Functional Requirements-Based Automated Testing for Avionics

Youcheng Sun Martin Brain Daniel Kroening
University of Oxford, UK

Andrew Hawthorn Thomas Wilson Florian Schanda
Altran, UK

Francisco Javier Guzmán Jiménez Simon Daniel
Rolls-Royce, UK

Chris Bryan Ian Broster
Rapita Systems, UK

***Abstract*—We propose and demonstrate a method for the reduction of testing effort in safety-critical software development using DO-178 guidance. We achieve this through the application of Bounded Model Checking (BMC) to formal low-level requirements, in order to generate tests automatically that are good enough to replace existing labor-intensive test writing procedures while maintaining independence from implementation artefacts. Given that existing manual processes are often empirical and subjective, we begin by formally defining a metric, which extends recognized best practice from code coverage analysis strategies to generate tests that adequately cover the requirements. We then implement it in automated requirements testing procedure that is applied in case studies with industrial partners. In review, the toolchain developed here is demonstrated to significantly reduce the human effort for the qualification of software products under DO-178 guidance.**

I. Introduction

DO-178C/ED-12C [1] provides guidance for the production of software for airbone systems. For each process of the life-cycle it lists the objectives for the life-cycle, the activities required to meet those objectives and explains what evidence is required to demonstrate that the objectives have been fulfilled. DO-178C is the most recent version of the DO-178 family. DO-178 is thorough, and when the highest level is chosen, aids the production of highly reliable software.

Developing to this guidance can be hard and time consuming, due to the rigid safety level it must ensure. A typical DO-178C test coverage analysis has two steps: 1) the first step is analysis of the coverage of requirements, and 2) the second step is structural coverage analysis on the implementation.

The requirements-based coverage analysis establishes whether the software requirements are covered adequately, which may prompt a revision of the requirements-based tests. Subsequently, structural coverage analysis is applied to determine which part of the code is exercised by the requirements-based tests. Inadequate structural coverage in the second step indicates that requirements are missing or that implementation behaviour is unspecified. To this end, requirements-based testing derives a suite of tests from software requirements only, and must not use internal structure of the implementation.

a) Structural Coverage Analysis: The most common form of structural coverage analysis is **code coverage analysis**, which measures the degree to which the source code of a program has been covered during execution. Criteria for code coverage include statement coverage (checking whether each statement in the program has been executed), branch coverage (checking whether each branch of conditional structures in the code has been taken), and Modified Condition/Decision coverage (MC/DC) [2]. In MC/DC analysis, a boolean decision consists of multiple boolean conditions such that every condition (predicate) shall be evaluated to true and false and it is required that this switch changes the outcome of the final decision. DO-178C guidance requires MC/DC coverage of function bodies.

b) Automating functional requirements-based testing: Common requirements-based testing techniques [3] include equivalence partitioning, boundary value analysis, decision tables, and state transition testing. Equivalence partitioning and boundary value analysis are most relevant to the coverage criterion we propose. Equivalence class partitioning is a software testing technique that partitions each of the inputs (or outputs) to the unit under test into a finite number of disjoint equivalence classes. It can also be applied to the inputs to a conditional statement. It is usually used in conjunction with boundary value analysis. Boundary value analysis is a technique that generates test cases that exercise an input or predicate (conditional) variable just below, on, and just above the limits of valid ranges (equivalence partitions), the rationale being that errors tend to occur near, or on the partition boundaries.

The key goal of this paper is the automated creation of a functional test suite from formal, low-level requirements. Software requirements are usually written in natural language and for our approach these must then be translated into a formal specification in the form of pre- and post-conditions. The pre-conditions capture the calling context required before the function call, and the post-conditions capture the state of the system required after the function call.

There is a variety of approaches to automating specification-based testing. In particular, [4] is among first that applied Model Checking to generate tests from requirements specifications. For each condition tested, a *trap property* that violates this condition is generated and instrumented into the code: if a trap property is satisfied by the model checking, it means that the corresponding condition will be never met by the program;

otherwise, a counterexample will be returned by the Model Checker that leads the program to the condition tested and a test vector can be thus derived. However, the method in [4] needs to call the function body in a black box fashion. In [5], structural coverage is applied to function specifications to generate testing conditions, which are then automatically matched with tests that already exist. [6] summarizes test selection strategies when using formal specifications to support testing. In such cases, post-conditions are only used to determine the outcome of the tests that are generated.

