CHALMERS

Evaluation of Error Handling Mechanisms for Automotive Embedded Systems

Master's Thesis in Computer Systems and Networks

ANTON HEMLIN
ANDREAS ÅKESSON

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2014

Evaluation of Error Handling Mechanisms for Automotive Embedded Systems

Anton Hemlin
Andreas Åkesson

Examiner: Johan Karlsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: The truck is from http://images.volvotrucks.com/ and the lightning is from www.clipartpal.com. The pictures are used in accordance to the respective page's image license agreement.

Department of Computer Science and Engineering
Gothenburg, Sweden June 2014

Abstract

This thesis presents an evaluation of the effectiveness and time overhead for plausibility checks in automotive electrical and electronic systems. Plausibility checks aim to detect errors caused by software bugs and random hardware failures. They are commonly used to ensure safety and robustness. There are two ways to implement plausibility checks, the traditional with checks directly in the source code and the use of a library. We have created a proof-of-concept implementation of an AUTOSAR compliant library that provides a standardised interface for common plausibility checks. We demonstrate the usefulness of the library for several AUTOSAR applications. We investigate the effectiveness of plausibility checks on three applications and found that plausibility checks are effective in detecting errors in input parameters to software modules. We compare the time overhead for implementing plausibility checks with the library versus implementing them directly in the source code. Using the library increases the execution time with 2.0 μs per library call on our hardware compared to having no checks at all. In comparison, having the plausibility checks directly in the source code increase the execution time with 0.25 μs per check.

Keywords: *AUTOSAR, functional safety, robustness, error handling, plausibility checks*

Acknowledgements

We would like to thank our examiner Prof. Johan Karlsson at Chalmers University of Technology for his help. We also wish to express gratitude to our supervisors at Volvo Group Trucks Technology, Dr. Mafijul Islam and Johan Haraldsson, for using their valuable time to give us tremendous support and expert guidance throughout the work. Finally, we also wish to offer special thanks to Fredrik Bernin for helping out with technical difficulties.

Andreas Åkesson & Anton Hemlin, Gothenburg June 2014

Contents

List of Abbreviations

Abbreviation	**Description**
ABS	Anti-lock Braking System
ALEM	Application Level Error Manager
ASIL	Automotive Safety Integrity Level
AUTOSAR	AUTomotive Open System ARchitecture
BBW	Brake-By-Wire
BLFI	Binary Level Fault Injection
BSW	Basic SoftWare
CAN	Controller Area Network
CRASH	Catastrophic, Restart, Abort, Silent and Hindering
E2E	End-to-End
ECU	Electronic Control Unit
E/E	Electrical and/or Electronic
FDIR	Fault Detection, Isolation and Recovery
FMECA	Failure Mode Effect and Criticality Analysis
GSPN	Generalised Stochastic Petri Nets
HWIFI	Hardware Implemented Fault Injection
LIN	Local Interconnect Network
RTE	Run-Time Environment
SW-C	SoftWare Component
SWIFI	Software Implemented Fault Injection
VFB	Virtual Functional Bus

1

Introduction

Since the 1970s the use of Electrical and/or Electronic (E/E) systems within the automotive industry has increased and all indications are that it will continue to increase [1]. The E/E systems replace mechanical equivalences or implement new functions to assist the driver. An E/E system with its hardware and software components is complex and prone to errors but is nevertheless trusted to control safety critical functions. This puts tremendous pressure on developers to design E/E systems which perform as expected during operation even in the presence of faults.

Building highly reliable E/E systems is however a demanding task. It was reported in March 2014 that nearly 60-70% of all vehicle recalls in North America and Europe are due to software errors [2]. In 2009 Toyota had problems with unintended acceleration of cars which are presumed to have been caused by software defects [3]. Recently General Motors was forced to large recalls due to a malfunctioning ignition switch in several car models that could be linked to multiple deaths [4].

Safety has always been an important property in the automotive industry and with the increased amount of functionality relying on E/E systems the need for standardising the requirements for functional safety has increased. To this end, a new standard for ensuring functional safety in automotive E/E systems called ISO 26262 was introduced in November 2011. This standard defines functional safety as ”absence of unreasonable risk due to hazards caused by malfunctioning behaviour of E/E systems” [5].

Another important standard in the automotive industry is Automotive Open System Architecture (AUTOSAR) which is an open standard that is developed in cooperation between automobile manufacturers, suppliers and tool developers. AUTOSAR is an architecture for automotive software and its main goals are to standardise basic software functionality, improve the scalability of solutions, simplify the collaboration between various players and to facilitate functional safety [6].

To ensure functional safety it is essential that an E/E system is equipped with effective error handling mechanisms which are implemented in either hardware or software.

These mechanisms handles errors by using detection, isolation and recovery. AUTOSAR and ISO 26262 specify several different software mechanisms for error handling [5, 6]. This thesis addresses implementation of plausibility checks in AUTOSAR. Plausibility checks is a mechanism that is commonly used to detect data errors caused by software bugs or transient hardware faults. Usually plausibility checks are implemented directly in the source code. Using a library with plausibility checks is another possibility. We will investigate the possibilities for creating an AUTOSAR compliant library with plausibility checks.

Robustness is defined as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [7]. To achieve a high degree of robustness for an application it is important to validate its input data. Plausibility checks can be used to check input values, which leads us to the first research question:

- **How effective are plausibility checks in improving the robustness of application software?**

We address this question by performing fault injection experiments on three applications. Fault injection is when faults are intentionally introduced in a system to study the behaviour. Plausibility checks are then implemented to detect errors in the input data.

One major concern is the additional overhead to the execution time that the use of a library might add. We will evaluate the execution time overhead when using an AUTOSAR compliant library for plausibility checks. We will compare with the execution time overhead when plausibility checks are implemented directly in the source code. The second research question is therefore:

- **What is the time overhead for implementing plausibility checks in an AUTOSAR library?**

The library is tested on two AUTOSAR applications, a simple LED blinking application and a complete Brake-By-Wire system. The impact on the execution time of the applications is evaluated with and without the library. We ask this research question because execution time is an important factor when deciding if the library should be used in an application.

1.1 Research Methodology

The research method for this Master's thesis is derived from Peffers et al. [8]. They describe the following five phases of a research process: *Problem Analysis & Motivation*, *Design & Development*, *Demonstration*, *Evaluation* and *Communication*. These phases are illustrated in Figure 1.1 where it also can be seen that iteration over *Design & Development*, *Demonstration* and *Evaluation* should be performed until a satisfying evaluation is achieved.

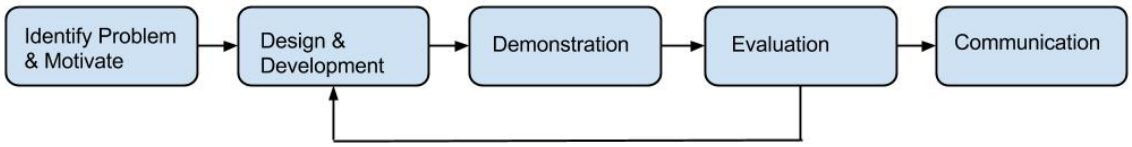


Figure 1.1: Research Methodology

In *Problem Analysis & Motivation* the specific research problem and a motivation to justify the value of a solution are defined in collaboration with the stakeholders. In *Design & Development* a solution is implemented for how to apply the error handling mechanisms and it is tested on different applications. In the *Demonstration* phase a demonstration/proof-of-concept is constructed in the AUTOSAR environment by using a realistic AUTOSAR software application. In *Evaluation* it is observed how well the work supports the objectives of a solution that were previously defined. In *Communication* the result is communicated to relevant audiences in the form of a written report and oral presentations.

1.2 Stakeholders

This thesis was performed at the Department of Electrical and Embedded Systems of Advanced Technology and Research at Volvo Group Trucks Technology. It was conducted within the scope of the VeTeSS (Verification and Testing to Support Functional Safety Standards) research project which aim to standardise methods and tools to verify safety properties [9].

1.3 Thesis Outline

The remainder of this thesis is organised in six chapters. Chapter 2 provides an introduction to dependable computing. Chapter 3 describes the AUTOSAR standard, functional safety and other basic concepts. Different error handling mechanisms are presented in Chapter 4. Chapter 5 contains the library design and an evaluation of plausibility checks and the library. A discussion of the results are presented in Chapter 6. Chapter 7 finishes the thesis by presenting our final conclusions.

2

Introduction to Dependable Computing

This chapter gives an overview of the definition of basic dependability concepts and taxonomy. Before the definition of dependability there are a few basic concepts we want to define. A *computer system* is an entity that interacts with other entities i.e. other computer systems, humans or the physical world. The *environment* for a given system is the other systems and the border between the system and the environment is called the *system boundary*. A service is the behaviour that a user expects from a system. One of Avizienis et al.'s [10] definitions of dependability is "the ability to avoid service failures that are more frequent and more severe than is acceptable". Dependability can be broken down into three elements: attributes, threats and means.

2.1 Attributes

The five key attributes for describing the dependability of a system are:

- **Availability:** readiness for correct service.
- **Reliability:** continuity of correct service.
- **Safety:** absence of catastrophic consequences on the user(s) and the environment.
- **Integrity:** absence of improper system alterations.
- **Maintainability:** ability to undergo modifications and repairs.

Availability, reliability, safety and maintainability are quantifiable by direct measurements. Availability is the amount of time a system delivers a correct service according to its specification. It is calculated by dividing the time the system works correctly by the total-time of operation including down-time. Reliability is the probability that the system works as expected at a given time. In safety the failures are divided in two parts -

catastrophic failures and safe failures. Safety is the probability that the system does not exhibit a catastrophic failure. Availability, reliability and safety can be calculated with for example reliability block diagrams, Markov models or Generalised Stochastic Petri Nets (GSPN) [11]. Maintainability is added as an extra parameter in the reliability calculations, where parts of a system can be repaired with different repair rates. Integrity cannot be directly measured and must be assessed with help of relevant information about the system.

2.2 Threats

Threats are the events that can affect the dependability of the system and they are called faults, errors and failures. A *service failure* (abbreviated failure) is the transition from a state where the system provides correct service according to its specifications to a state that provides incorrect service. The deviation is called an *error*. The real or guessed cause of the error is called a *fault*. An error occurs when a fault causes the system to enter another state than normal. Many errors never reach the system's external state causing a failure. One possibility for this to occur is if a data error is masked by a write operation before the value is read.

The concept of the Fault-Error-Failure chain can be explained by a simple example. We have two components - A and B, where A delivers a service to B. First a fault activates an error in component A which propagates to a failure which will cause A to deliver incorrect service to B. Then in component B the incorrect input will be seen as a fault that activates an error which may propagate to a service failure in B too.

2.3 Means

The means are ways to increase the dependability of a system and they are divided in four major groups by Avizienis et al. [10]:

- **Fault prevention** means to prevent the occurrence or introduction of faults.
- **Fault tolerance** means to avoid service failures in the presence of faults.
- **Fault removal** means to reduce the number and severity of faults.
- **Fault forecasting** means to estimate the present number, the future incidence, and the likely consequences of faults.

Fault prevention focuses on reducing the number of faults introduced in the system by improving the development process for both software (information hiding, modularisation, strong typed languages) and hardware (e.g. design rules).

Fault tolerance aims to avoid failures in the presence of faults with the help of different fault tolerance techniques. Figure 2.1 provides a classification of the techniques. Before the system can recover, the error needs to be detected. Concurrent detection is performed during normal operations while preemptive detection takes place while normal operations are suspended. Error handling removes the errors from the system in three different ways.

