CHALMERS

Evaluation of validity of verification methods

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

Abstract is often proportional to the thesis length.

Keywords
AUTOSAR, Semi formal verification, QuickCheck

Contents

Chapter 1

Introduction

In the first sections of this chapter we outline the background, purpose and objectives and the last section describes the scope of our research. Chapter 2 contains the theory needed for understanding the thesis, existing software and some verification methods described in the standards. Methods is described in Chapter 3 and the results in Chapter 4. In Chapter 5 and 6 we have a discussion and some conclusions about the results.

1.1 Background

The last decade of the 20th century resulted in a dramatic growth of technology(?). The rate is still increasing with up to 90% of all innovations being realized through electronics in the beginning of the 21st century (?). The software-based systems in motor vehicles become more complex and are moving toward handling more critical functions (?). Vehicles have already begun to communicate with each other [1] and it is soon expected that roadside traffic management systems will also interact with the systems in vehicles[2].

While the systems become more complex the software must be fault-tolerant and safe. Testing, validation and verification is a need and should follow all product development phases, from start to finish. The problem is that testing is both time consuming and labour intensive, accounting for up to 50 % of the development cost[3]. Unit tests adds additional complexity to the code. It is very important to test and verify all steps of the development.

The complexity issue is that in systems such as microprocessors the number of possible failure modes is so large it is considered infinite[4]. This makes it impossible exhaustively test the system and therefor also make the detection of failures unreliable.

Quviq QuickCheck has the ability to automate this process and allow the developer to write properties instead of tests. This makes it possible for QuickCheck to create arbitrary input vectors that can be feed to the code. Figure 1.1 shows the different steps of software development and the test and verification phases it has.

The first step is the system design. At this point the system specification is written and then the software specifications. When all specifications exists, the next phase is to

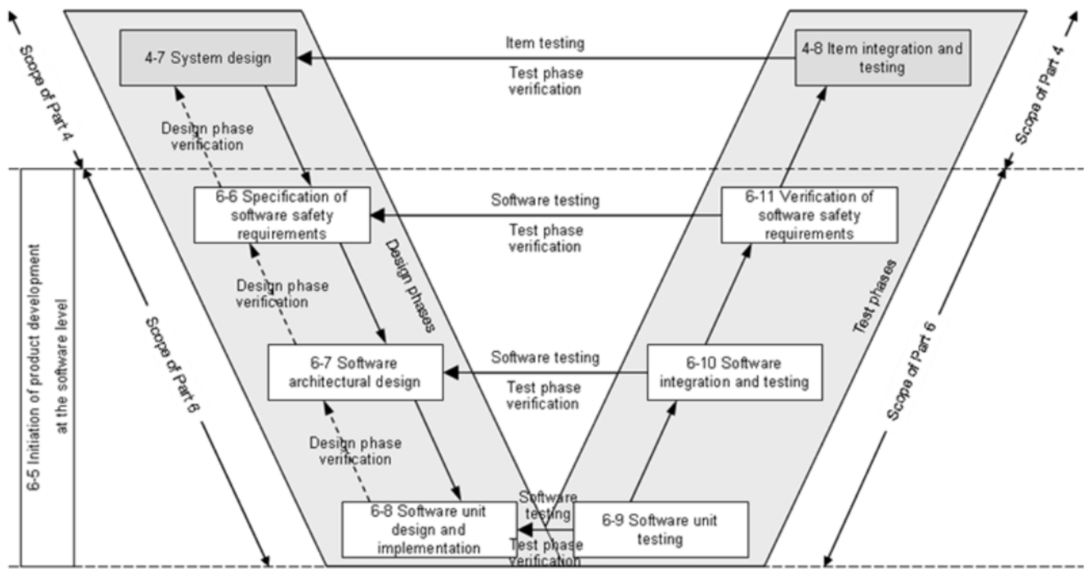
4-7 System design
Item testing
Test phase verification
4-8 Item integration and testing
Design phase verification
6-6 Specification of software safety requirements
Software testing
Test phase verification
6-11 Verification of software safety requirements
Design phase verification
6-7 Software architectural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

Figure 1.1: Phase model for software development

design the software architecture. The last part of the design is software unit design and implementation. All phases must be a verification of the former phase.

When the implementation is done, it is time for the test phases. These begins with software unit testing which tests the software unit design phase. If the tests verifies that the software unit design and implementation is correct, the test phase moves on to the architectural design and then to verification of software safety requirements. The last test phase verifies that the system is designed according to the specification.

1.2 Purpose

The purpose is to automate the testing process in an effective and good way preferably using QuickCheck. To make it possible to raise the Automotive Safety Integration Level (ASIL), in the software unit design and implementation phase and in the software architectural design phase, a check must be done to see if it exist a tool that can be used in order to perform a semi formal verification of a module, and then make this process generalized for modules in coexistence.