II. The Problem Formulation

The work in this paper targets the scenario depicted in Figure 1, with our focus on using automated low-level requirements (LLRs) testing to minimize human effort. Within such a scheme, after high-level requirements (HLRs) are (mostly) manually interpreted as low-level requirements (LLRs), the function implementation (Impl) and the requirements testing are two independent procedures. The function implementation can be either manually written by software programmers or automatically generated by source code generation tools as in many model-based development platforms. In the end, test cases generated from LLRs must be validated against the function implementation (e.g., by code coverage level measurement), which are supplied as evidence for fulfilling DO-178C guidance.

Figure 1 depicts a suitable scheme for developing avionics software under DO-178C. The work in this paper specializes in the LLRs testing, which is nowadays performed manually by experienced engineers. We aim to automating this procedure, that is, automatically creating test cases for LLRs.

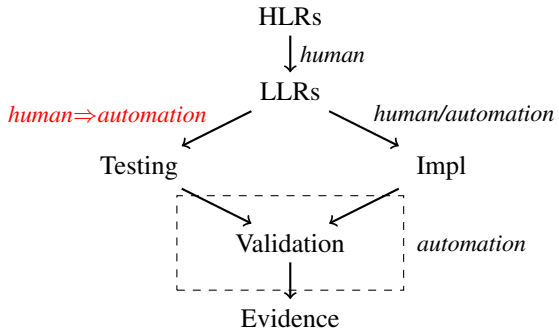

Fig. 1: The targeted software development scheme

Given the authors' experience in the field, the cost of developing tests against low-level requirements (as required by the objectives in DO-178C) equates to, on average, two days per procedure/function. This takes account of initial test development, test debugging and review. For a project with 250,000 lines of code and assuming an average size of 25 lines per procedure/function, this equates to 50 person-years of effort! Given this level of effort and expense, coupled with the generally held view that low-level testing is of relatively low value, there is a strong commercial incentive to consider ways of automating this aspect of development. The DO-333 Formal Methods Supplement [7] to RTCA DO-178C provides an approach to automating low-level verification through the use of proof, but few businesses are considering this option because of the perceived cost of using formal methods. This is because verification results from a formal methods tool (e.g., a Model Checker) can be used under DO-178C and DO-333 guidance, only if that tool has been certified or the verification process has been instrumented to generate the evidence. Either way is challenging.

The research described in this paper has used formal methods in another way: instead of mathematically proving the software, we use proof to generate tests that cover low-level verification objectives that are required in DO-178C. This approach features an *easy adoption*. The automated tests represent a like-to-like replacement for existing (manual) test-writing procedures. Generated tests can be reviewed in the same way as those written by a skilled engineer without a background in formal methods.

III. The Requirement Specification

For both the convenience and generality, we assume that every requirement is specified as a Boolean expression `bool_expr`, following the mini syntax in Listing (1).

```
bool_expr ≝ true | false | bool_var | ¬bool_expr
          | bool_expr ∧ bool_expr
          | bool_expr ∨ bool_expr
          | nbe {<, ≤, =, ≥, >, ≠} nbe
          | ITE(bool_expr, bool_expr, bool_expr)

non_bool_expr ≝ int_expr | float_expr
              | string_expr | char_expr | enum_expr
              | ITE(bool_expr, nbe, nbe)
```

Listing 1: The mini syntax of a requirement

This mini syntax does not aim to be comprehensive, and most parts of it are straightforward. The `ITE` is an if-then-else style operator like the ternary operator (? :) in C language, and *nbe* is short for *non_bool_expr*.

In addition, a *Boolean predicate* is defined as a Boolean expression of the form $nbe \sim nbe$ with $\sim \in \{<, \leq, =, \geq, >, \neq\}$. A Boolean predicate $nbe \sim nbe$ is also called *an ordered predicate* if $\sim \in \{<, =, >\}$. A *compound Boolean expression* is with multiple conjunctions or disjunctions of its Boolean sub-expressions.

