Evaluation of validity of verification methods: Automating functional safety with QuickCheck

Master of Science Thesis

Oskar Ingemarsson
Sebastian Weddmark Olsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden, September 2013

Evaluation of validity of verification methods:
Automating functional safety with QuickCheck

OSKAR INGEMARSSON
SEBASTIAN WEDDMARK OLSSON

Examiner: MENG WANG
Supervisor: JOSEF SVENNINGSSON

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

Department of Computer Science and Engineering
Göteborg, Sweden, September 2013

Abstract

Quviq QuickCheck can be used when testing and developing software within the automotive industry. A demonstration of QuickCheck with functional safety in mind has been made. Ambiguities in AUTOSAR 4.0.3 were discovered. Some ISO 26262 requirements are achievable with the use of QuickCheck, but it is not possible to achieve functional safety using only QuickCheck. This is mainly because AUTOSAR is written in informal syntax and can not help verify the model. Coverages have been measured and evaluated. To reach a higher level of coverage, one need both positive and negative testing, as well as more than one configuration.

Keywords
AUTOSAR, Verification, QuickCheck, Functional safety, ISO 26262

Contents

Glossary

ASIL Automotive safety integrity level.

Checkpoint A point in the supervised entity. These points can be configured and used for different supervision functions.

ECU Electronic control unit.

Mocking Mocking allows testing of code units without being reliant on dependencies.

Negative testing Errors should be triggered when generating negative and invalid input data or invalid command sequences.

Positive testing Trying to stay in a valid good state when testing by generating valid input data and command sequences.

Supervised Entity A critical section in the supervised program.

Supervision functions Logical-, Deadline or Alive supervision.

Wdg Watchdog (the hardware).

WdgIf Watchdog Interface.

WdgM Watchdog Manager.

Chapter 1

Introduction

1.1 Background

1.1.1 The Development within automotive industry

In the recent decades there has been a dramatic growth of information and communication innovations within the automotive industry. Analog vehicles have been transformed into complex electromechanical systems. New features are implemented (for example due to user demands, traffic safety or environmental regulations) requiring more computational power and less energy consumption. The average car has already 80 ECU's (electronic computation units) and to deal with the extra functionality, each ECU will need to become more complex. [1] [2] [3] [3] [4] [5]

1.1.2 Extent of software in modern vehicles

The cost of developing a new car model is up to one billion €, where electronics have reached a mean share of one third of the value. The electronics cost is divided into three categories: sensor, hardware and software. The share of software has been doubled the last 10 years and is now equal to the cost for hardware at 40%, and the sensor cost is 20%. [6] [7]

More and more functions will be implemented; Intelligent traffic systems which make the automotive vehicles communicate with the roadside management systems, infotainment systems will bring, among other, weather information through the Internet and emergency call support, traffic sign recognition, night vision and automated parking. [3] [8] [2] [6]

The number of lines of code running in a vehicle is another example of how complex the automotive software is. The software running on a F-22 Raptor and the F-35 Strike Fighter, two of the attack planes in the US air force, has about 1.7 million lines of code and 5.7 million lines of code respectively. The passenger plane Boeing 787 Dreamliner runs on 6.5 million lines of code, where the average premium-class car has close to 100 million lines of code. [9]

1.1.3 Introduction of standards

Because of the high development costs, and the complexity of modern cars, car manufacturers, suppliers and other companies related to the automotive industry joined efforts in 2003 and created AUTOSAR, short for Automotive Open System Architecture. The main purpose is to make it possible for car manufacturers to buy independent components from different software suppliers; the AUTOSAR motto is "Cooperate on standards, compete on implementation". [10]

Functional safety was introduced to the automotive industry with ISO 26262 in late 2011. ISO 26262 named "Road vehicles – Functional safety" is a general standard on how the implementation of functional safety in vehicle development should be carried out from beginning to end. [4]

This standard is built on top of another industrial standard, IEC 61508, named "Functional safety of electrical/electronic/programmable electronic safety-related systems", which purpose is to ensure functional safety in computer based systems' overall life cycles. [11] [4]

It is useful to distinguish between systems with different levels of dependability, and determine where the hazards exists. When this risk analysis is completed, and appropriate reliability and availability requirements are assigned to the system, the system can be identified by a certain automotive safety integrity level (ASIL). If this number is high, the system will experience a more rigorous design and testing than could be justified for a lesser demanding system. These levels are more defined in the standards IEC 61508 and ISO 26262. [12]

The concept of safety integrity levels exists because the implications of failure vary between applications. The safety integrity levels reflect the importance of correct behavior, and exists to ensure functional safety of the applications. [12, p.3,70]

1.1.4 Testing

Functional safety demands testing, and testing accounts for around half of all software development costs . Reducing the cost is motivated and can be done by automating the test generation process. [4] [13] [14] [15]

For simple devices it is possible to exhaustively test the functional safety of the system. For example consider a system consisting of a small number of switches, where each switch has only two states; open and closed. Then the number of possible failures can be determined by the combination of all possible failure states of each individual switch. The complexity issue is that in systems such as microprocessors or ECU's, the number of possible failure states is so large that it is considered infinite. This makes it impossible to exhaustively test the system, and therefore, make the detection of failures unreliable. This is called the combinatorial explosion problem. [12]

1.2 Purpose

This thesis purpose is to automate the testing process of automotive software in an effective and good way, and to make it possible to raise the Automotive Safety Integrity Level (ASIL), where applicable.

It is desired to do an evaluation of tools that can be used in order to perform at least semi formal verification of automotive software modules. The main purpose is however to evaluate if Quviq QuickCheck can be used to fulfill this.

1.3 Objective

The first problem is to evaluate what “semi formal verification” according to the ISO-standard means. In formal methods of mathematics, formal verification means to prove the correctness of algorithms. The ISO-standard mentions both “formal verification” and “semi formal verification” for software development, but it does not describe how to realize any of these. This evaluation must be performed to obtain knowledge of how to properly implement functional safety and reach an ASIL classification, using automated testing.

A model for an AUTOSAR module needs to be implemented. For this to be a good model, some questions must first be answered. How can one achieve good test cases for the model? How can one tweak the test generation to find test cases that are interesting in a safety critical point of view? Is the implemented model together with the generated tests good enough to reach the goals?

The test generation is a big problem when verifying a model. With unit tests, one can argue that each line of code has been executed (100% code coverage), but that is just a statement for that everything has been executed. Has it been executed correctly? Is every combination of computations in the system necessary to ensure correctness, or with other words, is it possible to collapse some states in the system’s state space without endangering the safety of the whole system?

There must be an evaluation of the solution after the model has been implemented. Does it ensure functional safety? How can one measure the size of the state space that is actually verified? Even if test generation is implemented properly, the solution might not fulfill the functional safety properties defined by the ISO-standard.

One must propose and motivate what should be done to be able to achieve a semi formal verification. This can include a confidence interval for how certain the verification is. The confidence interval would help describing the visited state space because it is probably not feasible to exhaustively visit all states due to combinatorial explosion problem. [16]

The main objective is to prove that it is possible to do semi formal verification for an AUTOSAR module and its specification. It should not matter which configuration that is used or how the module is implemented, because the specification should hold for all configurations and implementations. Every company that implements the module should be able to run the final code to achieve “semi formal verification”. Since modules in AUTOSAR are dependent, the work presented here should be generalized so it can be hooked on when implementing test suits, using the same techniques, for other modules.

1.4 Scope

We will use AUTOSAR 4.0 revision 3 for our thesis work. Since this version of AUTOSAR consists of more than 100 specifications and other auxiliary materials[17], we will limit our scope to one specification. The module of this thesis is the Watchdog Manager. This module provides monitoring services used to maintain correctness. The module is chosen because it got dependencies, and is used to report development and production errors, but mainly because it executes safety-critical work. The fact that it got dependencies is important when doing integration testing between different modules.

The aim of the work is to verify software components. In other words no work considered hardware or a combination of hardware and software will be prioritized. All implemented code for the verification will run on a standard PC-machine.

We will not implement deprecated API-functions in our model, nor will we test configurations which will give raise to segmentation faults.

Chapter 2

Theory

2.1 Software Testing

There are a lot different testing methods and the testing also varies in different faces of the development life cycle. Software testing can be divided into four different categories: correctness testing, performance testing, reliability testing and security testing. This thesis focuses of correctness testing. Correctness testing needs a set of rules, which defines the behavior of the software. [18]

2.1.1 Random Testing

When using random testing, test cases are simply selected at random from the input domain. Random testing has been shown to be cost efficient for a lot of programs. Subtle errors can be found at low cost and random testing combined with other testing methods may result in powerful and cost-effective testing solutions. [18]

Random testing usually provides low code coverage because test cases are uniformly distributed. As an example: To test an if-statement, which compares equality of an 32-bits integer variable $v \in \mathbb{Z}_{32}$ to a fixed value $v_{comp} \in \mathbb{Z}_{32}$, one must generate v_{comp}. Otherwise the if-statement will not evaluate to true. If test cases are uniformly chosen, the likelihood of generating v_{comp} is $P = \frac{1}{2^{32}}$. [19]

In white box testing, code knowledge can be added to the generation of test cases. This can be done by narrowing the input domain and thereby make it possible to drastically increase the number of needed test cases. Sub domains, subsets of the input domain, can be chosen to represent both valid conditions, as well as invalid conditions. In the example above, one sub domain can consist of only the value v_{comp}. Another sub domain can be defined as $\mathbb{Z}_{32} \backslash \{v_{comp}\}$, hence every other 32 bit integer. Now only one test case is needed from each of the two sub domains. The test case from $\{v_{comp}\}$ will make the if-statement evaluate to true and the test case form $\mathbb{Z}_{32} \backslash \{v_{comp}\}$ will make it evaluate to false. [20]

2.1.2 Property Based Testing

The purpose of property based testing is to establish formal validation results through testing. Property based testing assumes that the specified property captures everything of interest in the program. This because of that property based testing only validates the property. The property must hold whenever the program is executed. [21]

A problem with testing is to figure out when enough testing has been carried out. Property based testing solves this by performing an iterative strategy. A test is negative if it violates the property. The test is positive if a series of tests produces no errors and the test is "complete" under some coverage metric. A test is "incomplete" if a series of tests passes, but is not complete under the coverage metric. The iterative process comes from continuously modifying the property and selecting test cases which eventually makes the test "complete". [21]

Programs often consist of several independent properties, for example: array bounds, race conditions and authentication. Such properties can be put together as one property that should hold for the whole program. Hence property based testing is very likely an iterative process, where properties are put together until the test becomes "complete". [21]

2.1.3 Model Based Testing

Model based testing uses a model of a system to select test inputs that tries to find test cases that explores the behavior of the system. The output from the system can then be compared against the model. The advent of model based testing has given rise to new techniques that efficiently analyses system models with respect to the behaviour of the systems requirements. Such techniques can for instance find counter examples, when system requirements are violated. The meaning of counter examples are concrete and minimalistic cases when a system has an incorrect behavior with respect to its requirements. [22]

Model based testing makes it possible to manually select algorithms that automatically and systematically generates test cases from a model of the system. [23]

2.1.4 Model Checking

Model checking exhaustively checks the whole state space of a model that is constructed for a system. Only the model of the system is checked and there are no actual tests performed on the system. Instead the model is verified. Model checking uses properties, which generally can be classified into either *liveness* and *safety.* Without formality *liveness* means that eventually some wanted behavior will happen and with *safety* means that unwanted behaviour will never happen. [24]

Model checking systems can be built on a finite state automata. For instance the verification tool SPIN, see 3.1, has such a model checking system. [25]

2.1.5 Model Checking vs Testing

For larger system, when the state space is very large, it is more reasonable to use testing since it is infeasible to test the whole state space. It is, however, possible to extract certain parts of the system, which are considered important, and construct a model for those parts, which could be formally verified using model checking. [26]

2.2 Formal Methods and Verification

It is almost impossible to write a specification in a natural language and not make room for different interpretations and misunderstandings. Hence *Formal Methods* are introduced which are based on formal languages that have precise rules. Describing a system with formal notation gives the ability of automating test cases for the system, since it is precisely defined. Formal methods can be used everywhere in the development life cycle and not only when writing specification for the whole system. [12, p.272]

In the ISO standard ISO 26262, see section 2.3.1, something called semi-formal notation is also mentioned. The difference being that formal notation needs both semantics and syntax to be well defined but semi-formal notation needs only the definition of semantics. [27]

To determine whether the output of a life cycle phase fulfills the requirements specified by a previous phase, *verification* is needed. Formal verification is the verification of a system using formal methods. The exact meaning of verification is however confusing. The definition may differ in comparison of academic or industrial use. Even in different phases of the safety life cycle verification is conducted in various forms. [28][4, 8:9.2][12, p.309]

2.3 Industrial Standards

2.3.1 IEC 61508 and ISO 26262

IEC 61508 adopts a four level system for categorizing the severity of hazards. It also adopts a six level system for classifying the frequency of a hazard. There are four risk classes which are given the values 1-4 where 1 corresponds to the most serious accidents and 4 to the least serious. Based on this, IEC 61508 has a four level classification of safety integrity levels called SIL, ranged from 1, being the least critical, to 4, being the most critical. Each of the safety integrity levels has a criteria of maximum frequency of failures which a system built on that SIL must satisfy. In other words, a SIL is a level of measure of the reliability of a safety function. Due to the fact that the failure of a safety function can lead to a hazardous event, the safety integrity of a specific safety function must be of such a level to ensure that the failure frequency is sufficiently low or that the consequences of the hazardous event are modified to meet a tolerable risk. To ensure safety, functions with SIL 4 need to be tested and documented the most. [12][11][29][30]

The automotive functional safety standard, ISO 26262, has adopted a similar system of safety integrity levels, called automotive safety integrity levels or ASIL. As with

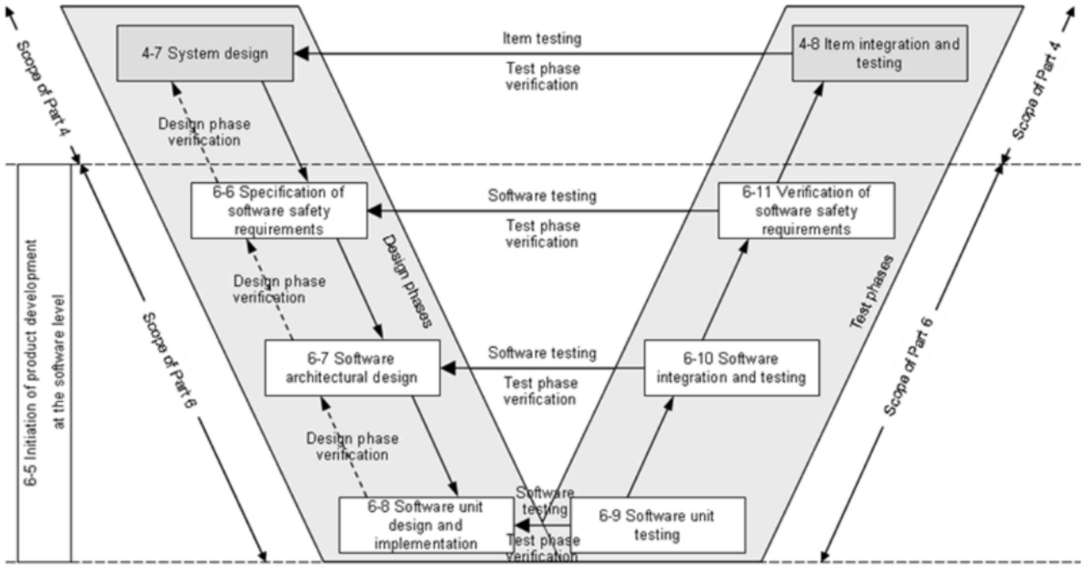

4-7 System design
Item testing
Test phase verification
4-8 Item integration and testing
Scope of Part 4
Design phase verification
6-6 Specification of software safety requirements
Software testing
Test phase verification
6-11 Verification of software safety requirements
Design phases
Test phases
6-5 Initiation of product development at the software level
Scope of Part 6
Design phase verification
6-7 Software architectural design
Software testing
Test phase verification
6-10 Software integration and testing
Design phase verification
6-8 Software unit design and implementation
Software testing
Test phase verification
6-9 Software unit testing
Scope of Part 6
Scope of Part 4

