CHALMERS

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

OSKAR INGEMARSSON
SEBASTIAN WEDDMARK OLSSON

Examiner: 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

{Cover:
an explanatory caption for the (possible) cover picture
with page reference to detailed information in this essay.}

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 is 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 has 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

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 [?][?].

1.1.2 Extent of software in modern vehicles

Developing a new car model costs up to one billion €, where electronics has reached a mean share of one third of the value, divided 20% sensor value, 40% hardware value and 40% software value [?][?]. The share of software has been doubled the last 10 years.

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 [?][?] [?][?].

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.

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".

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.

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 [?][?].

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 is 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.

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 [?].

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.

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. Have it been executed correctly? Are 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 be within the ISO-standards means of functional safety.

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 the state space explosion problem.

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[**?**], 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 Formal Methods and Verification

It is almost impossible to write 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. [**?**, p.272].

In the ISO standard ISO 26262, see section 2.2.1, 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 [**?**].

"**Verification** is the process of determining whether the output of a life cycle phase fulfills the requirements specified by the previous phase". 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 [**?**, 8:9.2].

2.2 Industrial Standards

2.2.1 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

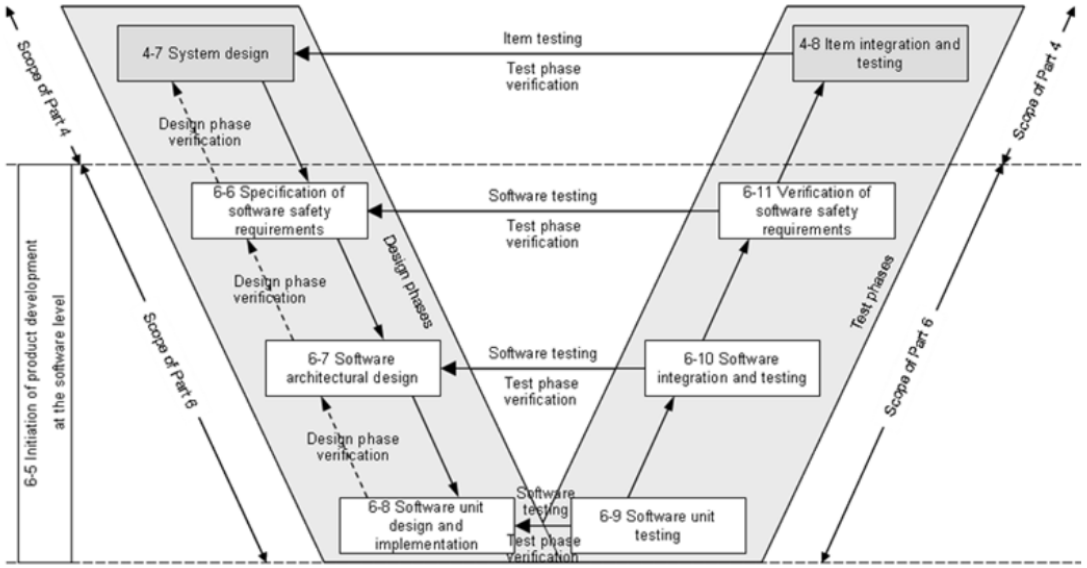


Figure 2.1: Phases of software development in the standard ISO 26262

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.

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

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 represent 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.

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 meet, is performed in the phase "verification of software safety requirements".

2.2.2 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.

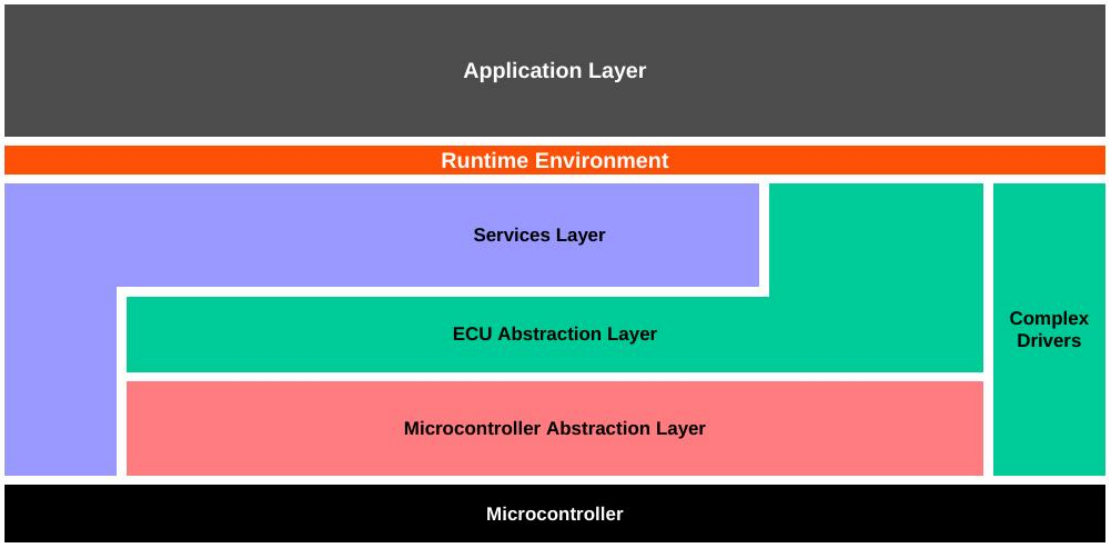


Figure 2.2: The AUTOSAR software architecture. Noticeable is that the basic software layer is divided further into four categories with even more subsections.

The runtime environment is the operating system, and the microcontroller is the hardware. The software running in the application layer is applications such as sensors and actuators. One example of how the different parts in the basic software layer is integrated, is the watchdog, which consist 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.

