Automatic Test Cases Generation From Formal Contracts

Extended abstract

Samuel Jiménez Gil[1][0000−0002−1632−2018], Manuel I Capel[2][0000−0003−2449−4394], and Gabriel Olea Olea[2][0000−0003−4596−1991]

[1] SatixFy Space Systems UK, Trident Unit 2, Styal Road, Manchester, M22 5XB, UK
Samuel.Gil@satixfy.com
[2] ETSIIT, University of Granada, 18071 Granada, Spain
manuelcapel@ugr.es, gabrieloo@correo.ugr.es

Abstract. Software verification is probably the greatest software engineering challenge for dependable systems and its associated costs grow with the code size. The use of proofs and tests are the most popular approaches for evaluating the correctness of a piece of software. Proofs require pre and postconditions for analysis, which are derived from a contract design, whereas software tests run test cases for specific instances. The availability of these contracts is particularly relevant to argue about the traceability of the verification activities. In our work, we use the formal contracts implemented in Spark/Ada programs to generate test cases, in this way, we use the most accepted methods for test generation but leveraging the contracts to achieve these high quality standards. The evaluation, which uses a wide array of open source examples of Spark contracts, shows a high level of statement coverage and how some contracts are more friendly to perform this test generation than others.

Keywords: Automatic test cases generation · software testing · formal methods · software verification

1 Introduction and objective

Software verification is a challenging and labor intensive task for dependable systems that may reach around 60% of the overall software development cost. Thus, the software industry has shown a significant interest in reducing this cost being automation the most promising solution. Software verification can be carried out either by code *review*, *proof* or *testing* [1].

Code review is normally the least preferred approach as it involves some manual and sometimes informal justification. On the one hand, software proof applies a deductive reasoning by means of formal specification analysis [1]. Such formal specifications generally consist of preconditions that input data must meet and output data postconditions. These verification conditions along with the implementation are used by a *theorem prover* to determine whether postconditions

are always satisfied.

On the other hand, software testing is an intrinsic deductive reasoning which consists of collecting observations from the real hardware or an equivalent simulator. This approach is based on the construction of test cases from the input data space, paired with their expected outputs which are compared with its actual outputs to determine whether the test passes or fails. The resulting evidence serves to argue about the confidence in the program correctness.

Proof and testing have their own pros and cons, a software proof is able to conclude correctness for all cases, but this requires a 'design by contract', which can be costly and requires a substantial effort from the point of view of a company value generation. In addition, the analysis of software correction performed by proofs is based on some assumptions about the hardware, which may not be always true, e.g., absence of hardware bugs or memory stability, which can suffer corruption due to radiation.

By contrast, software testing is able to collect representative observations from a real hardware platform. Nonetheless, it cannot generally cover the entire input space leaving some use cases or pieces of code untested. Despite so, software testing is often deemed the *gold-standard* in at least the aerospace domain [2].

Automated approaches to generating test input data have gained momentum in recent years, and only a few works focus on generating true test cases, i.e., providing input data and, in addition, expected results in the form of test cases or test suites [5]. Most of these approaches are code-driven, meaning that source code is read to generate only test data in a non TDD-methodological way, unstructured manner. From our point of view, these approaches are not acceptable for certain software quality standards, as they cannot be traced back to requirements.

The use of *formal contracts* provides the required traceability to requirements. Moreover, the technology implemented in Spark/Ada programming language – perhaps the most popular programming language for safety-critical applications – enables to integrate the formal contracts inside the specification of functions or procedures.

We hypothesize that 'design by contract' effort can be useful for both testing and proving dependable software. The objective of our work is to investigate a method to use these formal contracts to derive test cases using equivalence class partitioning supplemented with parameter domain boundary analysis and negated precondition tests, resulting in a solid testing method for achieving software quality standards.