Rollback brings the system back to a state before the error occurred and compensation uses redundancy information in the erroneous state to mask the error. Rollforward brings the program to a new state without the error where the state is loaded from another component. Fault handling prevents the fault from being activated again by the use of four main categories of techniques. Diagnosis identifies and records the cause of the error. Reconfiguration reassigns the task to another component. Isolation excludes the faulty component from further participation making the fault dormant. Reinitialisation checks, updates and records the new configuration and update system tables and records.

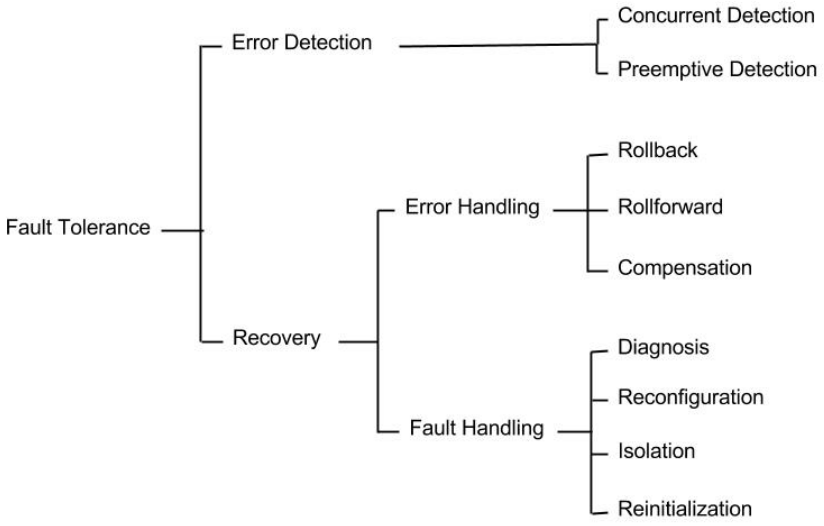


Figure 2.1: Fault tolerance techniques

Fault removal can be divided in two parts: fault removal during development and fault removal during use. Fault removal during development focuses on verification and used techniques are static analysis, theorem proving, model checking, symbolic execution and testing. The fault tolerance mechanisms are also tested, either by formal static verification or by fault injection. Fault removal during use consists of isolating and then removing reported faults.

Fault forecasting is performed by executing an evaluation of the system behaviour focusing on two different aspects: qualitative evaluation and quantitative evaluation. Qualitative evaluation aims to identify and classify failure modes that lead to system failures and quantitative evaluation investigates the probabilities for which some attributes are satisfied.

3

Basic Concepts

This chapter introduces and explains key concepts. First the AUTOSAR standard is explained. The functional safety standard ISO 26262 is introduced next. Then the concept of robustness is introduced and the different parts are explained, after that fault injection is described; a technique to test the robustness of systems or the effectiveness of fault handling mechanisms by injecting faults into them. The chapter is concluded with a description of different error models for how to choose test cases for a system.

3.1 AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) [6] is an open and standardised software architecture for the automotive industry. Several car manufacturers and subcontractors participate in the development of the standard. The purpose with the standard is to provide a software architecture which is scalable and able to fulfil the requirements of future vehicles. A primary goal with AUTOSAR is that components developed independently by different actors are able to be integrated through well-defined interfaces. AUTOSAR provides a layered architecture divided into three main layers: application, Run-Time Environment (RTE) and Basic SoftWare (BSW). This architecture is illustrated in Figure 3.1.

3.1.1 Application Layer

The application layer is the top layer and contains Software Components (SW-Cs) that run on an Electronic Control Unit (ECU). Each SW-C can implement a complete function or be a part of a function of several SW-Cs. A SW-C is atomic and cannot be distributed among different ECUs.

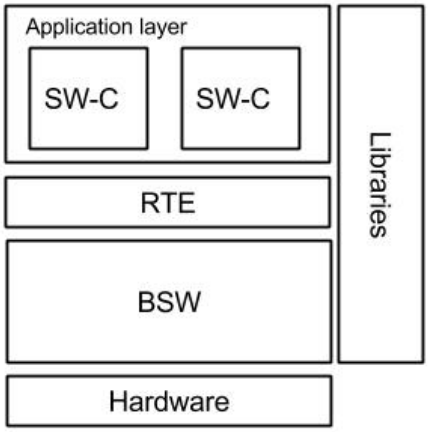


Figure 3.1: The layered AUTOSAR architecture.

3.1.2 Run-Time Environment

The RTE provides a communication interface to the application layer enabling SW-Cs to communicate with each other and the BSW. The interface is the same for communication within an ECU and between different ECUs using for example Controller Area Network (CAN). The RTE is automatically generated to be tailored for the specific SW-C and ECU. Several RTEs distributed over different ECUs are connected in the design model by the Virtual Function Bus (VFB). This abstract component interconnects the different SW-Cs and handles the information exchange between them. The VFB is the conceptualization of all hardware and system services in the vehicle. This makes it possible for the SW-Cs designers to focus more on the application instead of the infrastructure.

3.1.3 Basic Software

The BSW contains services which are required by SW-Cs to fulfil their tasks. There are standardised services such as communication (CAN, LIN, etc.) and operating system to support SW-Cs. There are also ECU specific services, for example Complex Device Drivers which have direct access to the hardware for critical applications.

3.1.4 Libraries

Libraries in AUTOSAR serve modules in BSW and SW-Cs with different appropriate functions. There exist eight different standard libraries with functions such as a CRC library that provides support for CRC calculations and End-to-End Communication Protection Library which protects data communication between SW-Cs. There is also a possibility for developers to specify their own libraries. When implementing libraries some rules must be followed to assure AUTOSAR compatibility [12].

3.1.5 Error Handling

AUTOSAR supplies a document [13] which addresses error handling at the application level. A chain of events consisting of Fault Detection, Isolation and Recovery (FDIR) is described. Detection: how to discover that a fault has occurred - for example that a value is out of range. Isolation: how to prevent the erroneous system from affecting other systems dependent on the erroneous system. Recovery: how to recover from an failure and make the system function as usual or with a degraded functionality.

The following error types are listed:

- **Data:** when the error is manifested in the value of a variable, parameter or message.
- **Program flow:** when the program executes different paths than expected.
- **Access:** occurs when the program accesses a partition without permission.
- **Timing:** when a message arrives early, late or does not arrive at all.
- **Asymmetric:** an error where a component produces different output with the same input.

AUTOSAR lists thirteen different error handling mechanisms that can be used at the different levels of FDIR. It is only suggested which mechanisms to consider and the developers have to decide which mechanisms to use. The thirteen different mechanisms are explained in Chapter 4.

3.2 Functional Safety

Functional safety is defined as ”absence of unreasonable risk due to hazards caused by malfunctioning behaviour of E/E systems” [5]. ISO 26262, published in November 2011, is a functional safety standard for the automotive industry [5] which was adapted from the generic functional safety standard IEC 61508 [14]. Functional safety features form an important part of each product development phase: specification, design, implementation, integration, verification, validation, and production release. Product development for the software level is described in part 6 of the standard. The first version of the standard from 2011 only applies to serial produced passenger cars with a maximum gross weight of 3500 kg. It is expected that motorcycles, trucks and buses will be included in next version of the standard that is expected to be released around 2020.

ISO 26262 specifies risk-based integrity levels called Automotive Safety Integrity Level (ASIL) where each safety critical function needs to be associated with an ASIL depending on the risk for that particular function. There exist four ASIL levels A, B, C and D with the risk increasing from A up to D. Level A has the lowest requirements whereas level D has the highest requirements [5].

In Germany and several other countries the law of product liability states that car manufacturers are generally liable for deaths and any damage to the health of a person caused by a malfunction of the product [15]. The liability may be excluded only if the potential malfunction could not have been detected according to the so-called technical

state-of-the-art at the time of placing the product on the market [15]. Since ISO 26262 has been published it must be viewed as state-of-the-art and therefore car companies are obliged to follow it.

3.3 Robustness

Robustness is defined as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [7]. A project that investigated robustness faults is the Ballista project which was a 3-year research project between 1996 and 1999. The purpose of Ballista was to evaluate the robustness of OS API using black box testing [16]. Ballista decided a set of properties that should hold for robustness testing. The result was a scale called CRASH consisting of five different properties: catastrophic, restart, abort, silent and hindering. A test that causes a crash of the system would be categorised as catastrophic. If a hang occurs then the test is classified as restart. When a crash of the tested module happens it is an abort. Sometimes no errors is reported although the input was invalid then the failure is silent. The last one is hindering which means that the error code reported is misleading. Robustness of multi-layered software in the automotive domain is investigated in [17].

3.4 Fault Injection

The ISO26262 standard recommends that fault injection is used during the whole development process. The earliest work about fault injection was released as early as 1972 by Harlan Mill [18]. Since then several different fault injection techniques have been developed. In fault injection faults are intentionally inserted into the system under test and the resulting behaviour is observed. The difference between fault injection and other testing techniques are that testing focuses on using expected inputs while fault injection instead use expected faults.

There exist several different ways to classify fault injection techniques; one way is to divide it in two catagories depending on the target system. In hardware implemented fault injection (HWIFI), the hardware of the system is targeted and in software implemented fault injection (SWIFI) the software is targeted. In HWIFI the faults are injected through the hardware for example by using heavy ion rays or specially crafted test hardware [18]. SWIFI techniques use software to inject faults into the system under test. Fault injection can also be categorised as invasive or non-invasive. A non-invasive technique does not change the implementation in any way where invasive tecniques alter the implementation [18].

One way to classify faults is into representative and non-representative faults. [19] discusses which type of faults to inject and where to inject them. The authors describe an algorithm that could be used to determine where to inject faults to achieve good fault representativeness.

In binary level fault injection (BLFI) the faults are injected in the binary code. The differences with BLFI when the source code is unknown and when the source code is

known is described in [20]. Sometimes it can be difficult to find some programming constructs in the binary, which means that the BLFI can become inaccurate and the paper concludes that it is difficult to achieve accurate injections in binaries when it is a complex program. Some accuracy is lost due to macros, inline functions and other language constructs which information is not shown in the binary. BLFI for AUTOSAR based system is investigated in [21].

Because ISO26262 recommends fault injection during the whole development process, fault injection at different steps in the classical V-model is investigated in [22].

3.5 Error Models

There exist several different error models. In the subsections three models used for robustness testing are described.

3.5.1 Bit-Flip

A common error model is bit-flips where the idea is to simulate transient errors. The injection is done by flipping one bit at a time and thus the number of test cases is limited to the number of bits in the test vector. It is a very simple model which is both an advantage and a disadvantage. Simple to implement but may not be representative for all types of faults [23].

3.5.2 Data-Type Based Errors

Data-type based errors target a parameter and are therefore chosen depending on the type of the specific parameter. Boundary values are typically good values to use for data-type based errors. Suitable error cases for an integer are for example 1, 0, -1, MAX_INT and MIN_INT. The number of error cases needed depends on the complexity of the data type. For example a structure need a lot of error cases since each member of the structure would need to be injected [23].

3.5.3 Fuzzing

To fuzz a parameter is to inject pseudo-random data within the scope of the data type of the parameter. Due to the randomness different injections may achieve different results and therefore several experiments must be performed to draw valid conclusions [23]. The error model was first presented in [24] which also provides a tool to generate random data that can be used to fuzz input parameters [25].

4

Error Handling Mechanisms

In this chapter the 13 different AUTOSAR error handling mechanisms are described and their relations to the mechanisms in ISO26262. Table 4.1 describes what the error handling mechanisms can be used for in terms of fault detection, isolation and/or recovery. In Table 4.2 the ISO 26262 error handling mechanisms are showed together with the corresponding AUTOSAR mechanisms. It also contains information about at which ASIL levels the mechanisms are highly recommended (++ column) and recommended (+ column). The *italicised mechanism* may be considered as only partially related.