Figure 2.1: Phases of software development in the standard ISO 26262

IEC 61508 there are four integrity levels, ranged A-D, but there are no direct correlation between the two. [31]

ISO 26262 describes the full development process, from concept to production, with functional safety in mind. For software development, the standard has a reference model with different phases of the process, see figure 2.1. Each phase in the reference model is dependent on the earlier phases. The reference model has the shape of a V, where the left side contains all development phases, and the right side contains the test phases. The work flow from this view starts with the phase “specification of software safety requirements”. This phase specifies the software requirements needed to ensure the stability of the system. They are derived from the system design specification. This is part of the integration between software and hardware. The second phase is the “software architectural design”. It represents the interaction between all software components. The third phase and the last development phase, in the model is the “software unit design and implementation”. This phase contains the implementation of each module. If the implementation does not meet the specified safety, the product needs to go back to an earlier phase and be redesigned. [4, 6:5.4][12, p.8-9]

Each of these phases is tested thoroughly with the phases “software unit testing”, “software integration and testing” and “verification of software safety requirements”. The unit testing phase confirms that the implementation of the module fulfills the design specifications. If the product pass this phase, it continues to the integration and testing phase, otherwise it is sent back to the implementation phase. The objective in the phase software integration and testing is to integrate the software units and demonstrate that the architectural design is correct. A demonstration that the software safety requirements is met, is performed in the phase “verification of software safety requirements". [4, 6:5.4]

2.3.2 AUTOSAR (AUTomotive Open System ARchitecture)

The AUTOSAR platform has a layered software architecture. This means that the architecture is divided to a number of different layers, such as the application layer, runtime environment, the basic software layer, and the microcontroller. In the figure 2.2 the basic software layer is represented as four different parts; services, ECU abstraction, microcontroller abstraction, and complex drivers. [32]

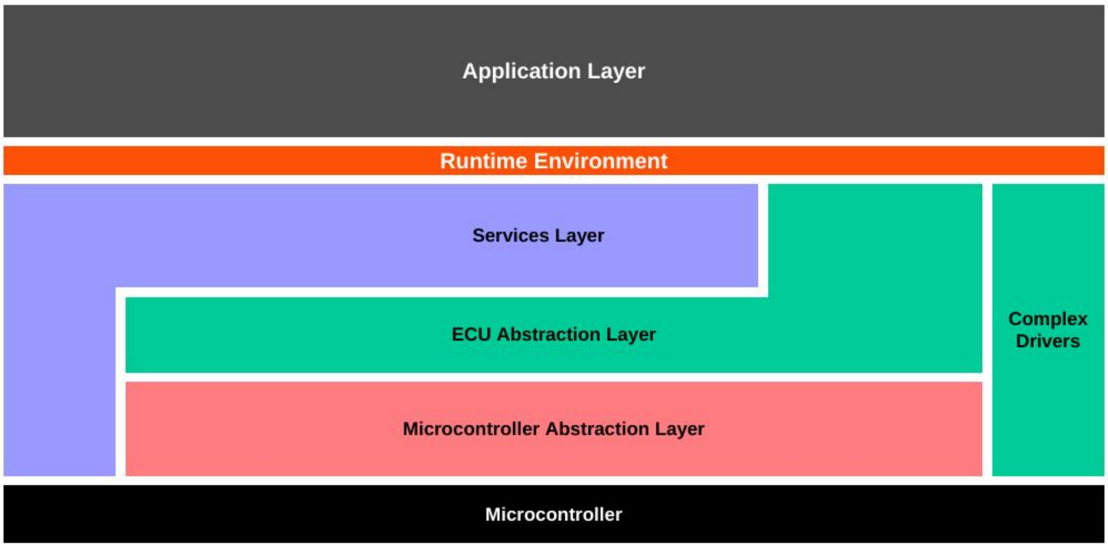


Figure 2.2: The AUTOSAR software architecture. Noticeable is that the basic software layer is divided further into four categories with even more subsections.

The runtime environment is the operating system, and the microcontroller is the hardware. The software running in the application layer is for example software components for sensors and actuators. One example of how the different parts in the basic software layer is integrated, is the watchdog, which consists of several parts as seen in figure 2.3. The microcontroller abstraction layer has the drivers for the watchdog; the interaction with the microcontroller. Then there is the watchdog interface at the ECU abstraction layer. The watchdog interface is the onboard device abstraction. Last is the watchdog manager (abbreviated WdgM) which runs as a system service in the service layer. [32]

The watchdog has a number of dependencies to other services in the basic software layer. For example when an error is found by the watchdog, it could either be reported to the diagnostic event manager or the development error tracer depending on the type of error. These are two services that are used for error management. [33]

AUTOSAR's concept is to make it possible for vehicle manufacturers to buy mod-

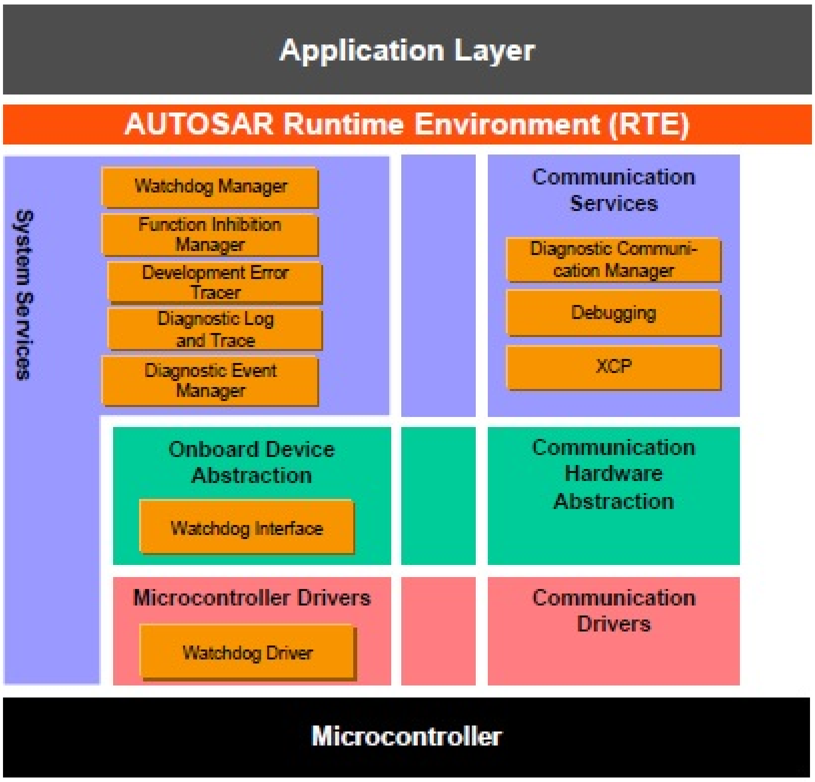


Figure 2.3: The Watchdog and some other related modules.

ules from different software developers, which will still work together in unison. For a software developer to present a software module with functionality that fits different vehicle manufacturers, the standard introduces configurations. The configurations specifies a number of parameters that can be configured in order to fit a specific vehicle manufacture. In the watchdog manager for example, there is parameters that specify if the watchdog manager should report errors to the diagnostic event manager (DEM), or which type of supervision that should be executed and what to supervise. [10][32]

The current version of AUTOSAR, version 4, has been designed with functional safety in mind. Essential concepts of ISO 26262 have been developed alongside AUTOSAR. [10]

2.4 Verification Methods

The standard IEC 61508 propose two methods to formal verify a program. The key is to model the program into one of the following state transition models. [11, p.127]

1. Finite state machines/state transition diagrams

2. Timed Petri nets

IEC 61508 emphasises that Timed Petri nets are best suited for concurrent programs. Regarding the finite state machine method, the following criteria needs to be satisfied for the implemented state machine to be formal verified [11, p.77-79][12, p.322]:

completeness the system must have an action and new state for every input in every state,

consistency only one state change is described for each state/input pair, and

reachability whether or not it is possible to get from one state to another by any sequence of inputs.

If the state machine is correctly implemented it represents a correct model of the original program. If it does not exist any unwanted transitions or states, then the original program is formal verified.

Since most program specifications are written in natural languages there may be a lot of ambiguities. Techniques have been developed to reduce such cases, and these techniques are often referred to as semi formal verification, because they often lack the mathematical rigor associated with formal verification. These methods use textual, graphical or other notation; often several techniques are used in unity. [12, p.91]

The description of semi formal verification in IEC 61508 states: "Semi-formal methods provide a means of developing a description of a system at some stage in its development, i.e. specification, design or coding. The description can in some cases be analyzed by machine or animated to display various aspects of the system behavior." [11, p.77]

2.4.1 QuickCheck

QuickCheck was invented by Koen Claessen and John Hughes, as a testing module for Haskell in 2000. In 2006 John Hughes founded the company Quviq together with Thomas Arts. Quviq offers a commercial version of QuickCheck for Erlang. One of the main differences, except from the programming language, is that the commercial version of QuickCheck has a C-testing interface. Hence it is possible to test C-code in Erlang with the help of QuickCheck. All test code is written in Erlang and checked against API calls to the C-code, this is called model based testing. It is not necessary to have the actual source code; it is enough to only supply the compiled byte code and some library files of the program to be able to test it. [34][35]

QuickCheck uses property based testing, which means that system requirements are implemented and tested as properties. QuickCheck also makes use of random testing, but has guided random test generation. This means that the samples can be weighted to cover certain parts of the state space with more likelihood. [35]

Chapter 3

Method

3.1 Existing verification tools

Software unit testing can be achieved by using a lot of different tools. For example unit testing can be done with the help of static methods such as code reviews and static path analyzers, and dynamic methods such as automatic generation of test inputs or designing test cases to be used alongside the actual code. The choice of verification tool was therefore not the most essential when it comes to pure unit testing. One can of course take the simplicity to achieve good unit testing into account, but this was not the goal of this project. [36] [37]

The phase “verification of software safety requirements” in the V-model, will not influence the choice of benchmarking software; to be able to test this phase, a greater amount of components of the whole system must be available. Such components include hardware, and this report will not cover hardware integration. The scope of this project was to be able to run the implementation on a standard PC-machine.

The most interesting part of the V-model was the phase “software integration and testing”. If it exists a tool that could be used to easily combine tests and requirements from different modules, and if it was possible to test functional safety concept from this combination, for example by corrupting some software elements.

Two tools that could be used in order to achieve better code and some functional safety was SPIN and Parasoft C/C++test. While SPIN can be used to verify the model, Parasoft C/C++test is used to define policies on work flow as well as on coding.

3.1.1 SPIN

SPIN is used to trace logical design errors in distributed software. It supports a high level language, called Promela, to specify system descriptions. Promela is an acronym for PROcess MEta LAnguage, and is a verification model language. The system properties that should be checked are written in logical temporal language (LTL), and SPIN reports errors such as deadlocks, race conditions and incompleteness between these properties and the system model. It also supports embedded C-code as part of the model specifications. It supports random, interactive and guided simulation, with both partial and exhaustive proof techniques. [38]

3.1.2 Parasoft C/C++test

Parasoft C/C++test is a commercial integrated development testing solution for C and C++. It automates code analysis, and enforces code policies depending on given rules. The solution is part software, part practical rules for team collaboration. It can detect certain run-time errors such as memory access errors, null pointer referencing, buffer overflows, division by zero and the use of uninitialized memory or variables. It can create and execute unit tests and collect code coverage from application executions. [39]

Parasoft claims that it should be possible to satisfy some of the ASIL requirements using their solution. [40]

3.2 Specification

In AUTOSAR, specifications for each module is given in text form. Therefore before a module can be tested, that specification must first be implemented in code.

3.3 Testing

Properties of a module have to take the current state in consideration, since most functions written in an imperative language are not immutable. This gives raise to the idea of a state based testing tool.

3.4 Choice of AUTOSAR module to test

When deciding which AUTOSAR module to test, there were a number of modules up for discussion. Since the goal was to get a proof of concept; examining if it was possible to get an ASIL-classification and achieve functional safety using QuickCheck, it seemed preferable to choose a less complicated module. It was also desirable to have the actual C-code and not just library files, because then ambiguities in AUTOSAR could be checked in a more efficient way.

3.5 The Watchdog Mangar (WdgM)

The AUTOSAR module that was chosen was the watchdog manager. This module seemed to fit the needs because it is a medium sized module which is highly state dependent and safety critical. It is safety critical since it monitors the hardware watchdogs. Since the objective was to be able to formally verify AUTOSAR modules and thereby examine if it was possible to higher the ASIL-classification, it seemed reasonable to chose a safety critical module. To evaluate if it was possible to reach the objectives, a module with some functionality, which still was not to complex, was desirable. The watchdog manager is described in detail in appendix B. As described in B.2 the watchdog manager has a global status that defines its general behavior. This status can be assigned 5 different values: WDGM_GLOBAL_STATUS_DEACTIVATED, WDGM_GLOBAL_STATUS_OK, WDGM_GLOBAL_STATUS_FAILED, WDGM_GLOBAL_STATUS_EXPIRED and WDGM_GLOBAL_STATUS_STOPPED. The transition between these statuses can be described with the use of a state machine. The global status was considered very important, since it specifies correct and incorrect behaviour of the watchdog manager.

3.5.1 The state machine

The watchdog manager's global status state machine is shown in figure 3.1. Its transitions depend on the changes of the supervision functions variables, and the current state. If the behavior of the watchdog manager is correct, it will stay in either the state WDGM_GLOBAL_STATUS_OK or WDGM_GLOBAL_STATUS_DEACTIVATED. There are however lots of reasons for the status to change from the correct state. It depends on the arguments of the API-calls but also the order of the commands that are called and which AUTOSAR configuration that is supplied. The configuration is important because it specifies how much faulty behavior the watchdog should tolerate. It could also disable some states and state transition or make some transition more likely to happen. The effect can for instance come from the number of checkpoints supplied in the configuration. A correct behavior of the watchdog manager depends on that checkpoints are reached with correct timing and does so in the right order.

