than the respective main versions.

On basis of these values, the hypothesis has been tested as follows:

$$H_02c : Md[\Delta\lambda_{FPGA}(V_f, V_m, f_{TB})] \geq Md[\Delta\lambda_{MCU}(V_f, V_m, f_{TB})]$$
$$\Rightarrow 0.17\% \geq -0.04\% \Rightarrow agreement$$

As expected after the previous observation, H_02c could not be refused. Therefore, the hypothesis $H2c$ could not be accepted, which is in accordance with the results presented in [98].

Table 5.5.: Differences between failure rates of main and final versions for different test benches (values printed bold if value >1 or if values of TBs do not have different signs)

	$\Delta\lambda_x = \lambda(V_{final_x}, f_{TB}) - \lambda(V_{main_x}, f_{TB})$				
Version	TB2	TB4	TB8	TB9	Σ TB
MCU 2	0.00%	0.00%	-0.01%	-0.10%	**-0.11**%
MCU 3	0.00%	0.00%	0.09%	-0.06%	0.03%
MCU 4	-0.05%	0.07%	-1.55%	-0.33%	**-1.85**%
MCU 5	0.05%	-0.01%	0.54%	-0.04%	0.54%
MCU 6	-1.19%	0.00%	-0.03%	0.03%	**-1.20**%
MCU 7	0.00%	17.14%	6.13%	4.36%	**27.63**%
MCU 8	0.00%	0.05%	0.06%	-0.01%	0.10%
MCU 9	0.24%	0.00%	4.79%	-0.65%	**4.37**%
MCU 10	0.10%	0.01%	-40.27%	-3.61%	**-43.77**%
MCU 11	-0.24%	0.02%	-7.07%	-4.74%	**-12.03**%
median	0.00%	0.01%	0.03%	-0.08%	-0.04%
FPGA2	-0.05%	0.00%	2.41%	0.01%	**2.37**%
FPGA3	0.33%	0.05%	-0.05%	-0.01%	0.33%
FPGA4	-0.29%	-0.02%	-0.20%	0.43%	-0.08%
FPGA5	0.81%	-0.60%	-1.21%	29.35%	**28.35**%
FPGA6	-0.14%	0.01%	-0.02%	0.03%	-0.12%
FPGA7	0.48%	0.06%	-0.48%	0.03%	0.09%
FPGA8	-0.19%	0.03%	-0.13%	0.84%	0.54%
FPGA9	0.00%	-0.01%	0.18%	0.07%	0.24%
FPGA10	0.00%	-0.03%	-0.46%	0.31%	-0.18%
FPGA11	-0.05%	-0.01%	-9.19%	-1.03%	**-10.28**%
median	-0.02%	-0.01%	-0.17%	0.05%	0.17%

Test of Hypothesis H2d

In order to investigate how the execution of the additional tasks is influencing the main functionality, a new test bench (TB4b) that contained also test signals for the three buttons was used. In this test bench, all buttons were driven sequentially and also in combination. The analysis of the test results revealed that three MCU versions included failures according to button activities. In this context, the versions MCU4 and MCU6 were failing in case of an activity of button 1 and version MCU3 in case of an activity of button 3 (see also last column of Tab. 5.4). Otherwise, no FPGA version was affected by button activities. The test of the corresponding hypothesis $H_0 2d$ was conducted accordingly:

$$H_0 2d : N[F_{FPGA}(V_f, f_{TB_{button}})] \geq N[F_{MCU}(V_f, f_{TB_{button}})]$$
$$\Rightarrow 0 \geq 3 \Rightarrow disagreement$$

Thus, the null hypothesis $H_0 2d$ could be rejected leading to the acceptance of the hypothesis $H2d$. The corresponding significance of the difference between MCUs and FPGAs determined by resampling has a value of 0.0549. As this value is slightly higher than the usual acceptance value of 0.05, the significance of the difference is limited. Nevertheless, the result is considered as relevant as no FPGA version revealed failures according to button activities (see also Section 7.1).

Further analysis and interpretation of the results achieved by the four hypothesis tests can be found in Section 6.3.

5.3. Software Review on Different Hardware Platforms (Exp.3)

In contrast to the previous two experiments, no source code was developed for this experiment. Instead, existing final versions developed in the previous experiment on MCUs and FPGAs were evaluated by review in order to test hypothesis H3 described in Section 4.4.

5.3.1. Preparation

Participants

As described in Section 4.6.2, the review process was conducted with student assistants. Accordingly, three student assistants familiar with the experiment task and both hardware platforms were selected as reviewers[4]. According to the fact that all student assistants had participated in one of our previous experiments, no additional introductory course was needed.

Materials

The guidance for the review process was given in form of a written review guideline and a review report form (see Section 4.6.2). The review report form included a section in which specific aspects of the specification should be evaluated and a section in which the function of the implementation had to be evaluated for specific scenarios. The scenarios had to be determined by the first

[4]for organizational reasons, the third review had to be split on two different student assistants.

two reviewers themselves (only one example was given), while the third reviewer had to review the same scenarios as the first one (to ease comparability).

Ideally, all versions available would have been used for this evaluation. However, the execution of reviews is considered as time consuming, especially as we allowed only one review per day to avoid reciprocal effects. To avoid that a single reviewer is repeatably reviewing too many versions, either the number of reviewers had to be increased or the number of versions had to be reduced. As the number of suitable reviewers was very limited in our case, we decided for the latter case. Therefore, 12 versions (6 MCU and 6 FPGA) were selected randomly from the 20 final versions available from the second experiment.

5.3.2. Execution

For the execution of the review, each reviewer had three hours (at most 4 hours) time for each version. Every reviewer had to review all 12 versions in a given order as described in 4.6.2 while only one review was allowed per day.

In the review form, each reviewer had to fill out how long it took him or her to review each version. Further on, a grading of the quality of each version as well as a grading of the reviewability of each version had to be added.

Finally, a certain control of the review activities was given by the time frame of three hours and the review report form (the report required to investigate scenarios and to list features of the specification which had been tested).

5.3.3. Variable Measurement

Certain variables could be measured and achieved directly as the time needed for review as well as the grading of the reviewability of each version. For the testing of the number of faults identified correctly by review, the statements given in the review reports were tested by the automatic test environment described in Section 4.7.4. Review results for three different scenarios were considered in this experiment and the corresponding results can be found in the following section.

5.3.4. Hypothesis Testing

The identification of faults by review is investigated in this section in order to test the hypotheses $H3a$ and $H3b$.

Test of Hypothesis H3a

A comparison of test and review results has been conducted for the 12 versions reviewed regarding three different scenarios as described above. The corresponding results are depicted in Fig. 5.3 for the three scenarios and each of the three reviewers Rev.1, Rev.2, and Rev.3.

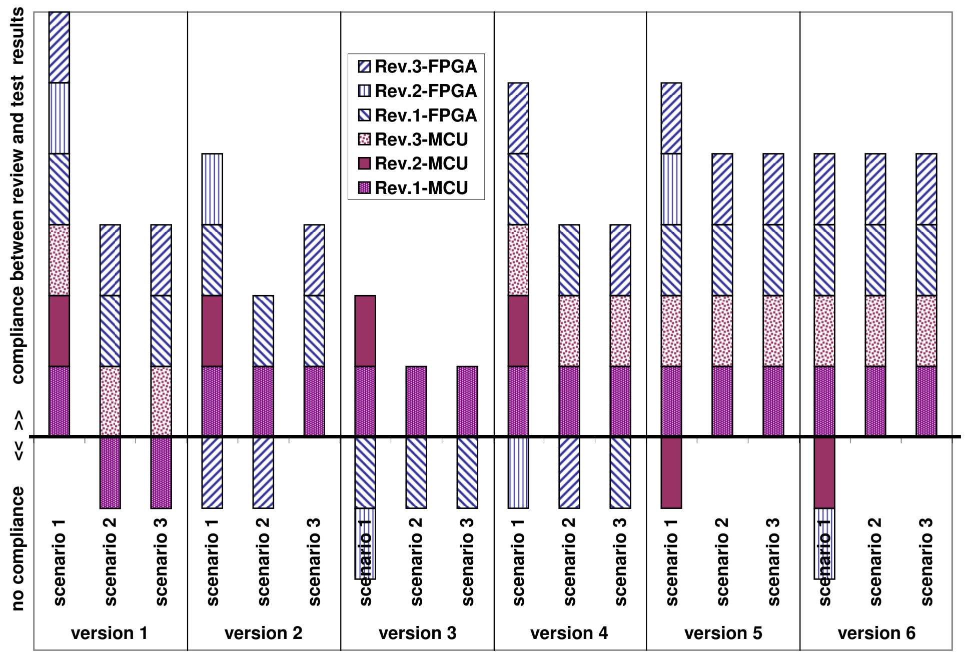


Figure 5.3.: Compliance of test and review results

The result of each evaluation (three reviewers and three scenarios) of each version (6 MCU and 6 FPGA versions) is represented in this figure by a box. All review results, which were in accordance with our test results, can be found above the zero line, while all results that led to different results can be found below this line. In some cases, a reviewer did not contribute to a specific scenario so that no results were included in the graphic for these scenarios. The hypothesis test compares the number of review results that are compliant with the test results:

$$H_0 3a : \sum_{1}^{i} \sum_{1}^{N} \sum_{1}^{n} \Psi_{FPGA}(R_i, V_{f_N}, S_n) \leq \sum_{1}^{i} \sum_{1}^{N} \sum_{1}^{n} \Psi_{MCU}(R_i, V_{f_N}, S_n)$$

$$\Rightarrow 29 \leq 32 \Rightarrow agreement$$

As no higher compliance could be observed for the FPGA versions, the null hypothesis $H_0 3a$ could not be rejected. Accordingly, hypothesis $H3a$ was not accepted.

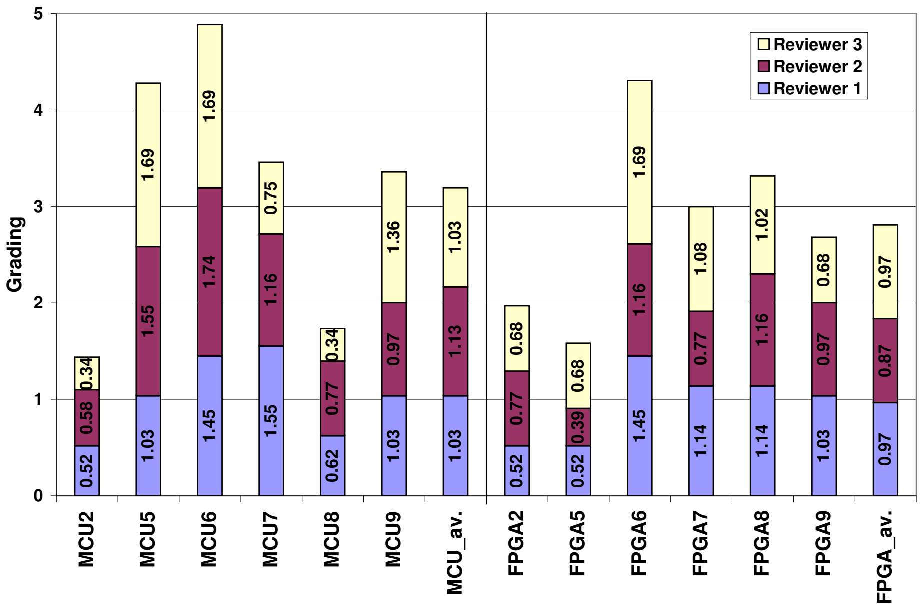


Figure 5.4.: Grading of the reviewability

Test of Hypothesis H3b

For the test of hypothesis H_3b, the grading of the reviewability given by all reviewers for each version was evaluated and presented graphically in Fig. 5.4 (grades: 1=very good,...,5=not possible). It has to be noted that the grading was normalized by dividing the values by the average grading of the corresponding reviewer. This normalization was applied to allow a better comparison between the different reviewers. As depicted in this graphic, very good as well as very bad gradings were assigned for MCU and FPGA versions while a team creating an MCU version with good reviewability did not necessarily create an FPGA version with good reviewability and vice versa.

The hypothesis test for H_3b, which compared the reviewability of versions implemented on MCUs and FPGAs showed the following:

$$H_03b : \frac{\sum_1^i \sum_1^N \Re_{FPGA}(V_{f_N}, R_i)}{i \cdot N} \geq \frac{\sum_1^i \sum_1^N \Re_{MCU}(V_{f_N}, R_i)}{i \cdot N}$$
$$\Rightarrow \frac{\sum \{0.97, 0.87, 0.97\}}{3} \geq \frac{\sum \{1.03, 1.13, 1.03\}}{3}$$
$$\Rightarrow 0.94 \geq 1.06 \Rightarrow disagreement$$

Therefore, H_03b could be refused. Again, the significance of the difference of the results for MCU and FPGA versions was determined by applying the resampling method described in Section 4.7.2. As the determined value is 0.176 > 0.05, the significance of the difference between MCU and FPGA versions is only limited. As the average grading of **each** reviewer was better for the FPGA than for the MCU versions, we still see a certain hint for a higher reviewability on FPGAs. Therefore, hypothesis $H03b$ was accepted in the end, although further investigations with higher numbers of reviewers are required to support this result (see Section 6.4.1 and Section 7.1).

Further analysis and interpretation of the results obtained by the testing of $H3a$ and $H3b$ can be found in Section 6.4.

5.4. Fault Removal by Test and Review (Exp.4)

While the previous experiments were based on two different hardware platforms, the investigations in this experiment focus on effects of review and testing on a single hardware platform as described in Section 4.6.3. The experiment was driven by hypothesis H4 (see Section 4.4 for details) aiming at a comparison of software versions improved by review and testing with those based on N-version programming developed in the first and second experiment.

5.4.1. Preparation

Participants

This forth experiment took place in a lab course as the first two experiments. In this case, 19 students (computer science, 5th semester or higher) participated forming 12 teams. Although only microcontrollers had to be programmed in this experiment, an introductory course was given prior to the experiment in the same way it was given in the first experiment. As the test environments, which the participants used for their testing activities, were based on FPGAs, back ground information for this device was considered as beneficial. Moreover, a brief introduction into testing and review was given.

Materials

Besides the materials provided in the second experiment (specification of the experiment task, the experiment guidelines, development guidelines and the initial and final questionnaire) further materials were required in this experiment. These additional materials included the test and review guidelines as well as review and test forms. Furthermore, a test environment had to be provided to the students for their testing activities. This test environment was based on a simplified version of the test environment used for evaluation testing in the previous experiments and is described at the end of Section 4.7.4.

Moreover, the acceptance test with the semi automated test environment used in the second experiment was applied in this experiment before and after the review and test phase.

5.4.2. Execution

The lab course, in which the experiment took place was comprised of 15 weekly appointments with three hours each. As in the first and second experiment, the students had to fill out the initial questionnaire before the experiment start while the final questionnaire had to be filled out after the completion of the experiment. The final questionnaire was modified as no FPGA implementation was required in this experiment. On the other hand, questions targeting the review and testing activities were added.

The students were asked to store their final and intermediate versions in the same way as it was done in the second experiment. Additionally, two acceptance tests were conducted on each version in this experiment, as it had been done in the second experiment (first acceptance test after the completion of the main task, second acceptance test after the implementation of all additional tasks). In this experiment, the versions of all 12 teams passed both acceptance tests and were considered for later evaluation.

In the following phase, review and test activities took part as described in the experiment design (see Section 4.6.3). After this phase, all participants had the chance to improve their versions based on the achieved review and test results. After these modifications, the acceptance test was conducted again to assure that the modified versions still fulfill the acceptance criteria.

The activity of storing the intermediate versions by the students was controlled as in the second experiment. Moreover, the test and review activities had to be controlled. This control was achieved by time slots given for both activities and by checking the test and review report forms for completeness. As the results should not be influenced by the experiment supervisor, the checks of the reports were limited to quantitative aspects (e.g. number of aspects tested from specification, number of scenarios used for evaluation, completeness of explanation fields). To improve the quality of the grading given by the reviewers and testers, each grading had to be explained. It was assumed that this explanation reduces those cases in which a grade is chosen randomly. Therefore, the completeness of these explanation fields was checked, too.

5.4.3. Variable Measurement

The most important variable in this experiment was the type and the number of faults identified in each version. Therefore, all review and test reports

No.	Problem	Type of problem	identified by	
			review	test
1	Spec.: Identify if one wheel is > 50% faster/slower than the other wheel on the same axis. *Problem: 50% faster ≠ 50% slower*	spec.	2	1
2	Spec.: Send a CAN message containing the peak speed values since the last request (…). *Problem: It is not defined if values occuring during test messages should be considered for peak value determination.*	spec.	1	1
3	As above, but: *Not defined if a second peak message must be sent if the peak message process is interrupted*	spec.	1	0
4	Spec.: Displaying of values on LEDs: *Not defined if LEDs must be switched ON or OFF in case of a logic 1*	spec.	2	0
5	Numbering of testmessage starts with 0x01 instead ot 0x00 (present in 5 versions, identified in 4 versions)	impl.	3	1
6	Fast changes in the input frequencies lead to response times longer than specified (present in all undebugged versions)	appl.	8	9
7	CAN transmit LED is activated in cases in which no CAN message is sent	impl.	1	0
8	No CAN messages sent if input frequency is zero or very low (present in at least two, identified in one version)	impl.	1	1

Figure 5.5.: Problems identified by review and test

were evaluated manually to determine whether problems known from previous experiments (especially Experiment 2) were identified. In this evaluation, two aspects had to be considered. First of all, it was of interest whether the students identified the problem itself (e.g. it could be challenging to fulfill a certain requirement in a certain situation). The second aspect was to determine, whether the identified problem was tested successfully in the corresponding implementation. To distinguish between these two aspects, a field was included in the test and review report forms in which the students could list all requirements they could not verify by review and/or testing.

A listing of the most important problems identified by review and testing can be found in Fig. 5.5. Moreover, the source of each problem was determined and included in this figure (spec. = specification specific problem, appl. = application specific problem, impl. = implementation specific problem).

Finally, all teams had the chance to improve their version on basis of the test and review results. To determine the quality of these improved versions, they were tested by using the automatic test environment. The failure distribution of these versions can be found in Tab. 5.6.

Table 5.6.: Mean and median values of the fraction of failures in debugged final versions (not all faults identified could be removed)

	TB2	TB4	TB8	TB9	TB4b
Mean MCU	1.73	1.88%	21.75%	6.16%	12.74%
Median MCU	0.8%	0.3%	3.9%	1.6%	3.7%

5.4.4. Hypothesis Testing

In this section, the hypotheses introduced in Section 4.4 are tested.

Test of Hypothesis H4a

The number of experiment teams, who identified the problem according to fast changes in the inputs by review and testing, can be found in Fig. 5.5. Therefore, the test of hypothesis $H4a$ (identification of failures according to fast changes in the input frequencies by at least 3/4 of the experiment teams, see Section 4.4) could be performed as presented below.

$$H_0 4a : \frac{N(V_f(F_{detected}))}{N(V_f(F_{detected})) + N(V_f(F_{undetected}))} < \frac{3}{4}$$

$$\Rightarrow \frac{4+3+5}{12} < \frac{3}{4}$$

$$\Rightarrow 1 < \frac{3}{4} \Rightarrow disagreement$$

As the failure was identified by review, testing or both techniques in case of all versions, the null hypothesis could be rejected, leading to the acceptance of hypothesis $H4a$. However, it has to be noted that the failure was even considered as detected, when just the problem of changes in the inputs had been identified and no further statements were given.

Moreover, the chosen experiment design allowed to increase the precision of the results by arranging that each team conducted a review as well as a test (see Section 4.6.3). However, learning effects during the first test or review activity on the verification of the second versions have to be considered. The problem was identified by 5 of the 6 initial review teams as well as by 5 of the 6 initial test teams. These results might have influenced the second verification part, but surprisingly the problem was identified in fewer versions in the second part of the verification (only 3/6 of review teams and 4/6 of test teams identified the problem). However, it has to be mentioned that the resulting failures according to this problem differ in intensity so that differences might exist in the possibilities of detecting these failures. Additionally, few review teams claimed that statements about the timing behavior of the code were not possible by review. To summarize, no undesired learning effects could be observed in this experiment.

Test of Hypothesis H4b

The four known ambiguous statements in the specification are listed in Fig. 5.5 (No. 1-4) together with the number of teams who identified these problems during their test and review activities. These values could be used directly to test hypothesis $H4b$ (the four known ambiguous statements in the specification are identified by at least 3/4 of the teams, see Section 4.4), respectively the corresponding null hypothesis:

$$H_0 4b : \frac{N(V_f(AS_{i_{detected}}))}{N(V_f(AS_{i_{detected}})) + N(V_f(AS_{i_{undetected}}))} < \frac{3}{4}, i \in \{1,2,3,4\}$$

$$\Rightarrow \frac{1}{12} < \frac{3}{4}, \frac{2}{12} < \frac{3}{4}, \frac{1}{12} < \frac{3}{4}, \frac{2}{12} < \frac{3}{4}$$

$$\Rightarrow agreement$$

In each case, only one or two teams identified each problem in the specification so that the null-hypothesis could not be refused. Therefore, hypothesis $H4b$ could not be accepted.