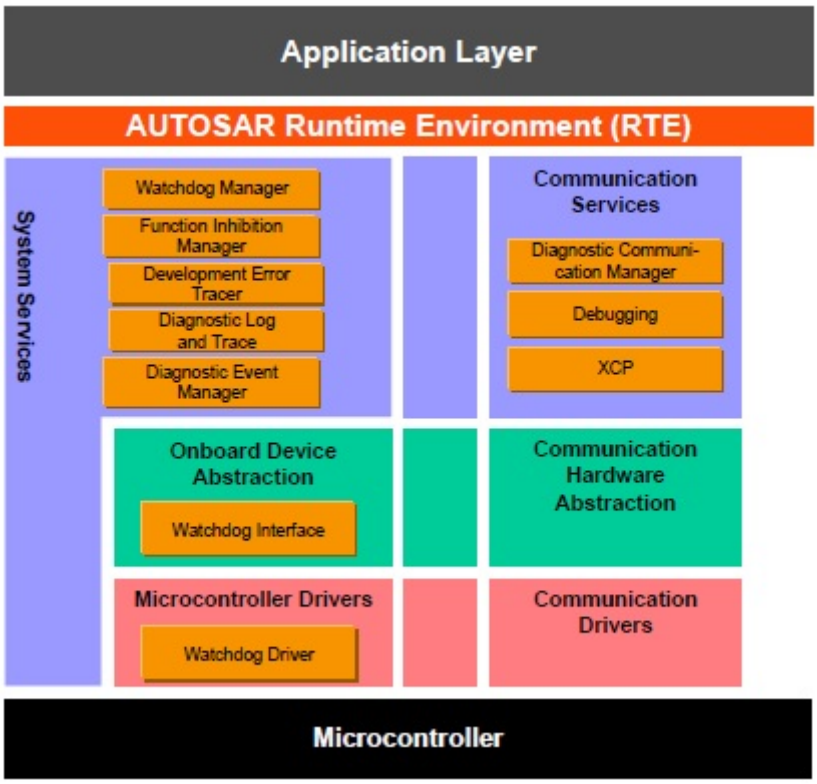


Figure 2.3: The Watchdog and some other related modules.

The watchdog have 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.

AUTOSAR's concept is to make it possible for vehicle manufacturers to buy modules 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.

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

2.3 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.

1. Finite state machines/state transition diagrams
2. Time Petri nets

IEC 61508 emphasises that Time 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:

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 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.

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."

2.3.1 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. The main difference, except from the programming language, is that the commercial version of QuickCheck has a C-testing interface. Hence it 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. It is not necessary to have the actual source code; it is enough to only have the compiled byte code and some library files of the program to test.

QuickCheck tests a program that has its specification implemented as properties that must hold for the program [**?**]. QuickCheck provides guided random test generation.

This means that the samples can be weighted to cover certain parts of the state-space with more likelihood.

Chapter 3

Method

3.1 Existing verification tools

Software unit testing can be achieved by almost any tool. The choice of verification tool is therefore not the most interesting when it comes to pure unit testing. One can of course take the simplicity to achieve good unit testing into account, but it is not what makes a verification tool unique for this projects goals.

Since the purpose is about bench marking software the phase "verification of software safety requirements" will not influence the choice. 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 implementation should be able to run on a standard PC-machine.

The most interesting part of the V-model is the phase "software integration and testing". Is there a tool that can be used to easily combine tests and requirements from different modules? Is it possible to test functional safety concept from this combination, for example by corrupting some software elements?

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

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.

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

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 we could check ambiguities in AUTOSAR in a more efficient way.

3.5 Implementation

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

The implementation of the properties was done to be able to test API calls, also described in appendix A, against the C-code. QuickCheck then checked that the post-conditions held, according to figure 3.1. The postconditions were written to test that AUTOSAR requirements held. In other words that the API calls were called correctly.

3.5.1 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.

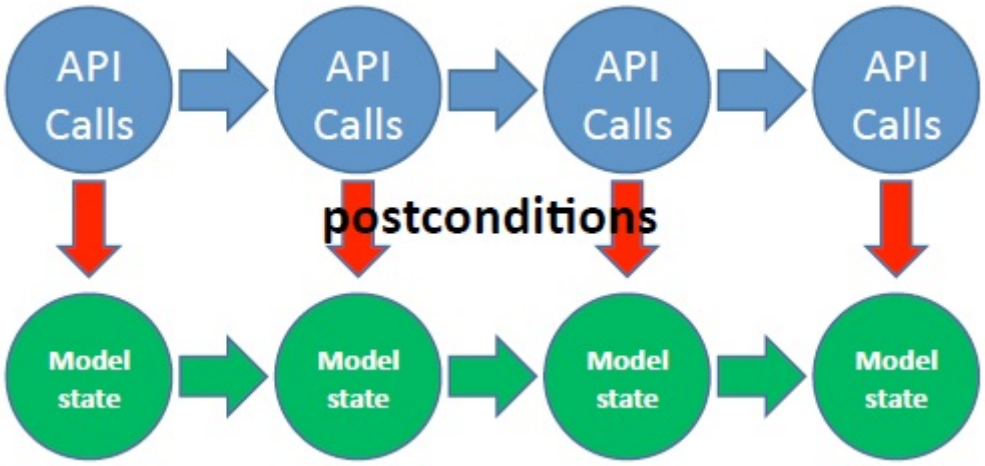


Figure 3.1: Shows Erlang modeled states with calls against the C-code

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 were room for different interpretations, which most likely would result in errors later. This is described more precise in section 3.5.4.

The translating process was done iteratively as described in section 3.5.3.

3.5.2 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. This since two different interpretation of the same specification would eventually be available.

3.5.3 Iterative strategy

The implementation of the AUTOSAR module in Erlang was done in an iterative way. Every piece of code were not 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.

3.5.4 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 written down.

Bugs in the C-code was found early, even though it already run actively in lab environments.

If a bug in the C-code was discovered how should the work be continued? Since QuickCheck terminates as soon as it finds a counter example, a method for resolving conflicts was needed to be able to run further tests. Some alternatives 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.

Item 1 was chosen because this was the most dynamic and allowed further bugs to be found, see section 4.4.

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.

3.5.5 Advantage of having the Actual C-code

When it did not exist a clear interpretation of the AUTOSAR specification, a great method for understanding, was to examine the C code. Even though QuickCheck can be used to test libraries when the source code is not available.

3.6 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 wdgm_helper
wdgm_checkpointreached	Erlang version of checkpointreached, see B.4.6
wdgm_main	Erlang version of the main function, see B.4.11
wdgm_pre	Checks for AUTOSAR preconditions
wdgm_post	Checks for AUTOSAR postconditions
wdgm_next	Defines the watchdog manager model, uses wdgm_checkpointreached, wdgm_main and wdgm_helper

3.7 Evaluation of the Implementation

If tests come back positive, it does not really say much more than that those tests evaluated to true. There is a need to evaluate what was actually tested.

3.7.1 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 those tools it was easy to see that the result was not good enough. The main problem seemed to be that the WdgM was put in an absorbing state and because of that the following commands interesting; they did not change the state. 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 3.2 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.