Besides the transition between the deactivated state and the OK state, the only function that can give rise to state transitions for the global status is the main function. In a working ECU, the main function should continuously be called by the run-time environment (RTE), in a configured time interval. Note that the timing is not used when using QuickCheck, see section 3.11.

3.5.2 Important Functions

The most interesting API-calls are the ones that modifies the internal state of the watchdog manager, see appendix B.4, namely WdgM_Init, WdgM_DeInit, WdgM_SetMode, WdgM_MainFunction, and WdgM_CheckpointReached. The reason for this is that they will influence the result of the following API-calls.

The Init and DeInit functions can just change the global status between two states and should only change the state of the watchdog when the watchdog is in either WDGM_GLOBAL_STATUS_OK or WDGM_GLOBAL_STATUS_DEACTIVATED, according to figure 3.1. If this happens they will change the internal state of the watchdog independently of previous called commands. The behavior of these commands will therefore not vary much.

WdgM_SetMode changes the mode, but should retain the global and local statuses of the supervised entities. It should not be possible to change the mode if the watch-

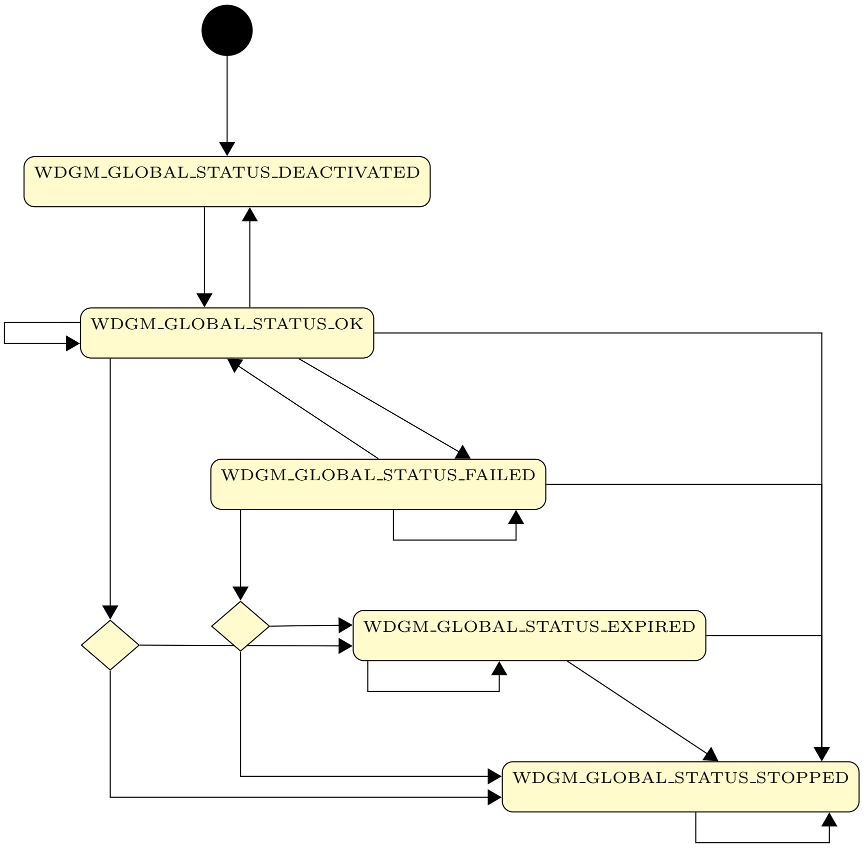

WDGM_GLOBAL_STATUS_DEACTIVATED
WDGM_GLOBAL_STATUS_OK
WDGM_GLOBAL_STATUS_FAILED
WDGM_GLOBAL_STATUS_EXPIRED
WDGM_GLOBAL_STATUS_STOPPED

Figure 3.1: State diagram that shows possible transitions between states

dog manager is in either WDGM_GLOBAL_STATUS_EXPIRED or WDGM_GLOBAL_STATUS_STOPPED.

The two remaining API-calls that needs to be discussed in details are the main function and the checkpoint reached function. As can be seen in figures 4.1(a), 4.2(a) and 4.3(a), they are also the two commands that are called the most.

WdgM_MainFunction

The WdgM_MainFunction handles alive supervision calculations, and the function WdgM_CheckpointReached handles the increasing of the alive counters, a certain number of calls to WdgM_CheckpointReached must be done before each WdgM_MainFunction. It does not end there. Each checkpoint may have some logical supervision, so the order of the called checkpoints is important as well. It is also possible to set deadline supervision for a supervised entity. Both deadline supervision and logical supervision is handled by

WdgM_CheckpointReached.

WdgM_CheckpointReached

Deadline supervision demands that a configured amount of time must have elapsed since the start checkpoint was visited. Because AUTOSAR does not specify how the handling of time should be implemented, see sections 3.11 and 4.5.1, we implemented the model as the source C-code was implemented, with the use of WdgM_MainFunction. This is possible because we know that WdgM_MainFunction is called periodically.

3.6 Implementation

The chosen module was already unit tested and run actively in the lab environment.

The implementation of the properties was done to be able to test API-calls, which is also described in appendix A, against the C-code. QuickCheck then checked that the postconditions held, according to figure 3.2. The postconditions were written to test that AUTOSAR requirements held. In other words that the API-calls were called correctly.

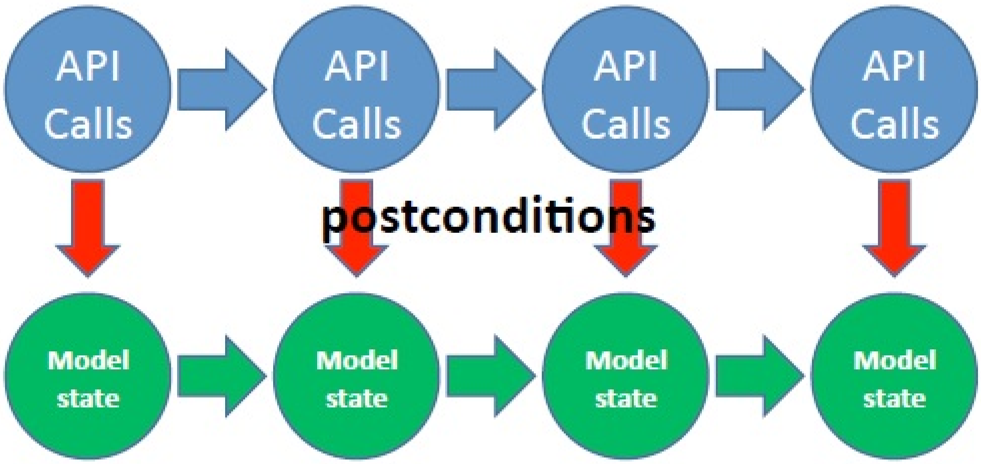


Figure 3.2: Shows Erlang modeled states with calls against the C-code

3.6.1 Formal Notation

For QuickCheck to be able to automatically generate test cases, AUTOSAR specifications written in a natural language, needed to be transformed into properties in Erlang code. In other words transforming informal notation into formal notation.

A problem when translating the AUTOSAR specifications into code was that there were ambiguities. It was easy to see that there was room for different interpretations, which most likely would result in implementation conflicts later. This is described more precise in section 3.7.

The translating process was done iteratively as described in section 3.6.3.

3.6.2 Independence of the Erlang implementation

The implementation in Erlang was done independent from the design choices in the C-code. The idea was to ensure an independent model; if the model was inspired by the C-code, it could have transmitted faults. Implementing the Erlang module independent of the C-implementation would also result in that ambiguities in the AUTOSAR specification would be easier to find, since two different interpretations of the same specification would eventually be available.

3.6.3 Iterative strategy

The implementation of the AUTOSAR module in Erlang was done in an iterative way. Not every piece of code were required to be implemented before tests could be run. This is because a module in AUTOSAR consist of a number of API-calls. It was enough to implement some of the specifications for one API-call before tests could be run. Of course this tested only the implemented part of the C-code. Early tests may not have fully tested the implemented API-call because some branches in the C-code will never have been reached before other unimplemented API-calls.

3.7 Conflicts and Bugs

Early in the implementation phase QuickCheck found differences between the Erlang and C-implementation. This was expected because every programmer makes mistakes. The question was whether the fault was in the C-code or the Erlang code. Then the API was thoroughly read and a conclusion was made based on this. Either a bug in the C-code was found or the Erlang code needed to be corrected. There were however cases when the API was ambiguous. In those cases the C-interpretation was chosen as correct and the ambiguous specification was documented.

There is a number of possible ways to handle bugs when QuickCheck encounters them. The problem is that QuickCheck generates arbitrary command sequences, it cannot "save" an error and proceed to find the next error. Either the C-code or the model needs to be adjusted. The best way, with the model in mind, would often be to let a third party correct the discovered bugs. However this is time consuming because the support line has often already much to do, and the releases does not come that often. Another way is to mock the faulty API-call. In other words simulate the output of the C-code in order to circumvent or hide a API-function, but then you will only find one bug per function, strictly limiting the probability to find bugs. There is a QuickCheck Erlang module for the purpose of mocking C-code, see appendix A.3.6. Then there are two equally good methods. Either the Erlang model needs to have the fault implemented, or the C-code needs to be fixed. There are pros and cons with both methods. If the

Erlang model introduce bugs, there may be secondary failures which are not discovered. This could also happen when correcting the C-code, but then more knowledge of the module is needed, and some of the secondary failures can easier be avoided. It also takes more time to get the extra knowledge of the C-code.

The alternatives listed below were discussed.

1. Fix the C-code, in other words change the source code. Knowledge about the structure in the C-code is needed.

2. Mocking, in other words simulate different C-code output. The pitfall is if that each mocked function eliminates all bugs in the function. Not only a selected subset; at most one bug per function can then be found.

3. Change the Erlang module to a faulty behavior to follow the C implementation. The problem is that other configurations or updated versions of the C-code will show up as faulty when using the same Erlang model, and it could be hard to discover secondary failures.

We choose to correct the C-code, item 1, because then we had direct feedback and could discover where in the code the bugs were introduced. Also this was the most dynamic of the alternatives and allowed further bugs to be found.

When thoroughly reading the AUTOSAR API not only ambiguous rules were found but also rules that contradicted each other were recognized. In those cases the implementation in the C-code was followed.

Although the C-code was used in lab environments, bugs were found early in the process.

3.7.1 Advantage of having the Actual C-code

A great method for understanding the AUTOSAR specification, when a clear interpretation of it did not exist, was to examine the C-code. QuickCheck can be used to test libraries when only the compiled source code is available. However, this makes the ambiguities harder to discover, because a third model would be needed to justify whether the C model or the Erlang model would be correct.

3.8 Implementation structure

The final implementation consisted of several Erlang modules. Table 3.1 lists the modules defining the watchdog manager. There are also other modules that reads configuration files, defines the generators, measuring code coverage etcetera, those modules have however no equivalence in the C-code.

Table 3.1: Erlang modules defining the watchdog manager

modules	descriptions
wdgm_helper	Helper module used by most of the other modules.
wdgm_checkpointreached	Erlang version of checkpointreached, see appendix B.4.6.
wdgm_main	Erlang version of the main function, see appendix B.4.11.
wdgm_pre	Checks for AUTOSAR preconditions.
wdgm_post	Checks for AUTOSAR postconditions.
wdgm_next	Defines the watchdog manager state model, utilizes both wdgm_checkpointreached and wdgm_main.

3.9 Evaluation of the Implementation

If tests return positive, it does not really say much more than that those tests evaluated to true. There was a need to evaluate what was actually tested. Coverage of the code and also coverage of visited states was needed to evaluate tests.

3.9.1 Verifying the tests

When the module was fully implemented in Erlang code there had to be some assurance of that every piece of code in the C implementation was actually tested. Code coverage for the Erlang implementation was measured using the Erlang module *cover*. The coverage were only measured on the modules listed in 3.1 since they are the only modules that defines the Erlang version of the watchdog manager.

To be able to measure the code coverage of the C-code the commercial tool Bullseye Coverage was used. When using these tools it was easy to see that the result was not good enough. The main problem was that WdgM was put in an absorbing state. All commands that were executed after that, were not able to change the state of the WdgM. The reason for that an absorbing state was reached was the availing of negative testing. The testing was negative because invalid command sequences and arguments were generated.

Figure 3.3 shows an example of how the status of the watchdog manager changed during the execution of API-calls. After a number of commands the absorbing state *stopped* was reached.

3.9.2 Finding better test cases

The next step was to tweak the generators that were used by QuickCheck to construct valid API-calls. This was done to find better test cases, i.e. there were a number of branches in the C-code that needed a specific sequence of API-calls with correct arguments, to be reached. For example, it was unreasonable to test functions often when the initialization function WdgM_Init has not yet been called.

Thanks to QuickCheck's weight feature, it was simple to change the ratio of the generation of certain API-calls; by matching the state and the function name of the

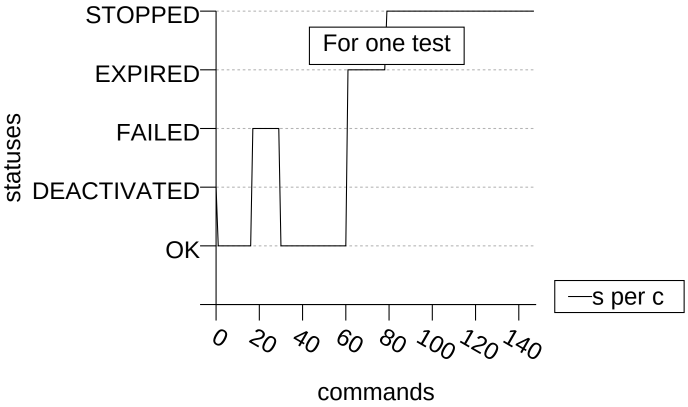


Figure 3.3: Shows changes to the global status in the execution of one QuickCheck test.

API-call, one can change the probability of generation of that call.

For example the initialization function WdgM_Init should have a high priority if it had not been called previously, and a low priority if it had been called previously.

```
weight(S, 'WdgM_Init') ->
  case S#state.initialized of
    true                          -> 1;
    _                             -> 200
  end;
```

It was a good idea not to lower some ratios to much, because then certain API-sequences would not be generated, and bugs could have been missed.

The tweaking of the generators were implemented in an iterative way by changing the probability properties of the generators and analyze the results and the coverage. After the analysis, the generators were tweaked even more to make the result and coverage even better.

To get a better picture of the work flow used in this thesis see figure 3.4.

A challenging step was the analysis of the results. If the testing tool returned zero errors what did that say about the robustness of the input byte code? Passing 100 of 100 tests is just a statement and does not say anything more than that some tests passed. Can tests be implemented in a clever way so that it is possible to get some kind of confidence on the correctness of the code?

3.10 Configurations

When the code coverage was calculated it was recognized that not every piece of code was executed. The reason seemed to be that the current configuration disallowed the execution of some parts of the code, even though the program behaved correctly. It was

1. Construct a model for an AUTOSAR module in Erlang
2. Run QuickCheck for this model and compare the results with the output from the C-code.
3. Tweak the generators for the test cases
4. Evaluate the results
   (a) Evaluate the state space
   (b) Evaluate if the test cases are relevant
   (c) Minimize irrelevant states
5. Are the results good enough, does it satisfy the requirements for the ASIL levels?
6. If not go the step 2

Figure 3.4: Work flow

easy to run tests on several configurations, because the implementation of the Erlang module was made independent of configurations. This resulted in almost full coverage.

