Testing AUTOSAR software with QuickCheck

Conference Paper · April 2015

DOI: 10.13140/RG.2.1.3348.6247

4 authors:

Thomas Arts
Chalmers University of Technology
85 PUBLICATIONS **1,641** CITATIONS
SEE PROFILE

John Hughes
Chalmers University of Technology
143 PUBLICATIONS **5,916** CITATIONS
SEE PROFILE

Ulf Norell
University of Gothenburg
26 PUBLICATIONS **739** CITATIONS
SEE PROFILE

Hans Svensson
Quviq AB
23 PUBLICATIONS **268** CITATIONS
SEE PROFILE

Testing AUTOSAR software with QuickCheck

Thomas Arts, John Hughes, Ulf Norell, Hans Svensson
Quviq AB
Gothenburg, Sweden
thomas.arts@quviq.com

***Abstract*—AUTOSAR (AUTomotive Open System ARchitecture) is an evolving standard for embedded software in vehicles, defined by the automotive industry, and implemented by many different vendors. A modern car may contain over 100 processors, running AUTOSAR software from a variety of different suppliers, which *must* work together for the car to function. On behalf of Volvo Cars, we have developed model-based acceptance tests for some critical AUTOSAR components, to guarantee that the implementations from different vendors are compatible. We translated over 3000 pages of textual specifications into QuickCheck models, and tested many different implementations using large volumes of random test cases generated from these models. This resulted in over 200 issues which we raised with Volvo and the software vendors. We compare our approach with an earlier manual approach, and argue that ours is more efficient, more effective, and more correct.**

I. Introduction

The automotive industry, with European manufacturers as the driving force, has decided on a software standard for automotive software: AUTOSAR [1]. Standardizing the software should lower costs by enabling manufacturers to purchase off-the-shelf components. AUTOSAR defines many software components, but in this paper we consider only the "Basic Software Modules" (BSW), which can be considered to be part of the operating system running on each ECU (Electronic Control Unit—automotive jargon for "processor"). The Basic Software consists of 24 modules, making up a number of protocol stacks (CAN, LIN, FlexRay and Ethernet), a communications-and-routing module, network management, and diagnostics (see Fig. 1). In this paper, we will focus on the CAN (Controller Area Network) [**?**] stack in particular, since the CAN bus is heavily used for communication between the ECUs in a vehicle.

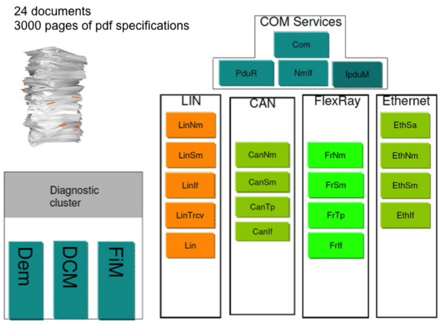

Fig. 1. VCC definition of exchangeable software clusters

Each BSW module is specified by a carefully written English document, with a clear API description (using the C programming language) and hundreds of requirements. Typically such a document is a few hundred pages long.

The first OEM to produce a car model completely based upon a new version of this software standard faces a risk of incompatibility between components from different vendors. The risk can be mitigated by buying all software from the *same* vendor, but most OEMs like to choose between different solutions. Of course, all components are already tested by their vendors, but only against the vendor's own interpretation of the standard. Therefore, Volvo Car Corporation (VCC) collaborated with SP and Quviq to define acceptance tests[1] for vendor specific implementations of the Basic Software Modules—in particular, for 20 modules including the communication part, the communication stacks, and diagnostics.

In Sect. II we highlight the challenges in testing the highly configurable AUTOSAR software. In Sect. III, we then outline our solution based upon model based testing with QuickCheck [2] models, from which large sets of random test sequences are generated. This methodology turned out to be far more efficient and effective than the manually created test sets developed in the traditional way, as we demonstrate in Sect. IV.

With QuickCheck models and automatic test case generation, we out-perform manually crafted automated tests in effort and fault-finding capabilities. This is particularly visible in complex, highly configurable software with high safety demands.

II. The testing challenge

The AUTOSAR standard specifies software as a large set of components, or modules. For example, the CAN communication stack consists of separate documents specifying driver, the transceiver (CanTrcv), network management (CanNm), the interface to other modules (CanIf), the transport layer (CanTp), and an internal state machine specification (CanSM). The requirements are local to these documents. In other words, it is clear for each document what requirements one would like to test, but there are no separate documents that specify how the complete stack should behave; that is left implicit in the requirements of the parts.