3.7.2 Finding better test cases

The next step was to tweak the generators that were used by QuickCheck to construct valid API-calls. This is 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 is unreasonable to test functions often when the initialization function WdgM_Init has not yet been called.

Thanks to QuickCheck's weight feature, it is simple to change the ratio of the generation of certain API calls; by matching the state and the function name of the API call, one can change the probability of generation of that call.

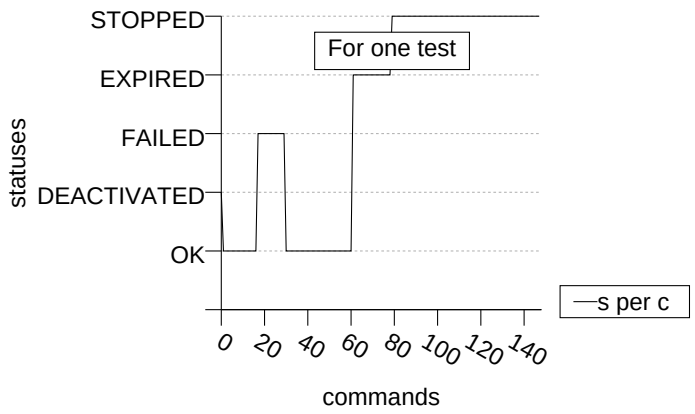


Figure 3.2: Shows changes to the global status in the execution of one QuickCheck test.

As an example we have the initialization function WdgM_Init were we want to have a high priority if it has not been called yet, and a low priority if it has been called previously.

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

It is a good idea not to lower some ratios to much, because then certain API sequences would not be generated, and bugs might be missed.

The tweaking of the generators were implemented in a 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 3.3.

A challenging step is the analysis of the results. If the testing tool returns zero errors what does that say about the robustness of the input byte code? Passed 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?

3.8 Configurations

When the code coverage was calculated it was recognized that every piece of code wasn't 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 easy

Figure 3.3: Work flow

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) Collapse irrelevant states
5. Are the results good enough, does it satisfy the requirements for the ASIL levels?
6. If not go the step 2

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 was 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.

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 is 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.

3.9 Calling the API commands

API calls were executed by QuickCheck using the *run_commands/1* function according to appendix A. The runtime 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. Accepting this solution it was easy to adopt the same approach in the Erlang module. More about this can be found in section 4.7.1.

3.10 Model State

The model state was constructed as minimal as possible. 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 when using *eqc_statem:show_states/1*, see appendix A. 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.

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.

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 to 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 4.2(b), 4.3(b), and 4.4(b) for how many commands that are executed before the absorbing state 'WDGM_GLOBAL_STATUS_EXPIRED' is reached.

4.1.3 Prioritized Testing

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 lean the results only 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 to many times can for example cause the alive supervision to fail because it goes outside of its bound. The same can happen if it is not generated enough. If a checkpoint is generated only because it needs to be inside of its alive supervision bound, then there is a chance that rules for deadline or logical supervision is violated. The easiest way to implement prioritized checkpoint generation is to start with logical supervision. This is because logical supervision follows certain graphs; transitions between checkpoints. Logical supervision maintains both internal graphs, inside of each supervised entity, and external graphs which is 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.

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 a 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 4.7.1.

4.1.5 Collapsing states

QuickCheck automatically collapse the states when a test fails. This process is called shrinking. After a command sequence have been generated and failed, QuickCheck tries to remove commands in the sequence in order to find the minimal sequence. This makes it easier to find where in the code the failure arouse from.

4.2 State Machine

The watchdog manager's state machine is shown in figure 4.1. 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 3.9.

4.3 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.

4.3.1 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 4.1. However, looking at the state transitions, as seen in figure 4.2(b) and table 4.1, only two states are visited. This happens because the configuration is to 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 it self, 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.

Figure 4.2(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 demand that a certain number of checkpoints

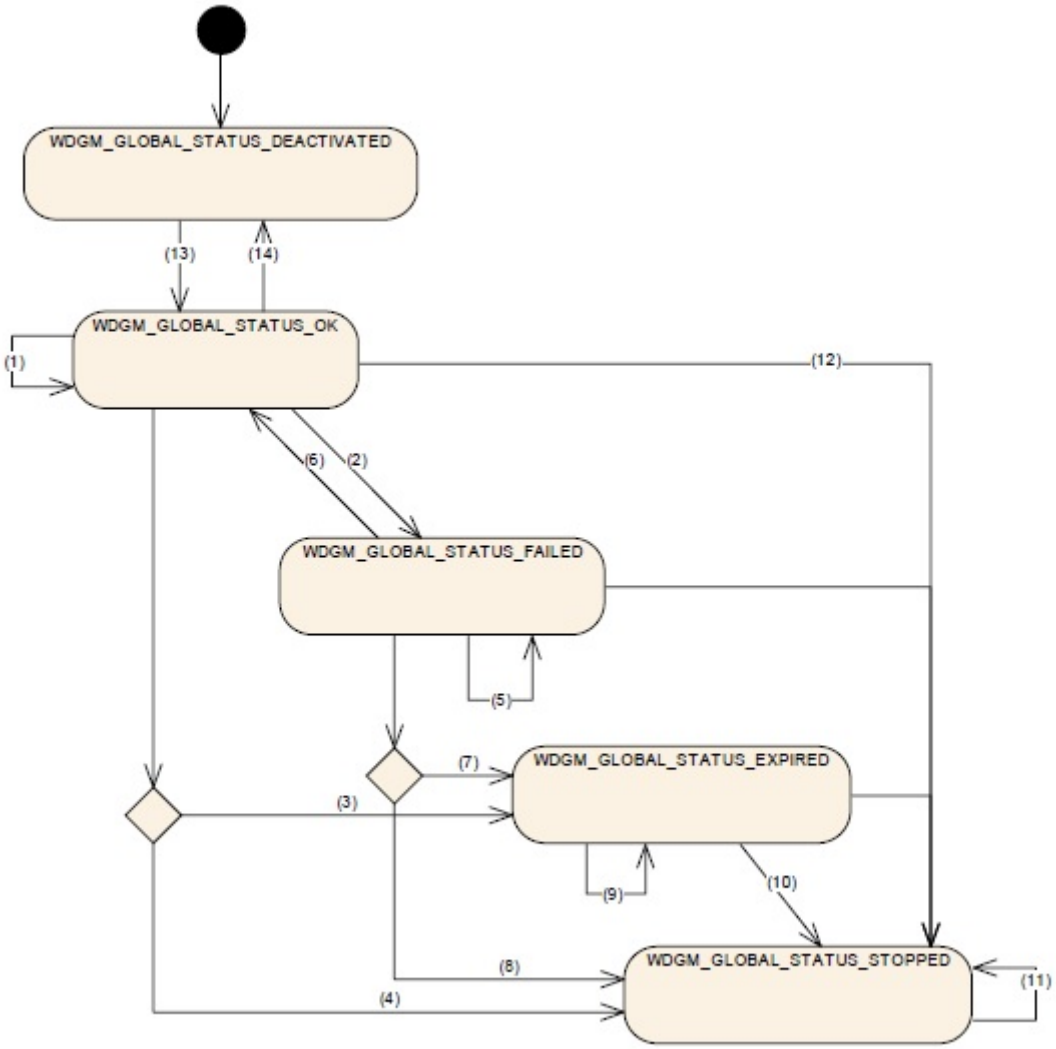
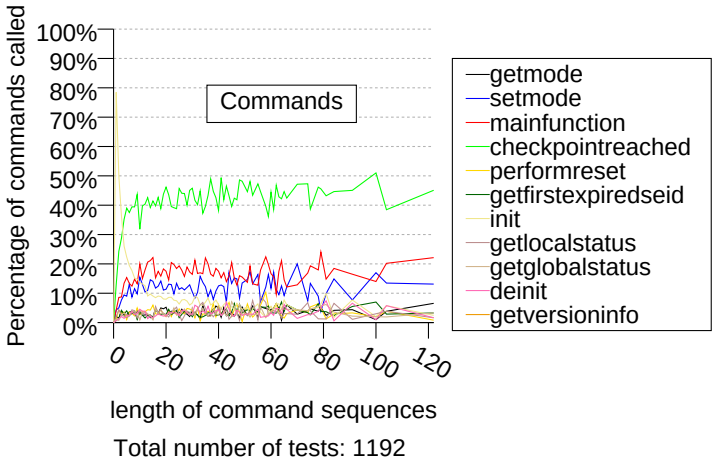