Mechanism	**Purpose (FDIR)**
Plausibility checks	Detection, Isolation
Substitute values	Recovery
Voting	Detection, Isolation, Recovery
Agreement	Detection, Isolation, Recovery
Checksums/Codes	Detection, Recovery
Execution sequence monitoring	Detection
Aliveness monitoring	Detection
Status and mode management	Isolation
Reconfiguration	Recovery
Reset	Recovery
Error filtering	Isolation
Memory protection	Detection, Recovery
Timing protection	Detection, Recovery

Table 4.1: AUTOSAR mechanisms. [26]

ISO 26262-6	**AUTOSAR (auxiliary documents)**	**ASIL(++)**	**ASIL(+)**
Range checks of input and output data	Plausibility checks [13]	A - D	
Plausibility check	Plausibility checks [13]	D	A - C
Control flow monitoring	Execution sequence monitoring [13], Logical program flow monitoring [27]	C - D	B
External monitoring facility	Execution sequence monitoring [13], Aliveness monitoring [13], Status and mode management [13], *Error filtering [13], Memory protection [13], Timing protection [13], Runtime timing protection and monitoring [27], Monitoring of local time [27]*	D	B - C
Detection of data errors	Checksums/Codes [13]	None	A - D
Diverse software design	*Voting [27], Agreement [13]*	D	C
Static recovery mechanism	Substitute values [13], Reconfiguration [13], *Reset [13]*	None	A - D
Graceful degradation	Substitute values [13], Reconfiguration [13]	C - D	A - B
Independent parallel redundancy	*Voting [13], Agreement [13]*	D	C
Correcting codes for data	Checksums/Codes [13]	None	A - D

Table 4.2: Error handling mechanisms of ISO 26262 and AUTOSAR. [26]

4.1 Overview of Mechanisms

An overview of the 13 different error handling mechanisms in the AUTOSAR standard is presented in the following subsections.

4.1.1 Plausibility Checks

A plausibility check verifies that a group of variables hold for a specific condition. There are in principle two different types of plausibility checks: validity check and comparison of values. Validity checks detect errors by controlling if a variable for example is within a specific range or equals a value. Comparison can detect errors by comparing multiple values and identify differences. To isolate the error, a validity check can be used to indicate which variable that is erroneous. Because plausibility checks verify if data values are plausible it is only possible to detect data errors.

4.1.2 Substitute Values

When a data error in a variable has been detected, the substitute value mechanism can be used. The value of the variable is then substituted with another value that would make the upcoming execution render a valid result, for example a default value or the last accepted value, however the result might be of degraded quality. Substitute value only masks an error as it does not provide recovery of the fault leading to the erroneous value. As the variable that is to be substituted is highly depending on the context this mechanism is application specific.

4.1.3 Voting

When voting is used several copies of redundantly software are executed and then a voting is performed with each result from the different executions. Examples of voting algorithms are simple majority and 2 out of 3 which often are implemented in a special voter component. Voting also supports recovery because a correct value can still be extracted even in the presence of an error.

4.1.4 Agreement

Components can agree on what value to use by exchanging their local version of the value. The difference from voting is that components reach a decision together instead of having a separate component for that.

4.1.5 Checksums/Codes

Checksums/Codes mechanisms detect modifications of data by adding extra information. Sometimes it is also possible to recover erroneous data values and return it to its original state.

4.1.6 Execution Sequence Monitoring

Execution sequence monitoring detects if the execution may result in an erroneous execution path. It is possible to detect program flow errors which may have been caused by data errors, timing errors or asymmetric errors. The monitor itself cannot decide which one of them that caused the error.

4.1.7 Aliveness Monitoring

This mechanism detects units that are not alive, which means that they are not executed as expected in terms of periodicity. This could be done by monitoring the heartbeats, a signal that should be sent periodically, of a unit.

4.1.8 Status and Mode Management

Status and mode management uses meta-information to investigate the system to be able to isolate faulty components. To support this mechanism, application level components must be able to access the meta-information.

4.1.9 Reconfiguration

When an error has been detected, this mechanism can be used to configure the system to not use the erroneous component. It is also possible to set the system in a degraded mode and only provide a limited set of services.

4.1.10 Reset

Reset can be done at different levels, such as application reset and ECU reset. Reset can only handle transient faults since permanent faults still will remain after the restart.

4.1.11 Error Filtering

Filtering of errors is used to prevent setting the system in a less safe state than before, during a recovery action.

4.1.12 Memory Protection

Memory protection is used to protect the memory from errors that propagate between different memory areas. For mutual protection of applications partitions are defined to create error confinement regions.

4.1.13 Timing Protection

The purpose of timing protection is to protect the system from activities that requires too much time to complete, and thereby hindering other tasks from executing. It can be used to detect if an application has exceeded its time slot.

4.2 Previous Work with Plausibility Checks

Korte et al. [28] describe plausibility checks in vehicles and they conclude that the plausibility checks improved the overall performance of their system.

Hiller [29] evaluates the detection effectiveness of executable assertions (e.g. plausibility checks) that is able to test one variable. The evaluation shows that there is a probability of 99% to detect an error in a monitored variable in his experiment. The corresponding probability to detect errors in random memory area locations are 81%. It is concluded that executable assertions is a feasible error detection technique when costs have to be low but detection coverage high.

Skarin and Karlsson [30] describe an application where plausibility checks perform well. Two different software error handling mechanisms are implemented in a brake-by-wire controller. One mechanism protects the stack pointer from being corrupted by storing an extra copy in a register. When returning from a function the restored stack pointer is compared with its copy. If the comparison fails either the original or the copy is corrupted, to solve this a soft reset of the controller is performed to reset the state. The other mechanism checks the state of the integrator in the brake by wire controller. There is a limit of how much the state could change between two iterations and if this limit is exceeded there is a rollback to the previous state. The evaluation of these two mechanisms implemented in a prototype showed that critical failures were reduced from 1.2% to 0.05%.

5

Library Design and Evaluation

This chapter describes three things. Firstly, the requirements for an AUTOSAR compliant library and how we have implemented one. Secondly, the effectiveness of plausibility checks is evaluated by adding it to three applications. Finally, our library is tested on two AUTOSAR applications and the time requirement is measured.

5.1 Library Design

In this section the AUTOSAR standard requirements for a library are investigated and then our implementation of a library is described.

5.1.1 Investigation of the Standard

AUTOSAR libraries were introduced in Section 3.1.4 where it was stated that an AUTOSAR library must follow certain requirements to assure compatibility [12], these requirements include but are not limited to:

- Not require an initialisation phase.
- Not require a shut-down phase.
- Re-entrant, shall not write to global or static variables.
- All function names must start with the library short name.
- Should use types defined in std_types.h and platform_types.h or define new types in its header file.
- Library functions are only allowed to call other library functions.
- Libraries should check input at run time and return error codes to the caller.
- Use of macros should be avoided.
- Those using the library should include its header file and not call via the RTE.

The reason for the first two requirements is that it should be possible for a BSW component to call a library function the first thing it does after a system restart or as a last call before a shut down. Then it is not possible to have an initialisation or a shut-down phase for the library. The library should be re-entrant, therefore can we not store any data in the library. Functions should start with the library short name to avoid name collision with other libraries and because it should be possible to easily locate library calls in the source code. As mentioned a library must be re-entrant and to guarantee this the library cannot call other components than libraries. Input should be checked to avoid that the library functions exhibit erroneous behaviour. Macros are easy to misuse because the types of the inputs and return values are not specified and they should therefore be avoided. A library should not be called via the RTE because the call of a library function should not affect the SW-C interface description. Also the function call should be efficient and going through the RTE will add overhead which will reduce the efficiency.

A standard library in AUTOSAR is the SW-C End-to-End (E2E) Communication Protection Library [31]. E2E protects from errors in the data communication between different SW-Cs. The errors could be caused by random faults in hardware or systematic faults in the implementation of the VFB. The library considers communication faults such as repetition, deletion, insertion, incorrect sequence, corruption, delay, addressing faults, inconsistency and masquerading. E2E protects the data to be sent by adding a control segment. The receiver can verify the received data using the control segment and if a deviation is detected it is up to the receiver to handle the situation.

To see how a standardised AUTOSAR library is implemented, a commercial AUTOSAR implementation of the E2E library, was examined. During the examination of the library it was noted that it followed all of the library requirements. We also observed how it was constructed, and that different versions of the functions exist depending on the size of the input (uint8, uint16 or uint32). The functions in the E2E library were tested with different input values to determine the robustness of the implementation. The library was found to withstand the majority of faulty inputs except for one function that did not check if its input pointer was null. That function crashed due to a segmentation fault when a null pointer was injected.

5.1.2 Library Implementation

We have implemented an AUTOSAR compliant library for plausibility checks including range checks. The purpose of the library is to provide a standardised interface for plausibility checks, to make it easier to add safety mechanisms to a program and minimising the risk of errors in the code by having one implementation instead of one in each program.

The library contains the following functions:

- **FS_CheckNull:** Checks if a value is null.
- **FS_CheckEq:** Checks if two values are equal.
- **FS_CheckNotEq:** Checks if two values are not equal.

- **FS_CheckRange:** Checks if a value is within a given range.
- **FS_CheckDifference:** Checks if the difference between two values is not larger than a given value.

These functions provide the most basic plausibility checks that nearly all applications need. We have chosen these functions because they were needed in order to test the applications in Section 5.2. All functions except CheckNull exist in three different versions for unsigned integers of 8, 16 and 32 bits (uint8, uint16 and uint32) to support functions with different parameter sizes.

The library follows all rules for AUTOSAR libraries described in Section 3.1.4. The functions return Std_ReturnType that is FS_E_OK or one of several error messages enumerated below. The most appropriate error message is chosen for each error. The caller must then decide how to handle about the error. The library does not store any state information and all functions start with the library short name FS.

- FS_E_TOO_LARGE
- FS_E_TOO_SMALL
- FS_E_NULL
- FS_E_INVALID
- FS_E_INVALID_ARGUMENTS

5.2 Plausibility Check Evaluation

To investigate how plausibility checks can increase the robustness of applications we have added plausibility checks to three different applications: Bitcount, Brake-By-Wire and Integer Converter.

5.2.1 Bitcount

Bitcount is a part of the embedded benchmark suite MiBench developed by scientists at the University of Michigan and it has been used as a test application in many research projects [32]. There are applications for six different categories: automotive and industrial control, network, security, consumer devices, office automation, and telecommunication. Bitcount is one of the application in the automotive package and it tests the processor's ability to manipulate bits. It does this by executing several algorithms that count the number of bits in an integer array.

In this section are plausibility checks evaluated with help of the bitcount application. Bitcount was compiled using GCC and run in Cygwin environment on a PC with Windows 7. As error model fuzzing, mentioned in Section 3.5.3, were chosen to see if the application behaved strangely when receiving random input. To generate the random data, that were injected into the application, a fuzz program provided from the University of Wisconsin [25] was used.

Three different versions of bitcount were used: one version with the original source code, another version with one plausibility check added and a third version with two

plausibility checks added. When an error was detected by the plausibility checks, a default value substituted the erroneous variable hence the execution of the application could proceed. Figure 5.1 shows the number of errors categorised as hangs or crashes that were exhibited for the different bitcount versions during 100 fault injections. As can be observed all hangs are handled with one plausibility check implemented, but the number of crashes increases when one plausibility check was introduced. When an additional plausibility check was added these crashes could be handled.