Test of Hypothesis H4c

Unfortunately, most students were not able to correct all faults identified within the given time. Especially the problem with the handling of fast changes in the inputs could not be solved by many teams. Thus, only minor corrections were applied successfully, which makes a suitable test of hypothesis $H4c$ impossible. Nevertheless, the test results obtained with the corrected versions showed improvements in case of the critical test benches TB8, TB9, and TB4b. In these cases, values lower than in Experiment 2 were determined for the medium and the arithmetic mean of the failure rates.

Further analysis and interpretation of the results obtained by the three hypothesis tests presented above can be found in Section 6.2.2.

5.5. Reusability on Different Hardware Platforms (Exp.5)

In this experiment a common specification was implemented on the same two hardware platforms applied in the second experiment. The difference to the second experiment conducted is the integration of an existing software component in the design (see Section 4.6.4). This integration enables the testing of hypothesis H5 described in Section 4.4.

5.5.1. Preparation

Initial Study

As described in Section 4.7.1, the software component, which had to be reused, was a life beat component (LB component). A first reuse of this LB component took place in an initial study conducted with one student assistant. During this first integration of the LB component, differences were discovered for the implementation on FPGAs and MCUs. These aspects will be discussed briefly in the following.

In case of the MCU, only a limited number of counter/timer units was available so that the frequencies of the four measurement channels had to be measured in a software based fashion (interrupts were applied). This approach had an influence on the measurement accuracy of the LB component. The LB itself was sampled periodically in the interrupt service routine of a timer unit, but the interrupt of this timer unit was assigned to a comparable low interrupt priority. In case of high interrupt activity in the measurement channels, this low priority led to measurement errors in case of the LB determination. To fix this problem, the interrupt priority of this timer had to be increased. Another problem in case of the MCU was that it had been forgotten to set back the timer after each *compare event*. While this fault led to problems in the new application, it was not detected in the original program, because this register was reset in the main routine in this case. After the problems identified were removed, the impacts of the LB component on the main application were tested. Tests with the corrected LB component showed no influence of the LB signal on the application. The reason can be seen in the timer unit used for this measurement, which allowed certain independence from other tasks executed on the common CPU. Eventually, the required integration of the functionality to read the status of the CAN controller[5] was comparatively easy to implement in case of the MCU.

In case of the FPGA, the original LB component could be reused by integrating the component in the overall circuitry. Undesired interactions between

[5]see also Fig. 4.11 for a description of the overall system.

the LB component and the remaining functions did not appear. However, the original LB component had to be modified in order to enable the read access to the CAN controller. The reason is the difficulty to access the interface to the controller directly from two processes within the VHDL design (additional measures are needed to assure that not both components are accessing this interface at the same time).

Finally, for the reuse of the LB component on the MCU, strict restrictions were needed to assure that the chip resources, which were reserved for the LB functionality, were not occupied by other components. On the other hand, no specific restrictions were needed in case of the FPGA. Only generic chip resources were used by the LB component on the FPGA and the allocation of these resources is done automatically by the corresponding synthesis tool.

Participants

The actual experiment took place in a lab course, which was hold in form of a two-week block course with 24 students (computer science, 5th semester or higher) forming 12 teams. As in the other experiments conducted in lab courses, we offered a mandatory two-day introductory course as described in Section 4.7.2.

Materials

In addition to the material provided in the second experiment (specification of the experiment task, the experiment guidelines, development guidelines and the initial and final questionnaire) further materials were required in this experiment. These additional materials included an interface description of the components to be reused in the MCU and the FPGA as well as guidelines for their integration. Finally, the components[6] for reuse had to be provided.

The acceptance test applied in this experiment was based on the semi automated test environment described in 4.7.4. In order to test also the basic functionality of the integrated software component, the test environment was extended by another output for the life beat signal. Moreover, failed acceptance tests were documented in this experiment as described in Section 4.6.6. For documentation, the sheets used for the acceptance tests were extended by fields for the time, the date and the reason of failed acceptance tests.

[6]Note: The same LB functionality was implemented in VHDL as well as in the language C.

5.5.2. Execution

According to the block structure of the lab course, this experiment took place at 9 appointments with 8 hours each. As in previous experiments, the initial questionnaire had to be filled out by the students before the start of the experiment while the final questionnaire was filled out after the completion of the experiment.

The procedure of acceptance tests as well as the storing of final and intermediate versions was conducted as in the second experiment, while a basic test of the life beat functionality was added to the acceptance tests as mentioned above. In this experiment, all 24 versions passed both acceptance tests and were considered for later evaluation.

5.5.3. Variable Measurement

For this experiment, all versions which passed the acceptance tests were evaluated by the test environment. Minor adaptations of the test environment had to be made to allow a testing of the additional software component integrated. These adaptations include a generation of the LB signal as well as the corresponding extensions in the analysis tools (see Section 4.7.4). A summary of the test results obtained with final versions is given in Tab. 5.7, while a more detailed listing of all failures observed can be found in Appendix A. Further small test benches were used to test the specific aspects of the hypotheses stated in Section 4.4. Results of two of these stress tests (TB12 and TB13) on the measurement results are included in Tab 5.7. It has to be noted that more than 50% of the failures listed for these two test benches did not occur in the measurement values but resulted from messages which had arrived too late.

Table 5.7.: Mean and median values of the fraction of failures made on different hardware platforms (final versions)

	TB2	TB4	TB4b	TB12	TB13
Mean MCU	11.82%	5.97%	32.65%	44.58%	40.22%
Mean FPGA	11.50%	3.96%	42.45%	45.63%	34.11%
Median MCU	0.52%	0.05%	27.56%	30.00%	28.00%
Median FPGA	3.20%	1.47%	33.25%	23.75%	14.67%

Moreover, the failures resulting in failed acceptance tests have been documented in this experiment. The overview of all failed acceptance tests presented in Fig. 5.6 includes the results of the 12 MCU and the 12 FPGA versions. Versions that were implemented first can be found in the left half of the table while the versions that were implemented second can be found on the right of

Team	1st version	acceptance test 1	acceptance test 2	2nd version	acceptance test 1	acceptance test 2
1	MCU	0	AT3(BTN)	FPGA	0	0
2	FPGA	0	0	MCU	0	0
3	MCU	MT (0/1)	AT3(blink), AT5	FPGA	0	AT6
4	FPGA	0	0	MCU	0	AT6
5	MCU	0	AT6	FPGA	0	AT2, AT3 (read)
6	FPGA	0	AT2, AT3 (read)	MCU	MT (init)	MT (overflow)
7	MCU	0	MT , AT5	FPGA	0	0
8	FPGA	0	AT3 (read)	MCU	MT (0/1)	0
9	MCU	MT (EMI)	0	FPGA	0	0
10	FPGA	0	AT6	MCU	MT (init)	AT3 (blink)
11	MCU	0	0	FPGA	0	MT*
12	FPGA	MT*	AT6	MCU	0	AT5

Note: MT = main task, AT = additional task, * = failure according to bad design practice

Figure 5.6.: Documentation of failed acceptance tests

the same table (see Section 4.6.4 for details of the *design of experiment*). The reasons for failing in the acceptances tests are indicated in this figure and will be described briefly in the following.

The first acceptance test was not passed in case of 5 MCU versions and 1 FPGA version. The version MCU3 failed to determine very low frequencies while the version MCU8 only showed this problem if a high differences was present between the input frequencies for the last two channels (Measurement of low frequencies had worked only if *all* channels received low frequencies). The version MCU6 did not pass the first acceptance test according to a wrong initialization of the CAN message contents and version MCU10 did not initialize one of the microcontroller ports correctly, leading to an incorrect LB-function. An interference problem, which could be mitigated easily later on, was the reason why version MCU9 did not pass the first acceptance test. The only FPGA version, which did not pass the first acceptance test, was version FPGA12. In this case, one of the measurement channels was not stable enough, resulting in faulty results. Bad design practice was the reason for these faults, which could be mitigated by a restructuring of the design.

In the following second acceptance test, the additional tasks were tested beside the main task. At the first try, 8 MCU and 7 FPGA versions failed in this second acceptance test for different reasons. First, problems in the main task occurred in the versions MCU6, MCU7 and FPGA11. The problem in the version MCU6 was that certain values remained zero after changes of the input values. The reason for this behavior was identified as an overflow in a variable. Next, version MCU7 failed the second acceptance test according to highly increased reaction times in case of changes in the measurement channels. In the version FPGA11, the CAN communication was stopped at a specific input frequency (1750Hz). This failure in sending included the LB component, which should have taken over the sending in case of errors in the remaining circuitry. The reason for this behavior has been identified as bad design practice (e.g.: different clock frequencies have been used without specific measures at the interfaces of the clock domains). While the circuitry still worked at the first acceptance test, the implementation of the additional tasks had changed the behavior of the main function. A redesign of the implementation removed this failure.

The remaining versions showed no failures in the main function, but in several additional tasks. In case of the additional task 2 (AT2: indicate *out of date values* in the CAN message), two FPGA versions (FPGA5, FPGA6) frequently indicated their measurement values causelessly as out of date. During the test of additional task 3 (AT3: if button is pressed: display status of CAN controller on LEDs, else: display input signals on LEDs in original frequency and a frequency divided by 16) it could be observed that two MCU versions (MCU 3, MCU10) had problems displaying the signals with 1/16 of the input frequency in case of low frequency input. Another team (MCU1) did not use the button at all and displayed the frequencies only. In case of three FPGA versions (FPGA5, FPGA6, FPGA8), the read access to the CAN controller did not work correctly and an incorrect CAN status was displayed on the LEDs for this reason. With respect to the additional task 5 (AT5: determine the peak values of each measurement channel and send them in a CAN message on request via a button), three failures were observed in the MCU versions. In the version MCU3, the interval of sending the CAN message containing the peak values was too long while the reset of the peak values did not work correctly in the versions MCU7 and MCU12. Moreover, the flag, indicating a message containing peak values, was not set correctly in version MCU7. Finally, bits indicating a difference in frequency greater 50% between two measurement channels (AT6) were set incorrectly in the versions MCU4, MCU5, FPGA10, FPGA12. The same was true for version FPGA3, but here the failure occured only if a difference had to be indicated between two pairs of measurement channels at the same time.

Beside the LB-problems in the versions MCU10 and FPGA11 mentioned above, no problems with the LB functionality could be revealed in the acceptance tests. As mentioned before, the results of failed acceptance tests are used within the later evaluation in Section 6.5.

Finally, a questionnaire had to be filled out by the experiment participants beside their implementation (see Section 4.7.2). In this questionnaire, students had to document the advantages and disadvantages of the two different hardware platforms they observed during implementation. Contents of these questionnaires will be used for evaluation in Section 6.7.

5.5.4. Hypothesis Testing

The hypotheses introduced for this experiment in Section 4.4 are tested in the following three subsections.

Test of Hypothesis H5a

In order to test hypothesis $H5a$ (functions implemented beside the reused function lead to less *content* failures in the reused function in case of the FPGA versions, see Section 4.4), test benches TB2 and TB4 were applied as test input for the four frequency measurement channels. The LB signal was taken from the fourth measurement channel in this case (the fourth channel and the input for the LB signal received the same frequency). Further on, the test bench TB4b was used to also test the additional tasks. In this context, the LB signal was switched on and off by a variable in the test bench. As a result, the LB function was working properly for all test inputs and in case of both platforms.

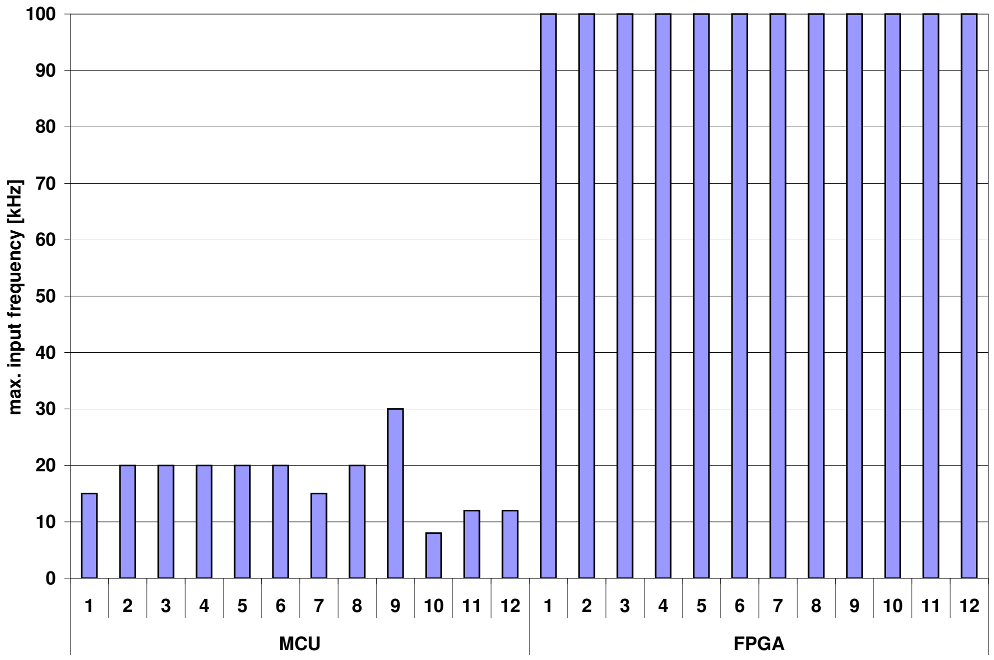


Figure 5.7.: Range of input frequencies (measurement channels)

Therefore, a new test bench was generated. In this case, two discrete frequencies (0Hz and 10Hz) were fed to the LB component while the frequency fed into the four frequency measurement channels of the main application varied from 0Hz to 100kHz ($\sim 16 \cdot f_{max}$). It has been assumed that the increased input frequency would lead to an increased computation time in case of the MCU versions and therefore influence the LB functionality. The effects of the input frequencies are depicted in Fig. 5.7. In this figure, the maximum frequency that could be fed into the measurement channels in order to remain a correct evaluation of the LB signal is displayed.

As introduced in the definition of the hypothesis, values up to 18kHz were intended. Based on this limit, 5 of the MCU versions failed in this test while the FPGA versions were not affected by the increased inputs. The results were used for the corresponding hypothesis test, in which the number of versions that failed in the described test is compared for the MCU and FPGA versions:

$$H_0 5a : N[F_{LB_{FPGA}}(V_f, f_M, f_{LB})] \geq N[F_{LB_{MCU}}(V_f, f_M, f_{LB})]$$
$$\Rightarrow 0 \geq 5 \Rightarrow disagreement$$

Based on these results, the null-hypothesis could be rejected as more failures occurred in the MCU versions. Thus the hypothesis $H5a$ was accepted. Again, the resampling approach was applied to determine the significance of the difference between the results achieved for the MCU and FPGA versions. As expected, the corresponding value of $0.011 < 0.05$ represents a very high significance of the results. Whether the MCU versions could have been designed in a better way to mitigate the problems in these versions is discussed as part of the evaluation in Section 6.5.

Test of Hypothesis H5b

Another aspect that was evaluated in this context was the interval of sending the LB messages. In the original application, the LB messages were sent every 109ms. In the new context, this timing was changed in case of 6 MCU versions while the timing was not changed in case of the FPGA versions (see Fig. 5.8). The values measured were used for the test of hypothesis $H5b$ introduced in Section 4.4. This hypothesis test is presented below (all values in [ms]):

$$H_0 5b : \frac{\sum_1^N |\Delta T_{LB_{FPGA}}(V_f, f_M, f_{LB})|}{N} \geq \frac{\sum_1^N |\Delta T_{LB_{MCU}}(V_f, f_M, f_{LB})|}{N}$$
$$\Rightarrow \frac{\sum\{0,0,0,0,0,0,0,0,0,0,0,0\}}{12} \geq \frac{\sum\{1,0,4,0,0,0,0,2,0,1,3,1\}}{12}$$
$$\Rightarrow 0 \geq 1 \Rightarrow disagreement$$

The null-hypothesis could be rejected, as the average change in the interval of sending was higher in case of the MCU versions. Accordingly, the corresponding

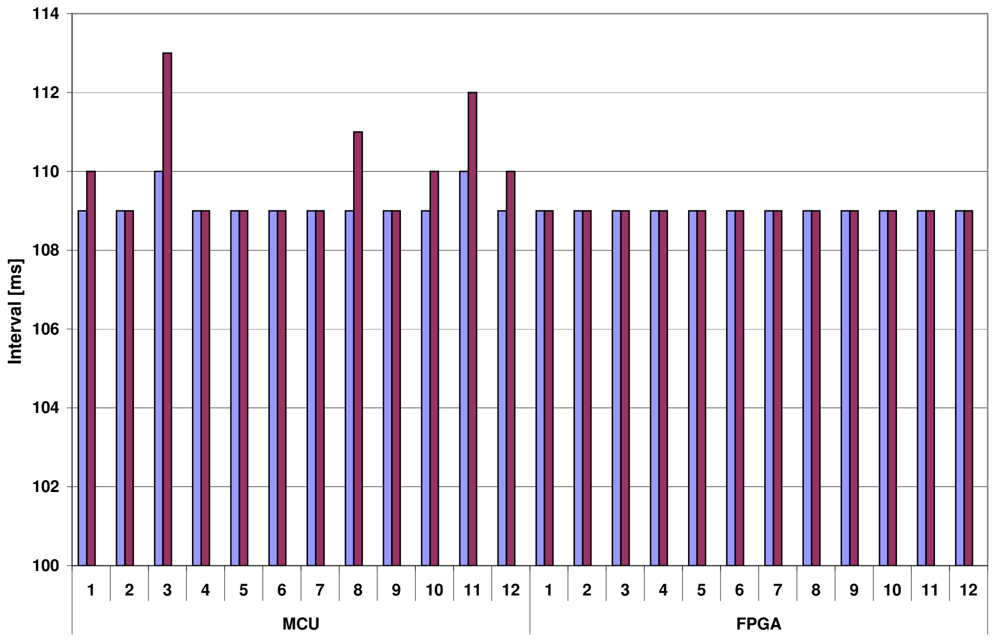


Figure 5.8.: Interval of sending LB messages (min. and max. values)

hypothesis $H5b$ was accepted. The significance of the difference between the results for the MCU and FPGA versions was determined as significant (Resampling approach: significance $= 0.013 < 0.05$). Moreover, further investigations were conducted to clarify whether the frequencies fed into the measurement channels also have an impact on the interval of sending of the LB messages. However, no dependencies between these four frequencies and the interval of sending could be determined.

Test of Hypothesis H5c

During the execution of the test mentioned in the previous sections, no impact of the LB function on the remaining functions could be observed. Thus, another test case was used that consisted of different frequencies for the LB signal ranging from 0Hz to >100kHz, while the frequencies fed into the measurement channels were kept at a moderate frequency ($\sim 0.5 \cdot f_{max}$). However, no impacts of the LB function could be identified. Therefore, the null hypothesis $H_0 5c$ introduced in Section 4.4 could not be refused. Hence, the hypothesis $H5c$ (inclusion of the function to be reused (LB component) leads to less failures in the remaining functions in case of the FPGA versions, see Section 4.4) was not accepted.

Further interpretation and analysis of the results presented in this and the previous two sections can be found in Section 6.5.

5.6. ISO26262 Development on Different HW Platforms (Exp.6)

This experiment was based on another experiment task than the previous experiments. Therefore, this experiment differs in preparation and execution as described in the following.

5.6.1. Preparation

The experiment task was more complex than in the previous experiments as described in Section 4.3.1. For this reason, the experiment could not be conducted in a lab course but was conducted within the context of two diploma theses.

Participants

The experiment was conducted with two computer science students within the scope of two diploma theses. Preconditions for the selection of the students were sufficient previous knowledge in the development of software and the particularities of embedded systems. To ease the work with the safety standard ISO26262 [48], an introduction to the structure and the basic principles of this standard was given before the actual experiment. Moreover, two student assistants were employed to assist the two students with the implementation work.

Materials

The most important material for this experiment was the specification document. While the task is based on a real application, only rudimentary specification documents in combination with a simulation of the functional behavior were available. The complete specification included confidential information and was not accessible for this reason. On basis of the limited material, a new specification document was developed. In contrast to the previous experiment, this specification document had to be adapted during the operation of the experiment, as a result of incompleteness of the specification and the existence of ambiguous statements.

Further materials were the safety standard ISO26262 [48] and the coding guidelines for the programming languages C and VHDL (see Section 4.7.3 for details), which were provided to the participants. Moreover, an operating system[7] with the corresponding documentation was provided.

Furthermore, a test and simulation environment was required for the development and test activities. This environment was developed alongside to the actual application and is further described in Section 4.7.4 and in [27]. A first working version of this environment was available in the last fifth of the experiment.

Finally, the development effort had to be documented. For this purpose, a suitable table (Excel sheet) was constructed. In addition, similar sheets were provided for the documentation of all failed acceptance tests.