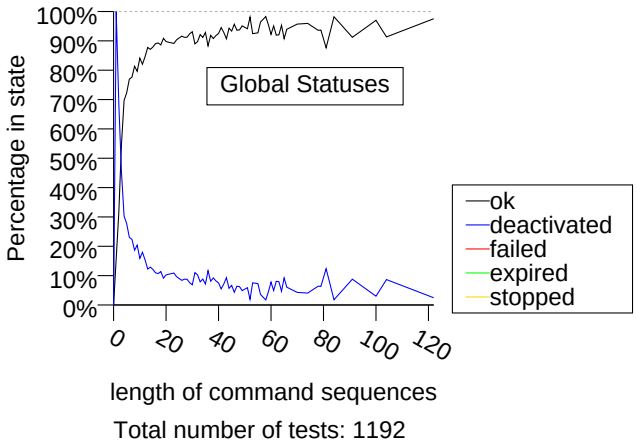
Figure 4.1: State diagram that shows possible transitions between states

is reached before the next main function is called[1]. There is also a dependency the other way around; the main function 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 stands out is 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 is ac-

[1]This is however not completely true, but it gives the general idea.



(a) Shows percentage of each possible command executed



(b) Shows percentage of each visited global status

Figure 4.2: Some statistics of the BSI minimal configuration.

tivated 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 lesser and lesser times. This function is only needed when the global state is deactivated. It has more likelihood to be generated among the first commands in the command sequence, or right after a WdgM_DeInit deactivation call.

Table 4.1: State transitions of the BSI configuration.

Number of tests: 1192					
From \ To	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%**

4.3.2 Freescale

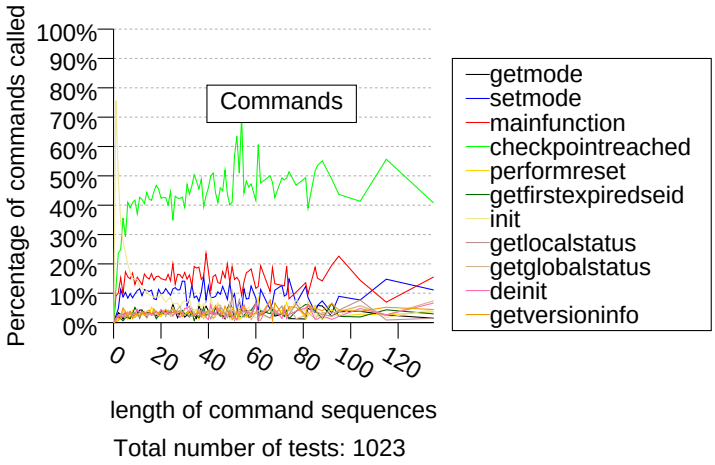
The Freescale configuration is, compared to BSI, a more realistic configuration. All supervision algorithms are configured and there are both external and internal graphs for logical supervision. It is also one of the configurations that is actively used in lab environments. The state machine for the global status is totally covered by running QuickCheck, see table 4.2 and figure 4.3(b). Locking at table 4.2 one can see that some transitions are done very seldom. This is due to 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.

Table 4.2: State transitions of the Freescale configuration.

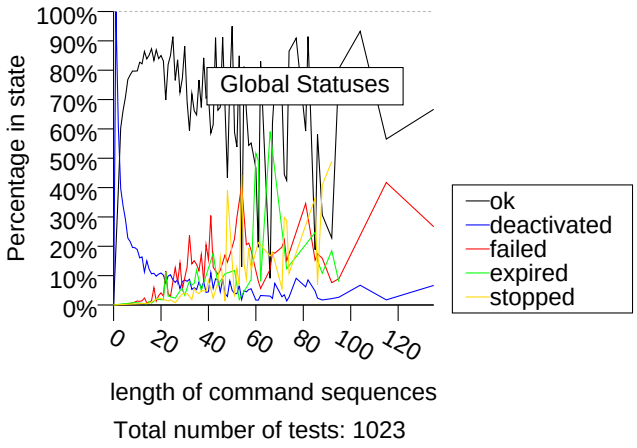
Number of tests: 1023					
From \ To	DEACTIVATED	EXPIRED	FAILED	OK	STOPPED
DEACTIVATED	**02.43%**	00.00%	00.00%	**08.32%**	00.00%
EXPIRED	00.00%	**03.36%**	00.00%	00.00%	**00.11%**
FAILED	00.00%	**00.17%**	**07.77%**	**00.12%**	**00.11%**
OK	**02.56%**	**00.18%**	**00.87%**	**69.53%**	**00.12%**
STOPPED	00.00%	00.00%	00.00%	00.00%	**04.34%**

4.3.3 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 transition is 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 4.1 is never made, see table 4.3. The reason for that is that the alive functions must fail once and then continue without failures.