Three configurations with different complexity were used. The first one, an example configuration (this will further on be called the Example configuration), had many supervision functions configured for each mode, and followed a strict execution of the program.

There were also a minimal configuration (BSI configuration) which, in lack of supervision functions, only could change the global status between WDGM_GLOBAL_STATUS_OK and WDGM_GLOBAL_STATUS_DEACTIVATED. This on the other hand, tested some null conditions, for example when there are no supervised entities. For more information about the states, see appendix B.2.

The last configuration (named Freescale configuration), which was one of the configurations that were used actively in lab equipment, was similar to the example configuration but a bit more relaxed. The global status stayed in a non-absorbing state more often; it was easier to do positive testing.

The tweaking of generators, where the aim was to generate better test cases, seemed in some sense to be configuration dependent. Better test cases were generated if the generators were tweaked according to a specific configuration, see chapter 4.

3.11 Calling the API-commands

API-calls were executed by QuickCheck using the *run_commands/1* function according to appendix A. The run-time environment module (RTE) is however responsible for the scheduling of the main function, which according to AUTOSAR, should be executed in a given time interval. Since the RTE was not available when testing the watchdog manager, the main function was called randomly and it was assumed that every time the main function was called, a given amount of time had passed.

Except for the main function, only one internal algorithm that was used by the watchdog manager was time dependent, namely the deadline supervision algorithm. A supervised entity with deadline supervision consists of two checkpoints. One start checkpoint, one stop checkpoint and a maximum time it should take to reach the stop checkpoint after the start checkpoint was reached. The AUTOSAR specification was however lacking of a clear definition of how time should be handled. The C-code just used ticks, not actual time stamps, which was incremented every time the main function was called. It was in other words assumed that the RTE was able to execute the main function correctly and a fixed amount of ticks would always represent the same amount of time. After accepting this implementation, it was easy to adopt the same approach in the Erlang module. More about this can be found in section 4.5.1.

3.12 Model State

The model state was constructed as minimal as possible. It is easier to get the model correct if the model state is kept simple. A complex structure means more data needs to be searched through when a bug is found. Even though it was tempting to use a more efficient data structure, a simple Erlang record was used to represent the model state. Using more efficient data structures could for instance speed up the execution of tests. The main reason for using a record was to make it easy to follow the model state and make it possible to use QuickCheck's function *eqc_statem:show_states/1*, see appendix A, for showing the state between command sequences. The efficiency of the test model was considered less relevant than the readability of the model state. The idea was to make it easy to find the actual bug, when conflicts arouse between the C-code and the Erlang module. Running the actual tests was also not considered time or memory critical.

3.13 Minimizing counter examples

After a generated command sequence fails, QuickCheck automatically tries to minimize the command sequence needed to prove that there is a difference between model and code; a so called counter example. This process is called shrinking. This makes it easier to find where in the code the failure arouse from. For example, a sequence of 30 commands could be shrunk to 5, if all those 5 commands is needed for the counter example, and none of the 25 other commands is needed. This is very useful, because it is easier to delouse a small number of command sequences than a large number with lots of unnecessary commands (for the counter example).

Chapter 4

Result

4.1 Achieving good test cases

To find good test cases is not trivial. It may not be enough to generate test cases which follows a correct behavior. Negative testing also needs to be taken into consideration, otherwise a module that is incorrectly implemented could return bad arguments or results to other modules.

4.1.1 Negative Testing

The problem with negative testing is that the watchdog manager quickly will be put in an absorbing state when an invalid API-call is executed. After such an invalid execution, all following API-calls will not test anything new since the absorbing state is reached. As a consequence it is not possible to test multiple invalid executions with one test. A problem using QuickCheck is that the test cases are generated before the actual execution of the program; it is likely that a lot of API-calls will be executed after an invalid execution of the program. This results in that negative testing may be time consuming using QuickCheck.

4.1.2 Positive Testing

There are a lot of things that can cause an invalid behavior of the watchdog manager. Because of this, there may be a lot of calculations needed to find test cases that are valid, so that the absorbing state is not reached. The complexity of finding such cases grows with the complexity of the configuration. However, properties are continuously tested as long the absorbing state is not reached. Eventually, even when trying to make use of positive testing, the absorbing state will be reached if the configuration is not too simple. This is because the order of commands will influence if the behavior is correct or not. Even if it is possible to prioritize certain commands the random factor of QuickCheck will eventually cause an invalid behavior. See figure 4.1(b), 4.2(b), and 4.3(b) for how many commands that are executed before the absorbing state 'WDGM_GLOBAL_STATUS_EXPIRED' is reached.

4.1.3 Prioritized supervision algorithms

The supervision algorithms are important parts of the watchdog manager since they specify transitions, liveness and timing properties of the program. It was therefore chosen to prioritize different algorithms when running some of the tests on the module. When doing so, more bugs were found. This emphasizes the importance of finding tests that are critical for the system and not only trust that results have been achieved based on line coverage.

Since there are different supervisions of checkpoints that can be configured at the same time, the supervision functions must be prioritized when generating command sequences and arguments. A checkpoint that is generated too many times can for example cause the alive supervision to fail because it goes outside of its bound. Alive supervision can also fail if a checkpoint is not generated enough times, according to the configuration. If a checkpoint is generated only because it needs to be inside of its alive supervision bound, then there is a risk that rules for deadline or logical supervision is violated. The easiest way to prioritize checkpoint generation is to start with logical supervision. This is because logical supervision follows certain graphs, where each vertex is represented by a checkpoint and each edge is a valid transition between two checkpoints. These graphs are defined by the logical supervision functions in the AUTOSAR configuration. Logical supervision maintains both internal graphs, inside of each supervised entity, and external graphs which are transitions between supervised entities. It is easy to get next possible checkpoints for all graphs, and then check whether one of those checkpoints also is configured for alive or deadline supervision. If it is, calculate the status for those supervision functions and then choose which checkpoint should be selected. If checkpoint generation is not prioritized with logical supervision as foundation, alive supervision or deadline supervision could be used. This is harder because it is more likely to end up with a checkpoint that violates logical supervision rules.

4.1.4 Tweaking the generators

The generators did not need to be tweaked much when performing negative testing since if the commands are uniformly random generated by QuickCheck an invalid behavior will quickly arise. However, with a small probability of generation, null pointers were also passed as arguments to the API-commands to see how the system behaved. Turning of the configuration parameter WdgMDevErrorDetect caused segmentation faults when passing null pointers. This does not follow the requirements for functional safety, see section 4.5.1.

4.2 Configurations

The watchdog manager (WdgM) was tested using three different configurations. The configurations were of different complexity. One was a minimal configuration, one an example configuration and one was a live configuration, used in actual implementations.

Because there is only a small number of commands that influences the state transitions, those commands were tweaked and therefore was generated more often. On the other hand, all get-functions were tweaked to not be generated as often.

4.2.1 BSI

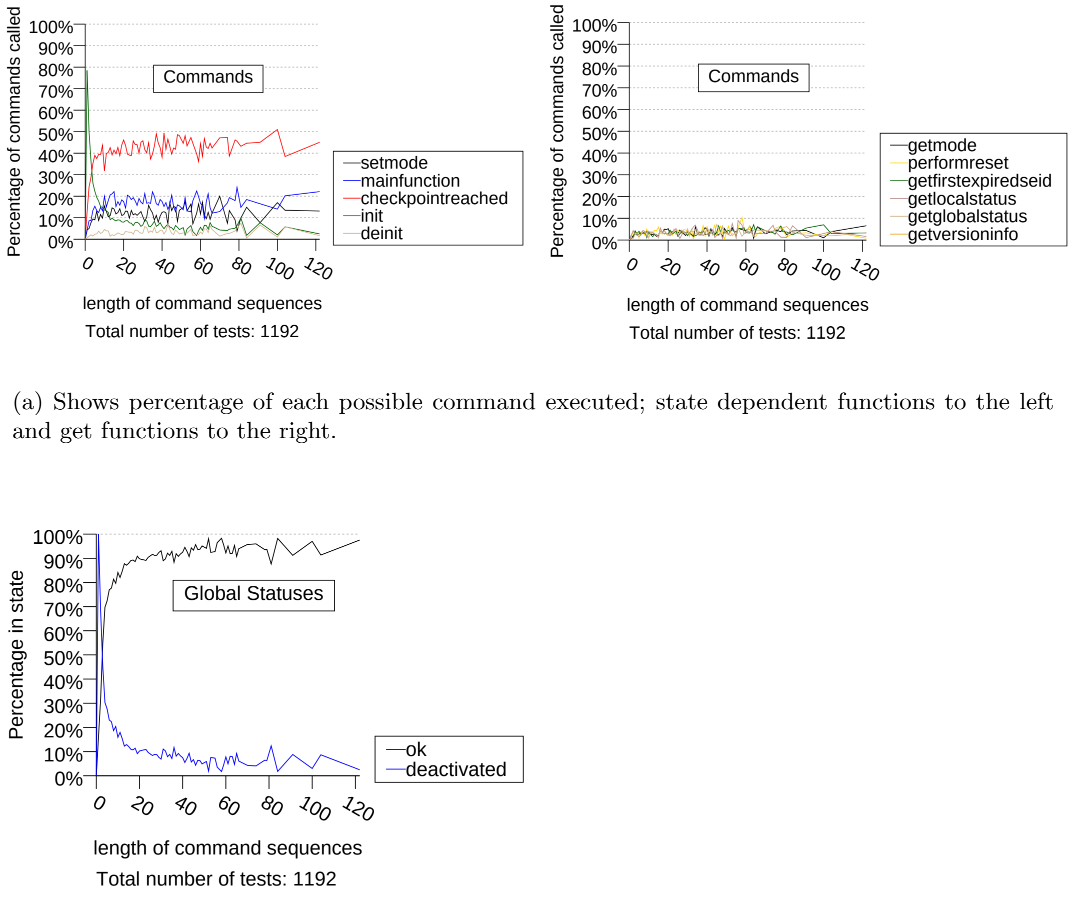
As a highly simplified configuration, *BSI* gives in some sense good results. Using this configuration the WdgM never visited the absorbing state according to figure 3.1. However, looking at the state transitions, as seen in figure 4.1(b) and table 4.1, only two states are visited. This happens because the configuration is too simple, it is actually impossible to hit any other states than WDGM_GLOBAL_STATUS_OK or WDGM_GLOBAL_STATUS_DEACTIVATED. There are no checkpoints or supervision functions configured for the *BSI* configuration. It is easy to run tests using this configuration, but it does not by itself, fully test the code because most of the specification requirements will never be tested. The untested requirements are mainly requirements for supervision functions that are, according to the configuration, never executed. Those untested requirements also leaves other requirements untested because the watchdog manager never reaches a state when those other requirements must hold.

Table 4.1: State transitions of the BSI configuration.

Number of tests: 1192

To From	DEACTIVATED	EXPIRED	FAILED	OK	STOPPED
DEACTIVATED	**02.87%**	00.00%	00.00%	**09.25%**	00.00%
EXPIRED	00.00%	**00.00%**	00.00%	00.00%	**00.00%**
FAILED	00.00%	**00.00%**	**00.00%**	**00.00%**	**00.00%**
OK	**03.12%**	**00.00%**	**00.00%**	**84.76%**	**00.00%**
STOPPED	00.00%	00.00%	00.00%	00.00%	**00.00%**

Figure 4.1(a) shows how many times a certain command was generated versus the length of the command sequence that was generated. E.g. the function WdgM_CheckpointReached was generated in average a little more than 40% of all calls. This is because, in any other configuration, the supervision functions often demand that a certain number of checkpoints is reached before the next main function is called. There is also a dependency the other way around; the main function often has to be called a certain number of times before WdgM_CheckpointReached is called on a certain supervised entity. This is why the main function also has quite high proportions. Other functions that stand out are WdgM_SetMode and WdgM_Init. WdgM_SetMode is called because different modes can have different supervision functions and supervised entities. That is why we need to call this function often. It should retain the states of supervised entities that are activated in the new mode and should reset the local state if the entity is deactivated in the new mode. The function WdgM_Init is in contrast called fewer and fewer times. This function is only needed when the global state is deactivated. It has more likelihood



(a) Shows percentage of each possible command executed; state dependent functions to the left and get functions to the right.

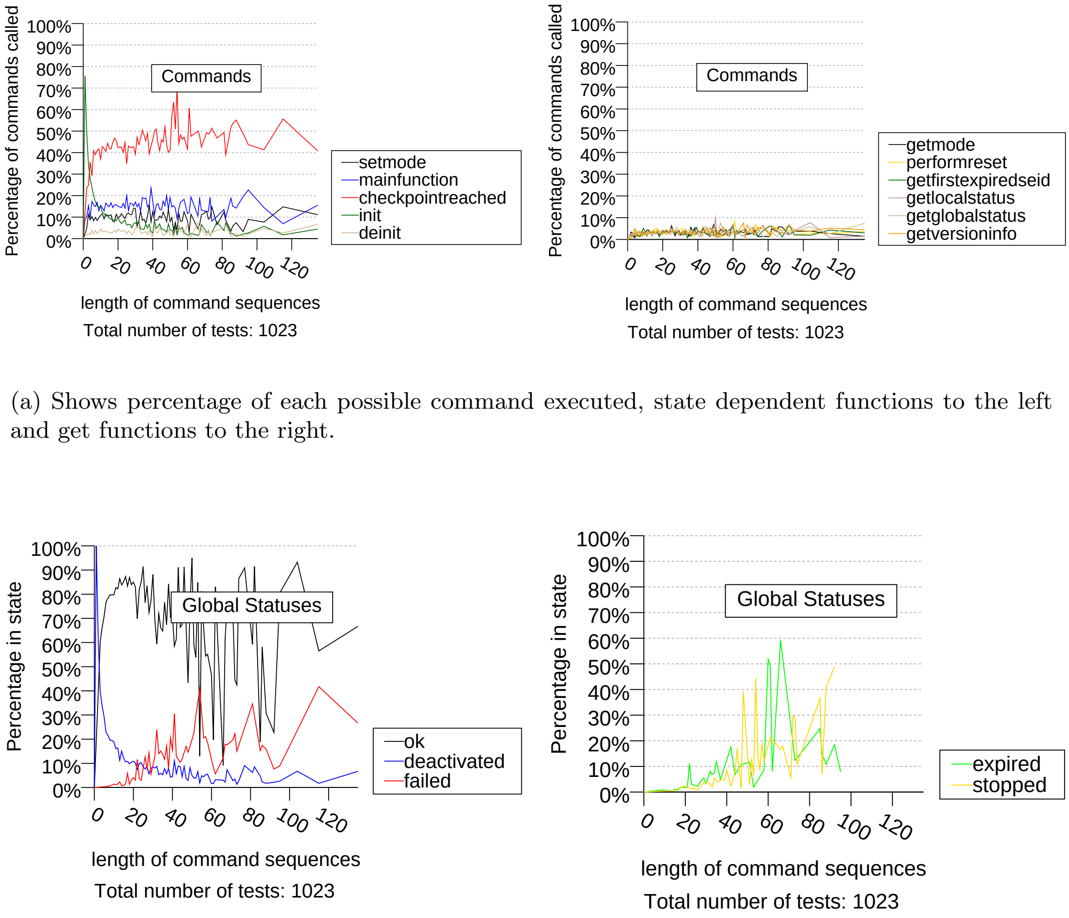
(b) Shows percentage of each visited global status

Figure 4.1: Some statistics of the BSI minimal configuration.

to be generated among the first commands in the command sequence, or right after a WdgM_DeInit deactivation call.