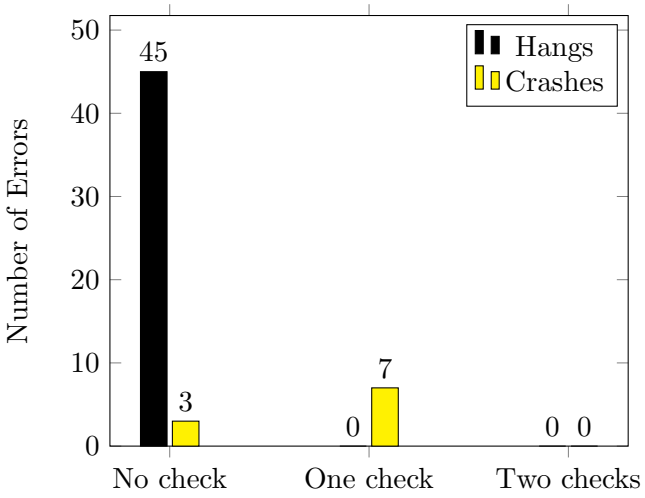


Figure 5.1: Number of errors with different versions of bitcount during 100 fault injections.

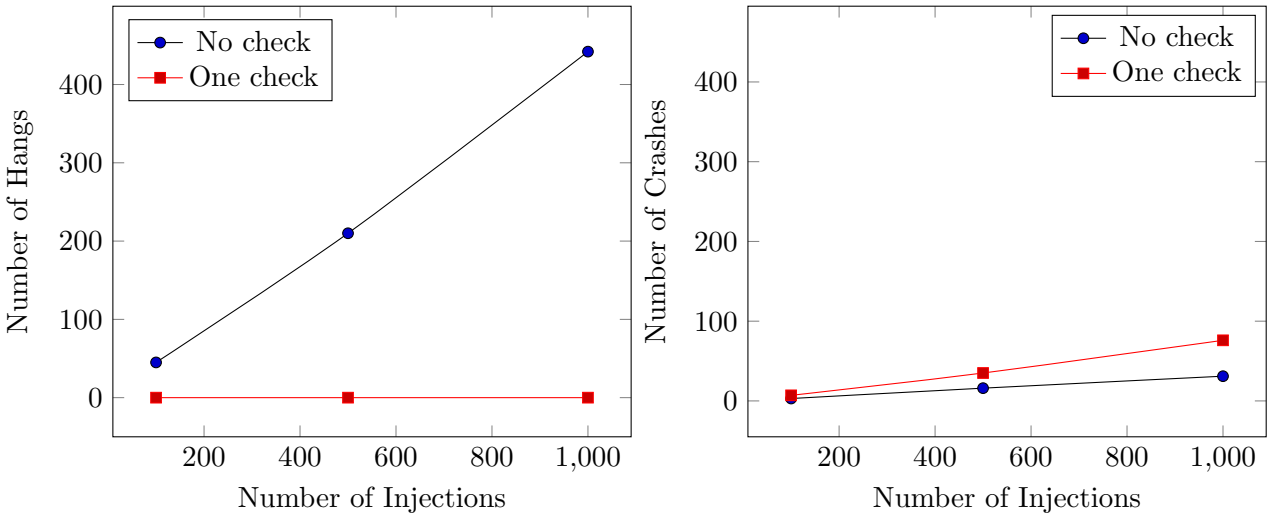
Each of the three versions of the application were tested with 100, 500 and 1000 fault injections where the results are shown in Figure 5.2. In Figure 5.2a it can be seen that the number of hangs increase with the number of tests using the original bitcount version. Similarly, in Figure 5.2b the number of crashes increases with the number of tests for both the original bitcount and also for the version with one plausibility check.

We noted that bitcount hanged when a negative value was injected. The implementation of bitcount was altered to interpret the input variable as an unsigned long instead of a signed long. With the input interpreted as an unsigned long the variable cannot turn negative and therefore this implementation executes correctly for all input values. We can of course add the second plausibility check here as well and remove all remaining errors.

5.2.2 Integer Converter

The integer converter application is used to convert integers to a byte stream. It handles integers of three different sizes: 32, 16 and 8 bits long. The input arguments to the application are a source pointer, a destination pointer and the number of integers to convert. The integers must be stored after each other where the source pointer point.

We tested this application with help of the programs Cygwin, GCC and Fuzz. The fuzz error model described in Section 3.5.3 was used because we wanted to test the



(a) Hangs **(b)** Crashes.

Figure 5.2: Number of errors during 100, 500 and 1000 fault injections.

function for all possible input data since a faulty source pointer can point anywhere in the memory space. The program was compiled with GCC using Cygwin and run on a PC with 64-bit Windows 7. The Fuzz program [25] was used to generate the input.

When the pointers were fuzzed they did always point outside the allowed memory space for the application causing a segmentation fault. This was because of the very small probability of pointing at a valid address in a 64-bit memory space, see Figure 5.3. Adding a range check caught all the faulty pointers. Fuzzing the size parameter could also cause failures if the value were to large depending on where in the memory the pointers pointed, for example was the return address overwritten on one of our tests which caused the program to fail.

5.2.3 Brake-By-Wire

A Brake-By-Wire (BBW) system is an electronic brake control system [33]. A schematic view of a BBW system is shown in Figure 5.4. The braking system is activated when the driver press down the brake pedal or by other systems requesting braking. There is a main ECU that reads the position of the brake pedal and calculates the brake torque from various sensor inputs such as vehicle speed and steering angle. This information is then distributed to an ECU at each wheel which is connected to an actuator. The ECUs located at the individual wheels will then compute the required brake force that the actuator will apply to the wheel.

A fully functional BBW system should brake the car with a force that is proportional to the driver's input on the brake pedal. Any unintended events of the system should not occur. Such events includes braking without the driver's request and release of the brakes when the driver want to brake [34].

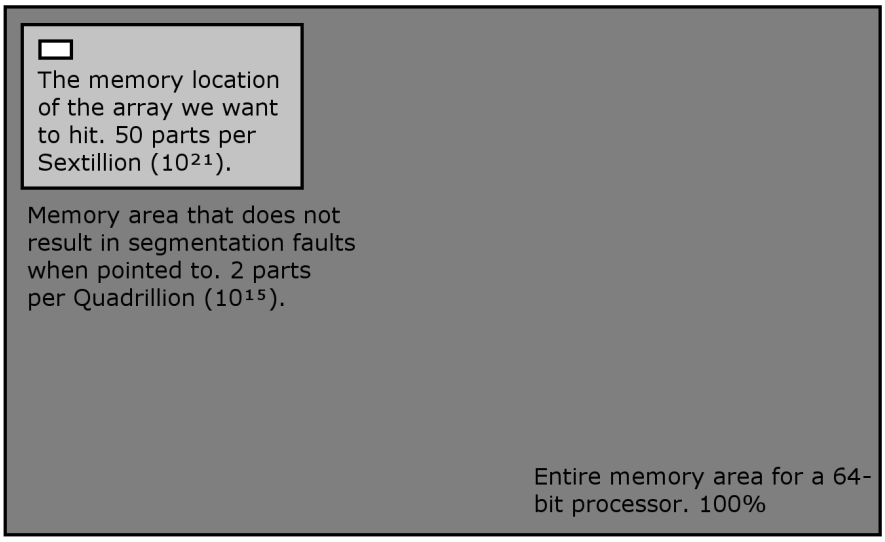


Figure 5.3: Illustration of how small part of the memory area that must be hit and the size of these areas compared to the entire 64-bit memory space.

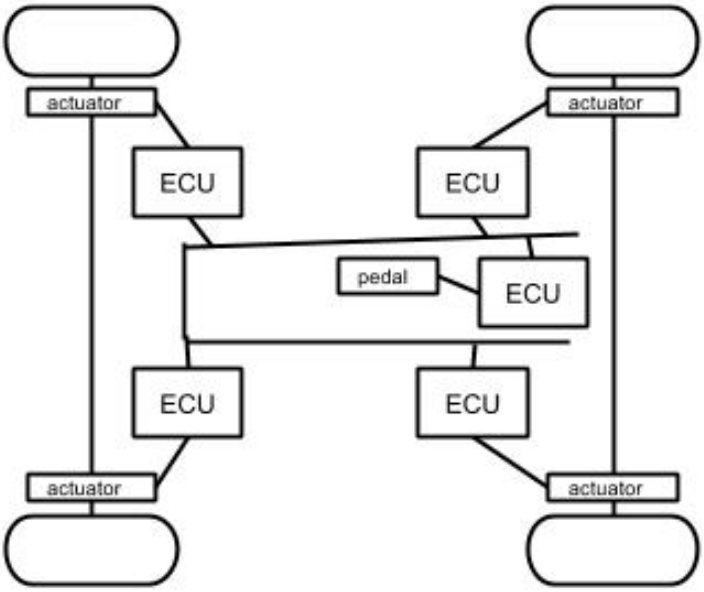


Figure 5.4: A schematic view of the Brake-By-Wire system.

In this section a Brake-By-Wire (BBW) model created in MathWorks Simulink was used. The model has been used in previous research projects by Volvo, for example in the BeSafe project - a 3-year Swedish research project. A simplified overview of the Simulink

model is shown in Figure 5.5. There is the global brake controller which reads requested brake torque from the brake pedal. The global brake controller then distributes the requested torque to the four different wheels. Each wheel has an Anti-lock Blocking System (ABS) function and as input it takes the requested torque from the global brake controller, the vehicle velocity and the rotation of its associated wheel. The local brake torque for that particular wheel is then calculated as output. The vehicle's behaviour is simulated by the model, which calculates the speed and wheel rotations based on inputs from the acceleration pedal and the ABS functions.

From the Simulink model standalone C source code was generated which was possible to compile and run independently from Simulink. The application was compiled with Visual Studio 2012 and run on a PC with Windows 7. There was a simulation of a predefined use case which was used as a reference. The reference run lasted for 30 seconds with acceleration from 0 km/h to approximately 70 km/h in around 10 seconds, steady speed from 10 to 20 seconds and braking from 20 seconds until the end of the simulation.

To determine how the model withstands erroneous input, fault injection was used on the different input parameters of the BBW. All parameters were of type double and the different test cases were derived according to data-type based error model described in Section 3.5.2. This resulted in the test cases described in Table 5.1, where DBL_MAX are $1.7977 \cdot 10^{308}$ and DBL_MIN are $2.2251 \cdot 10^{-308}$, the largest and smallest possible double values in the run-time environment. The inputs have limits which also should be tested; the global torque in the global brake controller for example expects a value between 0 - 3000. Therefore two additional test cases of max value and max value + 1.0 were added, where max value is 3000, to test the upper limit. The lower limit is already covered by the data type test cases.

Value
0.0
1.0
-1.0
±DBL_MAX
±DBL_MIN
max value
max value+1.0

Table 5.1: Test cases for BBW.

