Don't Judge Software by Its (Code) Coverage

Rolf Johansson, Hans Eriksson, Hans Svensson, Kenneth Östberg, Thomas Arts, Alex Gerdes, Martin Skoglund

▶ **To cite this version:**

Rolf Johansson, Hans Eriksson, Hans Svensson, Kenneth Östberg, Thomas Arts, et al.. Don't Judge Software by Its (Code) Coverage. SAFECOMP 2013 - Workshop CARS (2nd Workshop on Critical Automotive applications : Robustness & Safety) of the 32nd International Conference on Computer Safety, Reliability and Security, Sep 2013, Toulouse, France. pp.NA. hal-00848458

HAL Id: hal-00848458
https://hal.science/hal-00848458

Submitted on 26 Jul 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Don't Judge Software by Its (Code) Coverage

Rolf Johansson[1], Henrik Eriksson[1], Hans Svensson[2], Kenneth Östberg[1], Thomas Arts[2], Alex Gerdes[2], and Martin Skoglund[1]

[1] SP - Technical Research Institute of Sweden, Dep. of Electronics, SE-501 15 Borås, Sweden,
{rolf.johansson, henrik.eriksson, kenneth.ostberg, martin.skoglund}@sp.se
[2] Quviq AB, SE-412 88 Gothenburg, Sweden,
{hans.svensson, thomas.arts, alex.gerdes}@quviq.com

Abstract. This position paper argues that using code coverage metrics to evaluate the completeness of test cases as prescribed by e.g. ISO26262, is insufficient in a safety context. On the other hand it is impossible to execute test cases that achieve 100% completeness with respect to all possible input data combinations testing all requirements. We propose that existing requirements on code coverage shall be extended with requirements on 100% coverage of the safety requirements. As an example, the paper illustrates how this can be done for the AUTOSAR end-to-end protection library.

Keywords: Coverage metrics, Safety arguing, Automotive, ISO 26262, AUTOSAR

1 Introduction

Measuring code coverage is an explicit task in most safety standards for embedded systems, e.g. IEC 61508-3 [1], ISO 26262-6 [2], or DO 178B [3]. Coverage metrics are frequently used as stopping criteria for the testing activity, and to evaluate the adequacy of the test data. If the coverage is insufficient, additional test cases can be added if expected to be beneficial. On unit level, statement coverage is usually considered to be sufficient for low integrity software, whereas MC/DC is required for high integrity. IEC 61508-3 explicitly requires 100% code coverage regardless of metric. If 100% cannot be achieved, due to e.g. defensive code, an appropriate explanation shall be given. ISO 26262-6 and DO 178B are not as explicit as IEC 61508-3, but a motivation is needed when 100% is not reached.

In the context of the AcSäPt research project, we address the question: if 100% code coverage is reached, what is its evidential value in a safety argumentation? It is not hard to find examples where 100% coverage according to some of the common metrics leaves conditions untested and/or severe bugs unspotted in the code. When a specific code coverage metric is targeted, it can be used to guide automatic test case generation. However, it has been shown that such automatically generated test cases were less effective in finding faults than random ones, and shrinking large test suites

CARS 2013
© Springer-Verlag Berlin Heidelberg 2011

while maintaining code coverage (MC/DC in this case) significantly reduced their fault-finding capability [4]. These are examples of some of the pitfalls which could come with overconfidence in code coverage metrics. Despite these possible shortcomings, the largest potential problem is that there is no explicit relation between the functional requirements and the code coverage metrics. 100% code coverage can be achieved while the required functionality still is missing in the code. This is a fact which has been observed by Vanoverberghe et al. [5]. It is evident that besides covering code, it is more important and necessary to cover the specification too.

Of the three aforementioned safety standards, DO 178B is the only standard which explicitly talks about requirement coverage, but no coverage metric is given. The other two standards only mention requirements-based testing as a technique to achieve the same goal. According to DO 178B, requirement coverage is sufficient if: test cases exist for each requirement, and the test cases satisfy the criteria of normal and robustness testing. To normal range test cases belong: equivalence classes and boundary values of valid inputs, normal operation state transitions, and multiple iterations of time-related functionality. To robustness test cases belong: equivalence classes of invalid inputs, system initialization during abnormal conditions, provoking "not allowed state transitions", overflow, etc.

Besides coverage measurements on the requirements of the written specification as indicated by DO 178B, conventional coverage metrics can be used if the specification is modelled in a semi-formal way. If the specification is modelled using state-transition diagrams, state or transition coverage could be used. The specification could be an abstract model or a detailed, complete, model from which code could be generated.