4.2.2 Freescale

The Freescale configuration is, compared to BSI, a more realistic configuration. All supervision algorithms are configured and there are both external and internal graphs for logical supervision. It is also one of the configurations that is actively used in lab environments. The state machine for the global status is totally covered by running QuickCheck, see table 4.2 and figure 4.2(b). Looking at table 4.2 one can see that some transitions are done very seldom. This is due to the fact that a lot of things must be fulfilled for those transitions to occur, which also highly depend on the configuration supplied. Due to the randomness factor of QuickCheck such cases are hard to reach.



(a) Shows percentage of each possible command executed, state dependent functions to the left and get functions to the right.

(b) Shows percentage of all visited global status, recoverable statuses to the left and non recoverable to the right.

Figure 4.2: Some statistics for the Freescale configuration.

4.2.3 Example

The example configuration is somewhat more complex than the Freescale configuration. This is because it supports all functionality of an configuration. Because of the complexity, some transitions are harder or even impossible to reach. Noticeable is that the transition from the state WDGM_GLOBAL_STATUS_FAILED to the state WDGM_GLOBAL_STATUS_OK according to figure 3.1 is never made, see table 4.3. The reason

Table 4.2: State transitions of the Freescale configuration.

Number of tests: 1023

To From	DEACTIVATED	EXPIRED	FAILED	OK	STOPPED
DEACTIVATED	**02.43%**	00.00%	00.00%	**08.32%**	00.00%
EXPIRED	00.00%	**03.36%**	00.00%	00.00%	**00.11%**
FAILED	00.00%	**00.17%**	**07.77%**	**00.12%**	**00.11%**
OK	**02.56%**	**00.18%**	**00.87%**	**69.53%**	**00.12%**
STOPPED	00.00%	00.00%	00.00%	00.00%	**04.34%**

for that is that the alive functions must fail once and then continue without failures.

Table 4.3: State transitions of the Example configuration

Number of tests: 1067

To From	DEACTIVATED	EXPIRED	FAILED	OK	STOPPED
DEACTIVATED	**02.03%**	00.00%	00.00%	**07.23%**	00.00%
EXPIRED	00.00%	**27.00%**	00.00%	00.00%	**01.00%**
FAILED	00.00%	**00.11%**	**02.99%**	**00.00%**	**00.04%**
OK	**01.52%**	**02.50%**	**00.38%**	**36.93%**	**00.12%**
STOPPED	00.00%	00.00%	00.00%	00.00%	**18.15%**

4.3 Statistics

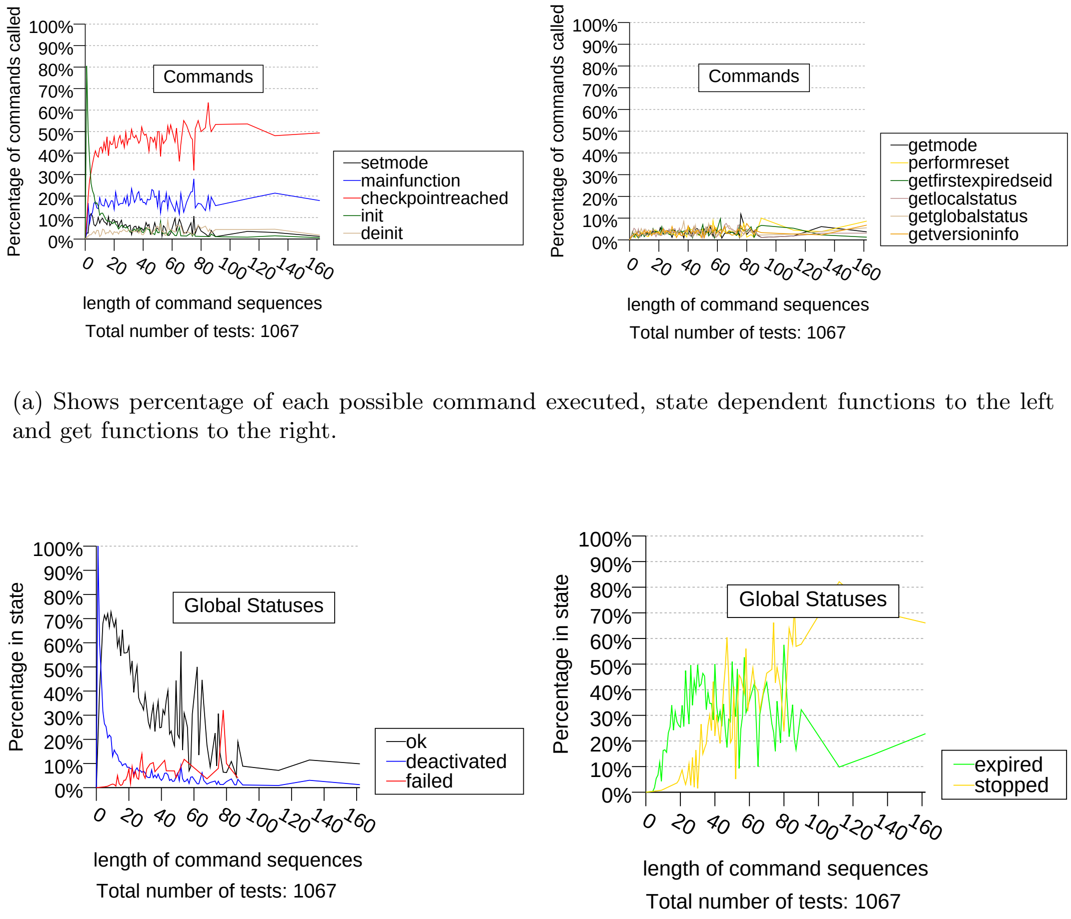
The distribution of API-calls seems, according to figure 4.1(a), 4.2(a) and 4.3(a), to be the same for all configurations. The arguments to the API-calls is however different, even though it is not seen in those plots.

4.4 Coverage

4.4.1 Erlang module

The Erlang module cover was used to calculate the line coverage for the Erlang module. To get an idea of how many tests that were needed to be executed, before the line coverage of the Erlang module converges against a certain value, the coverage was measured after each executed test for every configuration.

As can be seen in the figures 4.4, 4.5 and 4.6, the example configuration takes the longest time before it converges. It also becomes clear that the freescale configuration needs more tests to converge than the BSI configuration. The complexity of the configurations seem to play an important part. This is not surprising because a more complex configuration may drastically increase the state space.



(a) Shows percentage of each possible command executed, state dependent functions to the left and get functions to the right.

(b) Shows percentage of all visited global status, recoverable statuses to the left and non recoverable to the right.

Figure 4.3: Some statistics for the Example configuration

The Erlang model is separated into a number of files. The results of the coverages of these files, after running all configuration, can be seen in table 4.4. This table lists the same modules as table 3.1.

The module wdgm_pre checks preconditions of the model state; constraining the model states ability to change. This will affect the wdgm_next module. The wdgm_next module changes the model state, and is called after a call to the C-code is performed. Note that wdgm_next module are totally independent of the C-code, see appendix A. The wdgm_next module has two helper modules wdgm_main and wdgm_checkpointreached which changes the model state if the main function or the checkpoint reached function was the most recently called functions in the C-code. The module wdgm_post checks

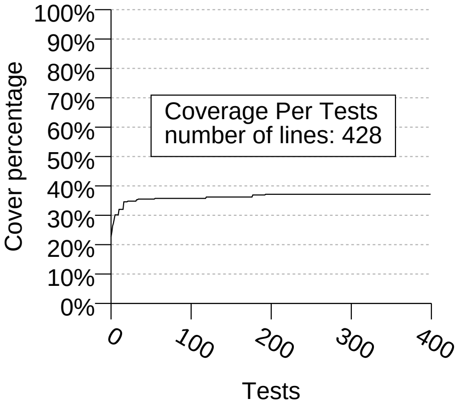


Figure 4.4: Coverage per tests using the BSI configuration

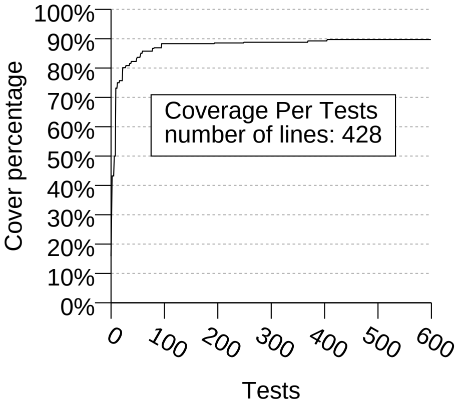


Figure 4.5: Coverage per tests using the Freescale configuration

that AUTOSAR specification holds, by comparing the models state and the actual state of the C-code.

The total line coverage results is 97.38%. The reason we do not achieve 100% code coverage depends on certain delimitations. Some lines can not be covered when the configuration parameter WdgMDevErrorDetect is true. On the other hand, if it is false, then the C-model will fail with a segmentation fault and the Erlang model will not be covered anyway. There is also a number of implementation specific lines, which another C-code model might reach but not the one that we had. There are also places that depended on the configuration to be more simple. A good idea is to supply configurations that only has specific supervision functions configured. Then it should be possible to prioritize only that supervision function and get better coverages.

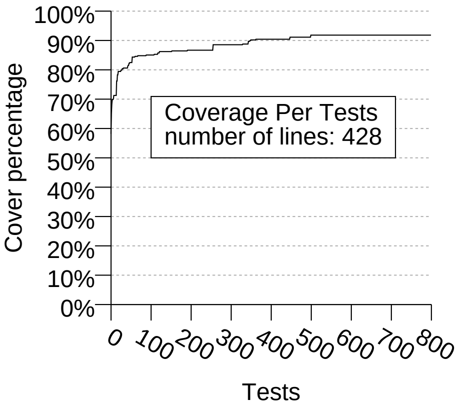


Figure 4.6: Coverage per tests using the Example configuration

Table 4.4: Shows coverage statistics generated by the Erlang module cover

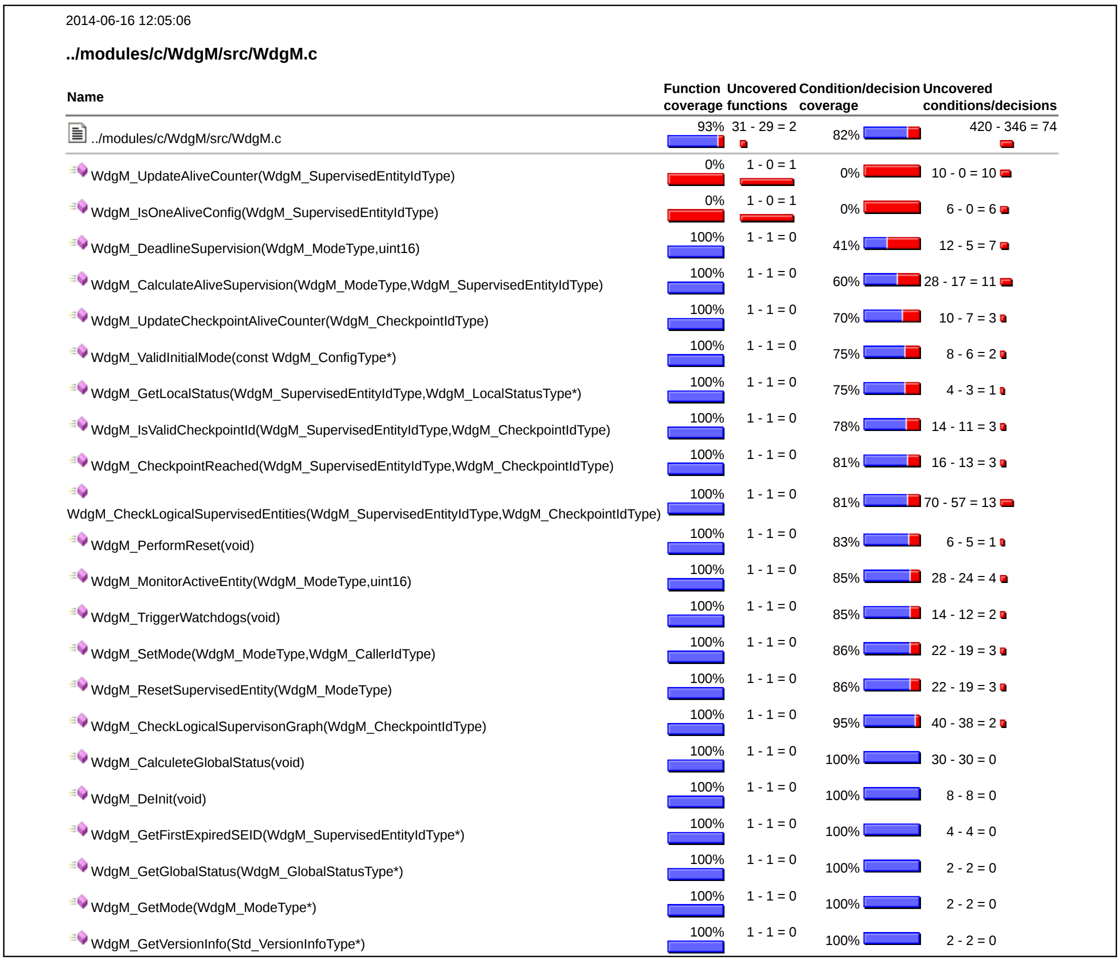
.

Total line coverage 97.38%			
module	total number of lines	lines covered	line coverage (%)
wdgm_helper	80	79	98.75%
wdgm_checkpointreached	104	98	94.23%
wdgm_main	81	80	98.77%
wdgm_pre	19	19	100%
wdgm_post	99	96	96.97%
wdgm_next	37	37	100%

4.4.2 C-code

Bullseye Coverage was used to analyze the coverage of the C-code. The results show that the condition/decision coverage is 85.64% and all functions except for two are covered, see figure 4.7. The reason that there are functions which are not covered, is that one of the functions is deprecated and the other is a support function to that function. One of our delimitations was to not implement deprecated functions into Erlang code. The missing condition/decision coverages in the C-code are for example checks for null pointers, some of which never evaluated to false. Many checks seems to be redundant and impossible to evaluate to true, if one excludes the possibility of hardware failures or other failures which may corrupt the memory of the watchdog manager. There is as well branches and conditions regarding the WdgMDevErrorDetect configuration parameter, which if turned off resulted in a segmentation fault.

The coverage statistics shown in figure 4.7 is constructed using the three configurations mentioned. Using several configurations gave better results since some code blocks were impossible to reach if not certain configuration parameters were set.