IV. The Coverage Criterion

We now formally define a coverage criterion, to analyze and generate the set of testing conditions that meaningfully interpret every requirement, specified as in Listing 1. It extends MC/DC with boundary value analysis (for compound expression analysis) and takes account of control flow structures in the requirement specification.

a) The compound expression: Our treatment of compound expressions is similar to [8], which uses MC/DC together with boundary value analysis. However, we provide a formal definition of rules for generating the testing conditions.

In code coverage analysis, the MC/DC criterion is often regarded as a good practice to generate testing conditions

to examine a compound expression. But MC/DC is not immediately applicable to software specifications, as it does not support equivalence class partitioning or boundary value analysis. Thus, subsequently to plain MC/DC, we perform two additional steps to obtain a larger set of testing conditions.

At first, we define $neg\text{-}free$ such that every negated predicate, in form of $e = \neg(nbe_1 \sim nbe_2)$, is converted into its negation free forms:

$$neg\text{-}free(e) = \begin{cases} \{nbe_1 = nbe_2, nbe_1 > nbe_2\} \text{ if } \sim \text{ is } < \\ \{nbe_1 > nbe_2\} \text{ if } \sim \text{ is } \leq \\ \{nbe_1 < nbe_2, nbe_1 > nbe_2\} \text{ if } \sim \text{ is } = \\ \{nbe_1 < nbe_2\} \text{ if } \sim \text{ is } \geq \\ \{nbe_1 = nbe_2, nbe_1 < nbe_2\} \text{ if } \sim \text{ is } > \\ \{nbe_1 = nbe_2\} \text{ if } \sim \text{ is } \neq \end{cases}$$

Furthermore, non-ordered predicates, in form of $e = nbe_1 \sim nbe_2$, are eliminated by $to\text{-}ordered$ operation:

$$to\text{-}ordered(e) = \begin{cases} \{nbe_1 < nbe_2, nbe_1 = nbe_2\} \text{ if } \sim \text{ is } \leq \\ \{nbe_1 > nbe_2, nbe_1 = nbe_2\} \text{ if } \sim \text{ is } \geq \\ \{nbe_1 < nbe_2, nbe_1 > nbe_2\} \text{ if } \sim \text{ is } \neq \end{cases}$$

The motivation behind this is that an ordered predicate typically represents one equivalence class for variables involved.

By recursively calling $neg\text{-}free$ and $to\text{-}ordered$, they can be naturally extended to apply to any compound Boolean expression.

As an example, let us consider the Boolean expression $M \geq 0 \wedge N \leq 1000$. Following MC/DC, the set of testing conditions generated is $\Phi = \{M \geq 0 \wedge N \leq 1000, \neg(M \geq 0) \wedge N \leq 1000, M \geq 0 \wedge \neg(N \leq 1000)\}$. The two colored Boolean expressions in Φ are not negation free. By applying the $neg\text{-}free$ operation, we obtain $\Phi' = \{M \geq 0 \wedge N \leq 1000, M < 0 \wedge N \leq 1000, M \geq 0 \wedge N > 1000\}$, where the colored predicates are due to $neg\text{-}free$. At last, after expanding all non-ordered predicates, the finally elaborated set of testing conditions is $\Phi'' = \{M = 0 \wedge N = 1000, M = 0 \wedge N < 1000, M > 0 \wedge N = 1000, M > 0 \wedge N < 1000, M < 0 \wedge N = 1000, M < 0 \wedge N < 1000, M > 0 \wedge N > 1000, M = 0 \wedge N > 1000\}$.

It is common to use a *tolerance level* σ for boundary value analysis. The value of tolerance level is based on the precision of the number representation and the requirements for the accuracy of calculation. This combines well with with our coverage criterion. In order to integrate this tolerance level, we extend the $neg\text{-}free$ and $to\text{-}ordered$ such that every time a testing condition with a predicate of the form $nbe_1 < nbe_2$ (resp. $nbe_1 > nbe_2$) is obtained, an extra testing condition with this predicate replaced by $nbe_1 = nbe_2 - \sigma$ (resp. $nbe_1 = nbe_2 + \sigma$) is added to the set.