5.6.2. Execution

As mentioned above, the experiment was conducted within the context of two diploma theses and had a duration of approximately 6 month. According to the design of the experiment presented in Section 4.6.5, the students developed the application in a first step following the safety requirements of the ISO26262. While the development activities are described in more detail in [92], the key aspects will be presented briefly in the following.

During safety analysis, two potential hazards were identified. First, the roof could start unintentional to open or close while the car is driving at high speed. This movement could result in the hazard that the roof breaks off and hits people and/or other vehicles on the street. Second, unintentional movement of the roof could clamp body parts of humans. Therefore, a fault tree analysis was performed to determine possible causes for these hazards. As depicted in Fig. 5.9, possible causes for the first hazard could be faults in the CAN message including the information of the vehicle's speed, faults in the control panel, and faults in the control unit itself. Similar causes are present for the second hazard as depicted in Fig. 5.10. In this case, faults in the CAN message including the status of the ignition key are an additional cause while the vehicle's speed is not relevant in this case. Based on these hazards, the safety goals were formulated representing the hazard free situations. Next, a safety concept was developed to achieve these safety goals, which is presented in Fig. 5.11. According to the safety analysis, the hydraulic pump that is responsible for any movement of the roof was identified as the safety-relevant actuator. Thus, a safety function was introduced which is monitoring the application and can disable the movement of this actuator. For the decision whether to disable the actuator or not, the safety function receives the corresponding information from relevant sensors

[7]Two different operating systems were applied which were both compliant to the OSEK standard. Further information on OSEK can be found here: http://www.osek-vdx.org

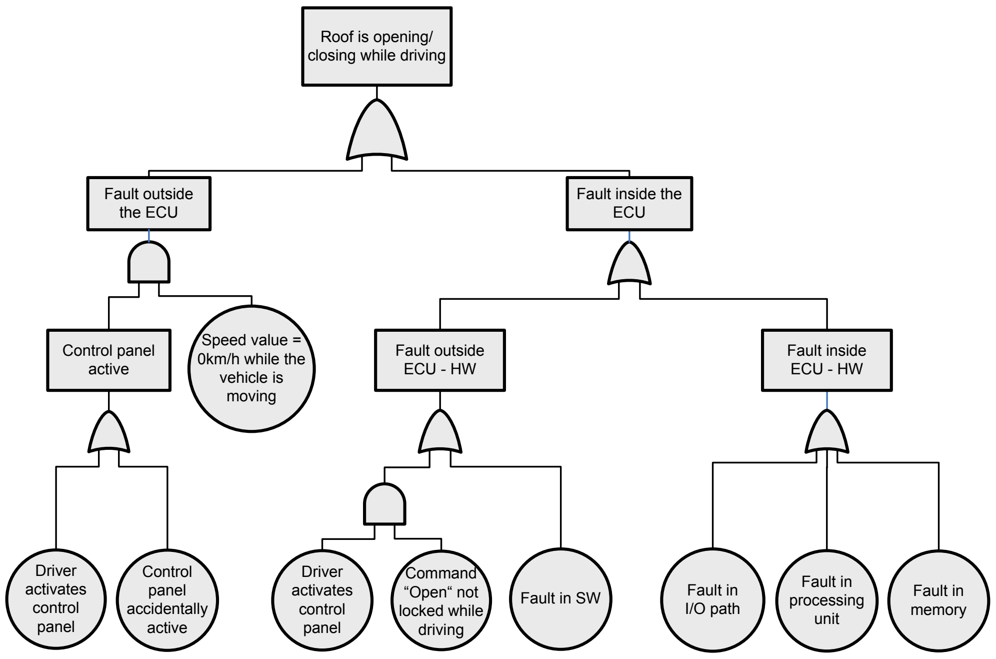

Roof is opening/ closing while driving
Fault outside the ECU
Fault inside the ECU
Control panel active
Speed value = 0km/h while the vehicle is moving
Fault outside ECU - HW
Fault inside ECU - HW
Driver activates control panel
Control panel accidentally active
Driver activates control panel
Command "Open" not locked while driving
Fault in SW
Fault in I/O path
Fault in processing unit
Fault in memory

Figure 5.9.: Fault tree analysis for first hazard

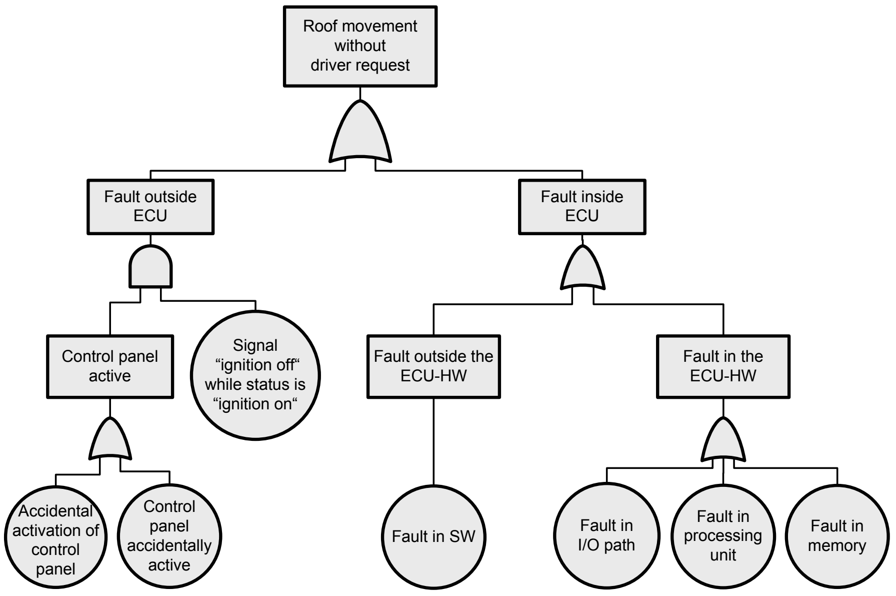


Figure 5.10.: Fault tree analysis for second hazard

or from the controller executing the application. In all cases in which safety-related information is received from the application itself (in this case: CAN messages), this information has to be checked for potential faults. These checks were achieved by including information redundancy into the CAN messages and checking the redundant information for consistency in the safety function. After completion of the safety concept, this concept was verified by another fault tree analysis including all safety measures introduced above. According to this analysis, the chosen safety concept was suitable to fulfill the considered safety goals (see [92] for details).

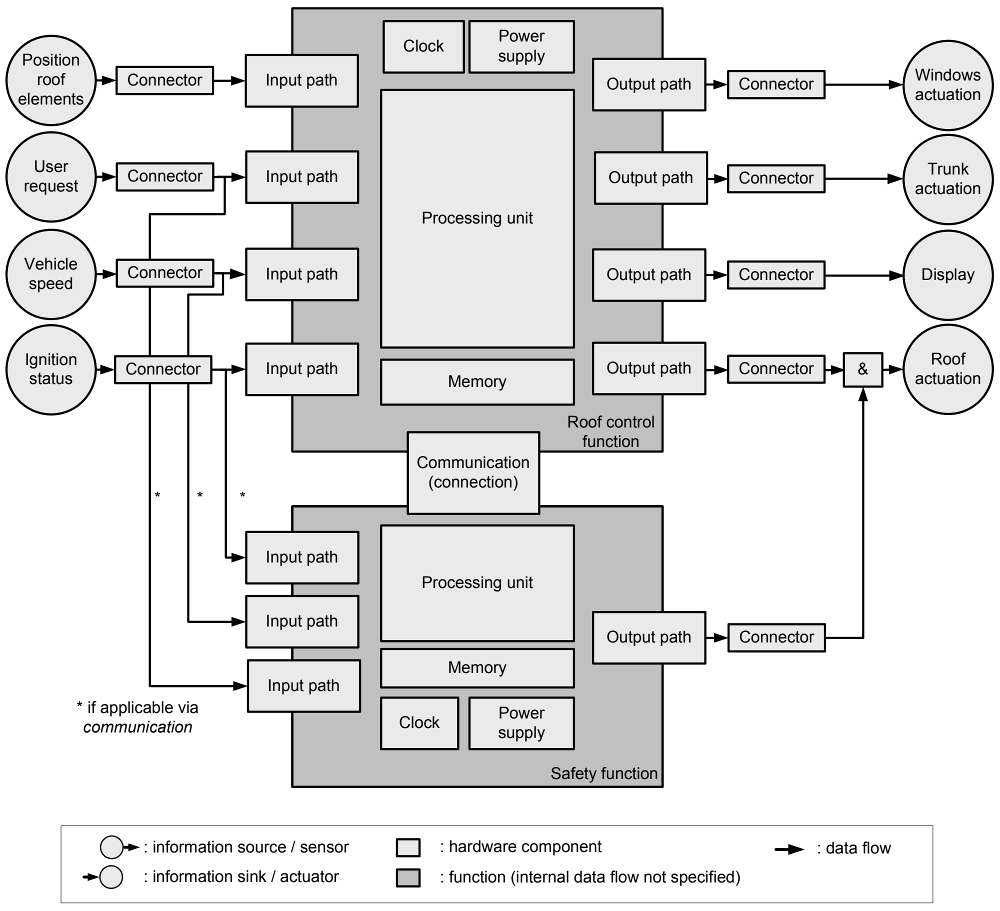


Figure 5.11.: Safety concept for the application

In the next step, a mapping of this safety concept on the two hardware platforms took place (design of technical safety concept). In case of the dual-core microcontroller, the application was mapped on one core while the safety function was executed on the second core. Additionally, the communication between the two cores was achieved via an exchange RAM available in this microcontroller device. During operation, both cores are equipped with their own program and data memories. However, common mode failures as faults in the common clock or power supply have to be considered. Therefore, a time windowed watchdog is triggered by the safety function to detect failures of the overall chip. Moreover, this watchdog includes a brown-out circuit performing a reset of the chip if supply voltages drop below a certain threshold. Further on, an operating system was used only on the first core while no operating system was applied on the second core. The decision not to use an operating system for the core executing the safety function was made for the following two reasons: First, an operating system is comparatively complex and therefore complicates verification activities. Moreover, the complexity of the safety function was low, which was the second reason an operating system was considered as not useful. Further details of this implementation can be found in [113].

For the combination of MCU and FPGA, the application was executed on the MCU as it was suited best for this hardware platform while the safety function was mapped on the FPGA. Moreover, the communication between the two devices takes place via a serial connection (SPI). To enable a safe communication, information redundancy was applied in combination with message counters. As both devices can be provided with independent clock and power supplies, no additional measures for the handling of common mode failures are needed. As in the case of the dual-core microcontroller, an operating system was applied only on the MCU executing the roof control application. Additional details of this implementation can be found in [17].

Finally, both technical safety concepts were evaluated regarding the requirements given in the ISO26262. Accordingly, two fault metrics[8] and the achieved diagnostic coverage were determined for both architectures. As no sufficient reliability data was available for the devices applied in this experiment, data available for similar components was used for an estimation. As a result, both technical safety concepts could fulfill the safety requirements (see [92] for details of the fault metrics and the determination of the diagnostic coverage).

After the design and implementation phase, which took about 4.5 months, the student assistants started to test both implementations with the test environment. As mentioned before, acceptance criteria present in the specification were used for testing. Moreover, fault injection was used to test the effectiveness of the implemented safety mechanisms. Faults identified during test activities were documented and removed in this phase. Further on, additional properties of the implementations (see Section 4.4) were evaluated by the students.

[8]Single Point Faults Metric and Latent Faults Metric

5.6.3. Variable Measurement

According to the design of the experiment described in Section 4.6.5, the development progress as well as existing development problems were documented and discussed with the experiment participants in weekly meetings. Moreover, the time needed for each implementation was measured. The testing of the acceptance criteria revealed several faults in the roof control application in case of both implementations. However, these faults could be removed later on. During the fault injection tests[9] applied for the testing of the safety measures, all faults could be handled correctly. Further properties of the implementations were analyzed later on. For an evaluation of the reliability, the behavior of the developed systems in defined fault scenarios was compared, while the modifiability was evaluated by given scenarios of modification.

5.6.4. Hypothesis Testing

The main hypothesis of this experiment stated differences in the development effort between the two hardware platforms ($H6a$, see Section 4.4). However, the safety concept developed for both hardware platforms presented in Fig. 5.11 differed only with respect to the implementation of the safety function on the two hardware platforms. As described in Section 5.6.2, this safety function was implemented on the second core of the dual-core microcontroller in one case. In the other case, the FPGA was used for the implementation of this function. In both cases, only minor problems occurred during implementation. In case of the dual-core device, certain challenges resulted from using the exchange RAM for communication between the two cores. In case of the MCU+FPGA approach, the implementation of a successful SPI connection was challenging. However, both problems were identified as "beginner problems". The main application was developed on a microcontroller core in both cases and determined the majority of the development effort. Therefore, the development effort was very similar for both hardware platforms. Finally, the time for implementation was measured. However, only small variations were present between the two groups. As only two teams were used, this difference was not significant. Therefore, it is obvious that the hypothesis $H_0 6a$ could not be refused.

Although not formulated in form of hypotheses, the remaining aspects were evaluated. The evaluation and further discussion of the results gained in this section are present in Section 6.6.

[9]Note: 20 fault types were injected into 6 components (ROM, RAM, I/O, CPU, CAN, supply voltage). The fault type "overvoltage", all clock faults, and *stuck at faults* in the memories were not tested for technical reasons.

6. Evaluation of Empirical Results

This chapter includes an evaluation of the results obtained by the variable measurement and hypothesis testing in the previous chapter. The evaluations are centered around the six main hypotheses and are presented in the sections 6.2-6.6. Before the actual evaluation, general threats to the validity of these results are discussed (Section 6.1). Furthermore, potential validity threats that are relevant for particular experiments only are considered within the sections of the corresponding experiment evaluations. Moreover, we summarize development problems we observed during the operation of our experiments in Section 6.7.

We published the results gained by Experiment 1 in [106]. Results of the second and third experiment can be found in [98] while parts of the results achieved by the forth experiment were published in [96, 97]. Finally, the results of the fifth experiment were presented in [101].

6.1. General Threats to Validity

The general threats to internal and external validity are discussed in this section. In this context, *internal validity* represents the correctness of the experiment itself while *external validity* represents the portability of the results to other applications [126].

6.1.1. Threats to Internal Validity

As mentioned in Section 4.7.2, the control of variables is a major concern in experiments. Since independent development of all experiment versions is a precondition for the evaluations, the possibility of plagiarism has to be considered. This consideration includes the exchange of ideas for solving certain problems but also the copying of complete parts of source code. The following measures were taken in the experiments 1, 2, 4, and 5 to overcome this problem: First, the students were asked explicitly to work on their own and to ask the experiment supervisor in case of any problem. As discussed in 4.7.2, the supervisor did not provide any concrete solutions, but offered help to find an individual solution (e.g.: recommended to test all modules separately to find out which one is not working). Second, a software tool[1] was used to search

[1]JPlag, URL: https://www.ipd.uni-karlsruhe.de/jplag/

for eventual plagiarism in the final MCU versions. This tool was also used in exercises and identified plagiarism in this context. Otherwise, it did not indicate any critical matches in our experiment outcomes. Versions with increased compliance were inspected manually, but the similarities found resulted from the common specification. In case of the FPGA versions, no tool was available which could consider the syntax of VHDL as it was the case for the language C. Therefore, the analysis could be performed on text level only. However, no critical matches were found by this evaluation. Independent activities could be also achieved in the remaining two experiments. In Experiment 3, the reviewers did not work at the same time and possibilities of exchange were therefore very low. Finally, the risk of uncontrolled exchange in Experiment 6 is considered as low as only one team implemented the application on each hardware platform.

Further on, the previous knowledge of the experiment participants might be a threat to validity. According to a questionnaire conducted at the beginning of all lab courses, students were generally more experienced in C/MCUs than in VHDL/FPGAs. This difference was tried to adjust by a two-day introductory course prior to the experiment, which is described in Section 4.7.2. Additionally, the personal contentment was rated similarly by the students in the final questionnaires for both implementations.

Another aspect we observed in the final questionnaires was the amount of work the students performed at home. In case of the experiments conducted in weekly lab courses, a high amount of the students also worked on their implementation at home. On the other hand, less than 50% of the participants worked on the code at home in case of the block course (experiment5). Since the amount of work conducted at home was similar for both treatments in most cases, the impact of this aspect is considered as low. As the main focus of these experiments was not the development effort but the quality of the implemented versions (assured by acceptance test), we see no threat on validity in this aspect.

Moreover, one compilation of VHDL code took up to 2 minutes while the C code was compiled in a few seconds. While this aspect might have influenced the development, no threat to internal validity is seen in this aspect since it is a given difference between FPGA and MCU programming.

6.1.2. Threats to External Validity

With respect to generalization of the experiment results, the results could depend on the type and the complexity of the task applied. Therefore, a minimum complexity of the experiment task is required to allow representative results. The task chosen for the experiments conducted in the lab courses included several real-time requirements (four channel frequency measurement, communication, etc.) typical for embedded systems. While the complexity with respect

Version	1	2	3	4	5	6	7	8	9	10	11	12
MCU	480	483	510	376	995	787	383	500	413	362	446	-
FPGA	-	689	747	384	871	862	574	394	424	616	623	505

Figure 6.1.: Lines of code, excluding comments and blank lines (Exp.2)

to the lines of code (see Fig.6.1 for final versions of Experiment 2) was slightly lower than in a comparable experiment [63], sufficient challenges were present in this application according to several concurrent real-time tasks. Hence, we are of the opinion that the given requirements are fulfilled by our experiment task, but additional experiments are desirable to minimize remaining dependencies between the task and the results. Finally, the application used in Experiment 6 is representative as it is taken from a real automotive application. The application was reduced only slightly in complexity to allow its implementation within the scope of a diploma thesis.

Another important threat to external validity is the quality of the participants regarding experience and development knowledge. Although this problem is applicable to our experiments, the aim here was to show general differences of effects and not to give an absolute assessment. In this context, using students to test the initial hypothesis is a viable approach referring to [115]. Additionally, it is stated in [19] that no difference of programming expertise between professional and non-professional developers could be found, while [45] states that at least last-year software engineering students and professional software developers have a comparable assessment ability.

In the experiments 1, 2, 4, and 5, most teams chose an interrupt based approach for their MCU implementation. However, the task might allow a cyclic scanning approach (static scheduling) which could have a positive impact on the aspect of encapsulation in MCUs (e.g. computation would depend less on inputs). However, this approach is suitable for periodic input signals only or requires very short cycles resulting in a need for higher CPU frequencies. For this reason, we expect no threat to external validity in this aspect, but nevertheless, an investigation with this alternative approach would be desirable.

6.2. Software Diversity and Fault Removal (H1,H4)

While in the sections 6.3 to 6.6 the results of single experiments are evaluated, this section represents a combined evaluation of results achieved in the experiments 1, 2, and 4. In the following subsection, the aspect of software diversity is discussed (based on experiments 1 and 2), while this approach is compared with the approach of testing and review (based on Experiment 4) in Section 6.2.2.

6.2.1. Software Diversity by Diverse Hardware (H1)

Our first experiment revealed an extremely low likelihood for the property of failure independence between pairs of MCUs and CPLDs (H1a, see Section 5.1.4). While an improvement of the failure independence by using diverse hardware platforms could be shown (H1b, see Section 5.1.4), the corresponding effect was small.

The results of both hypothesis tests indicate the existence of difficulties that remain difficult even in different development approaches. Therefore, the versions of the first experiment were inspected with respect to failures identified during evaluation. Moreover, this failure analysis was conducted also on the versions created in the second experiment. The corresponding failures are depicted as part of Fig. 6.2 and are discussed in the following sections.

Failures in Experiment 1

The hypothesis testing on the versions developed in Experiment 1 showed high numbers of dependent failures (see Section 5.1.4). For the analysis in this section, failures identified were examined and listed in Fig. 6.2 (Exp.1). The first failure mentioned in this table was present in the first messages after reset in all MCU versions and one CPLD version. The behavior after reset was not explicitly specified (resulting in the same requirements as during runtime) which could have led to this failure. Another reason might be improper initialization methods, especially in case of the MCUs. According to the parallel structure of the CPLD and the fact that less initializations are needed in case of this type of hardware (no interrupt handlers, timers, etc. have to be initialized), this specific fault occurred only in one CPLD version. Therefore, forced diversity by different hardware platforms was successful in this case.

The following failures (No. 2-4) result from not considering certain input situations like very low, very high or rapidly changing inputs. These failures occurred in many MCU and CPLD versions, however, differences can be observed between both hardware platforms. In the CPLD versions, the use of different measurement intervals was usually avoided (most probably because it was complicated to implement on this hardware platform). For this reason, changes of the input frequencies were handled faster with CPLDs (No. 4), coming at the cost that the accuracy in case of low input frequencies is insufficient (No. 2).