In the rest of the paper, it is assumed that the specification is correct, i.e. complete and consistent [6]. Complete refers to both internal and external completeness. Internal means that there is no information left unstated, and that the information does not contain any undefined entities. External means that the information needed for problem definition is found in the specification. When there are no contradictions in the specification, it is consistent.

2 How (not) to Ensure High Safety Integrity

In order to argue that a piece of software fulfils its safety requirements, it is necessary to show that it cannot fail in any unsafe way. This is a smaller task to show, than that the software component is completely correct.

The underlying question is what we really mean with a certain safety requirement allocated to a component. Johansson et al. [7] pointed out the importance of explicit fault and failure models for giving semantics to any safety requirement.

If the software fails in a safe way, it may be considered having bad quality, but still be safe. In that sense, it is easier to argue for safety than for quality. Depending on what is defined as unsafe for a particular software component, the difference may be small or large. Yet, the software component can unsafely fail in very many different ways. This implies that in order to show absence of all possible unsafe failures, a

complete testing of all possible input data sequences may become unachievable in practice due to the enormous length of such a test sequence. Still we need to show such completeness in order to argue for safety. How can this paradox be solved?

The solution to the paradox is to make reasonable assumptions about the implementations to test. For example, it is reasonable to assume that when adding a CRC checksum, the generator does not alter the data to be protected. Hence, all possible input data sequences do not need to be tested, but only so many different sequences that all reasonable failures can be detected. What is to be considered as reasonable can also be mentioned as the considered fault model. In the next section, reasonable fault models for the AUTOSAR end-to-end protection library are elaborated.

As stated in the introduction, the problem with traditional code coverage metrics is that they do not address the problem of completeness with respect to all relevant specification requirements. The coverage is measuring how complete the test is with respect to how well the implementation (code) is exercised, not with respect to how completely the requirements are checked. On the other hand, the problem with achieving true completeness with respect to all requirements is that it implies a too long test sequence. The situation is depicted in Fig. 1 where the different cases are categorized in two dimensions.

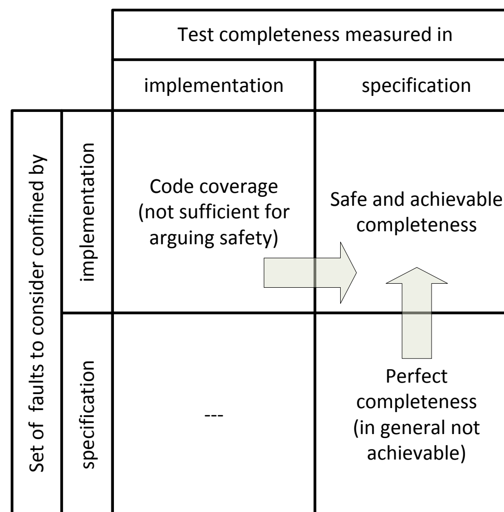

Fig. 1. Fault confinement entities versus completeness measurement entities

The first dimension is whether completeness (coverage metrics) is measuring how well the implementation is covered, or how well the specification is covered. The second dimension is whether the set of faults to check is confined by the specification or by the (assumed) implementation. Note that the implementations to test still may be black boxes. A fault model is what we assume as possible faults in the implementation, and we do not need to know how a certain implementation looks like. More information about an implementation means more specific fault models.

As depicted in Fig. 1, we argue that the solution to the paradox of combining achievable test length and test completeness with respect to safety arguing is to combine the coverage metric of completeness with respect to the specification, with the fault model to consider as confined by the assumed implementation. In this way we can both argue for completeness and for achievable test length. We may reach 100% coverage with respect to the specification, if only we can argue that the considered fault model is reasonable.

3 Example – AUTOSAR End-to-end protection library

As argued in the previous section, it is essential in a safety context to measure coverage with respect to the specification. In this example, this measuring is enabled by modelling the specification. It is possible to instantiate the method with other tools, but we have chosen to use QuickCheck [8]. In QuickCheck, models are done with Erlang as modelling language, and there we can measure everything within the model and the generated tests, and these tests all fall the in category of normal tests, no robustness testing is done.

We have made some initial studies on the AUTOSAR end-to-end (E2E) communication protection library [8], which is a library to protect safety-related data communication. Using the protection library, errors can be detected and handled at runtime. The API of the library is simple; before sending, data is protected by calling the function `protect`. The function `check` is used to check received data for errors. Configuration parameters are: message length, data id, and various message structure parameters. During communication, the sender keeps a (4-bit) counter that is incremented after each message send. The receiver likewise keeps a current state such that it knows what to expect in the next message. The E2E library will in some applications get ASILD integrity safety requirements [2].

3.1 Modelling protect

If we ignore the problem of modelling the CRC computation that is used by `protect`, it is straightforward to write a model for `protect`. The model consists of approximately 40 lines of Erlang code. The property compares the result of calling the C implementation with the result of calling the model.