Module level specification

All OEMs, including VCC, like to allow vendors a certain implementation freedom. Therefore, they place different demands on the parts of the software that they would like to be able to exchange. They might, for example, take the view that

[1] In VCC terminology the acceptance test is a test to see whether the software meets the functional requirements.

they only want to be able to replace the basic software from one vendor in its entirety by the complete software of another vendor, in which case only the external interface (API) of the complete software bundle needs to follow the standard—or they might want the freedom to replace groups of modules with the same group from another vendor. The smallest part that an OEM wants to be able to exchange can be called a cluster. VCC decided to set the cluster border at the level of protocol stacks, where the driver is so hardware specific that the border is above the driver level. For example, the CAN stack is made up of a cluster of four modules (Fig. 1).

The testing approach *must* allow deviations from the standard in the interfaces within a cluster. A strict check whether each module separately fulfils the specification is almost guaranteed to find false positives, where the defects are due to modified interfaces and short-cuts. Therefore, we must avoid testing each module in isolation, instead testing a whole cluster at once. The difficulty with that is that formulating a test at the cluster level is very difficult and error prone; there are no explicit requirements for the cluster. In practice, such a test would be written by taking a top level API function, interpreting all the layers it traverses, and predicting the outcome at the bottom of the stack. This is error prone, and, because many documents must be interpreted together, it is extremely hard to explain *why* a failing test violates the standard.

Highly configurable

AUTOSAR basic software is extremely configurable. All OEMs use different feature sets—in fact each car uses different features. For example, some ECUs do not need safety checksums on their messages, whereas others do. From one large configuration file (several thousands lines of XML), per-ECU sub-configurations are extracted, and from these software is partly generated and partly instantiated. The final generated C code is compiled and integrated as a library into the ECU. Software development is often driven by customer requirements for a specific ECU. Different implementations may implement a subset of all available features, although the intention is to be complete.

Since every ECU has a different configuration of the software, there are very many different instantiations of the software that need to be tested. Where each OEM may be happy with a test for their specific handful of configurations, the software developers would prefer to re-use tests, and deliver software tested for the same configurations to all the different OEMs. Moreover, software developers would like to keep the configurations and tests small in order to be able to predict the outcome of a test. In practice, this results in each vendor having up to a hundred different small configurations with an extensive test set for each of them. Because configuring the software is a manual process, then adding a new configuration and accompanying tests can take days or weeks.

Complex Scenarios

The software in cars caters for many very complex scenarios. There are so many different scenarios that it is virtually impossible to think of all of them. Moreover, scenarios may be configuration dependent. For example, signals transmitted over the CAN bus have priorities. High priority messages should be transmitted before low priority ones. There are two versions of the CAN standard, and this property should be tested for both. Moreover, an implementation running one standard should be able to communicate with an implementation running the other. This complexity gives rise to many more possible scenarios than a tester is likely to think of.

Implementation freedom

The AUTOSAR specifications leave some freedom to the implementors, which makes sense, since otherwise there would not be any room for competition. Very tightly specified tests may therefore detect "deviations" that OEMs and vendors would claim are not really important. For example, if a buffer of a certain size is configured in a router, then it is acceptable to reject a second message when a first is in transit. However, it is also perfectly ok to put *both* messages into the buffer, if they both fit. A simple test to check that a first message is sent and a second rejected would fail for such a smart solution, and this is unwanted behaviour.

III. Test methodology and tools

The method we have developed is based upon creating one large configuration covering all the features that VCC is interested in. This means that each vendor need configure their software only once for testing. The configured and compiled software is then submitted as a C library for testing.

Quviq have developed QuickCheck models for each module, with a close correspondence between model code and pdf specification. These QuickCheck models can be seen as executable formal specifications. Each model can be used to generate random test cases for the module it specifies. The reason we make one model per AUTOSAR module is that, otherwise, we would need to manually interpret the standard to see what the effect of a top level API call should be. Instead we want to *compute* that effect, from the models of each module. We created a mocking [3] and clustering solution that on the one hand is able to mock sequences of randomly generated calls to a lower layer, and on the other hand can glue a layer together with a lower layer to form a larger specification. Using these techniques, we can automatically build a composite model for *any* possible cluster of AUTOSAR Basic Software Modules.