Finally, cross talk between cables carrying the measurement signals was a problem in most CPLD versions while this problem did not occur in any MCU or FPGA version (No.13). While external measures[2] could mitigate the crosstalk, two CPLD versions still had major problems in processing four different input

[2]Note: Filter elements and shorter cables were applied

No.	Failure description	Type of problem	# versions with this failure in				
			Exp. 1		Exp. 2		Exp.3
			MCU	CPLD	MCU	FPGA	MCU
1	Wrong or delayed CAN messages after reset, especially if the input signal frequency is high.	Impl./ Spec.	100%	10%	36%	20%	25%
2	Wrong values in the CAN message as soon as input signals are of very low frequency (<5Hz). → measurement interval is probably too short	Appl.	33%	80%	18%	40%	25%
3	Wrong values in the CAN message as soon as input signals are close above a certain threshold* → Overflows or wrong determination of maximum output value	Impl./ Appl.	42%	20%	**0%**	**0%**	**0%**
4	Missing or delayed CAN messages or messages with wrong values if subsequent input values change quickly.	Appl.	83%	40%	91%	100%	75%
5	Wrong or delayed results as soon as different values are fed into the four measurement channels while changes of subsequent input signal values are limited to 200Hz → et al.: faulty determination of the measurement interval	Appl.	n.a.	n.a.	82%	90%	75%
6	Test cases with only equal test values at all inputs do not always lead to 4 identical results (not necessarily a failure).	Impl.	33%	30%	64%	20%	42%
7	Device stops sending of CAN messages as soon as the input frequency has been above a certain threshold*	Impl.	**0%**	**0%**	9%	**0%**	**0%**
8	No messages after reset if the input signal frequency is low	Impl.	**0%**	**0%**	**0%**	**0%**	8%
9	Device stops sending of CAN messages as soon as two buttons are pressed sequentially with high frequency → probably leading to an undefined state in state machine	Impl.	n.a.	n.a.	**0%**	10%	**0%**
10	Testmessage counter does not always start with 0 as specified → at least in some cases: faulty interaction between testmessage counter and sending method	Impl.	n.a.	n.a.	27%	60%	17%
11	Testmessage counter is not incremented correctly (is incremented by more than 1) → faulty interaction between testmessage counter and sending method	Impl.	n.a.	n.a.	**0%**	60%	**0%**
12	User input via buttons changes the number of wrong values for the worse → real-time properties affected by user input	Impl.	n.a.	n.a.	27%	**0%**	8%
13	Crosstalk between measurement channels leads to failures in the measurement	Impl.	0%	90%	0%	0%	0%
		number of versions used for analysis:	12	10	11	10	12

*close above the maximum input signal frequency explicitly specified if no button in pressed
Impl. = Implementation, Appl. = Application, Spec. = Specification
Exp.3: versions have been debugged according to individual test and review reports

Figure 6.2.: Failures found during evaluation

frequencies (see test results of TB3s in Annex A). Thus, different hardware platforms allowed an increase of the failure diversity in this specific case.

Failures in Experiment 2

A more complex task was applied for Experiment 2 as presented in Section 4.3.1. To avoid the comparatively simple errors found in the first experiment (overflows, etc.), we conducted a stronger acceptance test in this second experiment. The results can clearly be seen in Fig. 6.2 (Exp. 2, No. 2 and 3): Overflows were detected by the acceptance test as well as most of the problems with low frequency input. As before, a high number of dependent failures occurred as presented in Fig. 6.2 (Exp. 2, No. 4, 5, 6). Moreover, new requirements (additional tasks and a stricter acceptance test) increased the problems of response times and accuracy, especially in FPGA versions (compared to CPLD, No. 4). Further on, a dependent failure could be identified in the additional tasks (No. 10). Several versions did not implement the test message counter as specified so that it starts with an incorrect value (failure is described in more detail in Section 6.2.2). The second failure within the test message (No. 11) resulted from insufficient interaction between parallel processes in FPGA versions.

Obviously, some fault sources identified are *implementation dependent* (No. 1, No. 6-13) while others seem to be *implementation independent* (No. 2-5). According to our results, the N-version programming approach applied might be more suitable to mitigate implementation specific faults than to deal with implementation independent faults. Therefore, a classification of fault sources was conducted, which is presented in the following section.

Fault Classification & Limitations of N-version Programming

During evaluation of the experiments 1 and 2, it became obvious that different sources exist for the failures found. Those failure sources (faults) have been identified as follows:

- **Specification specific faults:** the specification was misleading, incomplete or ambiguous. Moreover, statements in the specification could be even wrong, but this was not the case in our experiments.
- **Application specific faults:** application specific problems and challenges have not been understood and thus have not been handled sufficiently (e.g. forget to handle a certain scenario/input constellation).
- **Implementation specific faults:** specification and application specific problems have been identified correctly, but faults have been made during implementation (e.g. incomplete case structure).

The failures found in the experiments have been analyzed with respect to these fault categories and many *implementation specific faults* identified in Fig. 6.2 could be mitigated by our approach of diverse hardware NVP. Otherwise, even implementation specific faults as No. 10 (test message counter, see Section 6.2.1) occurred in several versions developed independently on diverse hardware platforms.

Whereas it is stated in [10] that the specification must be correct to allow a successful application of NVP, potentials of NVP to deal with *specification specific faults* are seen in other publications [39]. Some specification specific faults could have been tolerated by diverse hardware NVP in our experiments if one hardware platform guided to the correct implementation, as it was the case for the failure No.1 (Fig. 6.2) in the first experiment. Further specification specific faults will only be found if different teams interpret the specification differently. According to our results, different developers interpret the specification differently, but for us this is no hint that the **majority** of the resulting versions is correct. For this reason, NVP might uncover specification problems, but redundancy concepts based on majority voting, as for example a two out of three (2oo3, TMR) system, would probably be no solution to mask the specification specific faults present in our experiment data.

As a third fault category, we introduced *application specific faults*, which revealed to be a challenging problem in NVP. Despite the immense effort put into different development processes, languages and programming styles by using completely different hardware platforms in our NVP experiments, several problems remained the same in all implementations leading to identical wrong results in several cases. These problems (No. 2-5, Fig. 6.2) result from the application itself, are implementation independent and thus cannot be avoided by this approach of NVP. Results of another NVP experiment published in [28] indicate that "lack of understanding by the programmers of key points in the specification was a major contributor to faults that caused coincident failures". This result goes well with our observations that NVP is not suitable to mitigate these implementation independent problems.

Accordingly, the approach of NVP applied for these experiments seems only useful to mitigate implementation specific faults. Nevertheless, even a dependent implementation specific fault was identified in versions developed independently on different hardware platforms. Therefore, possible solutions to this problem are discussed in Section 6.2.4. Nevertheless, recent publications as [21] support the application of NVP for mission-critical applications. According to their investigations, a supportive evidence for NVP could be provided. While our intention is not to neglect positive effects of NVP, our aim is to emphasize the specific limitations of this approach. As mentioned above, the most critical limitation is that certain faults are closely related to the specification and the application itself. This relation makes a mitigation of these faults very unlikely, even by the improved NVP approach presented in this thesis.

Otherwise, the failures present in the majority of the versions created in these two experiments seem to be found easily as soon as they have been identified once. Furthermore, we had added ambiguous statements to the specification which had not been identified by any experiment participant in the first and second experiment. Thus, the potentials of review and testing to identify these problems are discussed in the following section.

6.2.2. Software Diversity vs. Fault Removal (H4)

As part of the hypothesis testing in Section 5.4.4, the most common failures identified in Experiment 4 during the review and test activities were evaluated. Hypothesis testing investigated two aspects. One aspect referred to ambiguous and unclear statements in the specification (hypothesis $H4b$). While these problems occurred in every review and testing process, each of them was revealed only by one or two teams during their test and review activities. Therefore, the hypothesis $H4b$, stating that at least 3/4 of the teams would identify these problems, could not be accepted. One might wonder why they were not identified by all or at least the majority of the teams. However, ambiguities could have been identified only if the verification team understood the specification differently than the team that implemented the code or if the ambiguity itself was identified. Obviously, this difference in understanding or the identification of the problem in the statement itself occurred only in the minority of the cases.

On the other hand, hypothesis $H4a$ stating that at least 3/4 of the teams would identify the application specific problem according to fast changes of frequency values at the inputs could be accepted (see Section 5.4.4). Obviously, this fault was comparatively easy to detect. Thus, it seems interesting that the problem was not identified during the implementation activities. It is assumed that it will be more efficient if not the own code is evaluated but the code developed by a different team.

Moreover, implementation specific faults in one or more versions have been identified (see No. 5, 7, 8 of Fig. 5.5 in Section 5.4.4). Of special interest is problem No. 5, as it occurred in several versions: A test counter had to be realized that starts with 0x00 and increments by 1 with every CAN message sent. In 5 of the 12 undebugged versions, the test counter did not start with 0x00 but with 0x01. One reason for this failure was that the line of code for the incrementation was placed incorrectly (incrementation took place before the counter value was read for the first time). This problem was identified in 4/5 of the versions, mostly by review.

In a last step, all experiment participants had the opportunity to improve their versions. The results are listed in the last column of Fig. 6.2 (Exp.3). While many failures were identified and removed, the mitigation of some failures (as No. 4 and 5 of Fig. 6.2) would have needed major changes in the software architecture. According to the limited time for these changes, many final versions of Experiment 4 still contain faults. In this context, the inability to deal with fast changes of the input values within the specified time is a fault that remained in most versions.

In the following, the results of our application of NVP are compared with those of the fourth experiment in which review and testing were applied for reliability improvement. Originally, this comparison was planned by testing hypothesis $H4c$ (see Section 5.4.4). As certain faults could not be removed, the versions were compared with the help of the observed failures presented in Fig. 6.2 and Fig. 5.5.

As expected, diverse hardware NVP allowed to mitigate most of the *implementation specific faults* (e.g. No.7-9, 11, 12 in Fig. 6.2). One exception was the implementation of the test counter (No. 10). This easy task was faulty in several versions for the reasons already described. In case of review and testing, many, but not all implementation specific problems were identified. In case of the non real-time tasks, reviews uncovered more faults while testing was more successful in case of the real-time functionalities[3]. Thus, all three approaches showed potentials regarding the mitigation of implementation specific faults.

Application specific faults were present in the majority of all versions, even in those created on different hardware platforms. Some of these application specific faults occurred less often on the first hardware while others were found less often on the other hardware. However, these differences were usually small (exceptions are No.2, Exp.1 and No.6, Exp2. in Fig. 6.2). For this reason, only low to medium potentials are expected for diverse hardware NVP to mitigate application specific faults. On the other hand, testing and review discovered most application specific problem as No.6 in Fig. 5.5. With respect to application specific faults, testing showed slightly more advantages in comparison to review. The reason is that also unexpected faults were identified during testing while reviewers typically concentrated on finding known problems in the code.

Finally, *specification specific faults* were a problem for all of the three approaches. In case of diverse hardware NVP, only few specification specific faults could be avoided as described above. In case of testing and review, all known specification problems were identified, but in several cases only by a minority of the teams (No.1-4 in Fig. 5.5). Review seemed to have the highest potentials to reveal specification specific problems, since the specification had been analyzed

[3]Note: This aspect is supported by the review reports as 7 out of 12 teams stated problems in determining real-time properties by review.

closely for review, while it was only used for test case generation in the test process.

Comparisons of the software fault tolerance by NVP and the fault removal by verification activities have been published before. An analytical evaluation was presented in [83]. In this work, the effect of reliability growth on the dependence between the failures of diverse software versions was investigated. As a result, there is currently no evidence that the choice between design diversity and other means of reliability improvements can be decided by *general* arguments. We agree that each individual case has to be analyzed to allow a suitable decision. However, argumentation could be improved in our opinion by separating the comparison in the fault categories proposed above. Results of another empirical evaluation comparing fault tolerance and fault removal are presented in [112]. The techniques applied were *code reading, static analysis, software tests* and *back-to-back voting* based on the versions developed independently. As a result, multiversion voting is not considered as a substitute for functional testing and should therefore not be reduced when using this software fault tolerance technique. Additionally, multiversion voting tolerated different faults than were detected by fault removal techniques. Both results support our findings. While the focus of our work was to compare NVP with alternative approaches, further analysis and comparisons of these fault removal techniques as well as references to further experiments investigating this aspect can be found in [112].

Summarizing, diverse hardware NVP, review, and testing showed different potentials of fault mitigation with respect to the three categories of failure sources. While further empirical results are needed to help designers of safety-critical embedded software to apply the optimal combination of reliability improvement approaches, the presented results are considered as a first step in this direction. Moreover, the categorization into *specification specific, application specific* and *implementation specific* faults allows a more systematic comparison of alternative approaches.

6.2.3. Threats to Validity

All major potential threats to the validity of the results of the experiments presented are included in Section 6.1. A further impact on the internal validity could be seen in the common background of the experiment participants[4]. However, this threat is considered as uncritical for the investigation of the NVP approach, especially as the intention was to force the diversity by different hardware platforms.

[4]Note: Most of the participants were in the final stage of their computer science study at RWTH Aachen University.

Finally, education of the students with respect to test and review activities was limited to theoretical knowledge in most cases. Although an introduction into the topic as well as review and test guidelines were applied in Experiment 4, this lack of experience might be another threat to external validity. Nevertheless, it is expected that the typical benefits and limitations of the investigated approaches could have been determined in this experiment.

6.2.4. Alternative Approaches for Software Diversity

A solution to the problems of dependent failures in versions developed independently might be the approach of *functional diversity* described in [62]. This approach offers higher independence of failure behavior according to different functionalities implemented in the diverse software versions. An example is the implementation of an equal function with different functional ranges as proposed in [20]. The example given in this work is an autopilot, which is implemented with full functionality in one version (allows smooth movement). In the second version, this function is implemented with reduced functionality including only the functionality to keep the plane in the air. All failures according to the higher complexity of version 1 are therefore not present in version 2.

Another option is to realize a function with different functional principles as described in [62]. An example for this approach is the safety function of a chemical reactor, which should prevent an explosion of the tank. This prevention is achieved by shutting down the reactor if the temperature is too high in one version while the reactor is shut down if the pressure is above a critical threshold in another version. While this approach clearly leads to the highest level of independence, it might not be applicable in every application. Moreover, this approach still comprises certain risks of common mode failures [62].

6.3. Impact of Hardware Platforms on Encapsulation (H2)

In our second experiment, potentials of encapsulation were evaluated for software developed on MCUs and FPGAs. In this context, four hypotheses regarding the encapsulation of subtasks ($H2a - H2d$) have been tested as described in Section 5.2.4. According to the test of the hypotheses $H2a$ and $H2b$, the timing of the subtask *CAN communication* is affected by changes in the measurement functions as well as the inclusion of the additional tasks in more MCU than FPGA versions. Moreover, different impacts on the contents of the CAN messages occurred. On the one hand, higher failures rates in the contents of the CAN messages according to the inclusion of the additional tasks ($H2c$) could not be shown for MCU versions. On the other hand, the execution of the additional tasks that are activated via user buttons had only an impact on the contents in the CAN message in case of the MCU versions, while the FPGA versions were not affected ($H2d$).

The failures observed in the context of the additional tasks can be grouped in failures, which represent faults in the measurement itself and those which represent faults in the sending of the CAN messages. Accordingly, no FPGA version revealed any faults in the measurement caused by user inputs while three MCU versions contained major failures according to button inputs (MCU6: stopped CAN communication, MCU3 and MCU4: content failures in CAN messages, see also 4th column of Tab. 5.4). Otherwise, one MCU and four FPGA versions revealed failures[5] in the sending of CAN messages as soon as more than one button had been pressed.

Even though some additional tasks revealed higher potentials for encapsulation in FPGAs, the overall number of failures in the final FPGA versions was increased in comparison to the main version while final MCU versions tended to reveal less failures than their respective main versions. Further examination of the failures revealed advantages for encapsulation of real-time tasks on FPGAs if the functional interactions between the functions implemented was limited. This aspect can be clarified with the following example: The encapsulation between the frequency measurement/value processing and the CAN bus communication was successful on FPGAs, as only the values measured and processed by the measurement/processing component were given to the CAN communication component at defined points in time. Otherwise, stronger interaction between functions (as in case of additional tasks 2 and 3) tended to result in an undesired coupling between the components in several FPGA versions in our experiment. Summarizing, encapsulation seems to work better on FPGAs than on MCUs if several real-time functions with only limited functional interactions have to be implemented.

Finally, the aspect of encapsulation was not forced by explicit requirements in this experiment. The effect of encapsulation was evaluated in a given implementation without the explicit requirement of the independence of the different subfunctions. Therefore, encapsulation might be different, if this aspect is somehow forced, e.g. by reusing a given component that is not allowed to be changed. This aspect is discussed in Section 6.5.

[5]Common failure: As soon as two buttons are pressed, only the function with the highest priority is executed (as specified). However, if only the button with the highest priority is released, the function with the lower priority is not executed (probably edges are identified as inputs only). Moreover, two FPGA versions (FPGA5 and FPGA7) do not start sending CAN messages immediately after all buttons are released resulting in missing messages.

6.3.1. Threats to Validity

A discussion of all major potential threats to validity is included in Section 6.1. Furthermore, a minimum complexity of the task is necessary to allow reasonable encapsulation. According to the presence of several real-time tasks in the application used in this experiment, this requirement is considered as fulfilled. Therefore, no further threats to validity are present in this experiment.

6.4. Impact of Hardware Platforms on Reviewability (H3)

In the hypothesis testing in Section 5.3.4, a clear advantage of FPGAs over MCUs with respect to reviewability could not be identified. While the reviewability was rated better on average for the FPGA versions by every reviewer ($H3b$), the quality of the review results ($H3a$) was not higher in case of the FPGAs.

However, we revealed differences in the problems during review[6]. Several problems regarding the review of the MCU versions resulted from interrupts and difficulties in determining the execution times of the subtasks. As an example, the determination of the CAN transmission interval by review was exact for the FPGA versions, while only intervals were given for the MCU versions. Moreover, the use of a graphical representation for the structural descriptions in the VHDL (generally, the top level of the design was a schematic describing the interconnection of the individual VHDL components) facilitates the first steps of understanding the FPGA versions. Such a description of the concurrent functions was not available in the code of MCU versions, which required a description of these issues in the documentation of the code. Otherwise, certain properties of the FPGA versions were not recognized correctly. The reason is seen in problems of understanding of the behavior of several interconnected parallel processes in the FPGA versions.

Nonetheless, solutions might exist to improve the reviewability in case of both hardware platforms. The application of coding guidelines might be an approach that could improve the readability and understandability of C code as well as VHDL code (see e.g. [25, 68]). Further measures as the application of an operating system might improve the reviewability of the MCU software. However, the verification of the correctness of the operating system itself is a challenge in this approach [117] so that operating systems available for safety-critical applications are often suitable for low safety integrity levels only.

[6]mostly by the comments given in the review report forms

Table 6.1.: Time needed for review

Version	Reviewer 1	Reviewer 2	Reviewer 3	Mean
MCU2	2	1.5	3	2.17
MCU5	3	2.5	4	3.17
MCU6	3	3	3.5	3.17
MCU7	2	2	2.8	2.27
MCU8	0.5	1.5	3	1.67
MCU9	1.8	2	3	2.27
FPGA2	1.7	1.75	3	2.15
FPGA5	1.1	1.5	4	2.20
FPGA6	2.5	3	4	3.17
FPGA7	1.5	1.5	3	2.00
FPGA8	1.5	2	3.5	2.33
FPGA9	1	2.5	3	2.17

Finally, the time needed for the review is depicted in Tab. 6.1 for each version and reviewer. While a time frame of about 3 hours was given, it has been possible to terminate the review earlier if the reviewers were sure about their results (see Section 5.3). According to the results presented in Tab. 6.1 and Fig. 5.4, most versions that have been reviewed for a comparatively short period of time received a comparatively good grade for their reviewability. However, no general correlation could be determined between these two aspects.

6.4.1. Threats to Validity

Beside the potential threats presented in Section 6.1, the number of reviewers in this experiment could have an impact on the results (e.g. determination of the significance). While three reviewers conducted the reviews for this experiment, a higher number would be beneficial to support our results. However, we do not expect fundamental changes in the results.

The selection of the scenarios for the evaluation of the review might have influenced the test of hypothesis H_3a. Since the same scenarios were applied for both platforms, we expect no major threat to validity of this selection.

Another aspect is the review technique being used. In contrast to formalized reviews, as for example *code inspection* described in [30], only three reviewers worked independently in this experiment and the guidance of the reviewers was limited to the review report form and a brief review guideline. While the internal validity is not affected by this aspect (both platforms were reviewed by the same technique), the external validity is limited to the review technique used in this experiment.

Moreover, the different skills in the programming languages might have influenced the structure and complexity of the resulting codes. This influence might have affected the results with respect to reviewability and might have an impact on the internal validity for this reason. Nevertheless, it is assumed that higher skills in the VHDL language would not have led to codes with a lower reviewability.

6.5. Impact of Hardware Platforms on Reusability (H5)

In this section, the reuse of a given real-time function is evaluated for MCUs and FPGAs. The evaluation is based on the results gained by hypothesis testing in Section 5.5.4 as well as on a discussion of further review scenarios.