(a) Shows percentage of each possible command executed

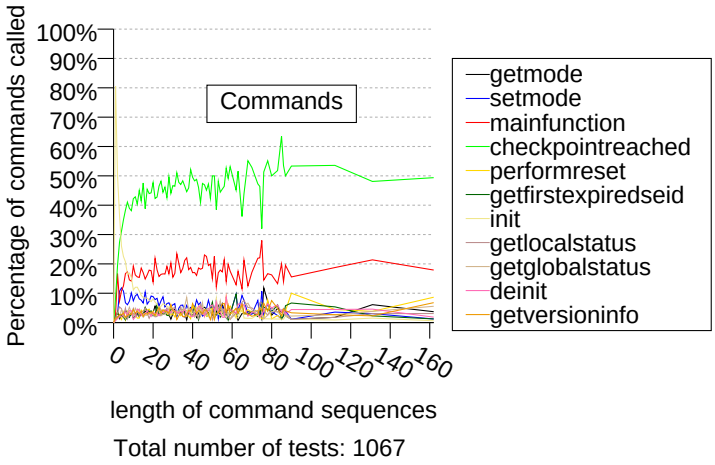


(b) Shows percentage of all visited global status

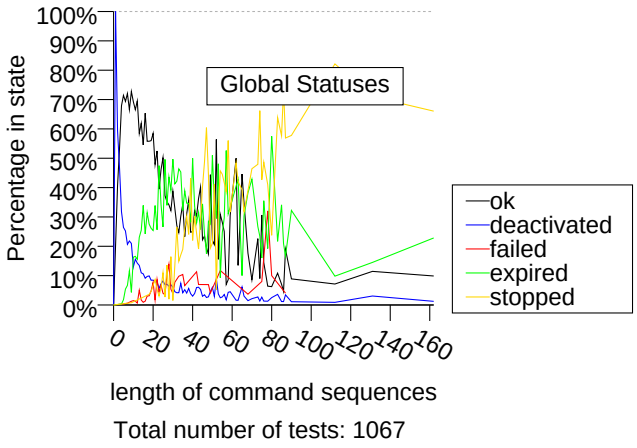
Figure 4.3: Some statistics for the Freescale configuration.

4.4 Handle bugs in the C code

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



(a) Shows percentage of each possible command executed



(b) Shows percentage of all visited global status

Figure 4.4: Some statistics for the Example configuration

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 doesn't come that often. Another way is to mock the faulty API call. In other words simulate the output of the C-code, but then you will only find one bug per function, strictly limiting the probability to find bugs.

Table 4.3: State transitions of the Example configuration

Number of tests: 1067					
From \ To	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%**

There is a QuickCheck Erlang module for the purpose of mocking C-code, see appendix A.1.6. Then there is 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.

We choose to correct the C-code, because then we had direct feedback and could discover where in the code the bugs were introduced.

4.5 Statistics

The distribution of API-calls seems, according to figure 4.2(a), 4.3(a) and 4.4(a), to be the same for all configuration. The arguments to the API calls is however different, even though it is not seen in those plots.

4.5.1 Important Functions

The most interesting API calls is 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 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 4.1. If this happens they will change the internal state of the watchdog independently of previous called commands. The behavior of these commands will therefor 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 watchdog 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 4.2(a), 4.3(a) and 4.4(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_CheckpointReach 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 3.9 and 4.7.1, we implemented the model as the C-code, with the use of WdgM_MainFunction. This is possible because we know that WdgM_MainFunction is called periodically.

4.6 Coverage

4.6.1 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 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.

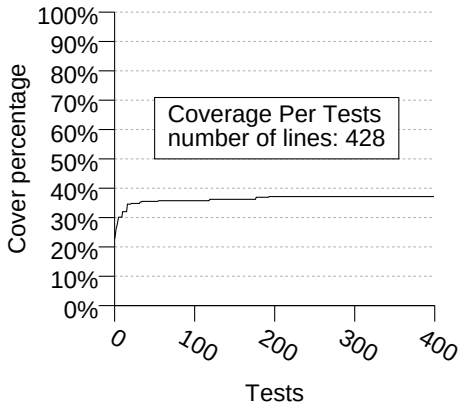


Figure 4.5: Coverage per tests using the BSI configuration

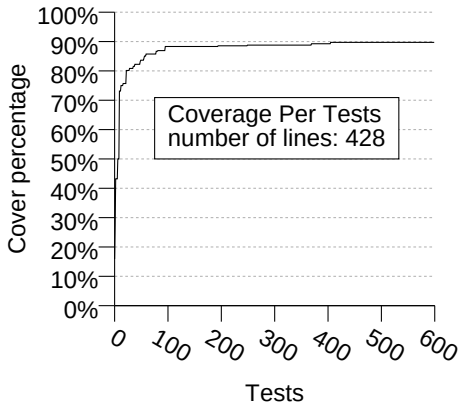


Figure 4.6: Coverage per tests using the Freescale configuration

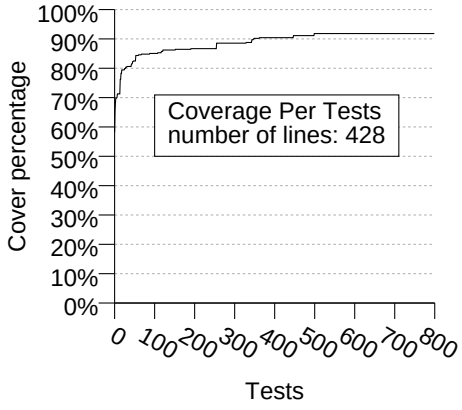


Figure 4.7: Coverage per tests using the Example configuration

As can be seen in the figures 4.5, 4.6 and 4.7, the example configuration takes the longest time before it converges. It also becomes clear that the freescale configuration needs more tests than the BSI configuration before it convergence. 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.