For any Boolean expression e, we use $\Phi(e)$ to denote its set of testing conditions after $neg\text{-}free$ and $to\text{-}ordered$ operations, following the MC/DC phase. Besides, $\Phi(e, +)$ contains all these testing conditions representing that the decision of e is true and $\Phi(e, -)$ is for e to be false such that $\Phi(e) = \Phi(e, +) \cup \Phi(e, -)$.

b) The ITE expression: Given an if-then-else (ITE) expression $ITE(e_1, e_2, e_3)$, the coverage analysis of e_2 (resp. e_3) will be only relevant if e_1 is true (resp. false). Therefore, we define the resultant testing conditions from an ITE expression e as follows.

$$\begin{aligned} \Phi(e) = &\{e' | \exists x \in \Phi(e_1, +)\ \exists y \in \Phi(e_2) \text{ s.t. } e' = x \wedge y\} \\ &\cup \{e' | \exists x \in \Phi(e_1, -)\ \exists y \in \Phi(e_3) \text{ s.t. } e' = x \wedge y\} \end{aligned}$$

This is how the control flow information of a requirement is taken into account for coverage analysis.

V. The Case Study

The coverage criterion in Section IV has been implemented in the Model Checking tool CBMC [9] to analyze function requirements and automatically generate test vectors. There are a number of works (e.g., [4]) that implement or integrate different kinds of coverage analyses with Bounded Model Checking. We have adopted a similar approach that embeds trap properties (i.e., negation of testing conditions) into the program model of these requirements. Because we aim to independently examine the requirements specified as pre-/post-conditions, the function implementation is simply ignored, whereas a proper over-approximation is also admissible depending on the application scenario.

As depicted in Figure 1, DO-178C certification requests evidence from both requirements coverage analysis and structural coverage analysis. Thus, CBMC has been integrated with RapiTestFramework, which is a GUI environment for subprogram selection, test compilation and execution, and review of results. The resultant toolchain was developed in the AUTOSAC (Automated Testing of SPARK Ada Contracts) project. A video demonstration of the toolchain (using open-source code under test) is available online[1].

This toolchain's performance and ability to replace manual testing effort is confirmed through the process of generating and executing tests for selected code made available by Rolls-Royce.

a) Case study results: The software architecture behind the case studies is composed of two layers. The first one is an application layer (AL) and the second one, is an operating layer (OL). In order to use the toolchain in a real environment, two case studies have been picked, one per each layer in order to observe how the tool works with different abstract levels. Concretely, a health monitoring component for the AL case study and a temperature sensor component for the OL case study, both written in Ada, have been chosen.

After both components have been isolated from non-related code, to make the toolchain work, the legacy code has to be modified to represent the requirements with the component/-functions by the use of SPARK Ada contracts [10], specifically, the use of SPARK pre- and post-conditions.

By applying the AUTOSAC toolchain, tests are automatically generated from the SPARK Ada contracts. We check these resultant tests, by measuring the corresponding statement code coverage level on each subprogram: $AFA(100\%)$, $AST(100\%)$,

[1] https://youtu.be/72NWFZQOvlM

$CTE1(100\%)$, $CTE2(100\%)$, $CTEC(100\%)$, $CVO(100\%)$, $FES(100\%)$, $ISCD(100\%)$, $ISCF(100\%)$, $ISCC(100\%)$, $RTC(68.75\%)$.

Overall, the automatically generated tests show good coverage and represent a good spectrum of the case scenarios to be tested. The gap on code coverage level in some cases (e.g., RTC from the OL) required further refinements on the SPARK Ada contracts under test and investigations on the intermediate language representation for SPARK Ada contracts in the toolchain.

Importantly, the verification role will retain its skillset, yet remove monotonous test scripting, as a result of writing SPARK Ada contracts instead of specific test harness. Besides writing such contracts, we believe minimal human observation for the review of the automatically generated tests is required.

The case study undertaken during the AUTOSAC research project demonstrated the successful implementation of automated testing in a real-world environment, integrating and extending existing tools for achieving DO-178-level certification. The toolchain has become one of the candidates for Rolls-Royce to reduce costs for verification, which is of great interest. We anticipate that it will undergo further development, including adding support for mixed-language test projects.

b) Experience and lessons learned: The main obstacle we encountered during the toolchain development phase is the intermediate representation of SPARK Ada. Regarding the automated tests generation, its implementation is based on CBMC, which does not have a direct front end for Ada. Thus, these requirements specified as in SPARK Ada contracts are at first converted into C language form, from which tests are then automatically generated. This choice is due to the limited timescale of the AUTOSAC project and is also for the purpose of evaluating the feasibility of such a requirements-based automated testing methodology. However, certain features in Ada cannot be trivially represented in C and the conversion procedure from Ada to C needs to be very carefully maintained. Currently, there is a continuing collaboration between partners to develop a formal Ada/SPARK front end for CBMC, thanks to the promising results shown from AUTOSAC toolchain for the automated test generation from software requirements.