Faults were injected into the global torque input in the global brake controller, hence overriding the actual requested torque from the brake pedal. Injections were performed during acceleration between 5 - 10 seconds and during braking between 20 - 25 seconds. The resulting graphs when ±DBL_MAX and ±DBL_MIN were injected are presented in

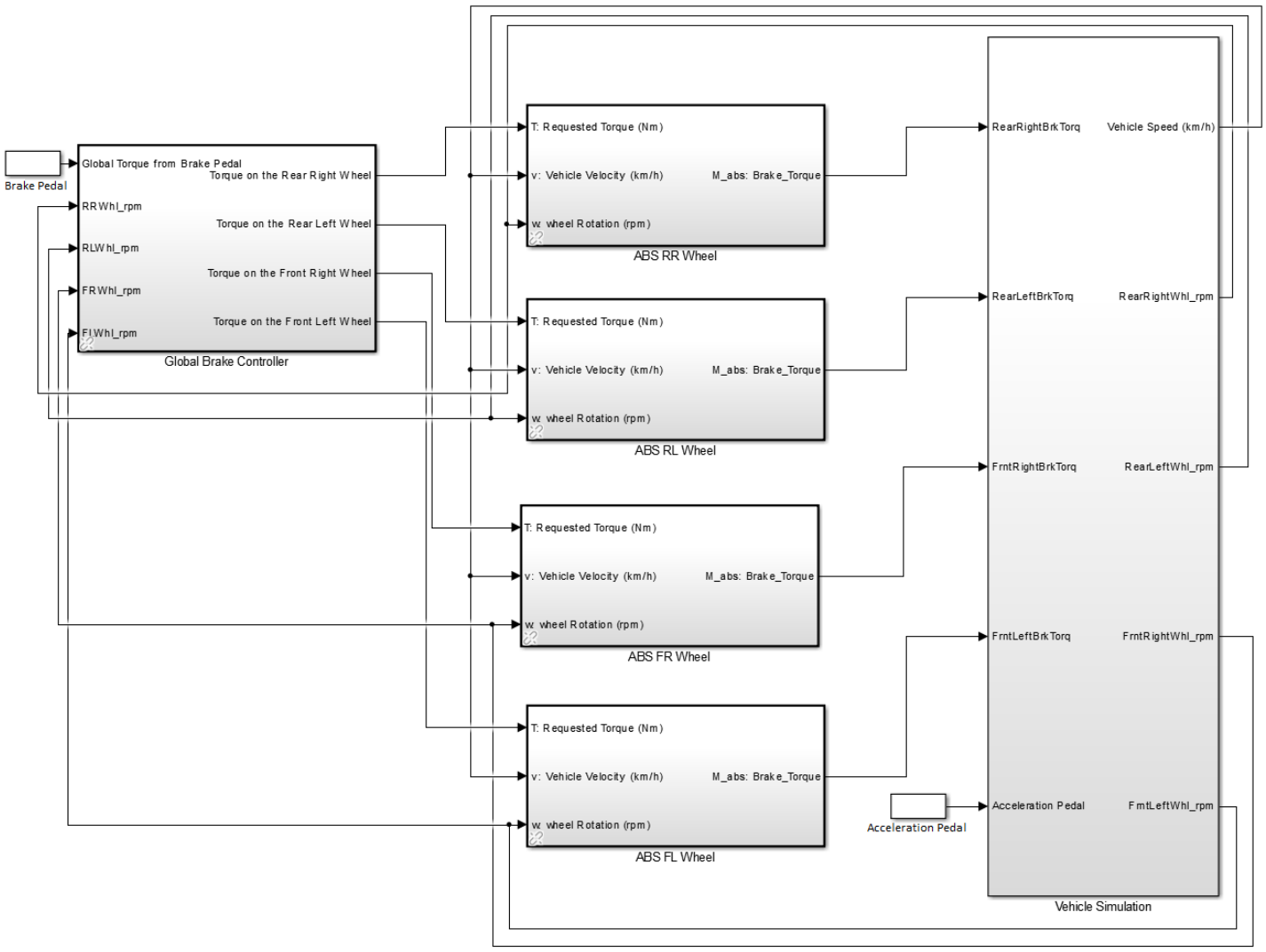


Figure 5.5: Simplified overview of the Simulink Model.

Figure 5.6 with the reference run plotted with a dashed line. The x-axis shows the time span of the simulation (0-30s) and the y-axis shows the speed in kilometres per hour. Figure 5.6 shows that -DBL_MAX and -DBL_MIN caused the system to behave as the reference run during the acceleration but during braking the injections prevented the system from braking. When DBL_MAX and DBL_MIN were injected Figure 5.6 shows that the system brakes both during acceleration and braking.

Figure 5.7 shows that injections with 0 and -1 exhibits the same behaviour as when -DBL_MAX and -DBL_MIN were injected. It is also observed that injecting 1 achieves the same result as DBL_MIN.

In Figure 5.8 the max value of 3000 and max value + 1 are injected in the global

torque. As can be seen the system brakes during both injections.

Figures 5.9, 5.10 and 5.11 show results from injections in requested torque of the right rear wheel. The same values as when injecting the global torque are used except that max value is 1500 instead of 3000. These figures show that the injection runs deviates slightly from the reference run. In these experiments, faults are injected in a single wheel only and the deviations from the reference run is less than when faults were injected in the global brake controller.

A plausibility check was added to check that the brake torque was within the allowed range. In that case the injections of -1.0, ±DBL_MAX, -DBL_MIN and max value + 1 are detected, because these values are not within the allowed range. During the fault injection experiments with the BBW model we found two unexpected scenarios. For the BBW model it was shown in Figure 5.8 what the speed graph should look like if the maximum value of the brake torque is applied. However, that Figure 5.6 indicates that the model brakes more when DBL_MAX is applied due to the model does not have any maximum boundary for the brake torque. Also in Figures 5.6 and 5.7, the graphs of DBL_MIN and 1 show that the acceleration stops during the period when faults are injected which should not be the case when a minimal brake torque is applied. The model was found to only allow either acceleration or braking where braking is prioritised. Therefore, if the acceleration pedal is triggered at the same time as the braking pedal the model only brakes. This explains the behaviour in Figures 5.6 and 5.7 in the cases with DBL_MIN and 1. A minimal brake torque is injected and therefore acceleration is blocked and the speed is maintained.

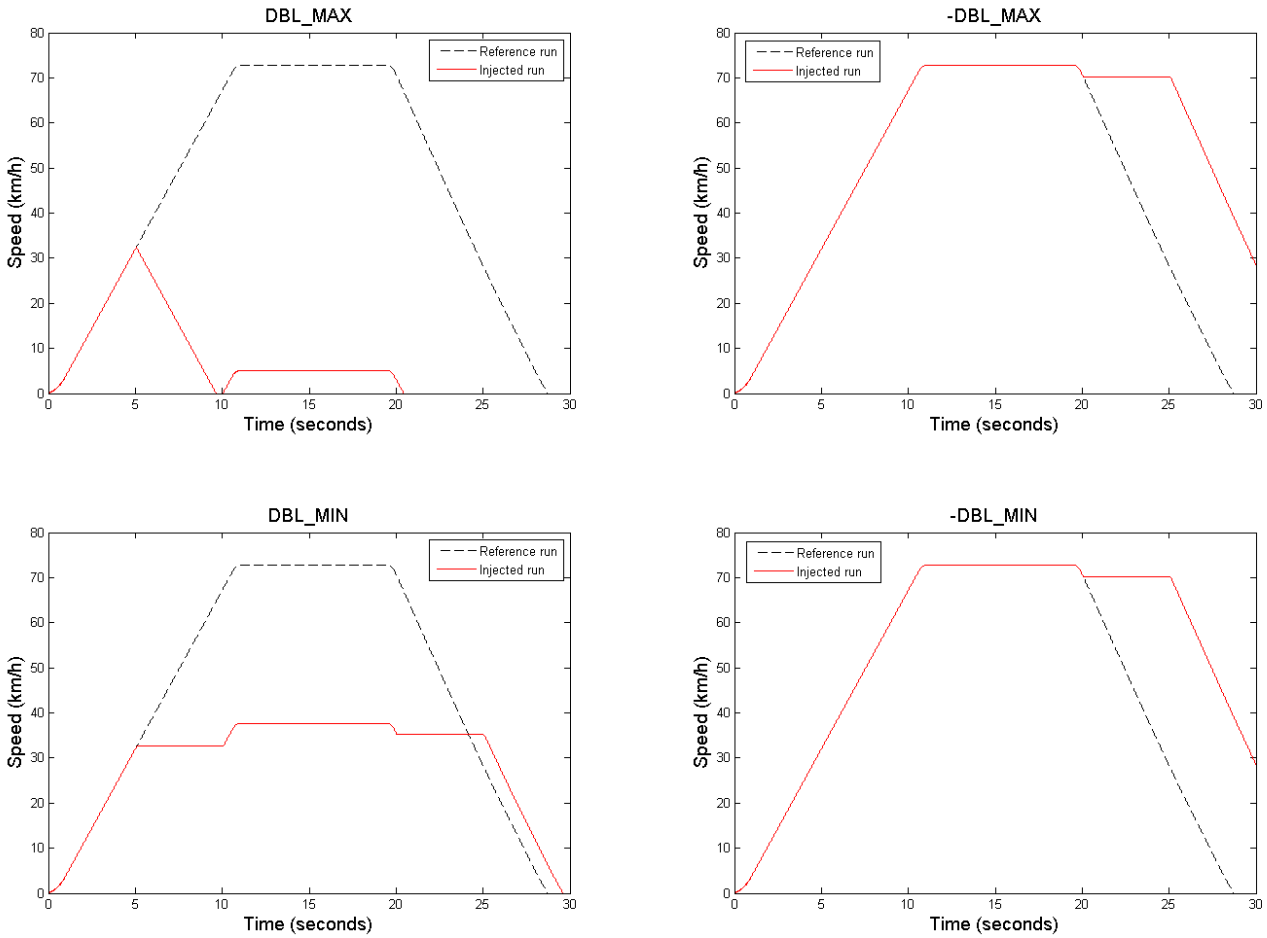


Figure 5.6: Speed graphs for the simulation with ±DBL_MAX and ±DBL_MIN injected in global torque between time 5-10 and 20-25. The reference run is dashed.