The purpose is also to be able to decrease the number of needed unit tests in the software unit design and implementation phase. Even further is the goal to introduce semi formal verification of the software architectural design phase. Functional safety concept will be most important in this phase.

1.3 Objective

Propose and motivate what should be done to be able to achieve a semi-formal verification. This should include a confidence interval for how certain the verification is.

Prove that it is possible to do a semi-formal verification for an AUTOSAR module and its specification, preferably using QuickCheck. This should be generalized so its possible to test other specifications and modules at a later moment. Also it should not matter which configuration that is active, because the specification holds for all configurations.

1.4 Scope

We will use AUTOSAR 4.0 revision 3 for our thesis work. Since AUTOSAR consists of around 80 specifications and other auxiliary materials[5], we will limit our scope to one or two specifications. The main module of this thesis is the CryptoServiceManager. This module provides cryptographic functionalities for synchronous and asynchronous services. This module is chosen because it only got a few dependencies, is used to trace development and production errors but also to incorporate cryptographic libraries.

It is hard to test that a call to another module gives the right results, that is why we have chosen a module with a small number of dependencies. The CryptoServiceManager should also have functionality which gives the same results no matter which state it is in, such as hash functions[6].

Also 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.

Chapter 2

Theory

2.1 Verification

The exact meaning of verification is confusing [7]. The definition may differ in comparison of academic or industrial sense. Even in different phases of the safety lifecycle verification is conducted in various forms [8, Chapter 9.2, Part 8].

2.1.1 Formal Verification

2.1.2 Semi foramal verification

2.2 Industrial Safety Standards

It is useful to distinguish between systems with different levels of dependability and determine where the hazards exists[4]. When this risk analysis is done and appropriate reliability and availability requirements is assigned to the system, the system can be identified by a certain safety integrity level (SIL). 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 is more defined in the standards IEC 61508 and ISO 26262.

2.2.1 IEC 61508 and ISO 26262

Automotive software for safety related systems is required to be designed, implemented and verified by the standard ISO 26262 that handles functional safety for automotive equipment. For a higher level of integrity ISO 26262 strongly recommends that a semi-formal verification of each module should exist [8, Part 6]. It also recommends a formal verification, but because of the state-space complexity this is hard to achieve.

ISO 26262 is built on IEC 61508 which is titled Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems which can be applied to any kind of industry. IEC 61508 have four safety integrity levels (SIL) ranked 1-4. SIL 4 is the highest and should be applied where a failure can do devastating damage to a large area. The automotive industry is improbable to have this risk. That is why ISO 26262 exists, it also has four levels of

SIL called automotive safety integrity level (ASIL). The ASIL range from A-D, where D is the highest and roughly translated to SIL 3.

2.2.2 AUTOSAR (AUTomotive Open System ARchitecture)

AUTOSAR is a worldwide development partnership of car manufacturers, suppliers and other companies from the electronics, semiconductor and software industry.

Some of the specifications in AUTOSAR is left quite open for interpretation. This makes it possible for vehicle developers to have different specifications for a configuration. Some parts of those configuration specifications is generated into code, while other is manually written or added as a configurable.

2.3 Existing Software

There are a number of existing software tools whose aim are to simplify, for a developer, the implementation of verification and testing.

- QuickCheck
- SPIN
- McErlang
- μCRL toolset
- CADP
- Parasoft C/C++test

2.3.1 QuickCheck

QuickCheck tests a program with specification implemented as properties that the program must hold [9]. QuickCheck has guided random test generation. This means that the samples can be weighted to cover certain parts of the state-space with more likelihood.

2.3.2 Erlang

Erlang can easily communicate with other programming language by using byte streams. There have been some work done including Erlang, AUTOSAR and QuickCheck, mostly by the QuviQ company[10]. Imperative coding requires state based testing(?). There are a library in Erlang developed by QuviQ for this purpose(?).

2.3.3 SPIN

SPIN is used to trace logical design errors in distributed software[11]. It supports a high level language, Promela, to specify system descriptions. Promela stands 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. Even further it supports random, interactive and guided simulation, with both partial or exhaustive proof techniques.

2.3.4 McErlang

McErlang is a model checker written in Erlang used for verifying distributed Erlang programs[12]. Its purpose is to check a program against a correctness property, but can also among other things check safety or liveness properties.

McErlang offers two depth-first state traversal model checking algorithms, one checks safety properties and the other is used for checking the liveness. McErlang also implements weak process fairness, by omitting non-fair loops from the accepting runs in its liveness algorithm.

2.3.5 μCRL toolset

μCRL is a process algebraic language which is suited for the analysis of distributed systems. The toolkit is built on this language and contains a theorem prover based on binary decision diagrams[13].

