CHALMERS

Evaluation of validity of verification methods

Master of Science Thesis
Project planning report

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

Contents

Chapter 1

Introduction

1.1 Background

Testing is time consuming and labour intensive, accounting for up to 50 % of the development cost[5]. Unit tests adds additional complexity to the code. It is still very important to test and verify all steps of the development. Figure 1.1 shows the different steps of software development and the test and verification phases it has.

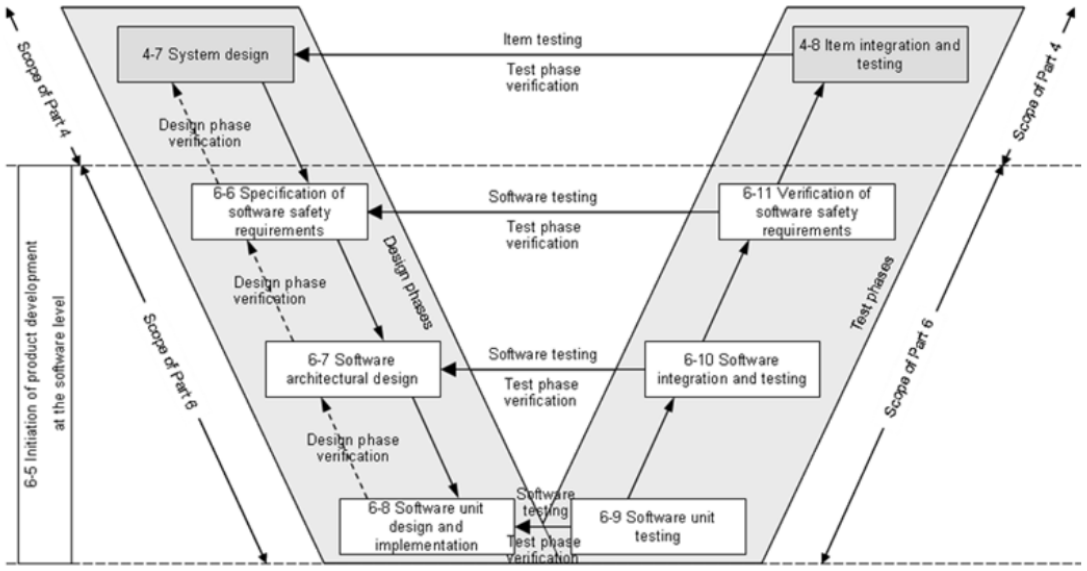


Figure 1.1: Phase model for software development

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 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.

1.1.1 QuickCheck

QuickCheck tests a program with a specification implemented as properties that the program must hold [3]. QuickCheck creates an arbitrary test vector for each of the properties. Because it is arbitrary it can't be used for true formal verification(?).

1.1.2 Industrial standards

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 [4, Table 9, part 6, p. 26]. 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 sould 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.

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 configurables.

1.1.3 Why Erlang?

First of all Erlang can easily communicate with other programming language by using byte streams. There have already been some work done including Erlang AUTOSAR and QuickCheck, mostly by the QuviQ company[6]. Imperative coding requires state based testing(?). There are a library in Erlang developed by QuviQ for this purpose(?).

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 face.

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[2], 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[1].

Chapter 2

Method

2.1 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.

2.2 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.

2.3 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?