The models are parameterised by the configuration. That is, the testing solution allows any configuration from any vendor and will instantiate the models in such a way that only tests valid for that configuration are created. For example, the configuration determines which signals can be sent, which of them expect a confirmation, etc. The configured signals, with their properties and features, are used in the test case generation process to decide what operations can be performed and what effect they should have. Similarly, we use timer values from the configuration to guide and predict events.

The test methodology is graphically presented in Fig. 2. The OEM provides a configuration file, which is input for both the vendor's process of creating the Software Under Test (SUT), as well as for the parameterised QuickCheck models. QuickCheck then generates thousands of randomly created test

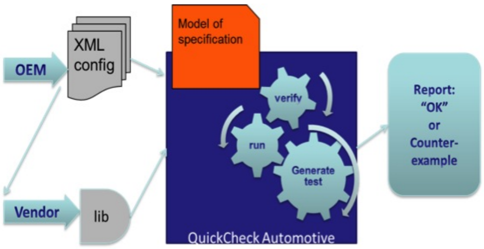

Fig. 2. QuickCheck Automotive solution

cases, which are executed against the vendor’s delivered code. If a failing test case is detected, QuickCheck automatically reduces the test case to the minimal test case that provokes a failure. This turns out to be necessary for failure reporting, since explaining and understanding why tests fail is non-trivial. Shorter tests are far easier to explain.

For detecting faults, this method works well, as we demonstrate in the next section. However, when we present failed random test cases to vendors or OEMs, we always face questions of the form ”why is this a valid test?” and ”why do you expect this test outcome?”. To answer those questions one needs both the configuration, and, crucially, the *requirement* that is violated by the test. We therefore annotated our models with requirements. Just as one might annotate C source code with comments of the form ”here we fulfil requirement X”, so we added annotations to the test cases ”here we test requirement X”. Given the operational character of the AUTOSAR specification, each requirement could be connected to only one or two places in the model. We still need to manually validate and explain each test failure, but the annotations make it much easier and quicker to figure out why a test is failing.

The last part of the puzzle is a stopping criterion for testing. Of course, we can stop testing as soon as errors are found, from an acceptance test point of view, since the SUT cannot be accepted. But if no violations are found, how many and which tests should be executed? We decided to address this by collecting the states the test model traverses while testing. In the state we record, for example, which requirements we have tested. We also record which routing paths we have taken, which kind of signals we have transmitted, etc. The state space of the test model is practically infinite, so we cannot possibly visit all states. However, by making an abstraction related to reasoning about safety, we define as a stopping criterion the moment we have covered all the *abstract* states in the test model.

Of course, our QuickCheck models initially contained errors. Just like the implementations, or models are a best-effort interpretation of the standard. We therefore evaluated our models against four independent implementations. For each difference detected, we needed to clarify what the correct behaviour should be.

IV. Results

We have modelled 20 AUTOSAR modules in two different versions of the specification, version 4.0.2 and version 4.0.3. Our models were ready before any vendor was able to submit a complete implementation. Developing a test model in a high level language is faster than writing corresponding C code. Moreover, the test models are not part of the same production process as the actual code and can therefore be developed in a more agile way.

In this section we explain the effectiveness of the approach in terms of the kinds of issues we have found during model development and testing. We evaluate the efficiency by comparing the test approach to an abandoned manual test approach developed by the AUTOSAR consortium. Finally, we report on a small experiment that shows that our method of evaluation against a number of different vendors turns out to be more correct than an approach by the standardisation committee in which high-level, manually written test cases are standardised.

Effectiveness

While developing the models, we regularly found ambiguities in the specification just by trying to implement tests. Every software development team has a similar experience, although domain knowledge often biases the reader to a certain interpretation of the ambiguous specification. We documented each ambiguity and deviation with the vendor code that we detected. In meetings with domain experts, these issues were discussed on a weekly basis. In total we registered 227 tickets during the length of the project, fairly evenly distributed over the different clusters. We classified them after resolution in a number of ways: whether we found the specification to be unclear or wrong (spec defect), whether the vendor admitted that their code was wrong (vendor defect), or whether the model needed to change (model defect). Of course, most model bugs could be resolved just by consulting the standard, without any issue being raised.

Of these 227 issues, about 180 are due to ambiguities, unclear formulations or deliberate implementation freedom in the specification. These vary in nature—figures may not agree with the text in the description, which error message to expect in certain situations may be unclear (negative cases are often less well specified than positive cases), behaviour may be unspecified, and there are even clear inconsistencies:

The description of the function says that ”Com_TriggerTransmit returns in the PduInfoPtr ... a pointer to the AUTOSAR COM module’s transmit buffer”. Requirement COM647 says ”the AUTOSAR COM module shall copy the contents of its I-PDU transmit buffer to the L-PDU buffer given by SduPtr.” These two are in conflict. Either the function should provide a pointer to where the data is or copy the data to the memory pointed to by the pointer passed in as an argument.