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

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%

The total line coverage results is 97.38%. The reason we don't achieve 100% code coverage depends on certain limitations. 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 any way. There are also a number of implementation specific lines, which an other C code model might reach but the one that we had. There is 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.

4.6.2 C-code

Bullseye Coverage was used to analyze the coverage of the C-code. The results show that the decision coverage is over 81% and all functions except for two is covered, see figure 4.8. 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 limitations were to not implemented 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 4.8 is constructed using three configuration.

Using several configurations gave better results since some code is impossible to reach if not certain configuration parameters are set.

Name	Function coverage	Uncovered functions	Condition/decision coverage	Uncovered conditions/decisions
../modules/c/WdgM/src/WdgM.c	93%	2	80%	82
WdgM_UpdateAliveCounter(WdgM_SupervisedEntityIdType)	0%	1	0%	10
WdgM_IsOneAliveConfig(WdgM_SupervisedEntityIdType)	0%	1	0%	6
WdgM_DeadlineSupervision(WdgM_ModeType,uint16)	100%	0	41%	7
WdgM_CalculateAliveSupervision(WdgM_ModeType,WdgM_SupervisedEntityIdType)	100%	0	60%	11
WdgM_CheckpointReached(WdgM_SupervisedEntityIdType,WdgM_CheckpointIdType)	100%	0	75%	4
WdgM_ValidInitialMode(const WdgM_ConfigType*)	100%	0	75%	2
WdgM_GetLocalStatus(WdgM_SupervisedEntityIdType,WdgM_LocalStatusType*)	100%	0	75%	1
WdgM_IsValidCheckpointId(WdgM_SupervisedEntityIdType,WdgM_CheckpointIdType)	100%	0	78%	3
WdgM_CheckLogicalSupervisedEntities(WdgM_SupervisedEntityIdType,WdgM_CheckpointIdType)	100%	0	80%	14
WdgM_ResetSupervisedEntity(WdgM_ModeType)	100%	0	80%	4
WdgM_UpdateCheckpointAliveCounter(WdgM_CheckpointIdType)	100%	0	80%	2
WdgM_MonitorActiveEntity(WdgM_ModeType,uint16)	100%	0	82%	5
WdgM_PerformReset(void)	100%	0	83%	1
WdgM_SwitchWdgMode(WdgM_ModeType)	100%	0	83%	1
WdgM_TriggerWatchdogs(void)	100%	0	85%	2
WdgM_SetMode(WdgM_ModeType,WdgM_CallerIdType)	100%	0	86%	3
WdgM_CheckLogicalSupervisonGraph(WdgM_CheckpointIdType)	100%	0	87%	5
WdgM_Init(const WdgM_ConfigType*)	100%	0	95%	1
WdgM_CalculeteGlobalStatus(void)	100%	0	100%	0
WdgM_DeInit(void)	100%	0	100%	0
WdgM_GetFirstExpiredSEID(WdgM_SupervisedEntityIdType*)	100%	0	100%	0
WdgM_GetGlobalStatus(WdgM_GlobalStatusType*)	100%	0	100%	0
WdgM_GetMode(WdgM_ModeType*)	100%	0	100%	0
WdgM_GetVersionInfo(Std_VersionInfoType*)	100%	0	100%	0
WdgM_IsSEActivated(WdgM_SupervisedEntityIdType)	100%	0	100%	0
WdgM_IsValidCallerId(WdgM_CallerIdType)	100%	0	100%	0
WdgM_IsValidMode(WdgM_ModeType)	100%	0	100%	0
WdgM_IsValidSEId(WdgM_SupervisedEntityIdType)	100%	0	100%	0
WdgM_MainFunction(void)	100%	0	100%	0
WdgM_SetGlobalStatus(WdgM_GlobalStatusType)	100%	0	100%	0
WdgM_ValidModeSwitch(WdgM_ModeType)	100%	0	100%	0

Figure 4.8: Shows coverage statistics generated by Bullseye Coverage

4.7 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 therefor hard to analyze code that is written without the standard in consideration and then examine if it fulfills that standards requirements.

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. There must first be a number of assumptions. 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.

4.7.1 AUTOSAR

Due to the informal syntax of AUTOSAR it is not good 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 but, in a functional safety point of view, 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, and then an other model was later 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 are 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 of WdgM_MainFunction tick a "time" counter.

4.7.2 Fulfilled ISO 26262 requirements

ISO 26262 mention several requirements that QuickCheck will be able to fulfill. Aside from general requirements like that a "safety plan" should be available, there are tests in which the hardware should be taken in consideration. This is beyond the scope of this thesis and can also not be tested using QuickCheck. 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.

4.7.3 Confidence interval

4.7.4 Measurements of the state space

One way when trying the measure the state space is to collect statistical data during the execution of the tests. It is hard to say much about the whole systems state space since it is very large, due to the state 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 are 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 state 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 alot of negative testing, because we didn't care about tweaking the generators. 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, because then you can really check the code and compare with the requirements.

We achieved fair coverage of the c-code, around 90%, and the Erlang code, around %. The problem was the requirements where we achieved around 50%. It would help if an 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

Some interesting thing we wanted to do from the beginning was to implementing another module, preferably one that has some kind of dependency with the watchdog manager. This is because then, it would be possible to testing 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 is not possible to achieve with only a software testing tool. For example requirements that verifies the hardware specifications or how the safety plan should be made. It is hard to use the ISO 26262 V reference 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 can't be used to verify the software units. This means one must translate the AUTOSAR requirements to formal syntax and verify that the formal requirements means 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 outputted 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 therefor 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.

Appendix A

Introductions to QuickCheck

Lets say that one created a *sort* function that takes a list Xs of any sortable items and returns the sorted version Ys of Xs. Then one can specify certain properties that must hold for to the 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 contains 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.

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.

A property in QuickCheck is written in the following way.

