Presentation: An application of KLEE to aerospace industrial software

JUAN FRANCISCO GARCÍA, DANIEL JURJO, FERNANDO MACÍAS, JOSÉ F. MORALES, and ALESSANDRA GORLA, IMDEA Software Institute, Spain

1 CONTEXT AND MOTIVATION

This work is framed in the MFoC project[1]. The general goal of the project is to explore hardware and software solutions to make the development of aerospace systems more cost-effective by lowering the development time, while maintaining high levels of reliability. In particular, our work so far has been oriented towards improving the testing workflow of an international aerospace company with several goals: automatise test input generation, improve test coverage, minimise test suites and streamline testing-related continuous integration tasks, i.e. regression testing. In this abstract we present the details of our approach to achieve the first two goals by using KLEE and the advantages of the Model-Driven Development (MDD) process used in the company. It is important to note that we cannot disclose the name of the company—so we keep using "the company" to refer to it in the remainder—neither the actual artefacts or very specific details due to a non-disclosure agreement.

Testing is a critical task in aerospace software development. The nature of the missions makes it hard or even impossible to debug or patch the software once the mission is launched. Moreover, a software failure that leaves the system unable to recover or to receive input from the ground is likely to result in a failure of the mission, causing a huge economic loss. To ensure that aerospace software is thoroughly tested, authorities like the European Space Agency impose severe requirements. This in turn increases the amount of resources that must be allocated for testing tasks during the development of such software, compared to other areas.

In the company we collaborate with, unit testing is used as the main source of verification. Test harnesses are used to automate the execution of tests and report the results, but test generation is still a mostly manual task, guided by structural coverage. In particular, statement (every instruction is executed at least once by one of the tests) and branch (every branch is executed at least once by one of the tests) coverage are used. An initial set of tests is created using a process called "ramping", where values are given for each input within a range (decided according to requirements and expertise of the engineer). Based on the coverage of this first set, engineers manually provide additional inputs to incrementally improve coverage until it reaches 100%.

The company uses MDD, meaning that the C code that is deployed is automatically generated from Simulink models using strict rules to ensure that it adheres to code standards like MISRA-C. The same models are also subject to exhaustive, worst-case simulations to ensure their correctness. Once this process is completed, these models serve as a reference for expected outputs of the unit tests in the generated code, i.e. the models become the test oracle for the C code. Therefore, the goal of the testing process is to ensure that the behaviour of the code predicted by the models is the same as the one exhibited by the code. Models and automatically generated code may exhibit different behaviour because of third-party libraries, hardware and configuration, which are sources of variability that may compromise the reliability of the code. The discussion regarding the process of verification of the models is out of scope of this work, so we also use the outputs predicted by the models as oracles in our approach.

In the following section, we illustrate our initial approach using KLEE in this project.

[1] https://flightonchip.es/

Authors' address: Juan Francisco García, juanfrancisco.garcia@imdea.org; Daniel Jurjo, daniel.jurjo@imdea.org; Fernando Macías, fernando.macias@imdea.org; José F. Morales, josef.morales@imdea.org; Alessandra Gorla, alessandra.gorla@imdea.org, IMDEA Software Institute, Campus Montegancedo s/n, Pozuelo de Alarcón, Madrid, Spain, 28223.

2 USING KLEE FOR TEST GENERATION

To understand how we managed to use KLEE successfully with the AOCS code, we must first comment on the features of that C code that we had to deal with. First, all the loops in the code of the component we tested are bounded by an integer constant, usually of a very low value, e.g. used to traverse a fixed-length array. Also, recursion is not used anywhere in the code, and the branching conditions are based in most cases on the values of Boolean flags. All these features make this software a perfect subject for achieving path coverage with KLEE, which provides better confidence than branch coverage, since exploring all paths subsumes the exploration of all branches. It is also worth mentioning that all constant values are declared in hierarchical C structs, which makes analysis and parsing easier. On the other hand, the software component also uses a small number of branch conditions based on floating-point arithmetic, usually normalised. The consequences of this fact are discussed in Section 4.

The development and testing workflow that we designed with our industrial partner in the project account for the following steps: **(1)** Build a Simulink model; **(2)** Verify the model using physics simulations; **(3)** Generate C code from the model; **(4)** Parse the C code and generate symbolic function calls; **(5)** Run KLEE to generate the test inputs; **(6)** Run the test inputs on the code and get the outputs; **(7)** Run the test inputs on the model (requires an adapted version) and compare with the output from the code.

Steps (1) to (3) are not discussed, since they remain unmodified from the company's original workflow. Hence, the key to our approach are steps (4) to (7), which have been automatised in a one-click fashion, as well as the adaptation of the Simulink model to use it as oracle. The current prototype implementation is based on a Python and bash driver scripts, and a simplified C parser tailored to the restricted subset of C necessary in this case study.