As a result of our modeling project, VCC has filed 20 issues as Bugzilla’s to the AUTOSAR consortium, therewith contributing to an improvement of the standard. Other issues were in parallel detected and filed or have been classified as implementation freedom, meaning that the test model should accept different correct behaviours.

We cannot reveal which software errors we detected in the production code we received, apart from one example. This example shows the power of randomly generated tests. There is always a scenario that is forgotten by the testers. Randomly generating large test sequences reveals such scenarios. Our

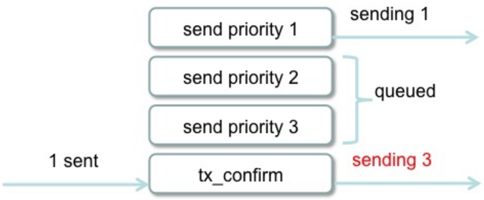

Fig. 3. CAN bus priority error

automatic test case shrinking, which extracts a minimal failing test sequence from a large one, then provides us with small test cases that are easy to understand. Each of those could have been supplied as a hand-written test case, but the point is, they are not, because there are so many possible scenarios and testers always overlook some of them.

For the software error we report on (Fig. 3) it is important to know that there are two version of the CAN protocol, standard CAN and extended CAN. The relevant difference is that the address field in standard CAN is 11 bits, whereas the address is 29 bits in extended CAN. In the CAN stack under test, the address is stored in both cases as an unsigned 32-bit integer, with the top bit set to indicate an extended CAN address. Lower-valued addresses take priority when there is contention for the bus. Nevertheless, we found a defect in which the CAN stack sent a message with address 3 before a message with address 2. The reason was that address 2 was an extended CAN signal, and so its address was stored in memory as $2^{31}+2$—and the most significant bit was not cleared before comparison with the address 3. This could be serious if, for example, your brake system is a modern one, working over extended CAN, but your tyre pressure is measured via the old CAN protocol.

After the models were evaluated and stable, additional vendors submitted production code for acceptance testing. In one month, we found 55 confirmed defects in clusters delivered for testing. Vendors were able to quickly understand the defects, correct them and re-submit their code. This indicates that the shrunk test sequences are indeed practically useful and that the procedure we have in place makes fast turn-around times possible.

A. *Efficiency*

In order to provide some data on the efficiency of the approach, we compare it with a manually created test suite for an earlier version of AUTOSAR. The AUTOSAR consortium defined standard AUTOSAR test suites written in TTCN for AUTOSAR version 3.1. These tests are defined for single modules, and clustering is not possible. This restricts the ability to test optimisations within the cluster; manual adaptation may be necessary to make use of some of these test cases.

The modules in the CAN cluster—CanIf, CanNm, CanSM and CanTp—have separate test suites. In total there are 245 tests for those four modules, implemented in 57983 lines of TTCN code. There is no exact way of determining whether our QuickCheck models could generate each of these 245 tests. We probably would need to insert a defect into the code that each such test case detects, and then see whether our model also detects that inserted defect. Unfortunately, we cannot do this in practice, since we do not have the source code of the AUTOSAR implementations in our testing project—only the binary object code. Instead we have manually inspected the test cases and their descriptions to learn what tests are performed. We believe that we cover all these cases in one way or another with our test model.

TABLE I. TTCN3 CODE VERSUS QUICKCHECK CODE

Module	TTCN3 lines of code	TTCN3 nr of tests	QuickCheck lines of code	QuickCheck time to generate 100 tests
CanIf	16930	65	1978	24 sec
CanSM	6751	17	1255	10 sec
CanNm	12318	58	1716	47 sec
CanTp	21984	105	2062	20 sec
cluster	57983	245	7011	33 sec

Our QuickCheck model consist of four models that are clustered: CanIf, CanSM, CanNM and CanTp. Together with all code for stubbing the upper and lower ends of the protocol, as well as stubs for the individual modules so that each QuickCheck model can also be used in isolation, we wrote 7011 lines of code; a factor 8 less than the TTCN code. At the same time, we can generate tens of thousands of different meaningful tests with these models. These tests have revealed software defects in 20 different implementations of already-well-tested software.