2014-06-16 12:05:06

../modules/c/WdgM/src/WdgM.c

Name	Function coverage	Uncovered functions	Condition/decision coverage	Uncovered conditions/decisions
../modules/c/WdgM/src/WdgM.c	93%	31 - 29 = 2	82%	420 - 346 = 74
WdgM_UpdateAliveCounter(WdgM_SupervisedEntityIdType)	0%	1 - 0 = 1	0%	10 - 0 = 10
WdgM_IsOneAliveConfig(WdgM_SupervisedEntityIdType)	0%	1 - 0 = 1	0%	6 - 0 = 6
WdgM_DeadlineSupervision(WdgM_ModeType,uint16)	100%	1 - 1 = 0	41%	12 - 5 = 7
WdgM_CalculateAliveSupervision(WdgM_ModeType,WdgM_SupervisedEntityIdType)	100%	1 - 1 = 0	60%	28 - 17 = 11
WdgM_UpdateCheckpointAliveCounter(WdgM_CheckpointIdType)	100%	1 - 1 = 0	70%	10 - 7 = 3
WdgM_ValidInitialMode(const WdgM_ConfigType*)	100%	1 - 1 = 0	75%	8 - 6 = 2
WdgM_GetLocalStatus(WdgM_SupervisedEntityIdType,WdgM_LocalStatusType*)	100%	1 - 1 = 0	75%	4 - 3 = 1
WdgM_IsValidCheckpointId(WdgM_SupervisedEntityIdType,WdgM_CheckpointIdType)	100%	1 - 1 = 0	78%	14 - 11 = 3
WdgM_CheckpointReached(WdgM_SupervisedEntityIdType,WdgM_CheckpointIdType)	100%	1 - 1 = 0	81%	16 - 13 = 3
WdgM_CheckLogicalSupervisedEntities(WdgM_SupervisedEntityIdType,WdgM_CheckpointIdType)	100%	1 - 1 = 0	81%	70 - 57 = 13
WdgM_PerformReset(void)	100%	1 - 1 = 0	83%	6 - 5 = 1
WdgM_MonitorActiveEntity(WdgM_ModeType,uint16)	100%	1 - 1 = 0	85%	28 - 24 = 4
WdgM_TriggerWatchdogs(void)	100%	1 - 1 = 0	85%	14 - 12 = 2
WdgM_SetMode(WdgM_ModeType,WdgM_CallerIdType)	100%	1 - 1 = 0	86%	22 - 19 = 3
WdgM_ResetSupervisedEntity(WdgM_ModeType)	100%	1 - 1 = 0	86%	22 - 19 = 3
WdgM_CheckLogicalSupervisonGraph(WdgM_CheckpointIdType)	100%	1 - 1 = 0	95%	40 - 38 = 2
WdgM_CalculeteGlobalStatus(void)	100%	1 - 1 = 0	100%	30 - 30 = 0
WdgM_DeInit(void)	100%	1 - 1 = 0	100%	8 - 8 = 0
WdgM_GetFirstExpiredSEID(WdgM_SupervisedEntityIdType*)	100%	1 - 1 = 0	100%	4 - 4 = 0
WdgM_GetGlobalStatus(WdgM_GlobalStatusType*)	100%	1 - 1 = 0	100%	2 - 2 = 0
WdgM_GetMode(WdgM_ModeType*)	100%	1 - 1 = 0	100%	2 - 2 = 0
WdgM_GetVersionInfo(Std_VersionInfoType*)	100%	1 - 1 = 0	100%	2 - 2 = 0

Figure 4.7: Shows coverage statistics generated by Bullseye Coverage

4.5 Functional Safety analysis

The V-model used by ISO 26262 requires that a certain work flow is taken into consideration during the whole development process. It is therefore hard to analyze code that is written without the standard in consideration and then examine if it fulfills the requirements of the given standard.

Since one important part of the functional safety concept is that it must be taken in consideration during the whole development process, one can not simply say that QuickCheck makes it possible to acquire functional safety. If every development step before the implementation of the watchdog manager satisfies the requirements for functional safety, one also must follow the same constraints in the remaining part of the development life cycle to achieve functional safety. If this is assumed, there is still one important assumption left before one can reason about how QuickCheck can benefit. This assumption is based on that the model for the watchdog manager is correct, namely the AUTOSAR specification.

4.5.1 AUTOSAR

Due to the informal syntax of AUTOSAR it is not well fitted for functional safety, since the informal syntax makes it possible for different interpretation. To be able to reach the requirements for a higher ASIL classification, AUTOSAR modules must be interpreted dependently. In other means they must agree on the same model.

One way of doing this is to interpret AUTOSAR in a model based language like SPIN and check that the model, after transforming it into formal syntax, is valid. The model in itself should then not contain any bugs. This can also be done using QuickCheck.

The model for the system must, in a functional safety point of view, be implemented before the C-code is written. It seems pointless to test the actual C-code before there is assurance for that the model actually is correct according to formal syntax.

C in itself is also a formal language but C is not good fitted to formally defining the actual requirements of AUTOSAR. This is for instance because of its low level nature.

In this thesis a defined AUTOSAR model already existed, written in C-code. The AUTOSAR model was implemented in Erlang and compared against the first model. A better work flow, with functional safety in mind, had been to define AUTOSAR in a model based language and check that this model holds. Then implement the actual C-code following the formal notation of the previous constructed model. Implementing the C-code would be easier because then there are no room for different interpretations. After the C-code is implemented, write the model in Erlang following the formal model written in the model based language. Then there is again no room for different interpretations. After the two implementations of the model, compare those using QuickCheck. If there are no bugs, then the original model was translated into C-code correctly.

The proposed work flow would require a lot of work, which is beyond the scope of this thesis. For example the C-code needs to be rewritten.

Development error detection

It was discovered that it existed a configuration parameter, WdgMDevErrorDetect, which would turn off functional safety checks. This made the C-code crash with segmentation fault as soon as negative testing was performed. This could for example be null pointers or improper identification numbers. AUTOSAR is not specified enough for the parameter WdgMDevErrorDetect to be switched off. With functional safety in mind, this parameter must be on!

Definition of time

In AUTOSAR time is specified as seconds and there are two functions that need to keep track of the time. First it is the main function that needs to be scheduled periodically by the run-time environment. This is done with a configuration parameter given for each mode. Time is also needed for deadline supervision. In deadline supervision when a start checkpoint is reached, a timer should start. If the final checkpoint is not reached within the configured time marginal, then the deadline supervision for the supervised entity with the given checkpoint will fail. Because it is known that WdgM_MainFunction should be called periodically, it could be used for the measurements of time. For each call to WdgM_MainFunction a counter could be incremented to keep track of the cycles.

4.5.2 Fulfilled ISO 26262 requirements

ISO 26262 mentions several requirements that QuickCheck will be able to fulfill. Aside from general requirements, e.g. that a "safety plan" should be available, there are also verifications in which the hardware should be taken into consideration. This is beyond the scope of this thesis. There are also tests that needs several modules implemented to make any sense. Such tests have not been executed, since only the watchdog manager module has been tested, but should be possible to run after implementing more modules in Erlang code. QuickCheck has a module for mocking C-code, see appendix A, which could possible also be used for running such tests.

4.5.3 Confidence interval

As seen in section 4.4, total coverage of a module could be hard to reach, because the code is configuration dependent. This makes it also hard to predict how well tested a module is with the use of a confidence interval. One can at least say that the more complex a configuration is, the more test cases is needed to improve the coverage. For the Freescale configuration for example, it took approximately 400 test cases to achieve 90% code coverage, while it only took around 200 test cases to achieve the maximum 38% coverage the BSI configuration could supply.

4.5.4 Measurements of the state space

One way of measuring the state space, is to collect statistical data during the execution of the tests. It is hard to say much about the whole system's state space since it is very large, due to the combinatorial explosion problem. In the case of an AUTOSAR module the state space also varies on the configuration in use, since features and supervision features can be configured. However looking at certain parts that are considered to be important for the system, much more can be said. For instance when examining the internal graphs of the watchdog manager, one can easily see that every node in those graphs is visited.

Measuring the code coverage can in itself tell if important parts of the state space are covered or not. This is possible because the parts of the code that changes the state of the watchdog manager are not covered. These parts can however be dead or redundant code, in the worst case, if something unexpected happens to the hardware, which is beyond this thesis. Total code coverage is hard, or may be even impossible to reach. Even if all lines of code in for example an algorithmic part of the watchdog manager are covered, this algorithm may not be totally verified. This is because the states are dependent of actual values of variables.

Chapter 5

Discussion

The model has been implemented in an iterative way. Function for function, requirement for requirement. This process is very easy with the use of QuickCheck. In the beginning there were a lot of negative testing, because we did not care about tweaking the generators. By limiting the state space for negative tests, we achieved a better ratio for positive testing when generating random tests. This was done by letting some command sequences weight more than others when QuickCheck generates the test cases. In figure 5.1 we show how the state space is minimized when QuickCheck has been tweaked.

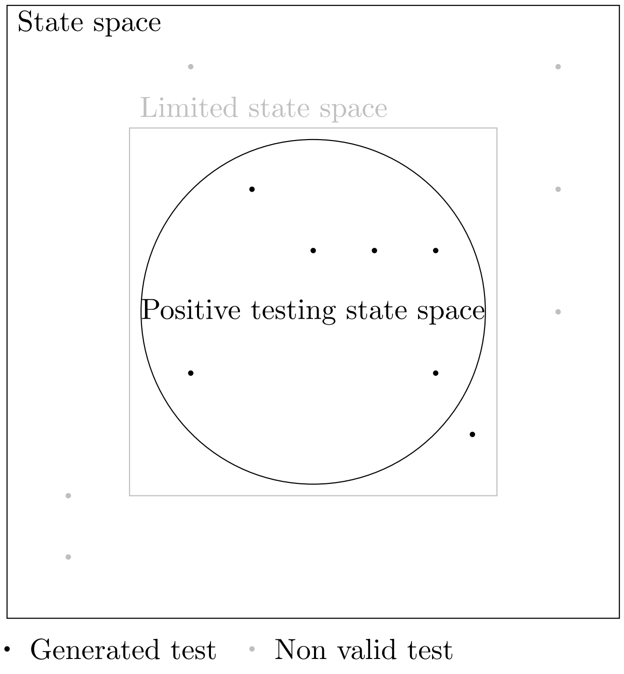


Figure 5.1: The state space when tweaking QuickCheck's generators.

The outer box in figure 5.1 represents the full state space, and the inner circle represents positive test cases. This means that all test cases outside of the circle represent negative test cases. When tweaking generators it means we can limit some of the negative test state space, in order to have QuickCheck generate more interesting positive test cases. The black dots in the figure represent different generated test cases, and the gray dots represent test cases which will not be generated after the new limitations.

As an example, it is unnecessary to test functions if the watchdog manager state machine has not been initialized with a call to the WdgM_Init function. We can weight the generation of the WdgM_Init function to be called with a higher ratio if the state machine is not initialized yet.

```
weight(S, 'WdgM_Init') ->
  case S#state.initialized of
    true                              -> 1;
    _                                 -> 200
  end;
```

We want to do this because there are only some functions that actually change the state of the watchdog manager; there is a lot of so called get-functions which only retrieve information.

Sometimes the model needs to be corrected because of ambiguities in AUTOSAR or errors in the model. Quite often it was the C-code that had the errors and needed to be corrected. Easier when you do white box testing, when the source code is known, because then you can really check the code and compare with the requirements.

The C-code coverage was measured differently from the Erlang code coverage, which used line coverage instead of condition/decision coverage. Recursion is often used in functional programming languages as Erlang, therefore it is not as suited for condition/decision coverage as C-code is. Erlang also comes with a coverage library, which makes it easy to use. On the other hand, measuring line coverage in an imperative language like C is a bit redundant since statements are executed sequentially. Therefore conditions/decision coverage seems more reasonable.

We achieved fair coverage of the C-code, around 85%, and the Erlang code, around 97%. The problem was the requirements, where we achieved around 50%. It would help if a QuickCheck model was implemented for the whole system as well. Many of the requirements had dependencies in other modules, and some requirements for the file structure, the configuration or even the generation of files.

QuickCheck is good for overall testing, and can help with raising the functional safety of modules.

Stripped of comments and blank lines, the implemented Erlang model is almost 1300 lines of code. This is to be compared with the C-code which is over 14500 lines of code.

5.1 Future work

An interesting thing we wanted to do from the beginning was to implement another module, preferably one that has some kind of dependency with the watchdog manager. This is because then, it would be possible to test towards the phase “system integration and testing” in ISO 26262 and get even better results.

Another good idea is to do more negative testing, and testing of null pointers. This should be done to raise the coverage for the C-code to even better levels.

We could also try new configurations. More configurations = better testing.

Chapter 6

Conclusion

It is possible to achieve some functional safety using QuickCheck, at least within the software units. There are however a number of ISO 26262 requirements that are not possible to achieve with only a software testing tool. For example requirements that verify the hardware specifications or how the safety plan should be made. It is hard to use the ISO 26262 V-model if the software units do not follow system properties that has been verified by a system model. Because AUTOSAR is written with informal syntax, it cannot be used to verify the software units. This means one must translate the AUTOSAR requirements to formal syntax and verify that the formal requirements mean the same as the informal requirements.

It is important to not only measure the state space, but also the code coverage. This can also be done with the use of QuickCheck. It is easy to measure Erlang coverages, and it is also easy to specify which compiler QuickCheck should use to perform C-code coverages. QuickCheck gathers information about the state space, which is output after a test has been run.

Both negative and positive testing can be implemented with the use of QuickCheck. Negative testing can be time-consuming because the program quickly comes to an absorbing state. It is therefore important to tweak the generators correctly.

Integration tests can also easily be implemented, by connecting several AUTOSAR modules. When doing this even more ISO 26262 requirements can be evaluated and verified.

One needs to be aware of the configuration of the AUTOSAR module which is going to be implemented, because it may contain variables that is not safe to turn off. It may also be difficult to reach the whole state space if the configuration is to simple or to complex.

Bibliography

[1] Navet N, Simonot-Lion F. Automotive Embedded Systems Handbook. Hoboken: CRC Press; 2008.

[2] Ulsoy, Galip A, Peng H, Çakmakci M. Automotive Control Systems. New York; Cambridge [U.K.]: Cambridge University Press; 2012.

[3] Gut G, Allmann C. In: Meyer G, editor. In-Research Project E-Performance - In-Car-Network Optimization for Electric Vehicles. Berlin, Heidelberg: Springer Berlin Heidelberg; 2012. p. 69–78.

[4] ISO 26262. Road vehicles - Functional safety. ISO, Geneva, Switzerland; 2011. Available from: `www.iso.org/iso/search.htm?qt=ISO+26262&published=on`.

[5] Hughes J, Arts T, Gerdes A, Svensson H. Quviq Course material; 2013.

[6] Strassberger M, Schroth C, Bechler M, Kosch T. Automotive Internetworking. Wiley-Blackwell; 2012.

[7] Hiraoka C. Technology Acceptance of Connected Services in the Automotive Industry. Gabler Verlag; 2009.