3 EXPERIENCES AND EARLY RESULTS

We tested the workflow presented in Section 2 in two laptops, both with an Intel Core i5 CPUs and 8 GB RAM. The execution time for

the whole process explained in the previous section is in the order of magnitude of seconds, which extrapolated to the full AOCS system would keep it in the order of magnitude of minutes. Besides, our process has shown better results than human-made tests, since it reduces both bias and dependence on expertise. Moreover, the test suite is heavily minimised, while maintaining the highest coverage even with a strict metric like path coverage. In one of the modules of the AOCS for which we had the original test suite, KLEE just needed 54 tests to achieve path coverage, whereas the ramping-based together with the manual test generation of the company yielded more than 6000 tests. This fact does not mean that the test suite could not be extended if necessary, based on variable ranges specified in requirements (see Section 4). We believe that our process can run in parallel or as an enhancement of the company's current approach. Specifically, we believe that our solution could be employed to generate tests daily, in order to save time and manual work of engineers. The solution currently employed by the company could still be used every now and then, for instance on a major release or when it is time to certify the software.

4 CHALLENGES AND LIMITATIONS

As said before, we found mixing floating-point with other data types a challenge for test generation. In our experience, extensions of KLEE to support floats (e.g., KLEE-Float) works efficiently when the model is small, but it seems to suffer from path explosion and inefficiencies on the bigger ones, potentially due to the bit-blasting approach of Z3 to solve floating point constraints (known to be good to find counter-examples). On the other hand, other theories like real arithmetic (unbounded rational numbers) are not applicable in this scenario where accurate floating point modeling is mandatory.

For those cases where a white-box may not scale for larger programs, or is impractical due existence of closed software/hardware components, we are considering a mixed approach where KLEE is combined with a black or grey-box steps (fuzzers). This has the added potential of detecting divergent behaviors in components such as the Simulink simulator, the different implementations of floating point libraries, or even the actual hardware. In practice this is one of the main challenges for testing-based code certification in our context. On the technical side, we found that coverage-guided fuzzers like `libfuzzer` are easy to integrate and customize in our KLEE workflow (Section 2) via corpus sharing and custom mutation.

5 RELATED WORKS

Researchers have proposed several techniques to automatically generate test cases from Simulink models. Gonzalez et al. propose a SysML-based modeling methodology for model testing of Cyber-Physical Systems, and an efficient SysML-Simulink cosimulation framework to conduct testing at early stages and over executable models [3]. Liu et al. aim to improve fault localization in automotive Simulink models by adding new automatically generated test cases. They resort to a four-objectives search-based algorithm to achieve diversity in the test suite, and to keep it minimized at the same time [5]. This technique follows the same intuition of a previous work by the same research group [7]. The key contribution is a test generation approach applicable to Simulink models which resorts to a meta-heuristic search to produce test outputs signals that differ as much as possible. Holling et al. present an automatic test case generator for generated C code that aims to identify over-/underflows and divisions by zero failures that may not occur when simulating the original Simulink model [4].

Other works focus on test oracles. Matinnejad et al. visualize the controller behavior over its input space to help the manual work of engineers during testing [6]. Menghi et el. use Signal First Order logic to specify requirements on Simulink models that can later be checked automatically at runtime [8].

Out of the CPS domain, several approaches can automatically produce test inputs to achieve high structural coverage using symbolic and concolic execution, search based algorithms, and random/fuzzing strategies [1, 2, 9, 11].

6 CONCLUSIONS AND FUTURE WORK

We report here our initial experiences on the application of KLEE to industry-level aerospace software. We achieved positive results in the comparatively small amount of tests generated that still provides better coverage than the original ones, and also in the high degree of automation and the short execution times of our approach. Some initial attempts with KLEE-Float and fuzzing have also provided promising results.

We envision several lines of future work. First, we plan to evaluate the combination of KLEE with different fuzzers and configurations. Second, we plan to keep experimenting with KLEE-Float for the specific cases in our code where branch conditions are based on floating-point arithmetic. And third, we would like to improve the oracles by not relying on the comparison of outputs, but rather defining a memory-based equivalence relation [10].

REFERENCES

[1] Gordon Fraser and Andreas Zeller. 2011. Generating parameterized unit tests. In *ISSTA 2011*. 364–374.
[2] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *PLDI 2005*. 213–223.
[3] Carlos A. González, Mojtaba Varmazyar, Shiva Nejati, Lionel C. Briand, and Yago Isasi. 2018. Enabling Model Testing of Cyber-Physical Systems. In *MODELS 2018*. 176–186.
[4] Dominik Holling, Alexander Pretschner, and Matthias Gemmar. 2014. 8Cage: lightweight fault-based test generation for simulink. In *ASE 2014*. 859–862.
[5] Bing Liu, Shiva Nejati, Lucia, and Lionel C. Briand. 2019. Effective fault localization of automotive Simulink models: achieving the trade-off between test oracle effort and fault localization accuracy. *EMSE* 24, 1 (2019), 444–490.
[6] Reza Matinnejad, Shiva Nejati, and Lionel C. Briand. 2017. Automated testing of hybrid Simulink/Stateflow controllers: industrial case studies. In *ESEC/FSE 2017*. 938–943.
[7] Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. 2016. Automated test suite generation for time-continuous simulink models. In *ICSE 2016*. 595–606.
[8] Claudio Menghi, Shiva Nejati, Khouloud Gaaloul, and Lionel C. Briand. 2019. Generating automated and online test oracles for Simulink models with continuous and uncertain behaviors. In *ESEC/FSE 2019*. 27–38.
[9] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *ICSE 2007*. 75–84.
[10] David A Ramos and Dawson R Engler. 2011. Practical, low-effort equivalence verification of real code. In *CAV 2011*. Springer, 669–685.
[11] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *ESEC/FSE 2005*. 263–272.