Testing AUTOSAR software with QuickCheck

Thomas Arts*, John Hughes[†], Ulf Norell*, Hans Svensson*
Quviq AB*, Gothenburg, Sweden. thomas.arts@quviq.com
Quviq and Chalmers University of Technology[†], Gothenburg, Sweden.

***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. On behalf of Volvo Cars, we have developed model-based acceptance tests for some critical AUTOSAR components, to guarantee that 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 generated tests. This exposed over 200 issues, which we raised with Volvo and the software vendors. Compared to an earlier manual approach, ours is more efficient, more effective, and more correct.**

I. Introduction

AUTOSAR is a software standard developed by the automotive industry, to lower costs by enabling manufacturers to buy off-the-shelf components [1]. In this paper we consider the AUTOSAR “Basic Software Modules” (BSW), part of the operating system running on each ECU (Electronic Control Unit—automotive jargon for “processor”). The Basic Software consists of 24 modules, comprising 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 focus on the CAN (Controller Area Network) bus stack in particular, since it is heavily used for communication between the ECUs in a vehicle.

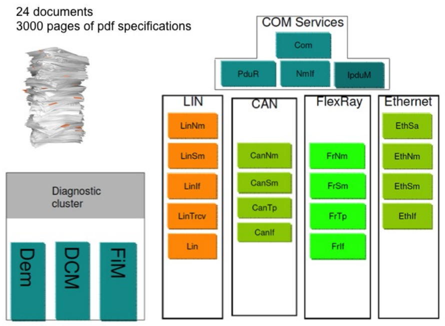

Fig. 1. VCC definition of exchangeable software clusters

Each BSW module is specified by a carefully written English document—typically a few hundred pages—describing a C API and hundreds of requirements.

2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
10th Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)
978-1-4799-1885-0/15/$31.00 ©2015 IEEE

The first OEM to base a car on a new version of this standard risks incompatibility between components supplied by different vendors, if the vendors have interpreted the standard differently. This risk can be mitigated by buying all software from the *same* vendor, but most OEMs dislike being locked-in like this. Volvo Car Corporation (VCC) therefore worked with SP and Quviq to define acceptance tests for implementations of the Basic Software Modules.

In Sect. II we explain the challenges in testing highly configurable AUTOSAR software. In Sect. III, we outline our approach based on QuickCheck [2], [3] models, from which large sets of random test sequences are generated. This approach proved to be far more efficient and effective than manually created test sets developed traditionally, as we report in Sect. IV. Automatic test generation from QuickCheck models out-performed manually crafted tests both in terms of effort and fault-finding capability.

II. The testing challenge

The AUTOSAR CAN stack comprises five separate modules, each described in its own document: the transceiver (CanTrcv, not considered further here), network management (CanNm), the interface to other modules (CanIf), the transport layer (CanTp), and an internal state machine specification (CanSM). Requirements are stated per module—so it is explicitly stated how each module should behave, but not how the complete stack should behave; that is implicit in the requirements for its parts.

Module level specification

All OEMs allow vendors to deliver *clusters* of modules, specified by the OEM, rather than single modules; internal interfaces within a cluster need not follow the standard. This enables implementors to optimize across module boundaries. OEMs expect to be able to mix clusters from different vendors, but not to mix modules within one cluster. VCC decided to set the cluster borders at the level of protocol stacks (excluding low-level drivers such as CanTrcv).

Thus testing *must* allow deviations from the standard internally within a cluster—modules tested in isolation *will* fail. We must test at the cluster level—even though there are no stated cluster-level requirements. We have to *infer* the intended behaviour of the cluster, by tracing the effect of a top-level API call through all the layers it traverses, to predict the outcome at the bottom of the stack. This is error prone, because we must interpret many documents together, and also hinders explaining exactly *why* a failing test violates the standard.

Highly configurable

AUTOSAR basic software is extremely configurable. Each OEM (and, in fact, each car) uses a different set of features. For example, some ECUs need safety checksums on their messages, whereas others do not. A car configuration is several thousand lines of XML; from this, for each ECU, a sub-configuration is extracted, C code is partly generated and partly instantiated, then compiled and integrated as a library onto the ECU. Software development is often driven by customer requirements for a specific ECU, which means that different vendors' implementations may implement different subsets of the standard.

This means the software needs to be tested in many different configurations—but which ones? The OEMs might prefer their own configurations—but the vendors 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 configurations and tests small in order to ease prediction of test outcomes. In practice, each vendor has up to a hundred different small configurations with an extensive test set for each of them. Because configuring the software is a manual process, adding a new configuration and accompanying tests can take days or weeks.

Complex Scenarios

The software in cars caters for many very complex, configuration-dependent scenarios. For example, signals transmitted over the CAN bus have priorities, each of which can be represented in two different ways, which gives rise to many more scenarios than a human tester is likely to think of.