6.5.1. Results of Hypothesis Testing

As described in Section 5.5.4, possible side effects of the reused function on the remaining functions were evaluated by feeding increased frequencies into the reused life beat (LB) function. The idea was that the increased value could increase the processing time needed for the determination of the LB signal and thus influence the overall system (test of $H5a$). But in case of both hardware platforms (MCU and FPGA), no impact of the increased LB frequency on the main functionality or the LB function itself could be determined. This independence has been expected for the FPGA versions since the evaluation of the LB signal was realized in dedicated logic and parallel to the remaining tasks. In case of the MCU, the independence was achieved by using a timer unit as mentioned before. Based on this timer unit, the LB signal was sampled at dedicated instances in time, which made this operation independent of the input signal. While the low frequency of the valid LB signal (10Hz) made this approach possible in this application, the measurement of a LB signal with a higher frequency would require higher sampling rates. To handle these increased sampling rates, higher clock frequencies are required for the MCU or another timer unit has to be applied for the count process. On the contrary, the implementation on the FPGA is not affected by the frequency of the signal measured (if the frequency is sufficiently smaller than the operating frequency of the FPGA).

Regarding the impacts of the remaining functions on the LB function, impacts could be determined during the test of hypothesis $H5b$. While no impact could be shown for the FPGA versions, all MCU versions were affected by high inputs for the measurement channels. Moreover, this effect clearly depended on the specific implementations of the individual teams as different maximum frequencies could be determined for the 12 MCU versions. Finally, the test of hypothesis $H5c$ revealed changes in the interval of sending LB message in case of the reused software components. These changes occurred on the MCU versions while the interval remained constant in all FPGA versions.

To summarize, side effects between the reused software component and the remaining functions could be determined in the MCU versions only.

6.5.2. Discussion of Further Reuse

The experiment considered the reuse of the LB component in a new but similar context. According to the specific requirements in the domain of embedded systems, this simple reuse already represented certain challenges. However, to determine potentials for effective reuse, possibilities of further reuse have to be examined (see e.g. [59]). For this reason, further reuse scenarios are discussed in the following.

Reuse scenario 1

New functional requirements (e.g. new algorithm, new user interface etc.) make a change of the hardware platform necessary. The interesting aspect is how easy the transfer of the given LB component to the new target is. The LB component implemented on the FPGA is device independent (device independent VHDL description). An easy transfer to another FPGA device is possible for this reason. A change of the MCU platform can have a high impact on the LB component. Since specific hardware resources are used in the LB component (counter unit, I/O interface, interrupts), the C code has to be adapted with respect to those resources which change from one MCU device to another. If both, the original and the new device belong to the same microcontroller family, only limited changes are required. On the other hand, major adaptations might be necessary if the design is transferred to other types of microcontrollers.

Reuse scenario 2

While the determination of the correctness of the LB signal is limited to the length of the high and low phase of the signal, a more complicated evaluation might be needed in another application. An example could be a specific pattern (e.g. 100ms high, 100ms low, 50ms high, 50ms low,...). This requirement would complicate the evaluation in case of both hardware platforms, but the implementation should be possible on the MCU as well as on the FPGA. If the evaluation of the signal needs significantly more computation time, problems could arise in case of the MCU (side effects with other tasks running on the same CPU).

Reuse scenario 3

In this case, not one but two or more LB signals have to be evaluated. In case of the FPGA, the part of the LB component implementing the LB test could simply be multiplied. The results of the additional LB tests would have to be included into the CAN messages then. The LB component implemented in the MCU samples the LB signal in a timer interrupt service routine. Therefore, additional LB signals could be sampled in the same routine. However, it has to be noted that the evaluation of further signals would increase the execution time of this interrupt service routine which might have an impact on other real-time functions.

Reuse scenario 4

While the considered application is only able to send CAN messages, the main function should also be able to read CAN messages in this final scenario. For this purpose, the receive buffer of the external CAN controller can be read via the read function implemented in the LB function each time a new incoming message is signaled (e.g. via interrupt line). In case of the MCU, the function for reading CAN messages could be integrated easily, e.g. in the main function by another function call. However, a bottle neck in CPU time could occur if a high rate of incoming messages has to be processed. For the FPGA versions, this function could be included easily in the LB component itself (easy access to all signals required) while an access outside the LB component would require the design of a corresponding interface managing the multiple access to the CAN controller interface. The development of this interface would probably require an increased development effort. On the other hand, the data transfer from the CAN controller to the main application could be implemented in dedicated hardware on the FPGA, which would reduce the dependability with other tasks present in the application.

6.5.3. Discussion of Results

The results of the initial study showed that the use of interrupts could have an negative influence on the reuse of real-time software components. While this problem is not present in FPGAs, the compatibility of reused components with the new context has to be checked especially careful for all components using interrupts. The inclusion of the read functionality indicated that the implementation of new functionalities in the FPGA could request modifications of the interface of the reused component. The reason is, in contrast to the MCU, that the write access of more than one component on a specific signal has to be implemented explicitly. Further on, a documentation of all on-chip peripherals used by a component has to be included in the reuse guidelines of an MCU component. This documentation is usually not needed in FPGAs as illustrated in Section 5.5.1.

The hypotheses $H5a$ and $H5b$ were accepted since it could be shown in the experiment evaluation that the LB function was affected by the remaining functions in several MCU versions but in no FPGA version that has passed the acceptance test. Reasons for the side effects in case of the MCU resulted from the common CPU used by all components and the use of interrupts (needed in this kind of resource restricted hard real-time application). On the other hand, the evaluation of failed acceptance tests also showed undesired side effects in case of the FPGAs. In one of the FPGA versions, these side effects even disabled the CAN communication of the LB component and the remaining functions. The cause was identified as bad design practice as aforementioned. Therefore, is has to be assured that design guidelines are followed strictly in FPGA design to avoid these side effects.

Otherwise, hypothesis $H5c$ tested in the experiment could not be shown. Moreover, even the evaluation of the failed acceptance tests could not identify impacts of the LB component on the remaining system. It had been assumed that the additional LB function would have a negative impact on the remaining real-time functions on the MCU. However, on-chip peripherals, in this case the timer unit, allow a sufficient computing of parallel tasks, which is only possible in parallel hardware or several CPUs otherwise. Thus, the application of further on-chip peripherals might also have avoided the side affects observed during the hypothesis testing of $H5a$ and $H5b$. Nevertheless, these on-chip resources have only limited flexibility and their number is usually limited. They also lead to a strong interdependency between hardware and software components which could lead to problems if a change of the hardware platform is needed, as has been discussed in reuse scenario one in the previous section. Moreover, the reuse scenarios two and three (see Section 6.5.2) identified potential bottlenecks in CPU time if the evaluation of the LB signal has to be changed.

Further on, challenges for reuse were identified in reuse scenario four. If the functionality outside the LB component is changed in a way that it requires a read access to the CAN bus, this change could affect the MCU as well as the FPGA versions. In the first case, the changes in the LB component are not considered critical, but the additional computation time needed for the read functionality has to be provided in case of the MCUs. While the read functionality could be implemented independently from the remaining system in the FPGA, changes in the internal structure as well as in the interface of the LB component would be needed in this case.

6.5.4. Threats to Validity

Potential threats to the validity of the results are presented in Section 6.1. Furthermore, the components used for reuse in this experiment could not be identical for MCUs and FPGAs (one was written in C while the other had to be written in a hardware description language). Differences in these two components, which were not given by the different hardware platforms, could have influenced the experiment results (threat to internal validity). To overcome the problem of specific advantages/disadvantages in one of the two components, both components were adapted to the new context by a single developer with great care (as described in Section 5.5.1) and have been reviewed by another developer. Nevertheless, further experiments reusing software components different from the one used in this experiment are desirable to support the results gained in this experiment.

6.6. Impact of HW Platforms on ISO26262 Development (H6)

For the evaluation of potential impacts of different hardware platforms on the development according to the standard ISO26262, a common safety-critical application was implemented on two diverse hardware platforms. During the hypothesis testing in Section 5.6.4, no differences in development effort could be identified between the two hardware platforms investigated. However, the actual application and the safety function are more similar in case of the dual-core microcontroller than in case of the platform that includes the MCU and the FPGA. This similarity is beneficial for the development effort, as parts of the application can be reused for the safety function. However, higher diversity between the application and the safety function might be desirable with respect to fault diversity. Which aspect is considered as more important depends on the individual application and the complexity of the safety function. As mentioned before, the complexity of the safety function implemented in this experiment is considered as low. Thus, no considerable benefits for the development effort could be observed in this case.

Further on, an aspect that is considered to have a strong impact on the development is the chosen functional safety concept. Therefore, potential impacts of the safety concept are discussed in the following section. Moreover, differences between the implementations with respect to reliability and modifiability are discussed in the sections 6.6.2 and 6.6.3.

6.6.1. Impact of Functional Safety Concept

The most important difference between the two hardware platforms is the implementation of the functional safety concept. In both cases, an approach was chosen that is based on the same safety concept (see Section 5.6.2). Only the mapping of the safety concept depicted in Fig. 5.11 on the two hardware platforms resulted in differences described in Section 5.6.2. The safety function was implemented on the FPGA in case of the MCU+FPGA architecture and on the second core of the dual-core approach. The development effort for both approaches did not differ significantly as mentioned before. The reason is that the implementation of the main application, which was very similar on both platforms, required most development resources.

An obvious disadvantage of the safety concept chosen is the dependence on the application. Thus, a new implementation of the safety function is required for each new application as discussed in Section 6.6.3. Moreover, the success of this approach depends on a sufficient safety analysis. Only aspects identified as safety-related in this analysis are included in the safety function. Thus, aspects not considered in the safety analysis are not covered by this approach.

An alternative approach is not to monitor the behavior of the application, but to monitor the behavior of the hardware executing the application. Accordingly, this approach could be realized independently from the application. An example would be to use the second core of a dual-core microcontroller to monitor the behavior of the first core. An intuitive option to achieve this monitoring is to execute the same application on both cores and then to compare their results. This way, a fault in one of the cores can be detected as the results of both cores differ in this case. Comparison of the results could be achieved by comparisons between the two cores at defined points in time. While this approach is possible by applying general purpose devices, the implementation of the comparison in software has a negative impact on the performance (impact depends on how often comparisons are required). Otherwise, a *lock-step approach* could be chosen (see e.g. [35]). In this approach, the results of both cores are compared after each step of computation by dedicated logic, which is implemented in the dual-core device. Furthermore, the second cores could periodically perform application independent checks on the main core as proposed in [18]. However, if one of these approaches is chosen, all possible faults in the hardware that could violate the safety goal have to be considered. Therefore, not only faults in the two cores have to be handled, but also faults in the memories, the bus systems, the I/O peripherals and further on-chip resources. While this comprehensive fault handling is possible in general, it generally requires further logic (specific chip, see e.g. [35, 107]) or is resulting in an run-time overhead. Moreover, the verification of sufficient fault coverage for all components is expected to be more challenging than the verification of sufficient functional supervision.

Another drawback of the generic approach is that the handling of hardware faults without the consideration of the actual application does not allow to distinguish between critical and uncritical faults. Approaches of fault recovery could partly overcome this problem (see e.g. [107]), but each uncritical fault that cannot be recovered by these measures is resulting in an undesired and unnecessary reduction of the availability of the system. Finally, only hardware faults could be mitigated by this approach, while the approach of functional supervision could also handle most software faults. It cannot be assumed that all software faults are handled as dependent faults might be present in the application and the supervision function. Nevertheless, functional diversity (see Section 6.2.4) could usually be achieved between application and safety function, which is expected to sufficiently reduce the risk of dependent failures.

Table 6.2.: Comparison of fault handling approaches

	functional supervision		device supervision
−	application dependent	+	application independent
−	mitigates only aspects considered in the specific safety concept	+	considers faults in the microcontroller hardware independently from the safety concept
+	allows handling of SW faults	−	no handling of SW faults
+	knowledge of criticality of faults	−	no knowledge of criticality of faults
+	intuitive fault handling concept	−	complex fault handling concept

An overview of the advantages and disadvantages of the two approaches compared above is given in table 6.2. Thus, the disadvantage of the approach chosen is the dependency of the supervision function on the application. As discussed in Section 6.6.3, the impact of this disadvantage is application specific. Another disadvantage might be the dependence of this approach on a successful safety analysis identifying all safety-related aspects, while the generic approach allows to tolerate certain aspects independently from the safety concept (example: one of the motors in Fig. 5.11 turns out to be safety-related, but is not considered in the safety concept. While the safety function implemented in the chosen approach does not handle any fault that leads to a critical movement of this motor, the generic approach could at least handle hardware faults in the microcontroller that could lead to this critical behavior). Summarizing, the decision for one of these approaches depends a lot on the application as well as the costs and the technological maturity of devices that allow device supervision. At the moment, we see significantly lower development effort in our approach compared to the alternative considered, at least if the application is comparable to the one investigated for this work (specific constellation of values could violate the safety goal, safe state by deactivating the safety-related actuators).

6.6.2. Impact on Reliability

For an evaluation of the reliability[7], the behavior of the developed systems was compared for defined fault scenarios[8]. However, only minor differences could be observed, which is not surprising as the same safety concept was applied in both systems. Nevertheless, the following differences could be observed. First, the communication between the MCU and the FPGA is considered as less reliable than the on-chip communication on the dual-core microcontroller, as soldering points and longer transmission lines are involved in this communication. Second, the clock of the MCU is not supervised in the MCU+FPGA approach, while an external watchdog is applied in the dual-core approach. Although this missing clock supervision is considered as a drawback on the reliability of the application, the supervision could be implemented subsequently on the FPGA with low effort. As a third aspect, the FLASH memory containing the program code of the application is not protected in the MCU+FPGA approach and is only checked during system start in the dual-core approach. However, the second core could perform cyclic checks of this memory and therefore increase the reliability of the dual-core approach. Summarizing, the differences between the considered platforms are low, but slight advantages of the dual-core approach could be determined with respect to the reliability of the implementation.

6.6.3. Impact on Modifiability

Finally, the modifiability was evaluated by given scenarios of modification. The first scenario is a change from a *semi automatic* roof control to a *full automatic* roof control. As the roof is moving in this case even without the driver constantly pressing the button, the hazard that a human is clamped in the roof mechanic has to be handled by measuring the torque during roof movement. Therefore, the measurement of this value has to be integrated into the safety function. Moreover, the original safety function has to be modified in a way

[7]Reliability is defined in this context as the probability that the system is acting as specified.
[8]The following scenarios were considered: Faults in the RAM, the ROM, the SPI connection, the exchange RAM, the I/O path, the CAN controller, the CPU, the clock, and the power supply (see [92] for details).

that the movement starts with the first activation of the button and has to stop with the activation of the *stop button* or by reaching an end position. As only single aspects of the safety concept have to be modified while the overall structure remains unchanged, it is expected that these changes in the safety function could be implemented easily.

A second scenario is the implementation of a door control unit responsible for the movement of the side windows. Again, the safety function has to control all safety-related signals. While the application is different, the same concept as developed for the original application can be used in this case for both hardware platforms (monitoring of user panel and torque, deactivation of motor in case of hazardous conditions).

Summarizing, the safety concept developed in the experiment could be easily transferred to other applications if the safety concept is similar (monitoring of safety-relevant signals, deactivation of safety-related actuators if signals indicate a violation of safety goals). Otherwise, modifications might require more effort if the safety function is more complex, as it is, for example, expected in most driver assistance applications.

6.6.4. Threats to Validity

As discussed in Section 6.1, the task selected might have an impact on the external validity. While the task selected for this experiment is certainly complex enough, impacts of a task modification on the results have to be considered. In this case, the results are transferable to applications that allow a comparable safety concept, as discussed in Section 6.6.3.

Moreover, the knowledge and the skills of the experiment participants is a potential threat to the validity. The participants had a comparatively long time to get familiar with the design environments and hardware used. Therefore, we see no major impact on validity in this aspect, especially as we investigated only an effect (e.g. treatment A leads to better results than treatment B without further quantification of the differences). Furthermore, the participants were not experienced in the application of the safety standard ISO26262. While this lack of experience might have influenced the development time, we see no threat to the validity of our results in this aspect, especially as both approaches were affected in the same way.

Finally, only two teams participated in this experiment for the reasons described in Section 5.6. Thus, the results might depend on the individual experiment participants. While we expect no changes in the results obtained, further experiments as the one presented are desirable to support our results.

6.7. Identified Development Problems

The development of different applications on different hardware platforms identified certain development problems. While most problems were of general nature (e.g. the application specific problems described in 6.2.1), several problems were specific for the hardware platform applied.

A first difference was observed during debugging activities of the experiment participants. Debugging activities on the MCU could be performed on the actual hardware by applying breakpoints and the in-system emulation hardware. This procedure seemed intuitive and each time students had to debug their implementation they could analyze their code by step-wise execution. However, the application of this approach was limited during later design stages, as an increasing number of interrupts complicated this sequential analysis of the code. Nevertheless, the participants could usually mitigate this problem by switching off all interrupts that did not contribute to the function investigated or by using more than one breakpoint. Debugging activities were a little different on the dual-core microcontroller applied in the last experiment. Although debugging by using breakpoints in both cores was possible in general, the simultaneous debugging of both cores turned out to be unpractical with our debugging tools. Moreover, even the standard debugging process for one core was less comfortable than in case of the Atmel MCU, which was used in the previous experiments. However, it is expected that debugging capabilities for dual-core microcontrollers will improve in the future.

In case of the FPGAs, no pre-designed debugging facilities were available. A tool for the integration of debugging circuitry in the FPGA (Xilinx Chip Scope) could have been used. However, test applications with students showed that the successful application of this tool requires a comprehensive introduction to the use of the tool, which could not be given within the scope of our experiments. Therefore, the students were instructed to include their own debugging circuitry in combination with LEDs, switches, and 7-segment displays. While this approach worked well in many cases, students often missed the possibility to evaluate their program step-wise as in case of the MCUs. This problem was especially the case during the application of state machines. This aspect emphasized that MCUs are suited better for sequential operations, not only with respect to the implementation aspect itself, but also with respect to debugging options.

As another aspect, it has often been stated that parallel FPGA components interact only via their interfaces and show less side affects for this reason. While this separation is true for a correct design, side effects are possible according to bad design practice. An example of this unwanted interaction is a design with two clock domains that are connected without specific measures [88]. This structure may lead to a design which works intermittently as the two clocks are asynchronous and therefore have an unknown phase relationship leading to metastability as the clock domains are crossed. This effect occurred in one of our experiment versions. The design worked fine first (first acceptance test), but additional components in the system led to a different routing of the design, which resulted in wrong behavior later on (second acceptance test). The reason for this behavior, also described in Section 5.5.3, is assumed to be an increase of the clock skew by adding new components. A warning was generated by the design environment already before the first acceptance test, but was ignored by the development team. Similar problems can also be introduced if input signals are not correctly synchronized with the correct system clock.

The hardware related aspects mentioned above generally do not have to be considered during software development for microcontrollers (e.g. synchronization stages are present at all MCU inputs by default). Although challenges are also present in these systems (deadlocks, problems resulting from interrupts and memory violations) mitigation of these problems requires only limited knowledge of the underlying hardware.

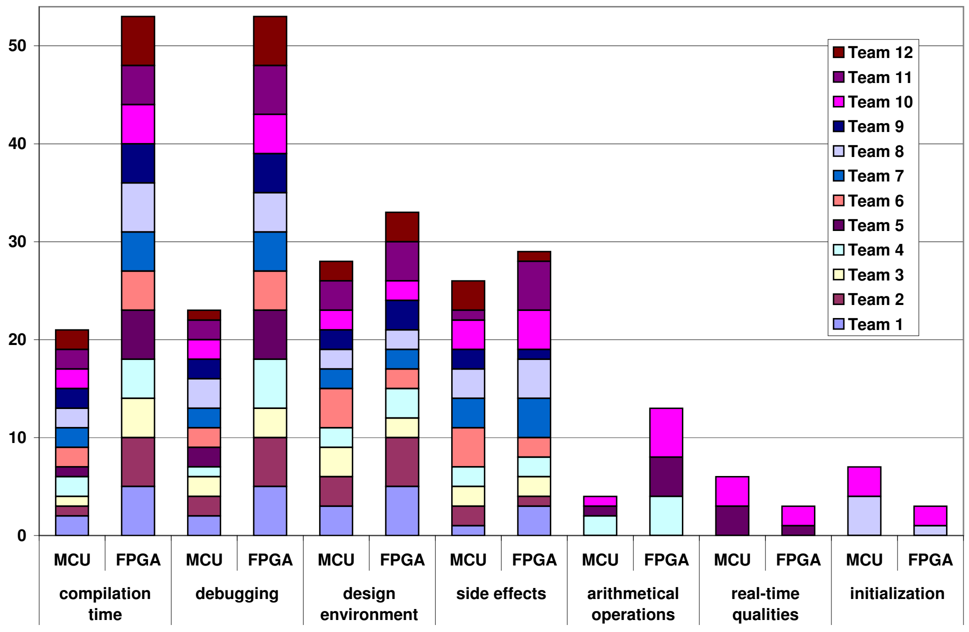


Figure 6.3.: Results of questionnaire MCU vs. FPGA

In order to collect concrete feedback from the students, they were asked to fill out an *MCU vs. FPGA evaluation sheet* in Experiment 5. While certain aspects were given, the students had the option to add further aspects. Each aspect was graded from 1 (very good/very sufficient) to 5 (bad/not sufficient) and the results are partly depicted in Fig. 6.3.