Notwithstanding this limitation, the case study presented here illustrates the success of the tools and approach developed during the AUTOSAC project. In particular, the successful integration of CBMC and RapiTestFramework demonstrated the powerful combination of test generation based on mathematical proof combined with a robust platform for test execution and review. It is expected the reduction in time and cost of low-level requirement validation described in this paper will begin to be achieved as this technology is adopted in large-scale commercial projects.

Meanwhile, the use of the AUTOSAC toolchain encourages software teams to adopt more efficient development practices, as quick tests of the code can be done while maintaining independence from implementation, something that is currently not possible within the constraints of the DO-178 guidance. Such an approach is preferable and it allows the same person to perform two roles on different modules. The first one is the verification role, which would initially make the specification of some modules using SPARK Ada contracts. The other one is the design role, which would design/implement a different software module, independently from the SPARK Ada contracts written by another person.

VI. Conclusions

In this paper, we successfully demonstrated feasibility of automatic test case generation from functional requirements, targeting software testing in avionics. The developed toolchain is applied to industrial case studies, and its applicability and usefulness are confirmed.

Regarding future exploitation, we are interested in investigating application of the technique in this paper to test case chains generation for reactive systems [11]. In particular, testing of model-based development processes (e.g., testing Simulink systems [12], [13], [14]) is of great relevance to the context of avionics software certification.

Acknowledgement

The work presented in this paper was sponsored by Birmingham City Council and Finance Birmingham and supported by Farnborough Aerospace Consortium (FAC) under NATEP project FAC043 – Automated Testing of SPARK Ada Contracts (AUTOSAC).

References

[1] RTCA, "DO-178C, Software considerations in airborne systems and equipment certification," 2011.

[2] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A practical tutorial on modified condition/decision coverage," 2001.

[3] S. C. Reid, "BS 7925-2: The software component testing standard," in *Quality Software. Proceedings. First Asia-Pacific Conference on*. IEEE, 2000, pp. 139–148.

[4] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *Software Engineering—ESEC/FSE*. Springer, 1999, pp. 146–162.

[5] J. Chang and D. J. Richardson, "Structural specification-based testing: automated support and experimental evaluation," in *Software Engineering—ESEC/FSE*. Springer, 1999, pp. 285–302.

[6] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause *et al.*, "Using formal specifications to support testing," *ACM Computing Surveys (CSUR)*, vol. 41, no. 2, p. 9, 2009.

[7] RTCA, "DO-333, Formal methods supplement to DO-178C and DO-278A," 2011.

[8] J. J. Chilenski, "An investigation of three forms of the modified condition decision coverage (MCDC) criterion," DTIC Document, Tech. Rep., 2001.

[9] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2004, pp. 168–176.

[10] B. John, "SPARK: The proven approach to high integrity software," 2012.

[11] P. Schrammel, T. Melham, and D. Kroening, "Generating test case chains for reactive systems," *Software Tools for Technology Transfer (STTT)*, vol. 18, pp. 319–334, June 2016.

[12] A. Brillout, N. He, M. Mazzucchi, D. Kroening, M. Purandare, P. Rümmer, and G. Weissenbacher, "Mutation-based test case generation for Simulink models," in *Formal Methods for Components and Objects (FMCO) 2009*, ser. LNCS, vol. 6286. Springer, 2010, pp. 208–227.

[13] N. He, P. Rümmer, and D. Kroening, "Test-case generation for embedded Simulink via formal concept analysis," in *Design Automation Conference (DAC)*. ACM, 2011, pp. 224–229.

[14] A. Balsini, M. Di Natale, M. Celia, and V. Tsachouridis, "Generation of Simulink monitors for control applications from formal requirements," in *Industrial Embedded Systems (SIES), 12th IEEE Symposium on*, 2017.