The test suite for the communication cluster consists of 124097 lines of TTCN3 test cases, whereas our QuickCheck models together with all stubs comprise 6777 lines of code. This strengthens the hypothesis that using QuickCheck models for generating test cases is more efficient.

Thus, the QuickCheck approach is scalable to real size projects, and out-performs existing TTCN approaches in scalability. Moreover, when we introduced an expert tester from one of the AUTOSAR vendors to our tool, then he explained us that he needed one full day of work for each of the 245 tests in the TTCN3 test suite—to set up the test, run it, and analyze the result. He took a QuickCheck course and after one day was able to use QuickCheck. The next day, he was able to run and analyze far more than 250 tests.

When we measure the time it takes to generate 100 test cases, we see no significant slow-down when we cluster the modules. The total time to generate 100 tests is about 30 seconds, which is quite fast enough for practical testing purposes. Test case generation is a factor 3 to 4 slower than when generating tests for each module separately, but this is acceptable given the benefits it offers in writing the models; testing is not that performance critical for the AUTOSAR testing project.

Another comparison one can make is how much work the test models are compared to writing the actual implementations (Fig. 4). It is well known that good testing may easily take up to 50% of the project budget. In our case we had C source code for one particular implementation, and found that the QuickCheck models are about 30% of the size of the C—slightly more than we expected, given that we use a much higher level language than C. The explanation is that the application area is so well suited to C that doing the same thing in our environment simply takes more code.

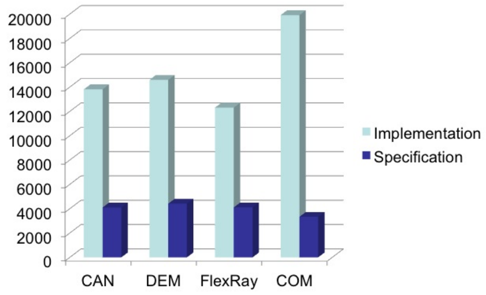

Fig. 4. Lines of C code versus lines of QuickCheck code

Correctness

In parallel with our project, the AUTOSAR consortium also formed a working group to define acceptance tests. For the CAN stack, 6 high level test descriptions were published in July 2014 [4]. Of course, we were curious to see whether we were testing the same thing or not. Therefore, we encoded those high-level test cases as executable tests compatible with our models, and executed them against the model, checking whether the preconditions for each API call were satisfied, and whether, when calling the API function with the arguments specified in the test, we got the same results.

Out of the 6 tests, 3 passed this evaluation. In the other 3 cases, the model did not accept the test cases—and so, we would never have been able to generate them from the model. After further analysis, we discovered that this was because the 3 standardised test cases did not consider the value of the `CanSMBorTimeTxEnsured` timer in BusOff recovery. This was acknowledged as an error in those tests, and is filed as Bugzilla 67170 for correction in the next release.

V. Conclusion

We translated over 3000 pages of textual specifications into QuickCheck models, and generated test cases from these models for clusters of AUTOSAR basic software modules. With our testing method, we have been able to find errors in production code in a more efficient and correct way than traditional testing methods are able to achieve. We estimate this method to be 5 to 8 times less work than standard testing approaches. The approach can be combined with formal methods to meet the ISO26262 safety standard.

We contributed to better software quality, but also to better specifications, by filing issues found during implementation of the models, or while testing code from different software vendors.

Acknowledgment

The research leading to these results has received funding from the EU FP7 Collaborative project *PROWESS*, grant number 317820, from the ARTEMIS JU under grant agreement n^o 295373, and from the Swedish Foundation for Strategic Research under grant RAWFP.

References

[1] AUTOSAR Consortium, “Automotive open system architecture, standard documents,” https://autosar.org/, 2013.

[2] T. Arts, J. Hughes, J. Johansson, and U. T. Wiger, “Testing telecoms software with Quviq QuickCheck,” in *Erlang Workshop*, M. Feeley and P. W. Trinder, Eds. ACM, 2006, pp. 2–10.

[3] J. Svenningsson, H. Svensson, N. Smallbone, T. Arts, U. Norell, and J. Hughes, “An expressive semantics of mocking,” in *Fundamental Approaches to Software Engineering*, ser. LNCS, S. Gnesi and A. Rensink, Eds. Springer Berlin Heidelberg, 2014, vol. 8411, pp. 385–399.

[4] AUTOSAR Consortium, “Acceptance Test Specification of Communication on CAN bus - Release 1.0.0,” July 2014.