One aspect that has been graded worse for the FPGA development than for the MCU development was the *compilation time*, which was significantly higher for the FPGAs. This aspect is also considered in the validity evaluation.

The second aspect was a rating of the *debugging capabilities* of MCUs and FPGAs. Also in this case, MCUs received a better grading than FPGAs. As mentioned above, effective debugging is possible also on FPGAs, but generally requires the development and integration of an own debugging circuitry in the device. Advantages of FPGAs are that also sophisticated debugging facilities could be implemented if sufficient chip resources are available. However, even simple debugging facilities require more effort than in case of the MCU.

Next, the *design environments* have been rated similar in both cases while the FPGA environment was rated slightly worse (reasons: too slow, less intuitive than the MCU environment).

Further on, the aspect *undesired side effects* was on average rated slightly better for the MCUs than for the FPGAs. While it has been expected that FPGAs allow better control of side effects according to their parallelism, this parallelism turned out to be the problem for several teams. Another disadvantage in the FPGA design was that undesired side effects could occur according to bad design practice as described above.

Another aspect rated better on MCUs than on FPGAs was the straightforwardness to use *arithmetical operations*. As a reason could be seen that e.g. floating point operations could be used in the C implementation (with a certain run time overhead), but even simple divisions could not be synthesized directly from VHDL descriptions.

Finally, *real-time qualities* have been rated better for FPGAs than for MCUs by two teams (reason given: no side effects in FPGA by interrupts). The *initialization effort* was also rated better for FPGAs (reason: all MCU resources had to be parameterized while they could be "simply designed" in the FPGA).

Summarizing, development problems between MCUs and FPGAs differ in several aspects and advantages of a specific hardware platform often depend on the individual application.

7. Further Evaluations

An evaluation of the validity of the results obtained in our experiments can be found in Section 6 while further aspects are considered in this chapter. In the following section, the problem of limited significance in the results of two experiments is discussed. Further on, the experiment design is evaluated from the organizational point of view in Section 7.2 while an evaluation from the educational point of view can be found in Section 7.3.

7.1. Significance of Experiment Outcomes

According to our evaluations by using the resampling method, significant results were obtained in the first and the fifth experiment, while the results in Experiment 4 and 6 are not suited for statistical evaluations[1].

In case of the hypothesis testing in Experiment 2, the value of the statistical significance determined by resampling is generally slightly above the accepted value of 5% (see also second column of Tab. 7.1). Accordingly, the effect measured is not strong enough to be demonstrated with the available number of versions. To determine the required number of versions to achieve the desired significance of the results, the resampling process introduced in Section 4.7.2 was modified. In a first step, not 20 but 24 samples were drawn in each resampling step simulating 24 versions. As can be seen in the third column of Tab. 7.1, this number of versions is expected to be sufficient to achieve a significant result in case of the hypotheses $H_0 2a$ and $H_0 2d$. In a second step, we increased the number of versions to 44, resulting in a sufficient significance for hypothesis $H_0 2b$.

A similar problem occurred in Experiment 3. Here, the significance of the result obtained by the testing of hypothesis $H_0 3b$ (17.6%) was a lot higher than the threshold of 5%. It might be possible to increase the significance of the observed results by increasing the number of the reviewers and/or the number of reviewed versions (in order to obtain more review results). According to another simulation a doubling of the considered review results would have lead

[1]Note: The reason is the type of hypothesis in Experiment 4 and the low number of experiment participants in Experiment 6.

Table 7.1.: Significance of the results in Experiment 2

Hypothesis	**Significance 20 versions**	**Significance 24 versions**	**Significance 44 versions**
$H_0 2a$	5.10%	3.62%	0.73%
$H_0 2b$	6.14%	5.68%	3.02%
$H_0 2d$	5.49%	2.26%	0.30%

to a significance of ∼9% while a quadrupling would have lead to a significance of 2.57%.

These considerations show that higher numbers of versions are required for the evaluations to allow an improvement of the significance of the results. In case of Experiment 2, a sufficient number of teams was available, but not all teams managed to pass the acceptance test with both implementations. Therefore, the possibility that not all teams finish the experiment successfully should be considered during the planning phase of each experiment.

7.2. Organizational Aspects of the Experiments

The conduction of the experiments revealed several challenges. First, the execution of the experiments 1, 2, 4, and 5 required good planning, as a high number of different devices (development boards, debugging hardware, CAN interface) and tools had to be provided to the students. In this context, the common CAN bus turned out to be a very useful output for the experiment application. The values of all teams present on the CAN bus could be recorded with a single CAN monitor and were simply displayed via a video projector. Moreover, each team was equipped with a small frequency generator, which we developed for this purpose, to test the main task of these experiments. Therefore, this application allowed comparatively easy tests by the experiment participants during their development phase while the acceptance test also profited from this structure. Summarizing, the chosen approach could support the idea of *design for testability* introduced in Section 4.6.6.

Further on, the acceptance and evaluation tests of the experiments 1-5 required specific equipment that was developed for this purpose. As described in Section 4.3, the use of a common basic task for these 5 experiments allowed the use of a common test environment, which could reduce the overall development effort in this field. In this context, the test environment turned out to be very flexible, which facilitated the adaptation of the test environment for the different experiments. One aspect that allowed this flexibility was the separation of the different test aspects, namely the test case generation, the test signal generation, the test run coordination and recording, and the later off-line analysis (see Section 4.7.4). Another aspect that allowed to support this high flexibility was the use of an FPGA for the test signal generation. This way, high numbers of independent and individual signals could be generated easily. While the performance of the test environment was sufficient for the chosen application, the bottleneck of the system can be seen in the serial connection between the PC and the remaining test hardware. However, this connection could be replaced if higher communication bandwidth is required in future test applications (e.g. by USB).

Moreover, the effort of developing a suitable specification was reduced significantly by using a common basic task for several experiments. Thus, only modifications had to be integrated into the specification and the amount of testing during experiment development could be reduced. Finally, comparisons of the results obtained in the different experiments were possible that way (e.g. comparison of the results of the experiments 1, 2 and 4 in Section 6.2).

Furthermore, the conduction of questionnaires turned out to be very useful, especially to evaluate threats to the validity of our results (see Section 6.1).

In case of experiments involving verification activities, it has to be distinguished between two main aspects: On the one hand, it could be evaluated which aspects are investigated by the experiment participants (What is verified?). On the other hand, it might be interesting to compare the verification results of several participants regarding a given aspect (What is the verification result?). In Experiment 4, we were interested in the first aspect. Therefore, we gave no guidelines to the students regarding the concrete aspects that should be verified. In Experiment 3, we were interested in a comparison between the review results of the three reviewers. Problems regarding comparability occurred in this case, as the second reviewer partly chose different review scenarios than the first one. To increase the comparability between the results, we provided the review scenarios selected by the first reviewer to the third reviewer. Therefore, we recommend always to determine the focus of the investigation first. If it is on a comparison of the review results, equal review scenarios should be provided to all reviewers to ease comparison of the review results.

An aspect, which could be approved for following experiments, was the time available for corrections in Experiment 4. However, the aspects of interest for this work could be evaluated without completed corrections as described in Section 6.2.2.

Finally, Experiment 6 was based on another application. The reason for choosing this application was the need for a more complex application with explicit safety goals. The new application was resulting in additional effort for the specification and the development of the test- and simulation environment. Nevertheless, it has to be noted that minor parts of the test environment developed for the first five experiments could be reused in this test environment (e.g. signal generation with an FPGA connected to a PC via serial connection, CAN controller interface on the FPGA). The reuse was limited as this application required a more sophisticated simulation of its environment (windows, roof, trunk,...), which necessitated to develop a model of the environment as described in Section 4.7.4. In the end, further experiments with this application are desirable to compensate the effort for the additional specification and the new test environment.

7.3. Educational Aspects of the Experiments

As mentioned before, experiments are often conducted in lab courses or as projects in universities. As a reason, it is often the only possibility to conduct experiments involving implementation activities with a sufficient number of participants. While the impacts on the experiment results of this approach are discussed in Section 6.1, the impacts of experiments on educational aspects are discussed briefly in this section.

First of all, experiments in lab courses and university projects come with a few educational advantages. In most cases, the experiment task represents a carefully designed specification of an application example or even a real application. This type of application is considered as a real-world task for the students in contrast to weekly step by step exercises. While this type of task is certainly more realistic, a higher level of knowledge and skills of the students is required for this form of teaching. Therefore, experiments seem to be especially suitable in later stages of the study period. Moreover, the combination of research and educational aspects allows higher investments of time and money in the design of the corresponding project than following these aspects independently. Further on, it is beneficial that by combining experiments with education, both parties are interested in completing the experiment task successfully. The students are interested in a successful implementation in order to receive their certificate, while the experiment supervisor has the same interest as successful implementations are usually required for later evaluations.

Moreover, the experiment design of the experiments 1, 2, and 5 was in line with the educational aspect behind these experiments. According to this structure, the participants were able to implement a common specification in two different ways, clarifying differences in their specific benefits and drawbacks (see [105] for further information on these educational aspects). While this approach was not possible in Experiment 4, feedback of the students on the review and test activities was also positive.

On the other hand, certain disadvantages could arise. If students fail to handle certain tasks of the experiment, only limited support can be given in order to avoid unintended influences of the experiment supervisor on the experiment outcome (see Section 4.7.2). Otherwise, the team could be removed from those teams which are considered for the experiment evaluation. In this case, individual help could be provided, but this treatment is regarded as problematic for the internal validity of the experiment as unwanted outcomes may be removed this way. In addition, it has to be assured that this special treatment is not affecting the motivation of the remaining teams. In our experiments, weak teams could be supported by giving additional time for completing the tasks. However, usually not all teams could pass all acceptance tests conducted in the corresponding experiment.

Additionally, the anonymization of the experiment participants is desirable for the conduction of the experiment (see e.g. [16], page 49). However, anonymization is often not possible in education as the certification of the successful participation in the lab course or project is generally dependent on the performance of each individual student. Therefore, each team received a random number in our experiments, but a document existed in which the students were assigned to these numbers. However, only for the issuing of the students' certificates we checked whether the team with the corresponding number fulfilled the acceptance criteria. Thus, a certain level of anonymization could be achieved.

Accordingly, educational and experimental responsibility tend to conflict so that well thought decisions have to be taken. Thus, additional effort is required to perform experiments in lab courses that suit experimental and educational needs. Nevertheless, we are of the opinion that this additional effort paid off very well in our experiments.

8. Conclusion and Future Work

The main aspects of this work are summarized briefly in this chapter and the corresponding conclusions are drawn. Moreover, a short discussion of possible future work is presented.

8.1. Conclusion

According to the importance of embedded systems in safety-critical applications, impacts of design decisions on safety aspects were discussed. The handling of faults in the software and hardware parts of these systems turned out to be an important aspect for the assessment of safety-related impacts. Moreover, we observed potential dependencies between the selection of a specific hardware platform and fault handling activities. Thus, two different hardware platforms (microcontrollers and FPGAs/CPLDs) were selected for the investigations conducted for this work followed by the evaluation of fault handling aspects in these devices in Chapter 3. During the evaluation, we observed a lack of well founded results in case of several fault handling aspects. Particularly few pieces of information could be found regarding the assessment of fault handling techniques that depend on *human factors* as many activities in the development of software for safety-critical applications. Accordingly, the impact of hardware platform selection on five of these aspects, *software diversity, encapsulation, reviewability, resusabilty*, and *development according to ISO26262*, were chosen for investigation in this work. Furthermore, the approach of *software diversity* was compared with a *fault removal approach.*

Empirical evaluations, experiments to be more precise, were identified as a suitable methodology for the evaluation of the selected aspects. The planning of these experiments, which is described in Chapter 4, required a consideration of the specialties of embedded systems. In contrast to pure software experiments, these experiments require specific materials. Therefore, development boards and the corresponding equipment had to be provided in sufficient numbers. Next, the experiment application had to allow a reasonable operation and testing during the execution of the experiment as well as an efficient evaluation testing of the experiment outcomes. An automotive application was chosen, which turned out to be very suitable for this purpose. In this context, the consideration of testing issues, as the *design for testability* and a suitable process for the conduction of the acceptance tests, were important aspect of the experiment design. Further on, specific test environments had to be developed for acceptance and evaluation testing. The development effort for test environments could be reduced by using a common basic task for the first five experiments. This task was extended by additional tasks to suit the needs of the individual experiments. Moreover, the use of a common basic task for these experiments made it possible to reduce the effort for the development of the experiment task specification. On the other hand, a different task was required for the last experiment. While minor parts of the existing test environment could be reused, a complete new specification had to be developed.

The operation of the experiments is described in Chapter 5. The experiments were mostly conducted in lab courses (experiments 1, 2, 4, and 5) to allow for sufficient numbers of experiment participants. The remaining experiments had to be conducted with lower numbers of student assistants (Experiment 3) or within the scope of diploma theses (Experiment 6) as the experiment task was not suitable for a conduction in lab courses. The results obtained by hypothesis testing and the corresponding analysis in Chapter 6 are summarized in the following for each aspect investigated.

8.1.1. Experiment Results

Software diversity

First, software diversity could be increased by the application of diverse hardware platforms. However, the effect was low and we disproved the hypothesis of statistical independence of the failures. Reasons for the dependent failures were identified in the specification, the application itself and the implementation. According to the results we obtained in the first and the second experiment, software diversity is not suited to tolerate application and specification specific faults. Moreover, even one implementation specific fault was identified in several versions developed independently on different hardware platforms.

Test and Review vs. N-Version Programming

The approach of fault tolerance was compared with fault removal by test and review in Experiment 4. As a result, test and review identified application specific faults, which were not covered by software diversity. Otherwise, specification specific faults (ambiguous and incomplete statements) were identified only by a minority of the teams during their review and test activities. Moreover, not all implementation specific faults were identified by review and testing. Accordingly, the approaches have their strength in different fault categories (software diversity for implementation specific faults, fault removal for application specific faults), while specification specific faults remain a problem for both approaches.

Encapsulation

Potentials of encapsulation were investigated for MCUs and FPGAs (Experiment 2). While it was expected that encapsulation of real-time tasks benefits from the parallel structure of FPGAs, not all hypotheses investigated could support this assumption. As a general result, this encapsulation is expected to work better on FPGAs than on MCUs if several real-time functions with only limited functional interactions have to be implemented. Stronger interaction between the implemented tasks counteracts this advantage.

Reusability

Encapsulation with respect to the reuse of real-time functions was investigated in Experiment 5. During evaluation, different impacts on the reused component could be observed in many MCU versions while no effects occurred in any FPGA version. According to our evaluations, FPGAs will have an advantage if real-time components can be reused without modifications. In this case, faults can occur only in case of an incorrect interface between the reused component and the new application. This is different in case of real-time software components reused in an MCU. While on-chip peripherals as timer/counter units could ease the concurrent execution of several real-time functions, undesired side effects by the common CPU have been identified in our evaluations. Moreover, the use of on-chip peripherals increases the hardware-software dependencies on MCUs, while software for FPGAs is in general developed completely hardware independent. However, problems could occur in FPGAs if design guidelines are not followed strictly. Furhtermore, changes in the functionality of the reused component could require major modifications of its internal structure and interface. These problems were identified as less critical for MCU software.

Reviewability

Moreover, the application of reviews was investigated for the software written for MCUs and FPGAs (Experiment 3). As in the case of encapsulation, it was assumed that the parallel nature of the FPGA would ease the process of evaluating real-time software. The reviewability of FPGA software was (on average) rated better by each of our reviewers than the reviewability of the MCU software. However, the significance of the observed difference is not given, but might be achieved by higher numbers of reviewers as demonstrated in Section 7.1. Otherwise, a check of the review results by testing showed higher compliance of test and review results in case of MCU versions. Therefore, the FPGA versions concealed certain faults from the reviewers, although their reviewability was rated higher on average.

ISO26262 Development

In the last experiment, the development of a safety-critical application following the safety standard ISO26262 was evaluated for two hardware platforms. According to the common safety concept chosen for both hardware platforms, only minor differences were identified between both implementations, as described in Section 6.6. Otherwise, differences were identified between the safety concept chosen, which is based on *functional supervision*, and an alternative approach based on *device supervision*. Several advantages could be identified for our approach (handling of software faults, knowledge of the criticality of faults, intuitive fault handling concept) while advantages of the alternative concept were limited to the independence of this approach on the safety concept and the application. A comparison of the advantages and disadvantages of both approaches revealed, that the decision for one of these approaches depends a lot on the application. While the costs and the technological maturity of today's devices allowing device supervision makes them unsuitable for many safety-critical embedded applications, this concept might gain importance in the future, especially for applications that do not allow an intuitive concept of functional supervision.

Identified Development Problems

Furthermore, the observed development problems regarding MCUs and FPGAs were discussed in Section 6.7. Aspects as long compilation times and a less intuitive way of debugging the software were identified as disadvantages in the FPGA design. Moreover, design guidelines are more important in FPGA design to allow a reliable design (hardware aspects have to be considered). Otherwise, trends to develop FPGA designs on higher levels (e.g. by using *Xilinx Embedded Development Kit*(EDK)) can be observed, which might ease the handling of hardware effects in future FPGA designs.

8.1.2. Evaluation

Potential threats on the validity of the presented results were discussed (Chapter 6). Accordingly, no major threats on the validity could be identified. Nevertheless, additional experiments are desirable to extend the general applicability of the results presented in this work. Further improvements of the experiment structure are discussed in the outlook on future work.

Moreover, the experiment design was evaluated in Chapter 7 regarding the significance of the experiment outcomes as well as organizational and educational issues. Accordingly, the significance of certain results observed in our experiments might be further improved by increasing the number of evaluated versions. To summarize the organizational aspects, the approach used for the experiments 1-5 presented in this work allowed to reduce the overall effort needed for preparation, execution and evaluation of the experiments. Thus, the approach presented could be applied for further experiments with a similar or different task to increase the validity of our results as mentioned above. Further experiments, based on the task applied in Experiment 6, are especially interesting to compensate the planning and development effort of this experiment. Finally, advantages and disadvantages with respect to education have been identified for experiments conducted in lab courses. In our opinion, the biggest advantage in the combination of educational and experimental aspects is the possibility of higher investments of time and money for each purpose compared to an independent following of both aspects. On the other hand, educational and experimental requirements tend to conflict. However, if well thought decisions are taken, we consider this approach as suitable to fit both needs.

Summarizing, we identified potential impacts of hardware platform selection and introduced a methodology for the investigation of open issues in this field. Further on, we presented empirical results based on the experiments conducted for the evaluations of these issues. The identified impacts of hardware platform selection on the safety of embedded systems hopefully help developers of these systems with their design decision. Finally, further investigations are desirable to support our results and to investigate the remaining open issues.

8.2. Future Work

Two main aspects are interesting for future work. First, further empirical evaluations are desirable to cover the remaining open issues presented in Section 3, which is discussed in the following section. Moreover, the results gained by empirical evaluations could be collected to support constructive approaches, which are presented in Section 8.2.2.

8.2.1. Further Empirical Evaluations

Further Impacts on Fault Handling

Investigations in experiments require a change of only the independent variable to allow precise results with respect to the dependent variable. Thus, several of the potential impacts listed in Chapter 3 could not be investigated in this work so that further experiments might be desirable to investigate these open issues. Aspects that might be of particular interest are the impact of hardware abstraction and operating systems or the impact of model based design techniques on fault handling aspects. Moreover, additional approaches of software fault handling could be further investigated, as the approach we presented in [22].

Further Experiment Designs

With a larger financial budget, improvements to increase the number of suitable participants could be achieved. One approach would be to employ students for the experiments. This approach allows higher numbers of experiment participants, particularly in case of experiments that are not suited for a conduction in lab courses for educational reasons (as Experiment 3).

Furthermore, alternative approaches might be suitable to increase the number of participants. An example would be an experiment application which could be accessed via the internet (e.g. application that can be programmed, executed and monitored via internet; supervision of application via webcam). This approach would allow participants all over the world to take part in the experiment, which could improve the significance of the results. However, these types of experiments might be more difficult to control.

Further Use of Experiment Data

The program versions developed in the different experiments can be used for further evaluations. As an example, several experiment versions were used for a case study of the model checker developed at our institute [111].