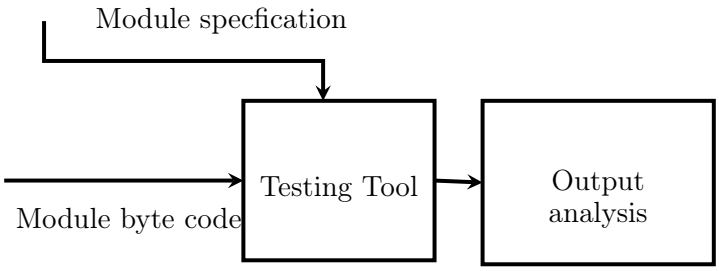


Figure 2.1: Abstract implementation module

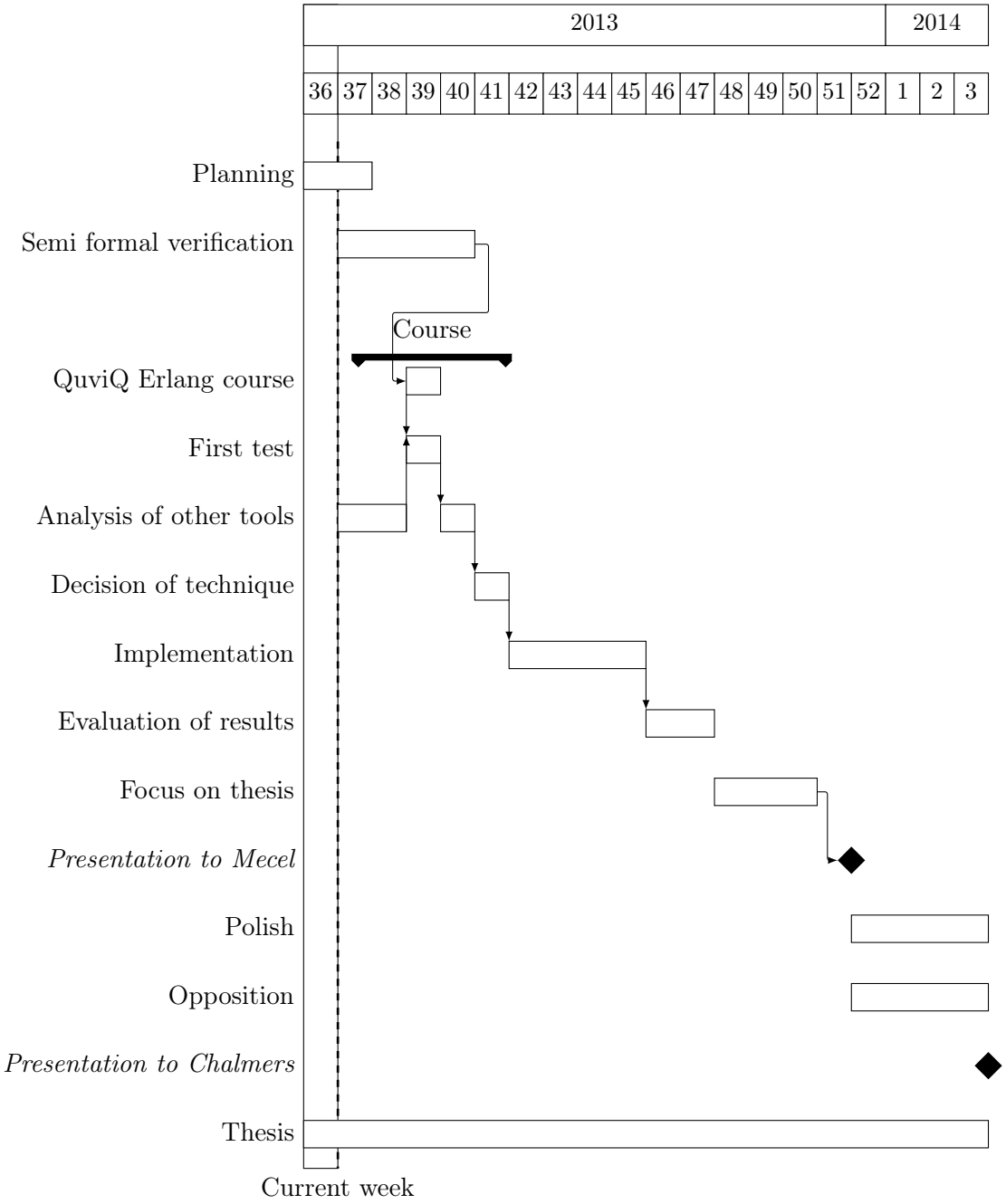
Chapter 3

Time-plan

3.1 Provisional plan

Exactly how things should be done isn't clear. Therefor we are proposing a more orderly plan to be able learn what can be done and where problems may lie a head. At first just try to implement one or two simple rules from an AUTOSAR specification, in Erlang, and try to run it on AUTOSAR module. It's important to notice that this may have nothing to do with our final solution and is mostly for evaluating what can be done in QuickCheck. Since our work is not just about implementing tests, but rather show that we can make semi formal verification on a module in AUTOSAR, the number of rules that we implement seems less relevant than what we can do with the rules that we implement.

The next step is to analyse the results and come to a conclusion for how to continue. Possible Erlang and QuickCheck is not suited for our goals. More information about this can be found on the Gantt scheme represented below.



Finding out the exact definition of semi formal verification and getting an idea of how this can be implemented is a fundamental step. Hence this is added explicitly, as shown in the Gantt Scheme, to the project plan. In the first test week we translate some specification demands to QuickCheck code, and test if it is possible to prove that this is semi formal verfied. The purpose of doing analysis of other tools is to see if there exists better tools for our goals, we should then decide which tools and techniques to use. A

presentation to Mecel is planned to the last week in December. In January we present our work to Chalmers.

Bibliography

[1] AUTOSAR. *Specification of Crypto Service Manager*. AUTOSAR, third edition. Document Identification Number: 402.

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

[3] John Hughes. *QuickCheck: An Automatic Testing Tool for Haskell*.

[4] ISO. Road vehicles - Functional safety. ISO 26262, International Organization for Standardization, Geneva, Switzerland, 2011.

[5] John Hughes Koen Claessen. Quickcheck: A lightweight tool for random testing of Haskell programs. Technical report, 2000.

[6] QuviQ. QuickCheck. `http://www.quviq.com/documents/QuviqFlyer.pdf`, September 2013.