Implementation freedom

The AUTOSAR specifications deliberately leave some freedom to implementors, which means that very tightly specified tests may detect "deviations" that OEMs and vendors consider unimportant. For example, if a router is configured with a buffer, then it is always acceptable to reject a second message if a first is in transit. However, it is also ok to put *both* messages into the buffer, if they both fit. A simple test checking that a first message is sent and a second rejected would fail for such a "smart" solution, which is undesirable.

III. Test methodology and tools

We decided to create one large configuration covering all the features that VCC is interested in, so that each vendor need configure their software only once, and submit a configured and compiled C library for acceptance testing.

We developed a QuickCheck state-machine model for each module, with a close correspondence between model code and PDF specification. These models can be seen as executable formal specifications; each operation is specified via a pre- and post-condition, a model state transition function, and a (state-dependent) generator for suitable arguments. Each model can be used to generate random test cases for the module it specifies, in which the other modules are *mocked*; our models include a specification of which (mocked) calls to other modules should be expected, and what results they should return [4]. We also developed a way to glue models together, matching mocked calls from one module to API calls of another, thus automatically constructing a model for a cluster. Using these techniques, we can build a composite model for *any* possible cluster of BSW modules.

Our models are parameterised on the configuration, and generate only tests valid for that configuration. 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 during test case generation to decide which operations can be performed and what effect they should have. Likewise, timer values from the configuration are used to predict when events will occur.

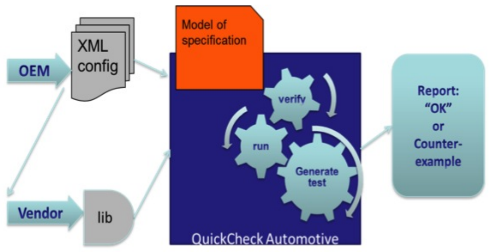

Fig. 2. QuickCheck Automotive solution

Our methodology is presented in Fig. 2. The OEM provides a configuration, which is input for both the vendor's generation of the Software Under Test (SUT), and the parameterised QuickCheck models. QuickCheck then generates many randomly created test cases, which are executed against the vendor's code. If a test fails, QuickCheck automatically reduces the test case to a minimal case that provokes a failure. These minimal test cases are much easier to understand and explain than the random cases that first fail, and so much better suited for reporting to VCC or the vendor.

When reporting failing tests, we need to explain why the test is valid, and why the outcome is wrong. To do so, we need both the configuration, and, crucially, the *requirement* that the test violated. We therefore annotated our models with requirements. Just as C source code may contain annotations of the form "here we fulfil requirement X", so we added annotations to the test cases "here we test requirement X"; we found that most requirements could be linked to just one or two places in the model. We do still need to interpret test failures for VCC and the vendors, but these annotations make it much easier and quicker for *us* to figure out and explain why a test is failing.

Finally, we need to decide when to stop testing. Of course, we can stop acceptance testing as soon as the first error is found, since the SUT should not be accepted. But if no violations are found, how many and which tests should be executed? We address this by collecting the model states traversed while testing, recording the requirements we have tested, the routing paths taken, the kind of signals transmitted, *etc*. The model state space is extremely large and we cannot possibly cover it all, so we make an abstraction related to safety argumentation, and stop as soon as we have covered all the possible *abstract* states.

Of course, our QuickCheck models were initially wrong—they are just a best-effort interpretation of the standard. We therefore tested them against four independent implementations. For every difference we found, we clarified the correct behaviour by first consulting the standard, and then if need be discussing it with VCC and/or the vendors concerned.

IV. Results

We modelled 20 AUTOSAR modules in two different versions of the standard, version 4.0.2 and version 4.0.3. Our models were ready before any vendor was able to submit a complete implementation (in part because we wrote in Erlang instead of C). Of course, 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 found during model development and testing. We evaluate efficiency and correctness by comparing our test approach to manual testing approaches by the AUTOSAR consortium.

Effectiveness

While developing the models, we regularly found ambiguities in the specification just by trying to implement tests. We found deviations between our understanding and vendors' understanding by running our models against their code. We documented each ambiguity and each deviation with the vendor code that we found, and discussed them with domain experts at weekly meetings. In total we registered 227 tickets during the project, fairly evenly distributed over the different clusters. We classified them after resolution as follows: wrong or unclear specification (spec defect), code defect admitted by the vendor (vendor defect), or model defect. Of course, most model bugs could be resolved outside this process, 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 disagree with the text, 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. For example, the function description 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 it should 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, thereby 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.