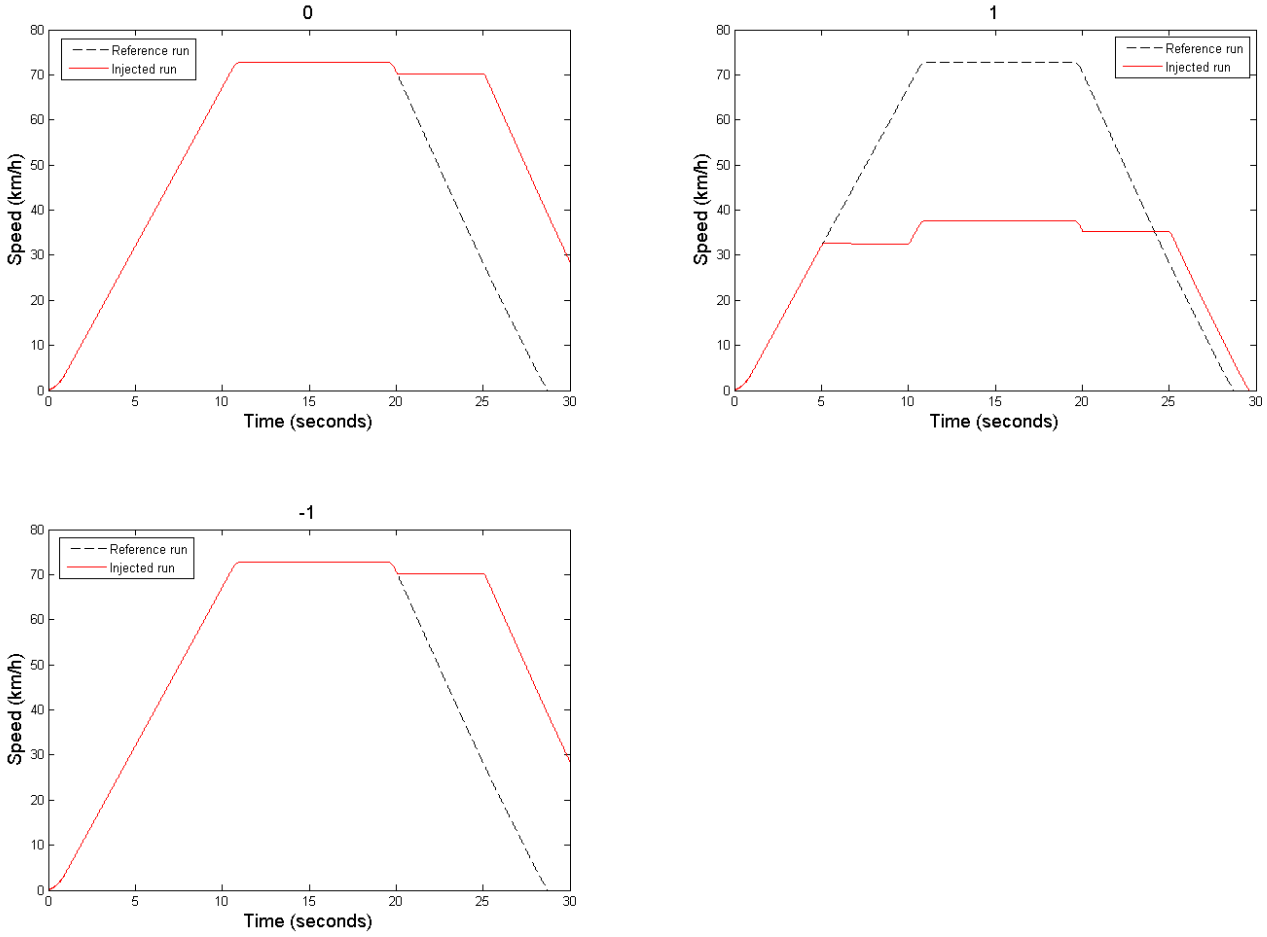


Figure 5.7: Speed graphs for the simulation with 0, 1 and -1 injected in global torque between time 5-10 and 20-25. The reference run is dashed.

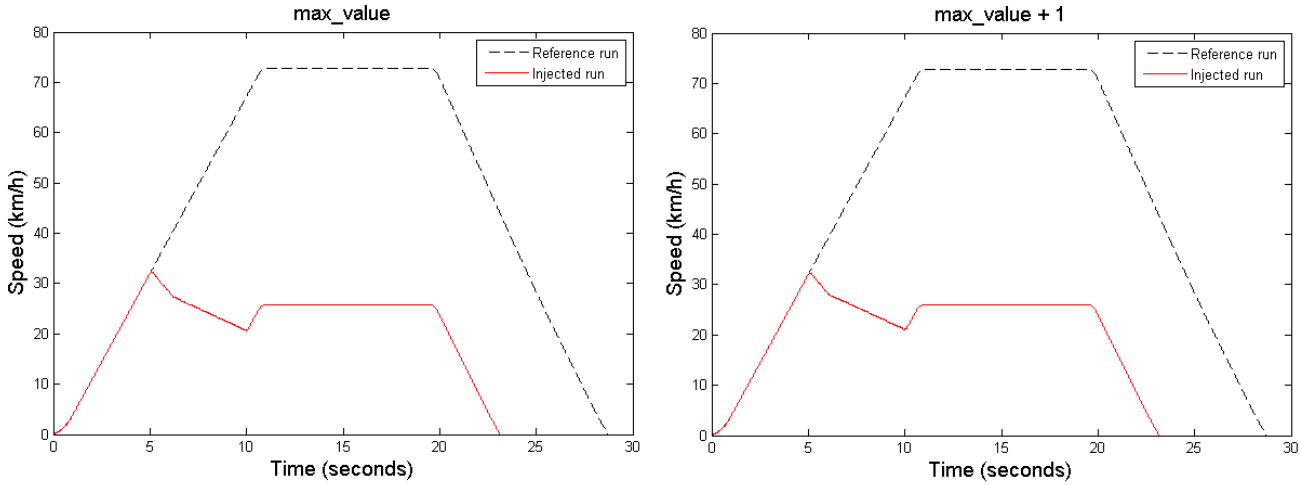


Figure 5.8: Speed graphs for the simulation with max_value (3000) and max_value+1 injected in global torque between time 5-10 and 20-25. The reference run is dashed.

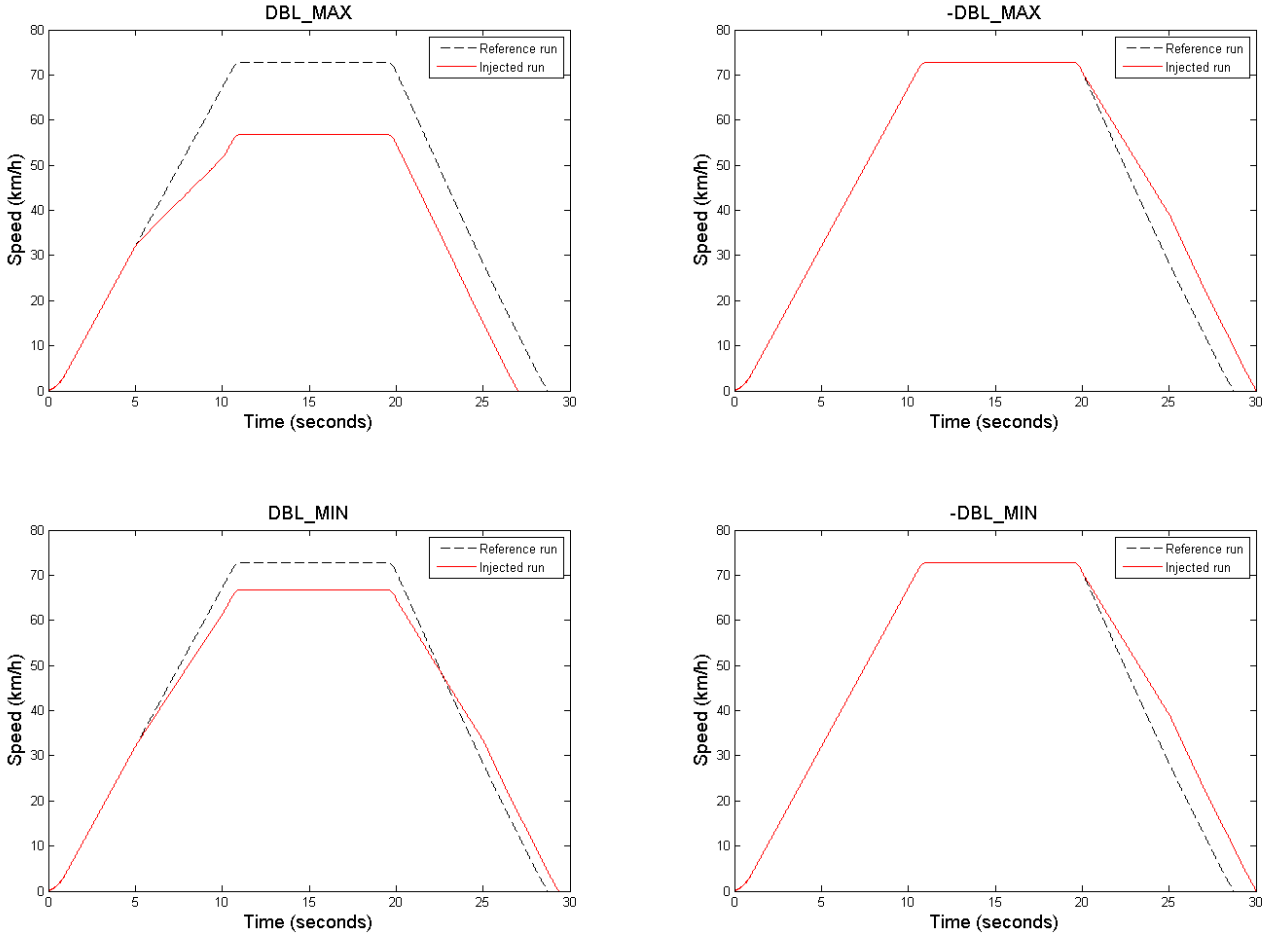


Figure 5.9: Speed graphs for the simulation with ±DBL_MAX and ±DBL_MIN injected in requested torque of the right rear wheel between time 5-10 and 20-25. The reference run is dashed.

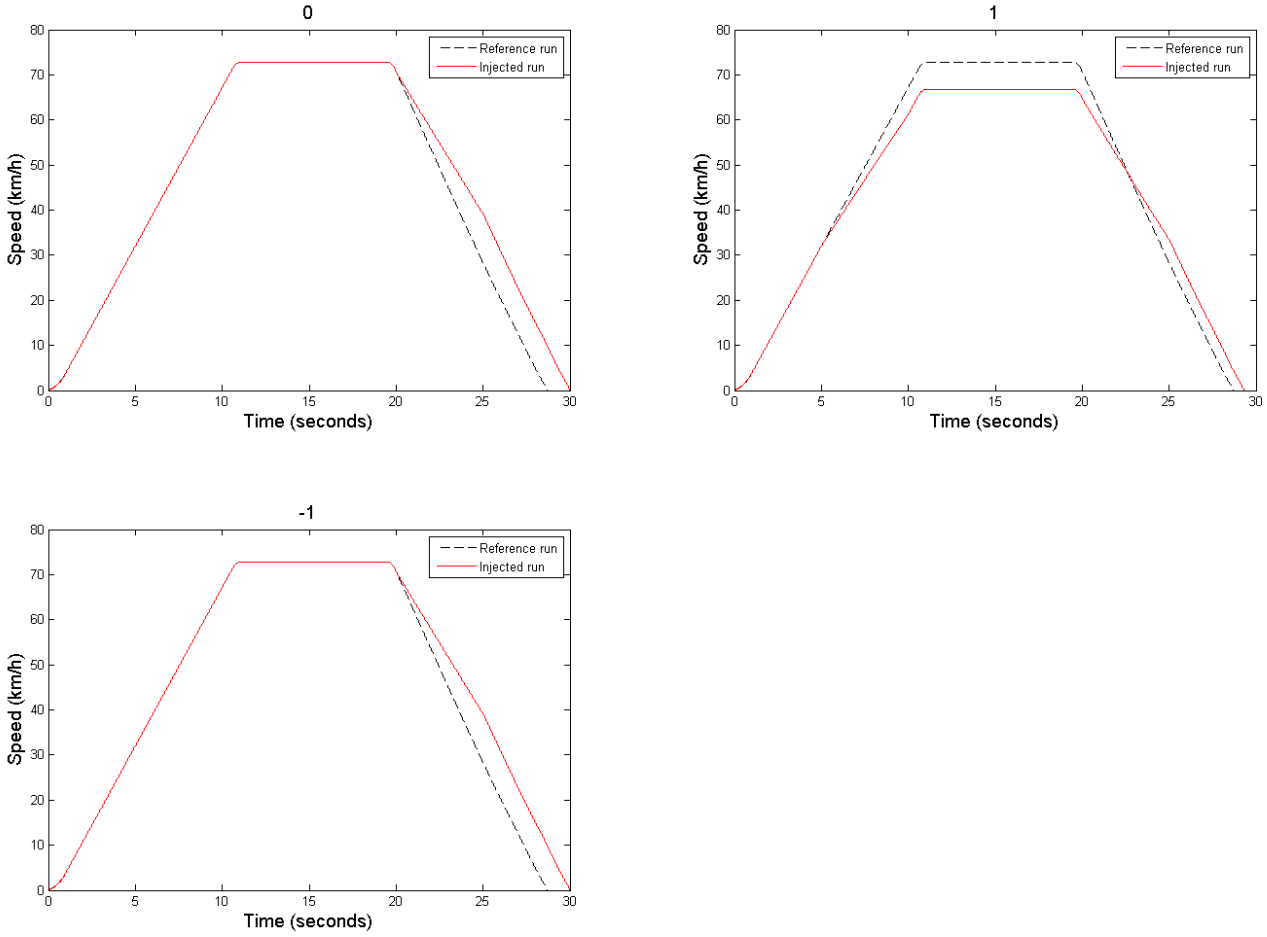


Figure 5.10: Speed graphs for the simulation with 0, 1 and -1 injected in requested torque of the right rear wheel between time 5-10 and 20-25. The reference run is dashed.