2 Software Testing Current Trends

Software testing is a broad area and when it comes to requirements-driven testing, probably the most popular testing techniques are: *equivalence class partitioning*, *boundary analysis*, *decision tables* and *combinatorial testing* [2]. To the best of our knowledge, software quality standards in the space [4], aerospace [2] and automotive [3] are comfortable with equivalence class partitioning augmented with boundary analysis [2]. This approach is based on spliting the input space of a program into disjoint sets depending on the expected results (output) and then choosing test cases located somewhere within and/or at the boundaries of these sets [2].

There are a few approaches to test vector generation [5] and amongst them, we deem Constraint-Based Testing as the most promising strategy for this problem domain given its accuracy. Constraint-Based Testing requires collecting constraints from the software under test so as to analyze them with a constraint solver later. This resulting solution (if there exists) is the test vector or test case.

The idea of combining test cases from Spark/Ada contracts is not new, in fact Comar et al. [7] investigate the idea of incorporating proof and testing as part of the verification system in Ada toolchain. Even though there are some interesting arguments about such an integration, the test cases in the single case study were produced manually.

Later on Sun et al. [6] apply a Constraint-Based Test Generation method using Modified Condition / Decision Coverage (MC/DC) with boundary analysis to derive test cases from Spark contracts. Unfortunately, their method does not seem to support universal or existential quantifiers usually available in Spark contracts. Their evaluation is also limited, given that only two industrial case studies were performed, no MC/DC in the software under test was measured, despite the capability of their tools, and the statistical significance could not be determined.

3 Method and Findings

Our approach starts from Constraint-Based Testing. Then, by reading the input data domain together with the precondition constraints, these are transformed to constraints to the input data or test vector. The postconditions are then mapped in similar way to the pass/fail criterion for the test cases. Based on these definitions, our test generation algorithm selects the test cases using equivalence class partitioning supplemented with parameter domain boundary analysis and negated precondition tests.

The evaluation measures the effect of our test generation in: 1) the resulting statement and branch coverage, which are common metrics for the completeness

of software testing, 2) the efficiency which is defined as the number of test cases per time unit, and 3) the number of pass/fail test cases. Additionally, a random test generator is included for comparison purposes.

The examples analyzed consists of representative open-source benchmarks exhibiting a wide array of formal contracts ranging from first order logic such as binary search to safety-critical signaling systems. The results display in general a high level of statement coverage from our method along with a high number of passed test cases. Efficiency is compared to that obtained using only the random test generator, and the results are discussed.

Lastly, some results are presented that complement the static analysis proof by showing how both techniques can work in conjunction. Another interesting aspect discussed in this section is why some formal contracts are more friendly for test generation than others and how these results can motivate more compatible contracts by design.

4 Conclusion

Automation is a very promising and probably the only solution to the escalation of software verifications costs. Automatic test cases generation is a major challenge that can help to alleviate this problem. Despite the substantial effort, a design by formal contracts not only can provide proof of code but it also provides traceability to the requirements. These contracts can be used to generate test cases as some works, and this one demonstrate.

Our approach has managed to establish a way in which formal contracts can generate test cases using a very accepted method in the software quality standards. Results show how our approach stresses the code substantially, the reason why some contracts are not that friendly to this test generation method and how proof and testing can work in conjunction.

References

1. John W. McCormick and Peter C. Chapin, Building High Integrity Applications with SPARK, Cambridge University Press, (2015).
2. Leanna Rierson, Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance, CRC Press, (2013).
3. ISO26262, Road Vehicles – Functional Safety, (2018).
4. ECSS-E-ST-40C, Space engineering: Software, (2009).
5. Samuel Jiménez Gil, Constraint-Based Testing and Tail Tests for Measurement-Based Probabilistic Timing Analysis. PhD thesis, University of York (2020).
6. Youcheng Sun et al, Functional Requirements-Based Automated Testing for Avionics. CoRR, (2017)
7. Cyrille Comar et al, Integrating Formal Program Verification with Testing, Open-Do. 2012.