8.2.2. Constructive Approaches

The idea of constructive approaches is to improve the reliability and safety of a system by applying suitable design rules. While certain guidance is given by safety standards [46, 48], several aspects remain untouched. For a successful application of constructive approaches further information, e.g. by empirical evaluations as applied in this thesis, is required for this reason. To enable an efficient application of this information obtained, it could be included in a design pattern catalog, as proposed in [3]. Moreover, the information could be provided in the form we proposed in [99] for supporting hardware platform design decisions. This approach is based on two steps presented in Fig. 8.1. First, *hardware attributes* of a specific hardware platform are identified and evaluated (left part of Fig. 8.1). They represent how much a given platform contributes to the corresponding system qualities (e.g. the *hardware attribute testability* of a

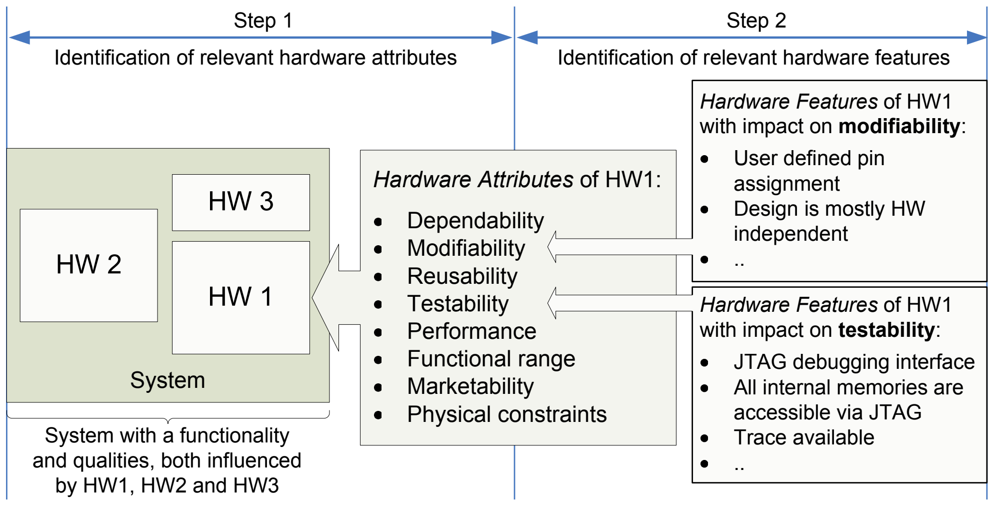

Step 1
Identification of relevant hardware attributes
Step 2
Identification of relevant hardware features
HW 3
HW 2
HW 1
System
System with a functionality and qualities, both influenced by HW1, HW2 and HW3
Hardware Attributes of HW1:
Dependability
Modifiability
Reusability
Testability
Performance
Functional range
Marketability
Physical constraints
Hardware Features of HW1 with impact on modifiability:
User defined pin assignment
Design is mostly HW independent
..
Hardware Features of HW1 with impact on testability:
JTAG debugging interface
All internal memories are accessible via JTAG
Trace available
..

Figure 8.1.: Approach to systematically identify impacts of hardware platform selection on the overall system

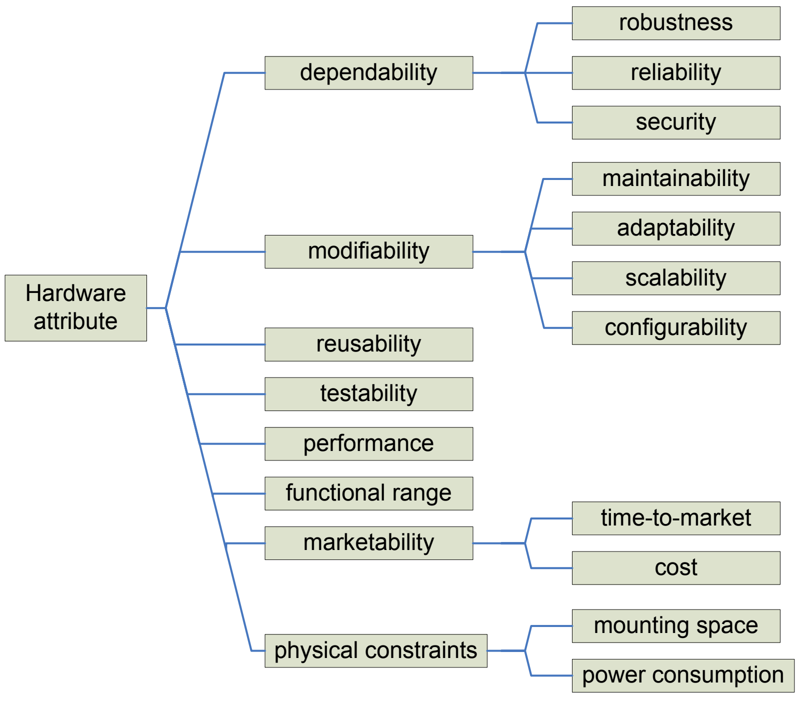


Figure 8.2.: The hardware attribute tree

specific hardware platform represents how this platform contributes to the *system quality testability*). The hardware attributes considered in our approach are depicted in Fig. 8.2 in form of an attribute tree. Second, all hardware features that have a possible impact on the different attributes have to be considered to allow an evaluation of the hardware attributes for a specific hardware platform (right part of Fig. 8.1). As an example, encapsulation capabilities regarding real-time functions (hardware feature) have an impact on different hardware attributes of the corresponding hardware platform, as reliability, safety and reusability. Based on the result of this two step approach, the hardware attributes of different hardware platforms can be compared. The designer can then decide which hardware attributes are the most important ones and which are of minor interest. Thus, an overall picture is used for comparison and single aspects are not forgotten which might turn out to be important later on. The application of the selection process, the hardware attribute tree as well as the relationship between *hardware attributes* and *hardware features* is described in more detail in [99]. Moreover, the approach was applied in [104] for a survey on the suitability of FPGAs for industrial applications.

Finally, trade-offs between different design solutions could be supported by the application of Kiviat graphs, as proposed in [72].

The integration of the results obtained in this work into the approaches mentioned above is desirable for two reasons. On the one hand, it would ease the access to the results obtained and simplifies their application in design decisions in safety-critical systems. On the other hand, the integration of empirical results into these approaches would hopefully motivate further empirical investigation in the field of safety-critical embedded systems.

A. Experiment Results

While only mean and median values of the test results are presented in Chapter 5, the results measured for each version are included in this appendix for the experiments 1, 2, 4, and 5. During the evaluation of the measurement results we distinguished between timing failures (message too late), silent failures (message lost), and content failures (values in the message are wrong). Moreover, we distinguished between major content failures (value differs more than 1 from the closest correct value) and minor content failures (value differs by 1 from the closest correct value). According to the limited space, only the sum of these failures could be included while a listing of all individual failures is available in electronic form. As a failure could be a content and a timing failure at the same time, the sum of all failures could exceed the value of 100%. Moreover, versions that were implemented in the first half of an experiment are indicated in the tables of the experiments 1, 2, and 5 as follows: "-1-" = this is an initial version, " 1 " = this version was developed in the second half of the experiment.

Table A.1.: Results Experiment 1

Team		Test bench				
		TB2	TB3	TB3s	TB4	TB8
MCU	-1-	0.4%	2.8%	-	0.0%	0.6%
	2	0.1%	1.9%	-	0.2%	20.7%
	3	10.8%	2.4%	-	7.0%	4.7%
	-4-	1.1%	0.7%	-	4.3%	8.6%
	5	0.2%	0.8%	-	0.0%	0.7%
	6	1.8%	15.6%	-	5.7%	5.2%
	-7-	1.7%	1.2%	-	0.1%	8.5%
	-8-	-	-	-	-	-
	-9-	0.7%	0.8%	-	0.2%	8.2%
	10	12.1%	2.9%	-	7.9%	4.6%
	-11-	0.9%	65.0%	-	0.0%	0.0%
	12	-	-	-	-	-
	Mean	2.98%	9.41%	-	2.54%	6.19%
	Median	1.00%	2.16%	-	0.18%	4.96%
CPLD	1	11.8%	98.6%	3.8%	3.2%	1.6%
	-2-	10.3%	94.8%	95.6%	6.2%	10.9%
	-3-	7.3%	98.6%	3.4%	2.7%	1.3%
	4	12.8%	99.7%	3.2%	3.1%	1.6%
	-5-	0.1%	94.5%	1.2%	0.0%	0.7%
	-6-	1.6%	95.7%	93.8%	1.1%	0.7%
	7	1.5%	0.7%	1.0%	1.0%	0.5%
	8	-	-	-	-	-
	9	11.1%	99.7%	3.8%	3.7%	2.3%
	-10-	2.1%	98.9%	1.2%	1.1%	0.4%
	11	1.9%	99.2%	1.2%	1.0%	0.4%
	-12-	-	-	-	-	-
	Mean	6.05%	88.04%	20.82%	2.32%	2.04%
	Median	4.69%	98.60%	3.32%	1.92%	1.00%

Note: TB3s = shortened version of TB3, used for the evaluation of crosstalk effects in CPLD versions.

Table A.2.: Results Experiment 2 (main versions)

Team		Test bench TB2	TB3	TB4	TB8	TB9
MCU	-1-	-	-	-	-	-
	2	0.8%	2.7%	0.0%	1.4%	1.8%
	3	1.4%	6.1%	0.0%	3.5%	2.7%
	-4-	0.5%	6.4%	0.1%	2.3%	0.6%
	-5-	0.8%	0.7%	0.0%	0.1%	1.0%
	-6-	2.7%	18.5%	0.0%	10.1%	6.7%
	7	2.1%	63.9%	0.2%	84.3%	38.4%
	-8-	0.2%	11.9%	0.1%	7.4%	2.5%
	9	0.3%	23.1%	0.0%	34.1%	5.1%
	10	1.4%	31.9%	0.2%	55.3%	6.8%
	-11-	0.5%	77.6%	0.0%	92.0%	22.3%
	12	2.1%	64.1%	0.2%	84.3%	17.1%
	Mean	1.15%	27.90%	0.09%	34.07%	9.53%
	Median	0.8%	18.5%	0.03%	10.1%	5.1%
FPGA	1	-	-	-	-	-
	-2-	0.9%	17.6%	0.0%	37.4%	2.5%
	-3-	7.8%	2.0%	2.3%	1.3%	1.8%
	4	1.9%	8.2%	0.5%	14.0%	2.9%
	5	0.2%	20.9%	0.6%	29.5%	3.9%
	6	0.5%	1.2%	0.0%	0.8%	0.5%
	-7-	18.9%	18.6%	1.9%	31.2%	6.8%
	8	7.0%	66.6%	2.3%	85.9%	17.6%
	-9-	0.2%	52.0%	0.1%	86.1%	19.4%
	-10-	1.4%	20.1%	0.3%	28.9%	4.0%
	11	6.8%	33.4%	2.4%	49.8%	12.2%
	-12-	4.7%	20.2%	18.1%	40.1%	25.4%
	Mean	4.56%	23.71%	2.59%	36.84%	8.83%
	Median	1.9%	20.1%	0.6%	31.2%	4.0%

Note: As team 1 did not deliver an FPGA version, also their MCU version was not considered for evaluation.

Table A.3.: Results Experiment 2 (final versions)

Team		Test bench TB2	TB4	TB8	TB9	TB4b
MCU	-1-	-	-	-	-	-
	2	0.8%	0.0%	1.4%	1.7%	0.6%
	3	1.4%	0.0%	3.6%	2.7%	12.4%
	-4-	0.4%	0.1%	0.7%	0.3%	0.0%
	-5-	0.9%	0.0%	0.7%	0.9%	49.7%
	-6-	1.5%	0.0%	10.1%	6.7%	61.5%
	7	2.1%	17.4%	90.4%	42.7%	31.8%
	-8-	0.2%	0.1%	7.4%	2.5%	25.0%
	9	0.5%	0.0%	38.9%	4.4%	45.9%
	10	1.5%	0.3%	15.0%	3.2%	9.9%
	-11-	0.3%	0.1%	84.9%	17.5%	42.0%
	12	-	-	-	-	-
	Mean	0.95%	1.80%	25.31%	8.26%	27.86%
	Median	0.81%	0.04%	8.75%	2.92%	28.36%
FPGA	1	-	-	-	-	-
	-2-	0.8%	0.0%	39.9%	2.6%	6.7%
	-3-	8.1%	2.4%	1.3%	1.8%	0.0%
	4	1.6%	0.5%	13.8%	3.4%	51.7%
	5	1.0%	0.0%	28.3%	33.2%	21.1%
	6	0.3%	0.1%	0.8%	0.6%	0.0%
	-7-	19.3%	1.9%	30.8%	6.9%	15.1%
	8	6.9%	2.4%	85.8%	18.4%	1.7%
	-9-	0.2%	0.1%	86.3%	19.5%	0.2%
	-10-	1.4%	0.2%	28.4%	4.3%	9.1%
	11	6.7%	2.4%	40.6%	11.2%	21.0%
	-12-	-	-	-	-	-
	Mean	4.64%	0.99%	35.59%	10.17%	12.68%
	Median	1.50%	0.36%	29.60%	5.58%	7.89%

Note: As team 12 did not deliver a final MCU version, also their final FPGA version was not considered for evaluation.

Table A.4.: Results Experiment 4 (debugged versions)

Team		Test bench				
		TB2	TB4	TB8	TB9	TB4b
MCU	1	1.5%	1.3%	0.3%	1.6%	2.8%
	2	0.0%	0.2%	1.8%	0.7%	2.8%
	3	1.4%	0.3%	2.3%	1.6%	0.0%
	4	0.1%	0.0%	19.7%	2.3%	65.7%
	5	2.6%	19.1%	44.9%	13.7%	13.2%
	6	0.5%	0.0%	50.3%	6.9%	14.6%
	7	0.6%	0.5%	1.6%	0.9%	3.9%
	8	0.0%	0.4%	0.2%	0.2%	3.1%
	9	10.9%	0.2%	0.3%	1.5%	2.8%
	10	0.8%	0.0%	5.5%	1.1%	36.6%
	11	1.5%	0.3%	39.1%	2.8%	3.5%
	12	0.8%	0.2%	95.0%	40.6%	4.0%
	Mean	1.73%	1.88%	21.75%	6.16%	12.74%
	Median	0.8%	0.3%	3.9%	1.6%	3.7%

Table A.5.: Results Experiment 5

Team		Test bench				
		TB2	TB4	TB4b	TB12	TB13
MCU	-1-	0.4%	0.4%	27.5%	22.5%	42.7%
	2	3.4%	2.3%	29.2%	52.5%	34.7%
	-3-	0.2%	0.1%	27.5%	17.5%	16.0%
	4	1.2%	1.8%	27.5%	30.0%	29.3%
	-5-	0.2%	0.0%	27.5%	37.5%	24.0%
	6	47.6%	22.4%	38.4%	112.5%	104.0%
	-7-	0.2%	0.1%	27.6%	30.0%	26.7%
	8	1.5%	0.0%	27.5%	22.5%	20.0%
	-9-	0.3%	0.1%	27.5%	30.0%	30.7%
	10	0.0%	0.0%	44.0%	27.5%	20.0%
	-11-	0.6%	0.1%	35.5%	22.5%	18.7%
	12	86.0%	44.5%	52.0%	130.0%	116.0%
	Mean	11.82%	5.97%	32.65%	44.58%	40.22%
	Median	0.52%	0.05%	27.56%	30.00%	28.00%
FPGA	1	3.5%	11.9%	33.0%	105.0%	97.3%
	-2-	25.2%	13.2%	33.5%	100.0%	98.7%
	3	1.1%	0.9%	27.6%	5.0%	0.0%
	-4-	7.0%	1.4%	27.6%	17.5%	8.0%
	5	0.9%	0.6%	28.4%	10.0%	6.7%
	-6-	15.8%	6.9%	52.3%	142.5%	98.7%
	7	1.2%	0.9%	27.6%	17.5%	8.0%
	-8-	2.9%	1.5%	89.8%	30.0%	17.3%
	9	0.2%	0.0%	35.9%	15.0%	6.7%
	-10-	6.0%	3.3%	35.7%	55.0%	33.3%
	11	1.2%	1.5%	89.8%	17.5%	12.0%
	-12-	72.9%	5.8%	28.2%	32.5%	22.7%
	Mean	11.50%	3.96%	42.45%	45.63%	34.11%
	Median	3.2%	1.5%	33.3%	23.8%	14.7%

Bibliography

[1] ALTERA. Error detection & recovery using CRC in Altera FPGA devices. In *Application Note 357*, April 2006.

[2] Atmel. *ATmega16(L) - Atmel ATmega16 data sheet.* URL: www.atmel.com/dyn/resources/prod_documents/doc2466.pdf (accessed 10.04.2008).

[3] M. Auerswald, M. Herrmann, S. Kowalewski, and V. Schulte-Coerne. Reliability-oriented product line engineering of embedded systems. In *Proceedings 4th Int. Workshop on Product Family Engineering, Bilbao, Spain*, 2001.

[4] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault-tolerance during program execution. In *Int. Computer Software and Applications Conference (COMPSAC'77)*, 1977.

[5] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing*, 1:11–33, 2004.

[6] A. Avizienis, M. Lyu, and W. Schutz. In search of effective diversity: A six-language study of fault-tolerant flight control software. In *Int. Symposium on Fault-Tolerant Computing ' Highlights from Twenty-Five Years'*, 1995.

[7] E. Bahl, T. Lindenkreuz, and R. Stephan. The fail-stop controller ae11. In *Proceedings of the International Test Conference.* IEEE, 1997.

[8] V. R. Basili and B. T. Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, 27:42–52, 1984.

[9] R. Baumann. The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction. In *Digest of the Internation Electron Devices Meeting IEDM'02.* IEEE, 2002.

[10] J. Bentley, P. Bishop, and M. van der Meulen. An empirical exploration of the difficulty function. In *Computer Safety, Reliability and Security (Safecomp)*, 2004.

[11] M. Berg. VHDL synthesis for high-reliability systems. In *2004 MAPLD International Conference*, 2004.

[12] P. Bernardi, L. Bolzani, M. Rebaudengo, M. S. Reorda, F. Vargas, and M. Violante. On-line detection of control-flow errors in socs by means of an infrastructure ip core. In *International Conference on Dependable Systems and Networks (DSN'05)*, 2005.

[13] L. Bolzani, M. Rebaudengo, M. S. Reorda, F. Vargas, and M. Violante. Hybrid soft error detection by means of infrastructure IP cores. In *10th IEEE International On-Line Testing Symposium (IOLTS'04)*, 2004.

[14] D. Borrione and P. Georgelin. Formal verification of VHDL using VHDL-like ACL2 models. In *Forum on Design Languages (FDL)*, 1999.

[15] J. Bortz. *Statistik*. Springer, 2005.

[16] J. Bortz and N. Döring. *Forschungsmethoden und Evaluation für Human- und Sozialwissenschaftler*. Springer, 2002.

[17] D. Boymanns. Untersuchung einer MCU-FPGA-Architektur für sicherheitskritische Automobilanwendungen nach ISO 26262. *Diplomarbeit, RWTH Aachen - Informatik 11*, 2008.

[18] Brewerton2007. Dual core processor solutions for IEC61508 SIL3 vehicle safety systems. In *Embedded World Conference*, 2007.

[19] J.-M. Burkhardt, F. Deétienne, and S. Wiedenbeck. Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, 7:115–156, 2002.

[20] A. Burns and J. McDermid. Real-time safety-critical stystems: analysis and synthesis. *Software Engineering Journal*, 1994.

[21] X. Cai, M. R. Lyu, and M. A. Vouk. An experimental evaluation on reliability features of n-version programming. In *Int. Symposium on Software Reliability Engineering (ISSRE'05)*, page 10 pp. IEEE, 2005.

[22] X. Chen, F. Salewski, S. Kowalewski, and K. Spisic. Concept and prototyping of a fault management framework for automotive safety relevant systems. In *Moderne Elektronik im Kraftfahrzeug II - Systeme von morgen - Technische Innovationen und Entwicklungstrends*. Expert Verlag, 2007.

[23] W. Cullyer, S. Goodenough, and B. Wichmann. The choice of computer languages for use in safety-critical systems. *Software Engineering Journal*, pages pp. 51–58, 1991.

[24] S. Dellacherie, L. Burgaud, and P. di Crescenzo. Improve-hdl: a DO-254 formal property checker used for design and verification of avionics protocol controllers. In *Digital Avionics Systems Conference (DASC'03)*. IEEE, 2003.

[25] Doulous. *The VHDL Golden Reference Guide*. 1995.

[26] DualCore-Semiconductor. *Dual Core Industrial Controller DCIC 9907 Data Sheet*, 2006. URL: www.dualcore.com/ downloads/ DCIC9907_Datasheet.pdf (accessed 18.04.2008).

[27] R. Dülks. Entwurf und Entwicklung einer modularen Simulations- und Testumgebung für Steuergeräte in Automobilanwendungen. *Diplomarbeit, RWTH Aachen - Informatik 11*, 2008.

[28] D. Eckhardt, A. Caglayan, J. Knight, L. Lee, D. McAllister, M. Vouk, and J. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transactions on Software Engineering*, 17(7):692–702, July 1991.

[29] A. En-Nouaary, F. Khendek, and R. Dssouli. Fault coverage in testing real-time systems. In *Int. Conf. on Real-Time Computing Systems and Applications (RTCSA'99)*, 1999.