2.3.6 CADP

Construction and Analysis of Distributed Processes (CADP) is a toolset for design of distributed systems[14]. It includes, among others, tools for equivalence checking, state space manipulation and model checking and also includes several verification algorithms. It provides functionality as step-by-step simulation to parallel model-checking.

2.3.7 Parasoft

2.4 Verification Methods

The standard 61508-7 © IEC:2000 propose to two methods to say that a program is formal verified. The key is to model the program into one of the following machines.

1. Finite state machines/state transition diagrams

2. Time Petri nets

The standard emphasises that Time Petri nets are best suited for concurrent programs. Regarding method 1 the following criteria needs to be satisfied for the implemented state machine to be formal verified.

1. completeness (the system must have an action and new state for every input in every state);
2. consistency (only one state change is described for each state/input pair); and
3. reachability (whether or not it is possible to get from one state to another by any sequence of inputs).

If the state machine is correct implemented, hence is a correct model of the original program, this also says that the original program is formal verified.

Since most program specification are written in natural languages there may be lots of ambiguities. Therefor a lots of techniques has been developed to reduce such cases. This techniques are often referred as semi formal verification, because they lack of mathematical rigour associated with formal verification. This methods use textual, graphical or other notation; often several techniques are used.

The description of semi formal verification in the 61508-7 © IEC:2000 standard 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 analysed by machine or animated to display various aspects of the system behaviour.”

Chapter 3

Method

3.1 Choose of tool for verification

Software unit testing can achieved by almost any tool. This since its just more or less about unit testing. This phase is hence not the most interesting when it comes to the choose of a tool verification. Of course one can take the simplicity to achieve good unit testing into account but still it is not what makes a verification tool especially unique for the project goals.

Since the purposes is about benchmarking the software the phase ”verification of software safety requirements” will not influence the choose. For be able to test this phase a greater amount of components of the hole system must be available. Such components may include hardware etcetera. Implementation wise should everything be able to run on a standard PC-machine.

The most interesting part is the software integration and testing. Is there a tool that one can use to easily combine test and requirements from different modules? Is it possible to test functional safety concept from this combination, for example corrupting some software elements?

3.1.1 Why QuickCheck and Erlang?

3.2 Specification

Specifications for what each module should do in AUTOSAR is given in text form. Hence one must first, before a module can be tested, implement the specification for that module in code.

3.3 Testing

Properties for 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 an state machine based testing tool.

- Choose a specification which will be translated to QuickCheck properties in parts.
- With the use of statistics and confidence intervals, show that, with enough tests the state-space will be exhausted.
- Evaluate other semi-formal techniques and show that the results from them shows that QuickCheck is reliable for verification.
- Generalize the technique.

3.4 Implementation

A challenging step is the analysing part. 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 interval on the correctness of the code?

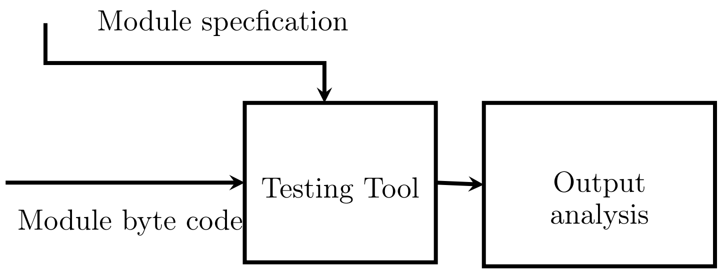

Figure 3.1: Abstract implementation module

Chapter 4

Result

Chapter 5

Discussion

Chapter 6

Conclusion

Bibliography

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

[2] Strandén L, Ström E, Uhlemann E. Wireless Communications Vehicle-to-Vehicle and Vehicle-to-Infrastructure. SP, Chalmers and Volvo Technology; 2008.

[3] Claessen K, Hughes J. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs; 2000.

[4] Storey N. Safety-Critical Computer Systems. ISBN 978-0-201-42787-5. Prentice-Hall; 1996.

[5] AUTOSAR. AUTOSAR; 2013. `http://autosar.org/index.php?p=3&up=2`.

[6] AUTOSAR. Specification of Crypto Service Manager;. Document Identification Number 402.

[7] Arts T; 2013. personal communication.

[8] ISO 26262. Road vehicles - Functional safety. ISO, Geneva, Switzerland; 2011.

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

[10] QuviQ. QuickCheck; 2013. `http://www.quviq.com/documents/QuviqFlyer.pdf`.

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

[12] Fredlund LÅ, Svensson H. McErlang: A Model checker for a Distributed Functional Programming Language. Universidad Politécnica de Madrid, Chalmers University of Technology; 2007.

[13] Grote JF, Lisser B. μCRL toolset. Kruislaan 413, Amsterdam, 1098 SJ (NL);. `http://homepages.cwi.nl/~mcrl/mutool.html`.

[14] CONVECS. Construction and Analysis of Distributed Processes;. `http://cadp.inria.fr/`.