[8] SARTRE. Safe road trains for the environment;. Available from: `http://www.sartre-project.eu/en/Sidor/default.aspx`.

[9] Charette RN. This Car Runs on Code. IEEE Spectrum. 2009 Feb;.

[10] AUTOSAR. Basic Information: Short Version; 2011.

[11] IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems. ISO, Geneva, Switzerland; 2000.

[12] Storey N. Safety-Critical Computer Systems. Harlow: Addison-Wesley; 1996.

[13] Claessen K, Hughes J. QuickCheck: a lightweight tool for random testing of Haskell programs. Acm sigplan notices. 2011;46(4):53–64.

[14] Hughes J. In: Software Testing with QuickCheck. vol. 6299. Berlin, Heidelberg: Springer Berlin Heidelberg; 2010. p. 183–223.

[15] Kantamneni HV, Pillai SR, Malaiya YK. Structurally Guided Black Box Testing. Dept. of Computer Science, Colorado State University; 1998.

[16] Grindal M. Handling combinatorial explosion in software testing [Dissertation]. Institutionen för datavetenskap Linköpings universitet; 2007.

[17] AUTOSAR. AUTOSAR 4.0; 2013. Available from: http://autosar.org/index.php?p=3&up=2.

[18] Pan J. Software testing. Retrieved September. 1999;2.

[19] Godefroid P. Random testing for security: blackbox vs. whitebox fuzzing. In: Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007). ACM; 2007. p. 1.

[20] Loo P, Tsai W. Random testing revisited. Information and Software Technology. 1988;30(7):402 – 417. Available from: http://www.sciencedirect.com/science/article/pii/0950584988900377.

[21] Fink G, Bishop M. Property-based Testing: A New Approach to Testing for Assurance. SIGSOFT Softw Eng Notes. 1997 Jul;22(4):74–80. Available from: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/263244.263267.

[22] Meinke K, Walkinshaw N. 1. In: Margaria T, Steffen B, editors. Model-Based Testing and Model Inference. vol. 7609 of Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg; 2012. p. 440–443. Available from: http://dx.doi.org/10.1007/978-3-642-34026-0_32.

[23] Schieferdecker I. Model-Based Testing. IEEE Software. 2012;29(1):14–18.

[24] Jhala R, Majumdar R. Software model checking. ACM Computing Surveys (CSUR). 2009;41(4):1–54.

[25] Holzmann GJ. Software Model Checking with SPIN. In: Advances in Computers. vol. 65; 2005. p. 77–108.

[26] Clarke EM, Grumberg O, Peled D. Model checking. Cambridge, Mass; London: MIT; 1999.

[27] Brockmeyer DU, Gros M, Valea A. ISO 26262 Compliant Automatic Requirements-Based Testing for TargetLink. An der Schmiede 4, 26135 Oldenburg, Germany: BTC Embedded Systems AG; 2012.

[28] Arts T; 2013. personal communication.

[29] Bell R. In: Dale C, Anderson T, editors. Introduction and Revision of IEC 61508. Springer Verlag London Limited; 2011. .

[30] Nordland O. In: Dale C, Anderson T, editors. A Devil's advocate on SIL 4. Springer Verlag London Limited; 2011. .

[31] Greb K, Seely A. Design of Microcontrollers for Safety Critical Operation. In: ARM TechCon3. Santa Clara Convention Center, California: Texas Instruments; 2009. .

[32] AUTOSAR. Layered Software Architecture; 2011.

[33] AUTOSAR. Specification of Watchdog Manager;. Document Identification Number 080.

[34] QuviQ. About us; 2014. Available from: `http://quviq.com/about.html`.

[35] Hughes J. QuickCheck: An Automatic Testing Tool for Haskell;. Available from: `http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html`.

[36] Naik K, Tripathy P. Unit Testing. In: Software testing and quality assurance: theory and practice. Hoboken, NJ, USA: John Wiley & Sons, Inc; 2011. .

[37] Myers GJ, Badgett T, Thomas TM, Sandler C. The art of software testing. Hoboken, N.J: Wiley; 2004.

[38] What is SPIN?;. Available from: `http://spinroot.com/spin/what.html`.

[39] Parasoft. Parasoft C/C++test;. Available from: `http://www.parasoft.com/printables/C++TestDataSheet.pdf`.

[40] Parasoft. Satisfying ASIL Requirements with Parasoft C++test;. Available from: `http://www.parasoft.com/printables/asil_automotive.pdf`.

Appendix A

Introductions to QuickCheck

A.1 The General Idea

The general idea with QuickCheck, is here explained by using an example. Let us say that one has a *sort* function that takes a list Xs of any sortable items and returns the sorted version Ys of Xs. To verify the correctness of this function it is possible to look at certain properties that must hold for a correctly implemented sort function. For instance:
The arity of Y must be the same.

$$|Xs| = |Ys| \tag{A.1}$$

The elements of Ys must actually be sorted.

$$y_{i-1} \leq y_i, \forall y_i \in Ys, i \neq 0 \tag{A.2}$$

For all permutations $Pe(Xs)$ holds:

$$sort(Zs) = Ys, \forall Zs \in Pe(Xs) \tag{A.3}$$

The sets Xs and Ys contain the same elements.

$$x \in Xs \leftrightarrow x \in Ys \tag{A.4}$$

Instead of specifying own test cases QuickCheck makes it possible to write such properties, automatically generates test cases and checks that the properties specified actually holds.

A.2 Testing C-code

QuickCheck is a testing tool for the programming language Erlang, but it is possible to efficiently test C-code, using QuickCheck, by performing API-calls against the C-code within Erlang. Lets say that there is a *Queue* implementation in C that has a function for creating a new queue as well as functions for inserting and retrieving elements from a queue structure.

```
typedef struct
{
    int size;
    int head;
    int tail;
    int* buffer;
} Queue;

Queue* new(int size) { ... }
int put(Queue* q, int element) { ... }
int get(Queue* q) { ... }
```

This program has a number of properties that should hold. For instance:

- When creating a new queue, the C function should return an address to the memory where it have allocated space.
- The function *put* should insert the element into the queue and return the element.
- It should be possible to dequeue an element in the order it was inserted with the function *get*.

Such properties can be used when using QuickCheck.

When testing C code with QuickCheck one uses state based testing. Which means that a model state S is passed around and checked against API calls according to figure A.1.

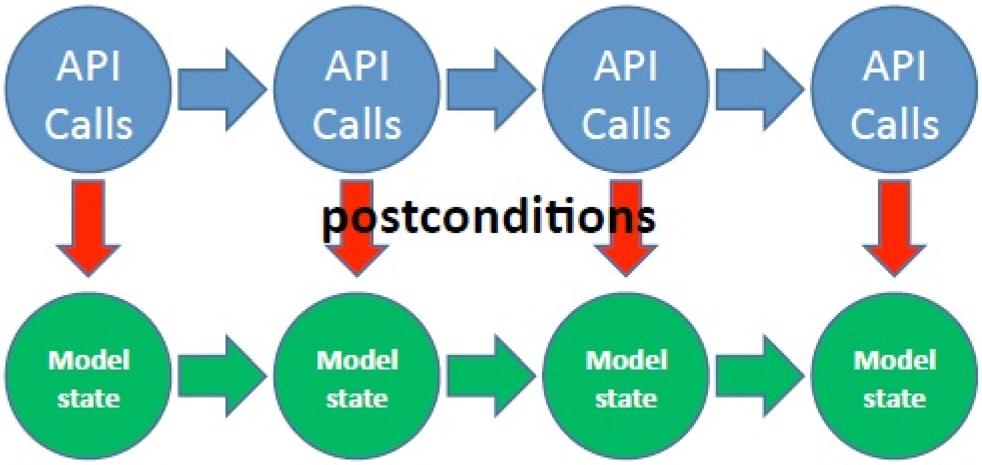


Figure A.1: Shows state based testing against API and model state

First a model of the queue must be implemented in Erlang code. It is also needed to implement a representation of the current state of the queue.

```
-record(state, {ptr, queue}).

new_next(_S, Pointer, _) ->
    #state{ptr=Pointer, queue=[]}.
put_next(S, _, [_, X]) ->
    S#state{queue = S#state.queue++[X]}.
get_next(S, _, _) ->
    S#state{queue = tl(S#state.queue)}.
```

The *record* will contain information needed to successfully test the states of the program. It has two variables; *ptr* which is the pointer to the queue and *queue* which is a list defining all inserted elements in the queue.

QuickCheck must also be told how to call the C-functions.

```
new(Size) ->
  q:new(Size).
put(Ptr, Val) ->
  q:put(Ptr, Val).
get(Ptr) ->
  q:get(Ptr).
```

Now the postconditions, program properties that must hold, are defined.

```
new_post(_S, _Arguments, ReturnValue) ->
  case ReturnValue of
    {ptr, "Queue", _} -> true;_
                       -> false
  end.
put_post(_S, [_, InsertedValue], ReturnValue) ->
  ReturnValue == InsertedValue.
get_post(S, _Arguments, ReturnValue) ->
  ReturnValue == hd(S#state.elements).
```

To be able to test these properties QuickCheck needs to know how to generate arguments for every functions.

```
new_args(_S) ->
    [nat()].
put_args(S) ->
    [S#state.ptr, int()].
get_args(S) ->
    [S#state.ptr].
```

The functions *nat()* and *int()* are generators defined by QuickCheck to generate arbitrary natural numbers and integers.

To test the functions *put*, *get* and *new* defined in the C-code a QuickCheck property is written in the following way.

```
prop() ->
  ?FORALL(Cmds, commands(),
          begin
             ... = run_commands(Cmds)
          end).
```

The function *commands()* is a generator that looks for functions defining the API-calls, such as *new*, *put* and *get*, in the Erlang module. The *commands()* function then combines every function api_i defining an API-call with a function arg_i which generates the arguments to api_i.

After the property has been implemented, it can be tested by:

```
eqc:quickcheck(prop()).
```

It is possible define how many tests QuickCheck should execute and also if the states of the model should be shown:

```
eqc:quickcheck(eqc:numtests(N, eqc_statem:show_states(prop()))).
```

A.3 QuickCheck Modules

QuickCheck consist of several Erlang modules.

A.3.1 eqc

The module *eqc* is the main QuickCheck module. This module defines a lot of macros that can be used when writing properties and also basic functions like *quickcheck*.

A.3.2 eqc_gen

The module is used for generations of test cases. The module contains various functions and macros for this purpose. There are some predefined generators, for instance for integers and characters etcetera, but it is quite easy to construct a generator for almost any data type. Just to get the idea follows code for a string generator.

```
    ?LET(Pat, nat(), vector(Pat, char()))
```

The macro *?LET* binds a generated value from the second argument, to *Pat* which can be used in the third argument. The above code binds a natural number, from the generator *nat()*, to *Pat* and creates a vector with length *Pat* of characters.

A generator can also be weighted, or in other words certain values can be more likely to be generated than others.

```
?LET(Pat, nat(), vector(Pat,
                        frequency([{1, choose(0,127)},
                                   {3, 32}])))
```

The code above will also generate a string of length *Pat*, but the generation of the white space character will be 3 times more likely to happen than a uniformly random character.

A.3.3 eqc_c

Contains the C-testing interface. In other words how to communicate with C-code.

```
eqc_c:start(q, [{c_src, "q_api.h"},
                {additional_files, ["queue.o"]}])
```

The code above starts the C-program *queue.o*, and an Erlang module is created with the name of the first parameter, *q*. This module can now be used within Erlang to call the C-program.

A.3.4 eqc_statem

Offers state based testing as shown above. A command has a definition, precondition, postcondition, and a next function.

Noticeable is that only the post function may depend on the C-code. QuickCheck has a generation step where tests are generated according to the precondition and the model state. The C-code is run first after the generation step and can only be used to check postconditions. This is actually what one want because it would be pointless to execute a program and then test it depending on the execution of the same program and not the model itself. For instance if we let the next state function depend on the C-program, then the model will be faulty if the C-program has incorrect behavior.

Possible preconditions for queue example above could be that the functions *put* and *get* can only be called if the queue has first been created and *new* can only be called with a size greater than zero.

```
new_pre(_S, [Size]) ->
  Size > 0.
put_pre(S) ->
  (S#state.ptr /= undefined) andalso
  (length(S#state.queue) < S#state.size).
get_pre(S) ->
  (S#state.ptr /= undefined) andalso
  (length(S#state.queue) > 0).
```

A.3.5 car_xml

Additional to the commercial version, there is a *car* module. This module is specifically created to parse AUTOSAR XML configuration files.

A.3.6 Other modules

There are also other modules; for instance a module for mocking C-code. Or in other words, if one has a C-function that is declared but the definition is missing, one can simulate its output. This is however not used in this thesis.

Appendix B

The Watchdog Manager (WdgM)

The watchdog is a basic AUTOSAR module. It's purpose is to supervise a programs execution by triggering hardware watchdogs entities. For the hole description of the module see the AUTOSAR specification.

B.1 Supervision, Checkpoints and Graphs

The watchdog supervises the execution of so called *Supervised Entities.* Important places in a supervised entity are marked as checkpoints. There are at least one checkpoint for every supervised entity. The checkpoints and transitions between checkpoints are defined as graphs. Checkpoints and transitions between checkpoints within a given supervised entity are marked as internal graphs. There may however be transitions between checkpoints of different supervised entities, such graphs are marked as external graphs. Available graphs are supplied by the configuration. There may be different graphs for different modes of the watchdog manager.

There are three supervision algorithms to verify the correctness of supervised entities.

- Logical Supervision:
  Logical supervision verifies if graphs are executed in the correct order.
  Let $G = (V, E)$ be a internal graph for a supervised entity S such that $\forall c_k \in V \rightarrow c_i \in S$. For the graph G there exist a start checkpoint $c_s \in V$ and a final checkpoint $c_f \in V$. The logical supervision checks that the first checkpoint c_1 that is reached has the property $c_1 = c_s$ and for every reached checkpoint c_j there exists an edge $(c_{j-1}, c_j) \in E, j \neq 1$.

- Alive Supervision:
  Alive supervision periodically verifies the timing of transitions and checkpoints reached in a graph.

- Deadline Supervision:
  Deadline supervision does the same as alive supervision but aperiodically.

B.2 Global Status

The global status represents the current state of the whole watchdog manager. There are five different statuses.

- WDGM_GLOBAL_STATUS_DEACTIVATED
  The watchdog manager is in a resting state, deactivated, and will not execute any supervision functions.
- WDGM_GLOBAL_STATUS_OK
  The watchdog manager is in a correct state.
- WDGM_GLOBAL_STATUS_FAILED
  A failure has occurred for an alive supervision and the watchdog is configured to have a tolerance against this kind of error.
- WDGM_GLOBAL_STATUS_EXPIRED
  A fault has happened and the watchdog is configured to postpone the error reaction. In contradiction to *WDGM_GLOBAL_STATUS_FAILED* there is no recovery mechanism for this state and the watchdog manager will eventually reach the state *WDGM_GLOBAL_STATUS_STOPPED*.
- WDGM_GLOBAL_STATUS_STOPPED
  This is an absorbing state of the watchdog state machine. Recovery mechanisms will be started and usually a watchdog reset will occur.