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

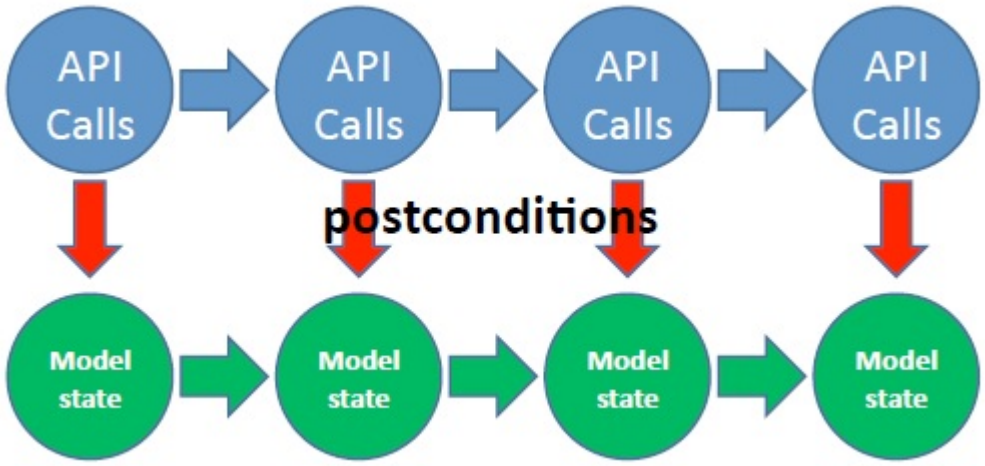


Figure A.1: Shows state based testing against API and model state

```
command(S) ->
      [{call, Mod, CFunction, arg_generator(S)},
        ... ].
```

The function *commands()* is a generator and looks in the module after a function called *command(S)* which takes a state and returns a list of functions that one want to test.

Every command has, additional to the command definition, a precondition, postcondition and a next state function. For more information about this, see section A.1.4.

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

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

One can 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.1 QuickCheck Modules

The commercial version QuickCheck consist of several Erlang modules.

A.1.1 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.1.2 eqc_gen

The module is used for generation 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.1.3 eqc_c

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

```
eqc_c:start(suit, [{c_src, "suit_api.h"},
                   {additional_files, ["myprog.o"]}])
```

The code above starts the C-program *myprog.o*, and an Erlang module is created with the name of the first parameter, *suit*. This module can now be used within Erlang to get information about the C-program. For example the following function, that counts the number of spaces in a string, might be declared in *suit_api.h* and defined in the *myprog.o*.

```
extern int spaces(char *str)
```

Then we can make the following call in the Erlang:

```
suit:spaces("this is a string")
```

If the function *spaces* is defined correctly the output will be 3.

A.1.4 eqc_statem

Offers state based testing. A command has a definition, precondition, postcondition, and a next function. If we wanted to test *spaces* function one may write:

```
% Defines the initial model state
initial_state() ->
```

```
    [].

% Takes the model state as a parameter
% Returns a tuple of the format
%   {call,Module,Function_to_call,Arguments_to_function}
spaces_command(_State) ->
  % Generates a natural number, binds it to Pat and generates a
  % list with length Pat with elements of type char
  Xs = ?LET(Pat,nat(),vector(Pat,char())),
  {call,?MODULE,spaces,[Xs]}.

% The actual call to the C-code
spaces(Xs) ->
  suite:spaces(eqc_c:create_string(Xs)).

%Takes the model state as a parameter
%The spaces function has no preconditions
spaces_pre(_State) ->
  true.

%Pramaters: S = ModelState, Args = arguments to the command
%Defines what should hold after the command
spaces_post(_S,Args=[Arg],Ret) ->
  length(lists:filter(fun(X) -> X == 32 end,Arg)) == Ret.

%Parameters: S    = model sate before the call,
%            Ret  = return value from command,
%            Args = argements to command
%Return the new model sate after the command is called
spaces_next(S,_Ret,_Args) ->
  S.
```

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 behaviour.

The example above is actually not state dependent since *spaces* only depend on the argument string. However one can imagine the same function being dependent on some global variable and according to that variable change its output.

A.1.5 car_xml

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

A.1.6 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 decide 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.