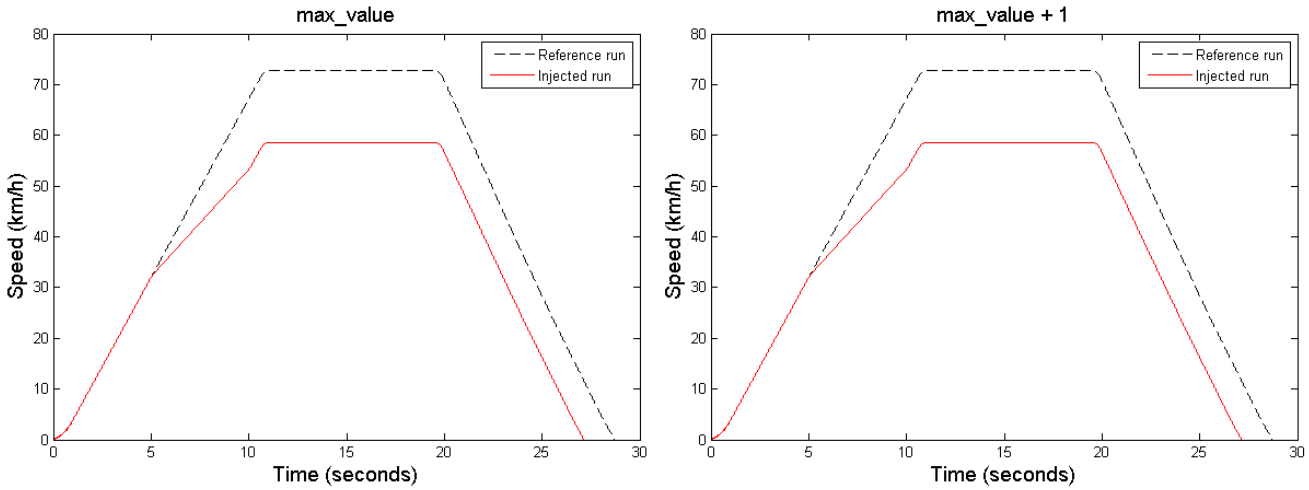


Figure 5.11: Speed graphs for the simulation with max_value (1500) and max_value+1 injected in requested torque of the right rear wheel between time 5-10 and 20-25. The reference run is dashed.

5.3 Library Time Evaluation

To evaluate the library it has first been tested on the three applications from Section 5.2. We then moved on with two AUTOSAR applications. First a LED blinking test application and then a Brake-By-Wire system. We have also measured the resource requirements of the library.

5.3.1 LED Blinking

LED blinking is a test application in ArcCore's distribution of AUTOSAR. The program blinks with a green LED on an ECU board. The purpose for using this relatively simple system is to ensure that the plausibility check library works with an AUTOSAR system. To test the application, a test setup with an ECU board with a Freescale MPC5567 microprocessor connected via a Lauterbach debugger to a PC was used. The source code was written in Arctic Studio and compiled using a powerpc-eabi cross GCC compiler for PowerPC target architecture.

A plausibility check from the library was added to the source code that controls that an allowed value is sent to the BSW for turning on/off the LED. There exist two different failure modes: the LED constantly off and the LED constantly on. The data-type based error model was used in this test, see Section 3.5.2. This error model was used because we think that the boundary values are the most interesting ones in this test. The input parameter is an unsigned integer and the suitable test cases are therefore 1, 0, -1, UINT8_MAX, UINT8_MAX+1 and UINT8_MAX-1. The results of the experiments are shown in Table 5.2.

Value	**Failure mode**
1	LED Off
0	LED On
-1	LED Off
UINT8_MAX	LED Off
UINT8_MAX+1	LED On
UINT8_MAX-1	LED Off

Table 5.2: Results from the tests with LED blinking.

From Table 5.2 and other tests with random numbers we observe that the LED is on if it receives 0 and off otherwise. When the application operates as intended it alternates between sending 1 and 0 to the BSW. The addition of a plausibility check that only allows 1 and 0 remove all problems with other values, but not if a 1 or 0 are mixed up. The time for using the library is measured to $1.71\mu s$ per check and placing the check directly in the source code takes $0.64\mu s$. That is $1.07\mu s$ longer with the library.

5.3.2 AUTOSAR Brake-By-Wire

This application is an AUTOSAR application unlike the one in Section 5.2.3. A setup with five ECU boards with Freescale MPC5567 microprocessors has been used. Focus has been on the global brake controller that was connected with CANoe and a Lauterbach debugger to a PC. The code was written in Arctic Studio and compiled using powerpc-eabi cross GCC compiler for PowerPC target architecture. Vector CANoe was used to send CAN messages to and from the global brake controller ECU. The DEDICATE framework case, which is a research framework developed in the DEDICATE project, with the ECUs and indicator lights is displayed in Figure 5.12. The PC environment, in CANoe, for sending messages to the system is pictured in Figure 5.13.

Figure 5.12: DEDICATE framework case.

Time measurements on the global brake controller have been performed. The controller has two values that is checked by plausibility checks. Three different cases were tested: no checks, checks directly in source code and using the library. The results are shown in Table 5.3. It is noted that adding the checks directly in the source code increases the execution time with $0.5\mu s$ and using the library instead increases the exe-

Figure 5.13: The environment in CANoe.

cution time with 4.0μs. Changing the implementation of the checks from directly in the source code to using the library increases the execution time with 3.0μs.

Configuration	**Processor cycles**	**Execution time**
No checks	466	11.65μs
Checks directly in source code	486	12.15μs
Checks in library	626	15.65μs

Table 5.3: Three different time measurements on the global brake controller.

To see how the time requirements change when more checks are added, more measurements were performed. The library has been used for two different plausibility checks in the source code of the global brake controller. The Lauterbach debugger was used to estimate the time spent in the plausibility checks when executing the AUTOSAR BBW application. The behaviour when many plausibility checks are implemented in an application was emulated by executing the checks 2, 10, 100, 500 and 1000 times

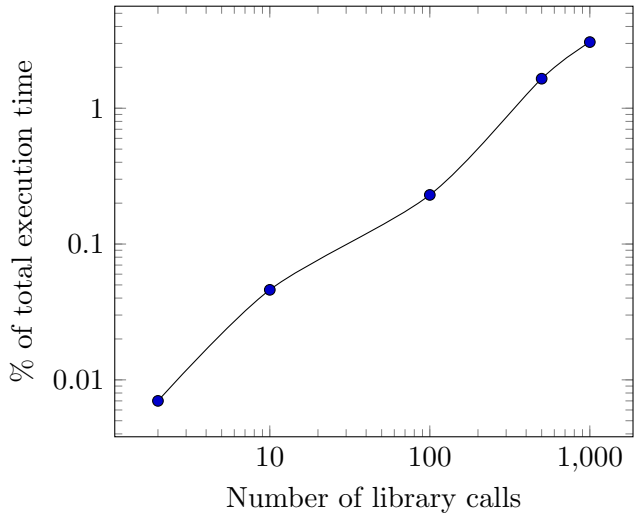


Figure 5.14: Scalability of the library

in each iteration of the global brake controller. Figure 5.14 shows the results from the time measurements. The x-axis denotes the number of calls to a plausibility check in the library and the y-axis denotes a percentage of the total execution time that is spent on the plausibility checks. It can be seen in the figure that the usage of plausibility checks scale linearly with the number of calls. With two checks the library is 4.5% of the application's execution time and at 500 checks it has increased to 55.6% of the time.

6

Discussion

We discuss the two different kinds of results that have been achieved. Firstly, that plausibility checks are important and effective means to lower the number of errors in various computer applications. Secondly, that an AUTOSAR compliant library for plausibility checks has been implemented and evaluated.

6.1 Plausibility Checks

Our experiments with the bitcount application show how plausibility checks can be used to detect errors and we also found that they improved the robustness of the application. The error causing the hangs in the bitcount application was a variable that got stuck at -1 causing an infinite loop. After inserting a plausibility check into the code that detected if a variable became -1, it was possible to terminate the infinite loop. In Figure 5.1 we observe that the number of crashes increases when the hangs are eliminated. This can be explained by that some of the runs that hanged exhibited a crash when their execution continued. Hence, recover from one error could cause another error to emerge. Another plausibility check detected if an index variable was outside the range of an array, thus detecting out of bound error which caused a crash. These two plausibility checks together detected all errors. Figure 5.2 shows that the number of faults increases linearly with the number of injections.

The importance of plausibility checks depends on how good the applications are designed. For example when we changed a variable to unsigned in bitcount (see Section 5.2.1) the need for a plausibility check was eliminated. On the other hand, without the plausibility checks we would not have found the error.

All the three applications tested in Section 5.2, Bitcount, Integer Converter and Brake-By-Wire, were improved by the use of plausibility checks and we therefore recommend the use of plausibility checks to ensure that values are plausible. Plausibility

checks also have some limitations. If the values are wrong but still plausible they cannot be detected by a plausibility check, for example only injections outside of the allowed range in the BBW application could be detected by a plausibility check. How effective plausibility checks are also depends on how stringent the plausible values are set. Usage of a state machine in an application makes it easier to find more stringent plausible values, because then it is possible to look at the previous state and detect if the state has changed more dramatically than what is realistic.

6.2 Library

In our plausibility check library only error detection is handled and neither isolation nor recovery. This is similar to the E2E library which signals a fault in the communication by an error code and then the calling function has to choose an appropriate action to handle the fault. An error handling method which is close at hands are substitute values. Another possibility is to reconfigure the system to no longer use the component that produced the erroneous result. This may be a good solution for a non-safety critical ABS system, but if for example the brake control ECU in a BBW system fails the safety critical brake system cannot be turned off. Another solution such as a backup system must be used instead.

The plausibility check library has been successfully used in the LED blinking and the AUTOSAR BBW applications. The library has also been used on all applications in Section 5.2. There are many advantages with using a library instead of implementing plausibility checks directly in the source code for each application. To have only one implementation of the checks in a program lowers the probability for bugs in the plausibility checks, and if there are bugs in the library they will probably be found relatively soon if it is commonly used. It becomes easier for the application programmers to add plausibility checks to their applications when using a library. It will also decrease development time because the checks are already implemented and the standardised interface hints of what kind of plausibility checks that can be used. The source code will also be clearer when a library is used, because of the descriptive function names of what conditions that are checked. Adding checks may increase the functional safety of applications that otherwise would not have been protected and this will lead to programs that are less vulnerable.

Using the library increases the size of the binary with 4.47 KB. The increase is not very large and it will probably not be a problem, for example it is only an increase of the program size with 0.3 % for the AUTOSAR BBW application.

From the time measurements of the Brake-By-Wire application in Table 5.3 we learned that the increase in time was $0.5\mu s$ from no plausibility checks at all to using two plausibility checks directly in the code. By using the library instead, the execution time increased with $4.0\mu s$ in that case. This increase in execution time is negligible, if the library is not called extremely often. Figure 5.14 shows that the time usage scale linearly with the number of library calls. The execution time therefore also depends on how many checks that are used.