The different statuses are related to each other according to figure B.2. There is only a small number of functions that is allowed to change the global status; those are the main function, the initialization function and the de-initialization function. The main function decides the next global status by checking the local statuses of the supervised entities and the current global status. The initialization function should only be able to change the global status from deactivated to ok, and the de-initialization function from ok to deactivated.

B.3 Local Status

A local status is a status of one supervised entity and could be set according to the current local status and the results of the supervision functions. There are four different local statuses. Init setmode mainfunction

- WDGM_LOCAL_STATUS_DEACTIVATED
  If a supervised entity is set to deactivated, it will not be checked by the supervision functions.
- WDGM_LOCAL_STATUS_OK
  The supervised entity is in a correct state.

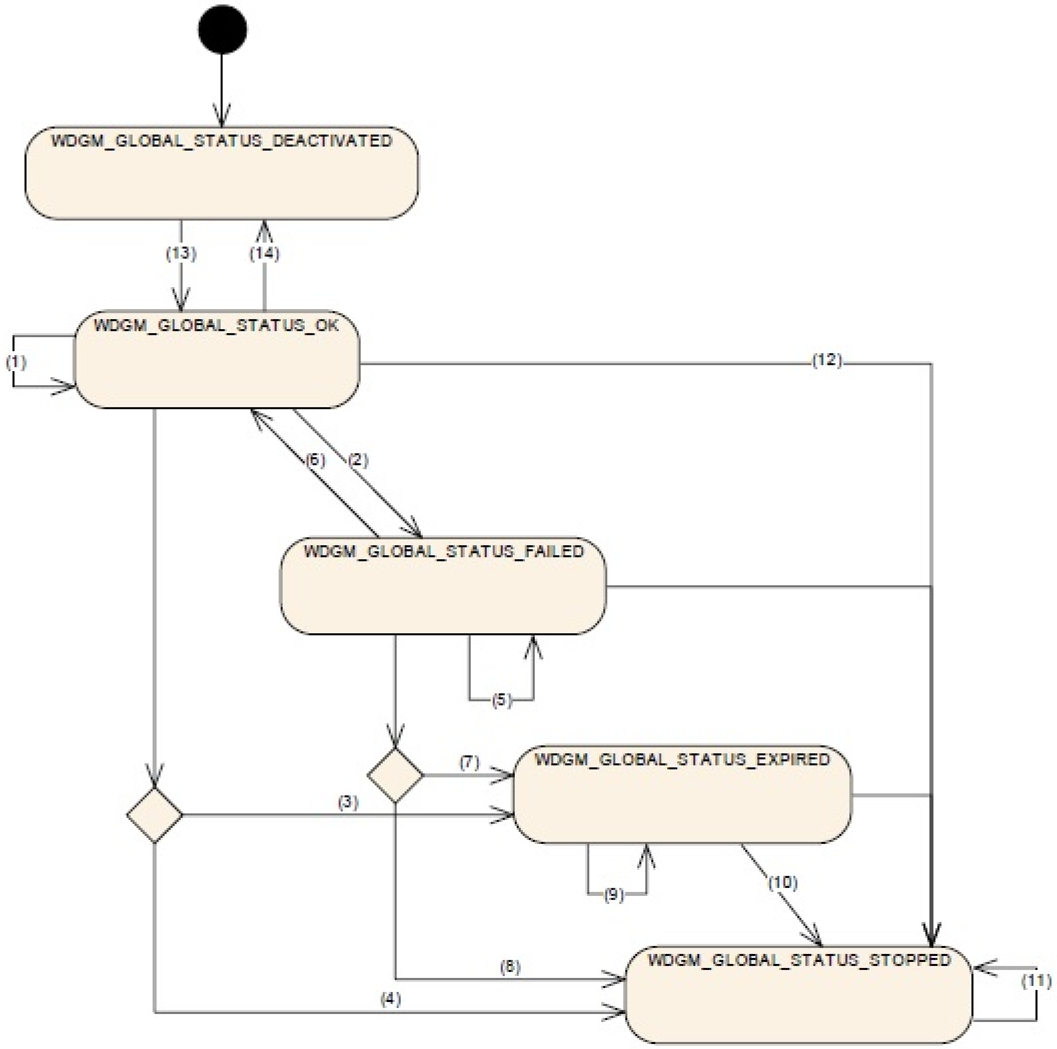

WDGM_GLOBAL_STATUS_DEACTIVATED
(13)
(14)
WDGM_GLOBAL_STATUS_OK
(1)
(12)
(6)
(2)
WDGM_GLOBAL_STATUS_FAILED
(5)
(7)
WDGM_GLOBAL_STATUS_EXPIRED
(3)
(9)
(10)
(8)
(4)
WDGM_GLOBAL_STATUS_STOPPED
(11)

Figure B.1: The possible global statuses represented as a graph

- WDGM_LOCAL_STATUS_FAILED
  Alive supervision for the supervised function has failed.

- WDGM_LOCAL_STATUS_EXPIRED
  A fault has been observed within the supervised function. The main function will save the identification of the first supervised entity which reaches this state.

Figure B.3 describes the state machine for the local status of a supervised entity.

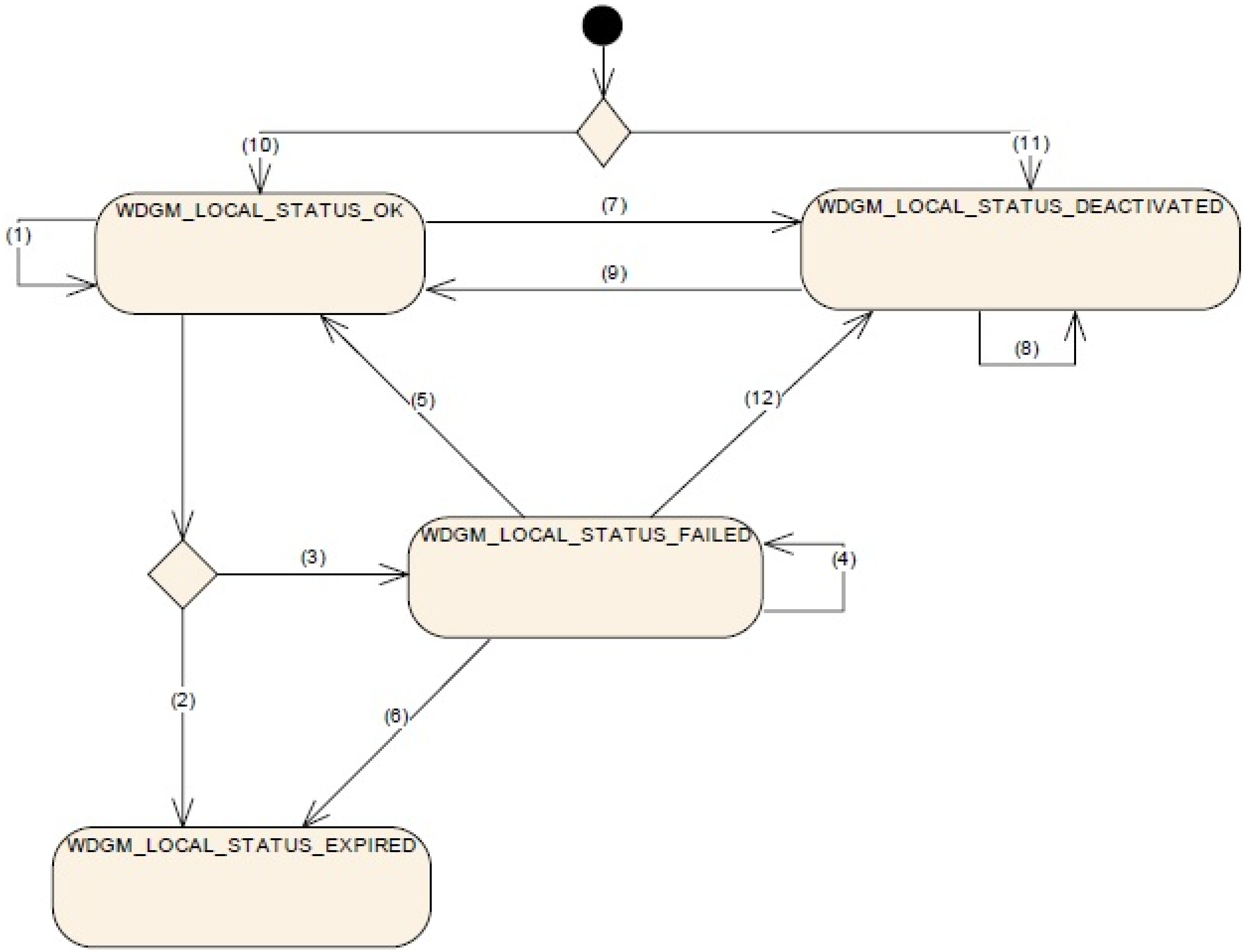


Figure B.2: The possible local statuses represented as a graph

B.4 API functions

B.4.1 WdgM_Init

Initializes the watchdog manager by setting, among other things, the local status of all supervised entities to either WDGM_LOCAL_STATUS_OK or WDGM_LOCAL_STATUS_DEACTIVATED. It also changes the global status to WDGM_GLOBAL_STATUS_OK.

B.4.2 WdgM_DeInit

Deinitializes the watchdog manger.

B.4.3 WdgM_GetVersionInfo

Returns the version info of the watchdog manager module.[1]

B.4.4 WdgM_SetMode

Sets a new mode for the watchdog manager.

B.4.5 WdgM_GetMode

Returns the current mode for the watchdog manager[1].

B.4.6 WdgM_CheckpointReached

Performs deadline and logical supervision for a given supervised entity.

B.4.7 WdgM_GetLocalStatus

Returns the local status of a supervised entity[1].

B.4.8 WdgM_GetGlobalStatus

Return the global status of the watchdog manager[1].

B.4.9 WdgM_PerformReset

Shall set the trigger condition for all configured watchdogs to zero and thereby causing the hardware watchdogs to cause an external hardware reset.

B.4.10 WdgM_GetFirstExpiredSEID.

Returns the supervised entity that first reached the state *WDGM_LOCAL_STATUS_EXPIRED*[1].

B.4.11 WdgM_MainFunction

The main function is periodically called, it first updates the local statuses by running alive supervision for the supervised entities and then sets the global status depending on the current state of the watchdog manager; this includes the new values of the local statuses.

[1] The function shall not change the internal state of the watchdog manager and should be side effect free.

Appendix C

Ambiguities in AUTOSAR

This section is used to describe some of the ambiguities we found in AUTOSAR. The highlighting refers to words or statements within the requirements that is to informal or even wrong.

C.1 Incorrect reference

C.1.1 Requirement description

[**WDGM273**] If the function WdgM_CheckpointReached determines that the result of the Logical Supervision for the given Checkpoint is true, and the Checkpoint is the initial one (WdgMInternalCheckpointInitialRef), then shall set the Activity Flag of the Graph corresponding to the Checkpoint to true . (BSW09221, BSW09222)

[**WDGM329**] If the function WdgM_CheckpointReached determines that the result of the Logical Supervision for the given Checkpoint is true, and the Checkpoint is the initial one (WdgMInternalCheckpointFinalRef), then shall set the Activity Flag of the Graph corresponding to the Checkpoint to true . ()

C.1.2 Problem description

Both requirements [**WDGM273**] and [**WDGM329**] refers to a “initial” checkpoint, but one of the requirements (preferably [**WDGM329**]) should instead refer to a “final” checkpoint. It should in that case also set the activity flag of the corresponding graph to false.

C.2 Optional or mandatory

C.2.1 Requirement description

[**WDGM344**] If development error detection for the Watchdog Manager module is enabled, then the function WdgM_GetGlobalStatus shall check whether the parameter Status is a NULL pointer (NULL_PTR). If Status is a NULL pointer, then the function shall raise the development error WDGM_E_INV_POINTER (i.e. invalid pointer) and return. ()
There are optional checks that are executed if and only if WdgMDevErrorDetect is enabled.

[**WDGM258**] If the configuration parameter WdgMDevErrorDetect [WDGM301_Conf] is enabled, the routine shall check if NULL pointers are passed for OUT parameters. In case of an error the service shall not be executed, the error shall be reported to the Development Error Tracer with the error code WDGM_E_INV_POINTER and the routine shall return the value E_NOT_OK. (BSW00323)

C.2.2 Problem description

The requirements [**WDGM344**] and [**WDGM258**] describes the same actions, with one difference: one is optional, the other mandatory.

C.3 Logical supervision results

C.3.1 Problem description

AUTOSAR does not specify if it is possible to overwrite logical supervision results from the same supervised entity.
I.e.

```
WdgM_CheckpointReached(SEx, Bad_CP)  -> incorrect result for SEx
WdgM_CheckpointReached(SEx, Good_CP) -> Correct result for SEx
```

C.4 Incorrect spelling

C.4.1 Requirement description

SWS Item	[**WDGM344_CONF**]
Name	WdgMInternallCheckpointFinalRef
Description	This is the reference to the final Checkpoint(s) for this Supervised Entity.

[**WDGM315**] If the current global status is WDGM_GLOBAL_STATUS_OK or WDGM_GLOBAL_STATUS_FAILED then for each Supervised Entity that is deactivated in the new mode (passed to function WdgM_SetMode as parameter), the function WdgM_SetMode shall change the state of the Supervised Entity to
WDGM_LOCAL_STATUS_DEACTIVATED; It shall set its Results of Active, Deadline and Logical Supervision to correct; It shall also clear its failed reference cycle counter to 0.

C.4.2 Problem description

Requirement [**WDGM344_CONF**] has miss-spelled its name *WdgMInternallCheckpointFinalRef*. This differs from references in other requirements, for example in [**WDGM329**].

C.5 Retain state

C.5.1 Requirement description

[**WDGM182**] If the current global status is WDGM_GLOBAL_STATUS_OK or WDGM_GLOBAL_STATUS_FAILED then for each Supervised Entity that is activated in the new mode (passed to function WdgM_SetMode as parameter), the function WdgM_SetMode shall retain the current state of the Supervised Entity. Switching to the mode where a Supervised Entity is deactivated clears also errors that had resulted with the WDGM_GLOBAL_STATUS_FAILED status. ()

C.5.2 Problem description

The problem is that it is unclear what the supervised entity state should contain. We know that the status, the results of alive, deadline and logical supervision and some counters should be part of this state. Should the supervision functions be part of the state?
This could be problematic because then there is a need to map out which supervision function should be retained (exists in the new mode as well as the old mode), which should be discarded (does not exist in the new mode) and which should be created (exists in the new mode but not the old).
If it is not part of the state, then all supervisions functions should be discarded and the new supervision functions should be added. This could also be problematic because, what if there exist a supervision function which has the status WDGM_INCORRECT and the only thing that keeps the supervised entity from setting the status
WDGM_LOCAL_STATUS_EXPIRED is a call to WdgM_MainFunction. Then a call to WdgM_SetMode with the same mode could reset all supervision functions and the expired state would not happen.
Another question arises; should internal logical supervision functions count? They are mode independent, but if the supervised entity is deactivated, the internal logical supervision should not be able to do anything.