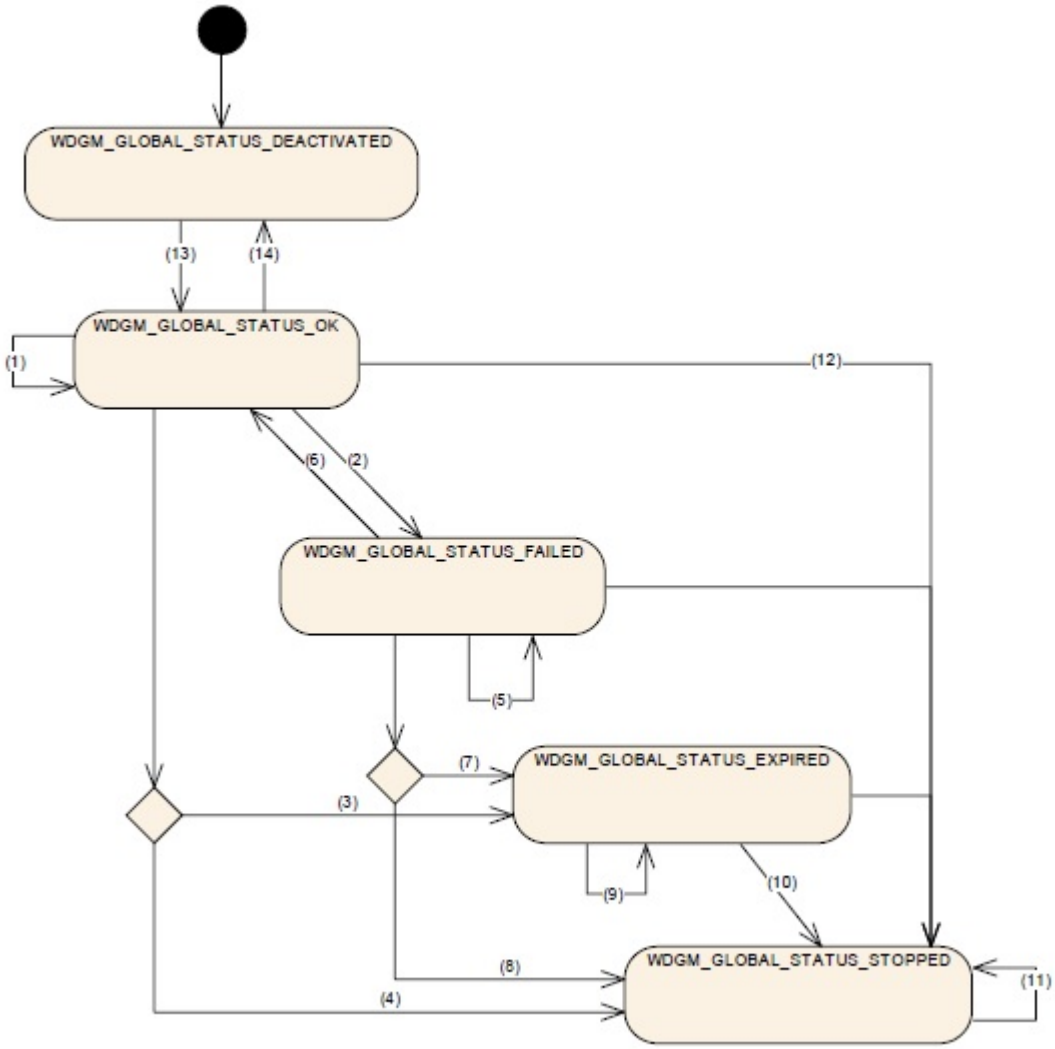


Figure B.1: The possible global statuses represented as a graph

- 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 reach this state.

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

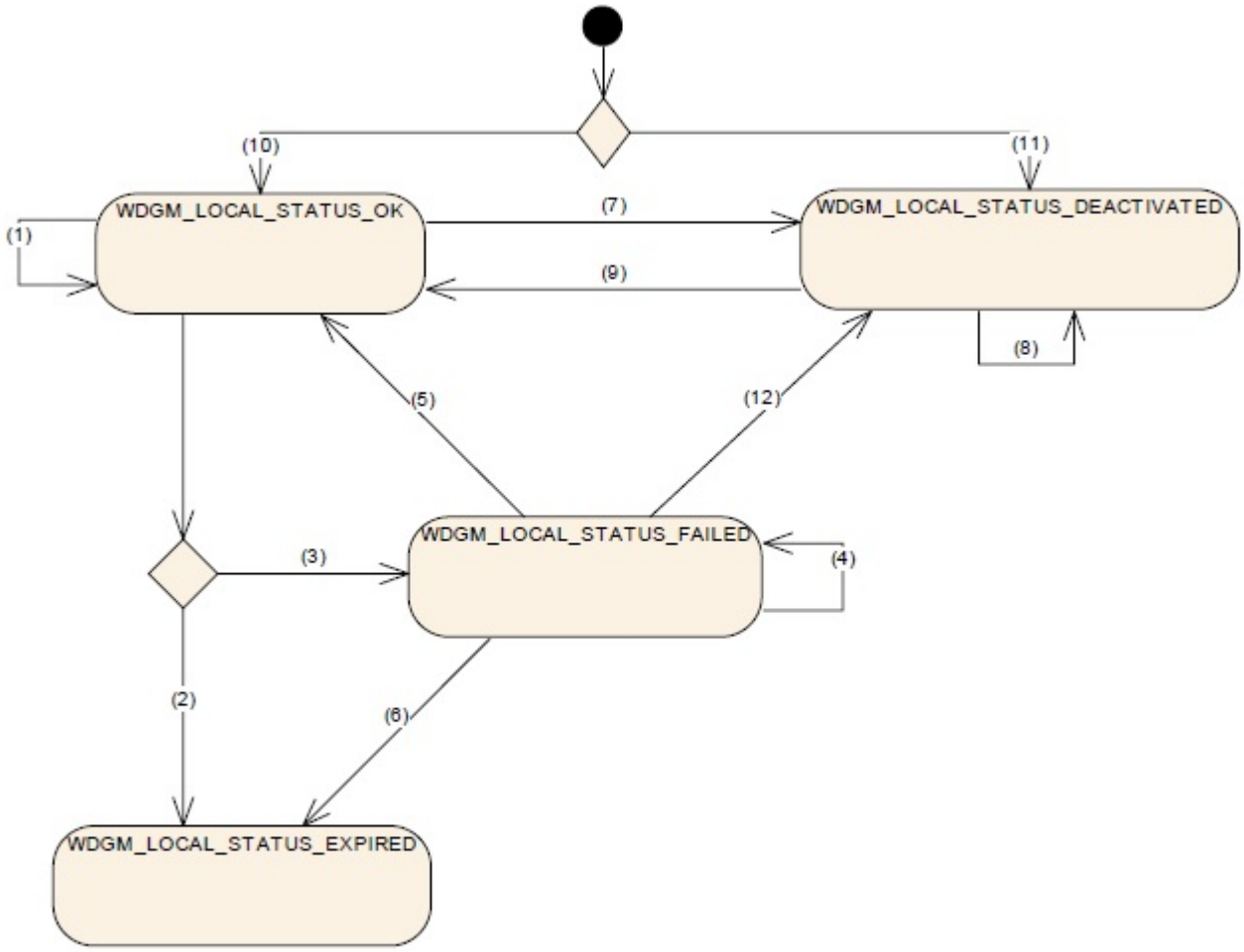


Figure B.2: The possible local statuses represented as a graph

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_DEACTIVATE
It also changes the global status to WDGM_GLOBAL_STATUS_OK.

B.4.2 WdgM_DeInit

De initializes 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

Bugs in C-code

Appendix D

Ambiguities in AUTOSAR

Bad transition

[WDGM285] claims that "if WdgM_Init was successfully called, change global supervision status to WDGM_GLOBAL_STATUS_OK".

[WDGM139] claims that "if a call to WdgIf_SetMode fails, the function shall assume a global supervision failure and set the global supervision status to WDGM_GLOBAL_STATUS_STOPPED ", with a reference to the transition between WDGM_GLOBAL_STATUS_OK and stopped.

What if it were not successfully called?

If WdgIf_SetMode is not successfully called, how can WdgM_Init be successfully, and get the global status WDGM_GLOBAL_STATUS_OK?

Incorrect reference

[WDGM329] could be the same as [WDGM273] The only difference is a parenthesis referencing to Initial/Final?

Optional or mandatory

[WDGM344] and [WDGM258] is the same, with one difference: one is optional, the other mandatory.

Logical supervision results

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
```

Wrong spelling

[WDGM344_CONF] WdgMInternallCheckpointFinalRef has incorrect spelling.

[WDGM315] "set its Results of Active, Deadline and Logical Supervision", what is "Active supervision"? Should be alive supervision.

Retain state

[WDGM182] if WdgM_SetMode is called, and a supervised entity which was activated in the previous mode, also is active in the new mode, then retain the state for the supervised entity.

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.