The input of `protect` has three parts: the configuration, the counter, and the data to be sent. An immediate observation is that the parameter value space is huge. Depending on the configuration, the data is up to 30 bytes long. Thus, it is not feasible to test all combinations of input parameters. Is there a way to reduce the value space? An important property of the `protect` function is that it should **not** change the data to be sent. It should only append a CRC checksum. Thus, not expecting the data to change, we simply generate random data bytes while we are studying the other parameters, and checking that the data does not change during the function invocation.

The configuration cannot change much. Basically, we can choose data length, data id, and the counter ranges from 0 to 15. Using a similar argument as for the data, we

simply select a data id at random. For the data length and counter value, we fully explore all combinations.

3.2 Modelling check

Just like for `protect`, we assume that the CRC checksum functionality is already modelled and verified. During reception, the checksum (thereby implicitly verifying the data id, etc.) and sequence number are checked. In some configurations it is allowed to lose a message or two in the sequence checking, therefore a state is needed in the QuickCheck model. This makes the model a bit more complicated, but in total the model is less than 100 lines of Erlang code.

The `check` function takes three arguments: the configuration, the current receive-state, and the data to be checked. The configuration and the data are the same as in the `protect` function. The receive-state structure is typical for C; it contains both in and out parameters: four in parameters and two out-parameters. The input consists of two 4-bit values and two Boolean values; giving a fairly small set of possible input parameter combinations. Combined with the data size in the configuration, we have no more than 26800 different combinations. Further, there is another 4-bit counter, and another Boolean parameter: the message sequence number (part of the data), and whether data has been corrupted or not.

In total there are 860160 different input combinations, and this number is small enough to test all combinations, since it only takes a few minutes to test all of them..

3.3 Justifying fault model

The argumentation for the assumed fault model (that actual data and data id does not change) build upon several different observations. The first observation is that we presuppose that the CRC checksum has already passed functional tests, which in itself requires a thorough validation effort. The second observation is that, and this argument should be supported by more precise fault models, the data is just read in the `protect/check` functions, and **not** changed. The same holds for the data id, it is only used in the CRC computation. Therefore, the risk should be low that the data and/or the data id are used incorrectly. In addition, in the end we are going to use 860160 random arrays of data (and 860160 randomly selected data ids), i.e. we are going validate these assumptions by testing a reasonable number of different combinations.

4 Conclusions

In this paper we address the challenge of arguing for software safety. Although code coverage measures are often prescribed by safety standards, it has been observed in literature and practice that this is an insufficient measure for safety integrity assurance. Code coverage is mainly a technique to spot uncovered areas of the code. In a

safety argumentation we propose to go beyond code coverage by using state coverage of a specification model. Input data is part of the state.

Our method is to first define both a fault model and an abstract functional model of the system. We then argue that 100% state coverage of the abstract functional model is sufficient for safety with regard to the defined fault model. The abstract functional model is used to automatically generate random tests cases from, which are used to test the software. Each test case covers part of the state of the abstract model. A sufficient test suite covers 100% of the abstract model. If all tests pass, we conclude that the implementation fulfils our safety requirements.

An example from the AUTOSAR domain indicates that our approach is feasible and effective.

Acknowledgements
We acknowledge VINNOVA for its support of the AcSäPt project (2012-00943).

5 References

[1] IEC, "Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 3: Software req.," IEC 61508-3:2010, 2010.

[2] ISO, "Road vehicles — Functional safety — Part 6: Product development at the software level," ISO 26262-6:2011, 2011.

[3] RCTA, "Software Considerations in Airborne Systems and Equipment Certification," DO-178B, 1992.

[4] M. Staats, et. al., "On the Danger of Coverage Directed Test Case Generation," in *Proc of FASE 2012*, 2012, pp. 409-424.

[5] D. Vanoverberghe, et. al., "State Coverage: Software Validation Metrics beyond Code Coverage," in *Proc of SAFSEM 2012*, 2012, pp. 542-553.

[6] D. Zowghi and V. Gervasi, "The Three C's of Requirements: Consistency, Completeness, and Correctness," in *Proc. of Int. Workshop on Requirements Engineering: Foundations for Software Quality*, 2002, pp. 155-164.

[7] Rolf Johansson et al., "A Road-Map for Enabling System Analysis of AUTOSAR Based Systems," in *Proc of the 1st Workshop on Critical Automotive applications: Robustness & Safety*, 2010.

[8] Thomas Arts, et al., "Testing Telecoms Software with Quviq QuickCheck," in *Proc of the ACM SIGPLAN Workshop on Erlang*, 2006.

[9] AUTOSAR, "Specification of SW-C End-to-End communication protection library," 2012.