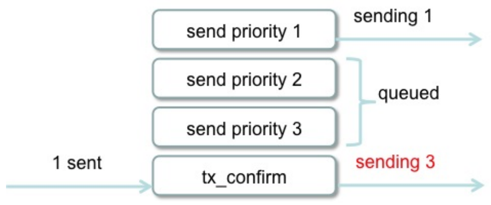

Fig. 3. CAN bus priority error

We are only at liberty to reveal one of the errors we detected in the production code we received—see Fig. 3. In this example, three messages are sent over the CAN bus, with message identifiers 1, 2 and 3. The latter two messages are queued until the driver confirms transmission of message 1 is complete. Lower-valued message identifiers take priority when there is contention for the bus, so the stack should then transmit message 2—but instead transmitted message 3. The reason is that message 2 used the standard CAN format, with 11 bits for the message identifier, while message 3 used the extended CAN format, with 29 bits available. In the CAN stack under test, this identifier is stored in both cases as an unsigned 32-bit integer, with the top bit set to indicate the extended format. The developer forgot to mask off this bit before comparing the two message identifiers for priority, and so message 2 was treated as a message with priority $2^{31} + 2$.

This example clearly shows the power of randomly generated tests, provoking an error that is easy for a human tester to miss. Clearly, any test of the message prioritization must include at least three sends (one to block the bus, and the other two to be prioritized between)—but few human testers would think of testing this for all combinations of standard and extended message identifiers. It also shows the power of test case minimization: the example we found is salient and to the point. Finally, the bug itself is important: imagine if 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 minimal failing tests we generate are indeed understandable to developers, and enable rapid debugging.

A. *Efficiency*

We compared our approach to a hand-written test suite for an earlier version of AUTOSAR. The AUTOSAR consortium defined standard test suites written in TTCN3 for AUTOSAR version 3.1. These tests are defined for single modules, and clustering is not possible, so manual adaptation may be necessary to make use of them.

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

In total there are 245 tests for the four modules in the CAN cluster, implemented in 57983 lines of TTCN3 (see Table I). While it is hard to be 100% certain, we have manually inspected the test cases and descriptions, and believe that we do cover all these cases one way or another.

Our QuickCheck model consists of four clustered models, one for each CAN module. Together with stubs for the upper and lower ends of the protocol (and for the individual modules so that each QuickCheck model can be used in isolation), we wrote 7011 lines of code; a factor 8 less. But we can generate tens of thousands of tests from these models, and our tests have revealed defects in 20 different implementations of already-well-tested software.

The test suite for the communication cluster is 124097 lines of TTCN3, whereas our QuickCheck models together with all stubs comprise 6777 lines of code—18× smaller. Taken together, these figures are strong evidence that QuickCheck specifications are more concise than TTCN3 test suites.

Moreover, when we introduced an expert tester from one of the AUTOSAR vendors to our tool, then he explained to 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 and our models. 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 this project.

Another comparison we can make is how much work it was to write the test models, compared to writing the actual implementations (Fig. 4)—it is well known that testing can cost up to 50% of a 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.

Our team alternated between reading the standard, writing the models, testing the models against implementations, and consulting domain experts, so we cannot report in more detail on the work devoted to each aspect.

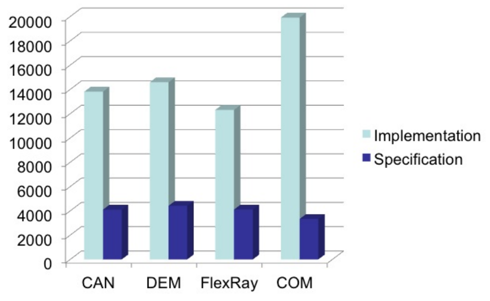

Fig. 4. Lines of C code versus lines of QuickCheck code

Correctness

In parallel with our project, the AUTOSAR consortium also developed acceptance tests—6 tests for the CAN stack, published in July 2014 [5]. We encoded these test cases in our format, and executed them against the model. 3 of the 6 tests agreed with our model; the other 3 were not accepted by it. So we could never have generated these tests—but this turned out to be a *good* thing: the 3 standardised test cases failed to consider the value of the CanSMBorTimeTxEnsured timer in BusOff recovery. This has been acknowledged as an error, and is filed as Bugzilla 67170 for correction in the next release.

It seems that testing our models against a number of real implementations has resulted in robust correctness.

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. In this way, we were able to find errors in production code more efficiently and correctly than by traditional approaches. We estimate this method to be 5 to 8 times less costly than traditional testing, and considerably more effective.

We contributed to better software, and also better specifications, by filing issues found during model implementation, and while testing code from different software vendors.

ACKNOWLEDGMENTS

This work is partially funded by the EU FP7 project *PROWESS*, grant number 317820, the ARTEMIS JU (grant agreement n^o 295373), and the Swedish Foundation for Strategic Research under grant RAWFP.

REFERENCES

[1] AUTOSAR Consortium, “Automotive open system architecture, standard documents,” https://autosar.org/, 2013.
[2] K. Claessen and J. Hughes, “QuickCheck: a lightweight tool for random testing of Haskell programs.” in *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 2000, pp. 268–279.
[3] 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.
[4] 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.
[5] AUTOSAR Consortium, “Acceptance Test Specification of Communication on CAN bus - Release 1.0.0,” July 2014.