Plausibility checks can be used to inspect inputs to functions to ensure that plausible values are used when the execution is started. That is useful to protect functions from errors in the inputs that were produced outside of the function. When a result from a function is completed it should be verified before it is passed on to another instance to prevent an error from propagating. Also, it is more important to target a function that a significant amount of other functions rely on to achieve a correct execution. Hiller [29] proposes a process for equipping a system with error detection mechanisms. First identify inputs, outputs and the pathway from the inputs to the outputs in the system. Then determine which of the signals that are crucial to a correct execution of the system e.g. by using FMECA (Failure Mode Effect and Criticality Analysis).

In Figure 6.1 the data flow graph for braking in the Brake-By-Wire application is shown. The ellipses represent the five ECUs and surround the SW-Cs they contain. Plausibility checks can be put at different places in the graph. At the moment we have put three checks in the ”central ECU”: the input to the brake torque calculation and the two inputs to the global brake controller (the sum of the four RPM inputs is checked). More checks can be added to increase the safety, for example the four outputs from the global brake controller can be checked. Using too few checks increase the risk of erroneous values to be undetected, but placing checks everywhere can increase the execution time too much.

In the specification of the RTE [35] it is specified that the RTE should be able to perform range checks of data to and from SW-Cs. However, since the RTE is separate from the SW-C it is not possible to use internal variables of the SW-C to specify the valid range. It is only possible to use predefined ranges specified at compile time. This is different from the plausibility check library that also is able to check dynamic ranges decided during run-time.

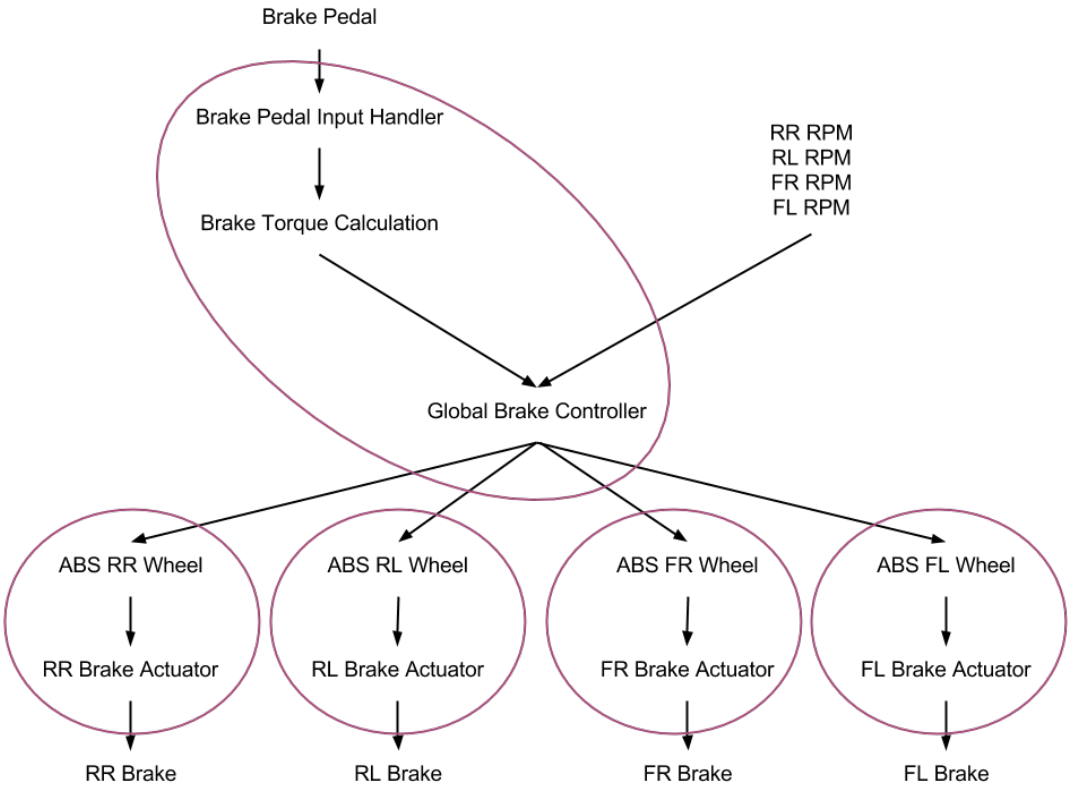


Figure 6.1: Data flow graph for a braking command. The ellipses represent the ECUs and surround the SW-Cs they contain. The ABS units receive the vehicle speed and the local wheel RPM as inputs but it has been omitted in the figure to increase clarity.

7

Conclusion and Future Work

In this thesis we have evaluated how plausibility checks can increase the robustness of AUTOSAR software components. We have also investigated the requirements for creating an AUTOSAR compliant library and a proof-of-concept implementation of a library with common plausibility checks was developed. The library was tested on two AUTOSAR applications and it was demonstrated that the library performed well.

There are two main types of plausibility checks: range checks and comparison of redundant data sources. We have focused on range checks in this thesis. Plausibility checks has been tested on several existing applications with good results. We used a randomised error model, fuzzing, to inject faults in the bitcount application from the MiBench benchmark suite. For thousand fault injections it was possible to detect all input errors with two plausibility checks.

We used a data-type based error model to inject faults in a non-AUTOSAR Brake-By-Wire system. As expected it was only possible to detect injections that were outside of the valid range 0-3000. However, to detect more errors the valid range can be reduced by comparing the current state with the previous state and check if the change is unrealistic. Then it is possible to have stringent conditions to achieve more effective detections with plausibility checks.

Two plausibility checks were inserted in the AUTOSAR Brake-By-Wire application. When the implementation of the plausibility checks was directly in the source code an overhead of $0.5\mu s$ was added. Using the library implementation instead added an overhead of $4.0\mu s$. This increase in overhead is small and most applications will not be affected by it. Applications with a very short execution time or applications that use the plausibility checks very frequently may however exceed their timing requirements. It is therefore important to consider how many plausibility checks that are used and where they are placed.

Using a library increases the execution time slightly more than plausibility checks directly in the source code. However, using the library has several design advantages. For

example, it makes it easier to add checks and it yields cleaner code. It is also important to remember that plausibility checks are not enough to protect an application on its own since for example control flow errors can be missed.

Use of plausibility checks in AUTOSAR applications needs further investigation. An investigation about whether the isolation and recovery parts of FDIR can be standardised/automated with a library is an important open issue. One concept mentioned in [13] is Application-Level Error Manager (ALEM) and it can be further explored.

Bibliography

[1] I. G. Insight, Resistance is futile - Electronics are on the rise: Electronic control units and communication protocols, IHS Global Insight, Inc 26 (2009) 24.

[2] B. Fleming, An overview of advances in automotive electronics, Vehicular Technology Magazine, IEEE 9 (1) (2014) 4–9.

[3] EmbeddedGurus: An Update on Toyota and Unintended Acceleration, `http://embeddedgurus.com/barr-code/2013/10/an-update-on-toyota-and-unintended-acceleration`, Accessed: 2014-04-08.

[4] Reuters: GM expands ignition switch recall to 2.6 million cars, `http://www.reuters.com/article/2014/03/28/us-gm-recall-expanded-idUSBREA2R1Y920140328`, Accessed: 2014-04-08.

[5] International standard, "ISO 26262 – Road vehicles – Functional safety", First Edition (Nov. 2011).

[6] AUTOSAR, www.autosar.org, accessed: 2014-01-30.

[7] IEEE standard glossary of software engineering terminology, IEEE Std 610.12-1990 (1990) 1–84.

[8] K. Peffers, T. Tuunanen, M. A. Rothenberger, S. Chatterjee, A design science research methodology for information systems research, Journal of management information systems 24 (3) (2007) 45–77.

[9] VeTeSS, `http://vetess.eu`, Accessed: 2014-04-08.

[10] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Transactions on Dependable and Secure Computing 1 (1) (2004) 11–33.

[11] N. R. Storey, Safety critical computer systems, Addison-Wesley Longman Publishing Co., Inc., 1996.

[12] AUTOSAR, Requirements on Libraries V4.0 (2013).

[13] AUTOSAR, Explanation of Error Handling on Application Level V4.0 (2013).

[14] D. Smith, K. Simpson, Functional safety, Routledge, 2012.

[15] M. Born, J. Favaro, O. Kath, Application of iso dis 26262 in practice, in: Proceedings of the 1st Workshop on Critical Automotive applications: Robustness & Safety, ACM, 2010, pp. 3–6.

[16] P. Koopman, K. DeVale, J. DeVale, Interface robustness testing: Experience and lessons learned from the ballista project, Dependability Benchmarking for Computer Systems 72 (2008) 201.

[17] C. Lu, J.-C. Fabre, M.-O. Killijian, Robustness of modular multi-layered software in the automotive domain: a wrapping-based approach, in: IEEE Conference on Emerging Technologies & Factory Automation, IEEE, 2009, pp. 1–8.

[18] J. M. Voas, G. McGraw, Software Fault Injection - Inoculating Programs Against Errors, Wiley Computer Publishing, 1998.

[19] R. Natella, D. Cotroneo, J. A. Duraes, H. S. Madeira, On fault representativeness of software fault injection, IEEE Transactions on Software Engineering 39 (1) (2013) 80–96.

[20] D. Cotroneo, A. Lanzaro, R. Natella, R. Barbosa, Experimental analysis of binary-level software fault injection in complex software, in: 9th European Dependable Computing Conference, IEEE, 2012, pp. 162–172.

[21] M. Islam, N. M. Karunakaran, J. Haraldsson, F. Bernin, J. Karlsson, Binary-level fault injection for autosar systems, in: 10th European Dependable Computing Conference, IEEE, 2014, pp. 162–172.

[22] L. Pintard, J.-C. Fabre, K. Kanoun, M. Leeman, M. Roy, Fault injection in the automotive standard iso 26262: An initial approach, in: Dependable Computing, Springer, 2013, pp. 126–133.

[23] A. Johansson, N. Suri, B. Murphy, On the selection of error model(s) for os robustness evaluation, in: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, IEEE, 2007, pp. 502–511.

[24] B. P. Miller, L. Fredriksen, B. So, An empirical study of the reliability of unix utilities, Communications of the ACM 33 (12) (1990) 32–44.

[25] Fuzz Testing of Application Reliability, `http://pages.cs.wisc.edu/~bart/fuzz`, Accessed: 2014-04-08.

[26] F. Bernin, M. Islam, Fault-Tolerant AUTOSAR SW Solutions, Internal Volvo document ER-653403 (Dec 2013).

[27] AUTOSAR, Technical Safety Concept Status Report V4.0 (2013).

[28] M. Korte, F. Holzmann, G. Kaiser, V. Scheuch, H. Roth, Design of a robust plausibility check for an adaptive vehicle observer in an electric vehicle, in: Advanced Microsystems for Automotive Applications, Springer, 2012, pp. 109–119.

[29] M. Hiller, Executable assertions for detecting data errors in embedded control systems, in: Proceedings International Conference on Dependable Systems and Networks, 2000. DSN 2000., IEEE, 2000, pp. 24–33.

[30] D. Skarin, J. Karlsson, Software implemented detection and recovery of soft errors in a brake-by-wire system, in: 7th European Dependable Computing Conference, IEEE, 2008, pp. 145–154.

[31] AUTOSAR, Specification of SW-C End-to-End Communication Protection Library V4.0 (2013).

[32] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown, Mibench: A free, commercially representative embedded benchmark suite, in: 4th International Workshop on Workload Characterization, IEEE, 2001, pp. 3–14.

[33] E. A. Bretz, By-wire cars turn the corner, IEEE Spectrum 38 (4) (2001) 68–73.

[34] VeTeSS, "VeTeSS: Verification and Testing to Support Functional Safety Standards", Version -0.01 (Not published yet).

[35] AUTOSAR, Specification of RTE V4.0 (2013).