[30] M. Fagan. Design and code inspections to reduce errors in program development. Technical report, IBM, 1976.

[31] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. In *IEEE Trans. Softw. Eng.*, volume 26, pages 797–814. IEEE Press, 2000.

[32] N. E. Fenton and S. L. Pfleeger. *Software Metrics - A Rigorous & Practical Approach.* PWS Publishing Company, 1997.

[33] M. Frate, Garg and Pasquini. On the correlation between code coverage and software reliability. 1995.

[34] Freescale. *MC9S12DP512 Users Guide*, 2005. URL: www.freescale.com (accessed 18.04.2008).

[35] T. L. Fruehling. Delphi secured microcontroller architecture. In *Design and Technologies for Automotive Safety-Critical Systems.* SAE World Congress, March 2000.

[36] M. G. Gericota and J. M. Ferreira. Restoring reliability in fault tolerant reconfigurable systems. In *Actas das Jornadas sobre Sistemas Reconfiguráveis (REC'2005)*, 2005.

[37] R. L. Glass. Reuse: what's wrong with this picture? *IEEE Software*, 15:57–59, 1998.

[38] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violoante. *Software-Implemented Hardware Fault Tolerance.* Springer, 2006.

[39] K. E. Grosspietsch and A. Romanovsky. An evolutionary and adaptive approach for n-version programming. In *IEEE Euromicro Conference*, 2001.

[40] J. Hammarberg and S. Nadjm-Tehrani. Formal verification of fault tolerance in safety-critical configurable modules. *International Journal of Software Tools for Technology Transfer*, 7, 2004.

[41] R. Harper and P. Lee. Research in programming languages for composability, safety, and performance. *ACM Comput. Surv.*, 28:195, 1996.

[42] Health and Safety Executive. *Out of Control - why control systems go wrong and how to prevent failure.* HSE books, 2003.

[43] A. Hilton and J. Hall. On applying software development best practice to FPGAs in safety-critical systems. In *10th International Conference on Field Programmable Logic and Applications.* Springer-Verlag, 2000.

[44] A. Hilton and J. Hall. Developing critical systems with PLD components. In *Formal Methods for Industrial Critical Systems (FMICS'05)*, 2005.

[45] M. Höst, B. Regnell, and C. Wohlin. Using students as subjects - a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5:201–214, 2000.

[46] IEC. *IEC61508: Functional safety for electrical / electronic / programmable electronic safety-related systems.* International Electrotechnical Comission, 1998.

[47] IEC. *IEC61508-2 Annex E: Special architecture requirements for ASICs with on-chip redundancy.* International Electrotechnical Comission, working draft.

[48] ISO. *ISO/WD 26262 - Road vehicles - Functional Safety.* International Organization for Standardization, working draft (2007).

[49] J. Jou and C. Liu. Coverage analysis techniques for HDL design validation. In *6th Asia Pacific Conference on Chip Design Languages*, 1999.

[50] F. L. Kastensmidt, L. Carro, and R. Reis. *Fault-Tolerance Techniques for SRAM-based FPGAs.* Springer, 2006.

[51] R. Katz, K. LaBel, J. Wang, B. Cronquist, R. Koga, S. Penzin, and G. Swift. Radiation effects on current field programmable technologies. *IEEE Transactions on Nuclear Science*, 44, 1997.

[52] B. Knegtering. *Safety Lifecycle Management In The Process Industries - The development of a qualitative safety-related information analysis technique.* PhD thesis, Technische Universiteit Eindhoven, 2002.

[53] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Softw. Eng.*, 12:96–109, 1986.

[54] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[55] J. Lach, W. H. Mangione-Smith, and M. Potkonjak. Low overhead fault-tolerant FPGA systems. *IEEE Transactions on VLSI Systems*, 6, 1998.

[56] J. H. Lala and R. E. Harper. Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE*, 82, 1994.

[57] G. Lehmann, B. Wunder, and K. Müller-Glaser. A VHDL reuse workbench. In *Design Automation Conference (EURO-DAC '96)*, pages 412–417. IEEE, 1996.

[58] N. G. Leveson. *Safeware - System Safety and Computers.* Addison-Wesley, 1995.

[59] N. G. Leveson. The role of software in spacecraft accidents. *AIAA Journal of Spacecraft and Rockets*, 41(4):564–575, July 2004.

[60] N. G. Leveson and K. A. Weiss. Making embedded software reuse practical and safe. In *SIGSOFT Software Engineering Notes.* ACM, 2004.

[61] B. Littlewood and D. R. Miller. Conceptual modeling of coincident failures in multiversion software. *IEEE Trans. Softw. Eng.*, 15(12):1596–1614, 1989.

[62] B. Littlewood, P. Popov, and L. Strigini. A note on modelling functional diversity. In *Reliability Engineering an System Safety*, 1999.

[63] M. Lyu, J. Horgan, and S. London. A coverage analysis tool for the effectiveness of software testing. *IEEE Transactions on Reliability*, 43(4):527–535, Dec 1994.

[64] M. R. Lyu and Y.-T. He. Improving the N-version programming process through the evolution of a design paradigm. *IEEE Transactions on Reliability*, 42(2):179–189, Jun 1993.

[65] R. Mariani and P. Fuhrmann. Comparing fail-safe microcontroller architectures in light of IEC 61508. In *22nd Int. Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT'07)*, pages 123 – 131. IEEE, Sept 2007.

[66] J. McCollum, R. Lambertson, J. Ranweera, J. Moriarta, J.-J. Wang, F. Hawley, and A. Kundu. Reliability of antifuse-based field programmable gate arrays for military and aerospace applications. In *4th Military and Aerospace Applications of Programmable Devices and Technologies International Conference*, 2001.

[67] A. Mili, S. Yacoub, E. Addy, and H. Mili. Toward an engineering discipline of software reuse. *IEEE Software*, 16:22–31, 1999.

[68] MISRA-C:2004. *Guidelines for the use of the C language in critical systems.* Motor Industry Software Reliability Association, 2004.

[69] K.-H. Moller and D. J. Paulish. An empirical investigation of software fault distribution. In *IEEE First International Software Metrics Symposium*, pages 82–90, 1993.

[70] E. Monmasson and M. N. Cirstea. FPGA design methodology for industrial control systems - a review. *IEEE Transactions on Industrial Electronics*, 54(4):1824–1842, 2007.

[71] S. Montenegro. *Sichere und fehlertolerante Steuerungen.* Hanser Verlag, 1999.

[72] J. Morris and P. Koopman. Representing design tradeoffs in safety-critical systems. In *Workshop on Architecting Dependable Systems (WADS)*, 2005.

[73] M. Mrva. Reuse factors in embedded systems design. *IEEE Computer*, 30(8):93–95, Aug 1997.

[74] NASA. TRST* and the IEEE JTAG 1149.1 Interface. NA-GSFC-2004-04. 2004.

[75] B. Nicolescu, R. Velazco, M. Sonza-Reorda, M. Rebaudengo, and M. Violante. A software fault tolerance method for safety-critical systems: Effectiveness and drawbacks. In *15th Symposium on Integrated Circuits and Systems Design*, 2002.

[76] F. Nicoli and L. Pierre. Formal verification of behavioral VHDL specifications: a case study. In *Conference on European Design Automation*, 1994.

[77] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43, 1996.

[78] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalarprocessors. *IEEE Trans. on Reliability*, 2002.

[79] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 55–64, 2002.

[80] D. L. Parnas and M. Lawford. Inspection's role in software quality assurance. *IEEE Software*, 20(4):16–20, July 2003.

[81] D. L. Parnas, J. van Schouwen, and S. P. Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33:636–648, 1990.

[82] Philips. *Data sheet: SJA1000 stand-alone CAN controller*, 2000. URL: www.nxp.com/ acrobat_download/ datasheets/ SJA1000_3.pdf (accessed: 10.04.2008).

[83] P. Popov, L. Strigini, and B. Littlewood. Choosing between fault-tolerance and increased V&V for improving reliability. In *Int. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA' 2000)*. CSREA Press, 2000.

[84] P. Popov, L. Strigini, and A. Romanovsky. Choosing effective methods for design diversity - how to progress from intuition to science. In *18th International Conference, SAFECOMP*, 1999.

[85] B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin. Improving FPGA design robustness with partial TMR. In *Military and Aerospace Programmable Logic Devices (MAPLD) International Conference*, 2005.

[86] L. Prechelt. *Kontrolierte Experimente in der Softwaretechnik.* Springer, 2001.

[87] R. B. R. Katz and K. Erickson. Logic design pathology and space flight electronics.

[88] C. Rockwood. Keeping time with clock domain crossings in FPGAs. *Chip Design Trends Reports*, 2007. www.chipdesignmag.com.

[89] J. J. Rodríguez-Andina, J. Alvarez, and E. Mandado. Design of safety systems using field programmable gate arrays. In *Workshop on Field-Programmable Logic and Applications (FPL'94)*, 1994.

[90] J. Rose, A. E. Gamal, and A. Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. In *Proceedings of the IEEE*, 1993.

[91] F. Salewski. Automatisches Testen eingebetteter Echtzeitsysteme: Testumgebung für eine Automobilanwendung. Technical Report 2006-1, RWTH Aachen - Lehrstuhl Informatik 11, 2006.

[92] F. Salewski. FAT/AK31 UAK Zuverlässigkeit - Bericht zur Studie. Technical report, RWTH Aachen - Lehrstuhl Informatik 11, to be published (2008).

[93] F. Salewski and S. Kowalewski. Zuverlässigkeitsmechanismen für Eingebettete Systeme. In *Workshop on Zuverlässigkeit in eingebetteten Systemen - Ada Deutschland Tagung*, pages 39–51. Shaker Verlag, 2005.

[94] F. Salewski and S. Kowalewski. Zuverlässigkeitsmechanismen für Eingebettete Systeme. *GI Softwaretechnik-Trends*, 25(4):7–8, 2005.

[95] F. Salewski and S. Kowalewski. Exploring the differences of FPGAs and microcontrollers for their use in safety-critical embedded applications. In *IEEE Symposium on Industrial Embedded Systems (IES'06)*, pages 1–4. IEEE, Oct. 2006.

[96] F. Salewski and S. Kowalewski. Achieving highly reliable embedded software: An empirical evaluation of different approaches. In F. Saglietti and N. Oster, editors, *Proc. of 26th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP'07)*, volume 4680/2007 of *Lecture Notes in Computer Science*, pages 270–275. Springer, Sept. 2007.

[97] F. Salewski and S. Kowalewski. Achieving highly reliable embedded software: An empirical evaluation of different approaches. Technical Report AIB-2007-08, Dep. of Computer Science, RWTH Aachen University, 2007.

[98] F. Salewski and S. Kowalewski. The effect of hardware platform selection on safety-critical software in embedded systems: Empirical evaluations.

In *IEEE Symposium on Industrial Embedded Systems (SIES'07)*, pages 78–85. IEEE, July 2007.

[99] F. Salewski and S. Kowalewski. Hardware platform design decisions in embedded systems - a systematic teaching approach. In *Special Issue on the Second Workshop on Embedded System Education (WESE)*, volume 4, pages 27–35. SIGBED Review, ACM, Jan. 2007.

[100] F. Salewski and S. Kowalewski. Testing issues in empirical reliability evaluation of embedded real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium, WiP session (RTAS'07)*, volume WUCSE-2007-17, pages 48–51. Washington University in St. Louis, March 2007.

[101] F. Salewski and S. Kowalewski. The effect of real-time software reuse in FPGAs and microcontrollers with respect to software faults. In *Symposium on Industrial Embedded Systems (SIES'08)*. IEEE, June 2008. to appear.

[102] F. Salewski and M. Lang. *FAT/AK31 UAK Zuverlässigkeit - Systemspezifikation für das Anwendungsbeispiel: Cabrioverdecksteuerung*, 2008.

[103] F. Salewski and A. Taylor. Fault handling in FPGAs and microcontrollers in safety-critical embedded applications: A comparative survey. In H. Kubatova, editor, *10th Euromicro Conference on Digital System Design (DSD'07)*, pages 124–131. IEEE, Aug. 2007.

[104] F. Salewski and A. Taylor. Systematic considerations for the application of FPGAs in industrial applications. In *International Symposium on Industrial Electronics (ISIE'08)*, pages 2009–2015. IEEE, July 2008.

[105] F. Salewski, D. Wilking, and S. Kowalewski. Diverse hardware platforms in embedded systems lab courses: A way to teach the differences. In *Special Issue: The First Workshop on Embedded System Education (WESE)*, volume 2, pages 70–74. SIGBED Review, ACM, Oct. 2005.

[106] F. Salewski, D. Wilking, and S. Kowalewski. The effect of diverse hardware platforms on N-version programming in embedded systems - an empirical evaluation. In *Proc. of the 3rd. Workshop on Dependable Embedded Sytems (WDES'06)*, volume TR 105/2006, pages 61–66. Vienna University of Technology, Nov. 2006.

[107] C. E. Salloum, A. Steininger, P. Tummeltshammer, and W. Harter. Recovery mechanisms for dual core architectures. In *21st IEEE Int. Symposium*

on Defect and Fault Tolerance in VLSI Systems (DFT '06), pages 380–388. IEEE, Oct 2006.

[108] P. Schaumont, R. Cmar, S. Vernalde, M. Engels, and I. Bolsens. Hardware reuse at the behavioral level. In *36th Design Automation Conference (DAC'99)*, pages 784–789. IEEE, 1999.

[109] M. Schlager, W. Herzner, A. Wolf, O. Gründonner, M. Rosenblattl, and E. Erkinger. Encapsulating application subsystems using the DECOS core OS. In *Int. Conference on Computer Safety, Reliability and Security (SAFECOMP'06)*, number 4166/2006 in LNCS, pages 386 – 397. Springer, Sept 2006.

[110] B. Schlich and S. Kowalewski. [mc]square: A model checker for microcontroller code. In *Int'l Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006)*. IEEE, 2006.

[111] B. Schlich, F. Salewski, and S. Kowalewski. Applying model checking to an automotive microcontroller application. In *Proc. IEEE 2nd Int'l Symp. Industrial Embedded Systems (SIES 2007)*, pages 209–216. IEEE, July 2007.

[112] T. Shimeall and N. Leveson. An empirical comparison of software fault tolerance and fault elimination. *IEEE Transactions on Software Engineering*, 17(2):173–182, 1991.

[113] T. Siegbert. Untersuchung eines Dual-Core-Mikrocontrollers für sicherheitskritische Automobilanwendungen nach ISO 26262 . *Diplomarbeit, RWTH Aachen - Informatik 11*, 2008.

[114] A. Sikora and R. Drechsler. *Software-Engineering und Hardware-Design.* Hanser Verlag, 2002.

[115] D. I. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanović, and M. Vokáč. Challenges and recommendations when increasing the realism of controlled software engineering experiments. *ESERNET 2001-2003, LNCS 2765*, pages 24–38, 2003.

[116] D. J. Smith and K. G. L. Simpson. *Functional Safety - A straightforward guide to IEC61508 and related standards.* Butterworth - Heinemann, 2001.

[117] N. Storey. *Safety-Critical Computer Systems.* Prentice Hall, 1996.

[118] H. Thane and H. Hansson. Handling interrupts in testing of distributed real-time systems. In *Real-Time Computing Systems and Applications (RTCSA '99)*, 1999.

[119] F. Vahid and T. Givargis. *Embedded System Design - A unified Hardware/Software Introduction.* Wiley, 2002.

[120] F. Vargas and A. Amory. Transient-fault tolerant VHDL descriptions: A case-study for area overhead analysis. In *Ninth Asian Test Symposium (ATS'00)*, 2000.

[121] R. Velazco and S. Rezgui. Transient bitflip injection in microprocessor embedded applications. In *6th IEEE International On-Line Testing Workshop (IOLTW)*, 2000.

[122] H. Vranken, M. Witteman, and R. van Wuijtswinkel. Design for testability in hardware software systems. *IEEE Design & Test of Computers*, 13(3):79–86, 1996.

[123] M. Wang. SEU Mitigation Techniques for Xilinx Virtex2 Pro FPGA. 2004.

[124] B. Wells and S. M. Loo. On the use of distributed reconfigurable hardware in launch control avionics. In *Digital Avionics Systems Conference (DASC).* IEEE, 2001.

[125] M. Wirthlin, E. Johnson, and N. Rollins. The reliability of FPGA circuit designs in the presence of radiation induced configuration upsets. In *IEEE Symposium on field-Programmable Custom computing Machines (FCCM)*, 2003.

[126] C. Wohlin, P. Runeson, M. Höst, M. Ohlson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering.* Kluwer Academic Publishers, 2000.

[127] Xilinx. *CoolRunner-II CPLD Family Data Sheet.* URL: www.xilinx.com/support/documentation/data_sheets/ds090.pdf (accessed 10.04.2008).

[128] Xilinx. *Spartan-3 FPGA Family Data Sheet.* URL: www.xilinx.com/support/documentation/data_sheets/ds099.pdf (accessed 10.04.2008).

[129] Xilinx. Xapp564 PPC405 lockstep system on ML310. Jan 2007.

[130] Xilinx. Correcting single-event upsets through virtex partial configuration. June 2000.

Aachener Informatik-Berichte

This list contains all technical reports published during the past five years. A complete list of reports dating back to 1987 is available from

`http://aib.informatik.rwth-aachen.de/.`

To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

2003-01 *	Jahresbericht 2002
2003-02	Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting
2003-03	Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations
2003-04	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs
2003-05	Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard
2003-06	Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates
2003-07	Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung
2003-08	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs
2004-01 *	Fachgruppe Informatik: Jahresbericht 2003
2004-02	Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic
2004-03	Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting
2004-04	Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming
2004-05	Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming

2004-06	Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming
2004-07	Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination
2004-08	Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information
2004-09	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity
2004-10	Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules
2005-01 *	Fachgruppe Informatik: Jahresbericht 2004
2005-02	Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: “Aachen Summer School Applied IT Security”
2005-03	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions
2005-04	Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem
2005-05	Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots
2005-06	Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information
2005-07	Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks
2005-08	Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut
2005-09	Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures
2005-10	Benedikt Bollig: Automata and Logics for Message Sequence Charts
2005-11	Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture
2005-12	Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems
2005-13	Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments

2005-14	Felix C. Freiling, Sukumar Ghosh: Code Stabilization
2005-15	Uwe Naumann: The Complexity of Derivative Computation
2005-16	Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)
2005-17	Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)
2005-18	Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"
2005-19	Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers
2005-20	Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.
2005-21	Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited
2005-22	Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins
2005-23	Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves
2005-24	Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks
2006-01 *	Fachgruppe Informatik: Jahresbericht 2005
2006-02	Michael Weber: Parallel Algorithms for Verification of Large Systems
2006-03	Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler
2006-04	Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation
2006-05	Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F
2006-06	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color

2006-07	Thomas Colcombet, Christof Löding: Transforming structures by set interpretations
2006-08	Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs
2006-09	Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking
2006-10	Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed
2006-11	Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers
2006-12	Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning
2006-13	Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities
2006-14	Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"
2006-15	Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices
2006-16	Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness
2006-17	Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines
2007-01 *	Fachgruppe Informatik: Jahresbericht 2006
2007-02	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations
2007-03	Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase
2007-04	Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation
2007-05	Uwe Naumann: On Optimal DAG Reversal
2007-06	Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking
2007-07	Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications

2007-08	Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches
2007-09	Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption
2007-10	Martin Neuhäußer, Joost-Pieter Katoen: Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes
2007-11	Klaus Wehrle (editor): 6. Fachgespräch Sensornetzwerke
2007-12	Uwe Naumann: An L-Attributed Grammar for Adjoint Code
2007-13	Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs
2007-14	Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes
2007-15	Volker Stolz: Temporal assertions for sequential and concurrent programs
2007-16	Sadeq Ali Makram, Mesut Güneç, Martin Wenig, Alexander Zimmermann: Adaptive Channel Assignment to Support QoS and Load Balancing for Wireless Mesh Networks
2007-17	René Thiemann: The DP Framework for Proving Termination of Term Rewriting
2007-18	Uwe Naumann: Call Tree Reversal is NP-Complete
2007-19	Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann: Adjoints for Time-Dependent Optimal Control
2007-20	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf: Three-Valued Abstraction for Probabilistic Systems
2007-21	Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre: Compositional Modeling and Minimization of Time-Inhomogeneous Markov Chains
2007-22	Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking: Uncoordinated Two-Sided Markets
2008-01 *	Fachgruppe Informatik: Jahresbericht 2007
2008-02	Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing
2008-03	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René :Thiemann, Harald Zankl: Maximal Termination
2008-04	Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler

Curriculum Vitae

Name	Dirk-Falk Salewski
Date of birth	13.05.1979
Place of birth	Siegen

2004 – 2008	Research assistant at the Embedded Software Laboratory at RWTH Aachen University, Germany
1998 – 2004	Study of electrical engineering at the University of Siegen, Germany
1989 – 1998	Grammar school Neunkirchen, Germany
1985 – 1989	Elementary school Burbach, Germany