Boost the Impact of Continuous Formal Verification in Industry

Felipe R. Monteiro[1], Mikhail R. Gadelha[2], and Lucas C. Cordeiro[3]

[1]Federal University of Amazonas, Brazil, felipemonteiro@ufam.edu.br
[2]SIDIA Instituto de Ciência e Tecnologia, Brazil, mikhail.gadelha@sidia.com
[3]University of Manchester, UK, lucas.cordeiro@manchester.ac.uk

Abstract. Software model checking has experienced significant progress in the last two decades, however, one of its major bottlenecks for practical applications remains its scalability and adaptability. Here, we describe an approach to integrate software model checking techniques into the DevOps culture by exploiting practices such as continuous integration and regression tests. In particular, our proposed approach looks at the modifications to the software system since its last verification, and submits them to a continuous formal verification process, guided by a set of regression test cases. Our vision is to focus on the developer in order to integrate formal verification techniques into the developer workflow by using their main software development methodologies and tools.

Keywords: Formal Software Verification, Model Checking, DevOps.

1 Motivation

The formal verification community faces a pressing problem to ensure security and reliability of large codebases, which have a significant impact in millions of users [1]. Even minor defects can lead to huge impacts for companies and costumers [2], e.g., in September 2018 attackers exploited three Facebook vulnerabilities and stole access tokens from as many as 50 million users in order to take over their accounts [3]. In this particular context, software verification plays an important role in ensuring the overall product reliability. Even though formal techniques have been dramatically evolved over the past 15 years, the main challenges in the formal methods community remain **scalability and adoptability** [4]. *So how can we scale formal verification techniques for real-world software systems? How can we increase adoption of formal verification techniques by software engineers in industry?*

In order to tackle both aforementioned questions, **our vision** is to integrate formal verification techniques into the workflow of the main software development methodologies and tools. Our work is inspired by recent insights described by Sadowski et al. [2] who describe a set of lessons from building static analysis tools at Google. We believe that formal methods can be effective in improving software quality assurance of a large number of organisations around the globe. In particular, our approach aims to provide a solution that applies formal verification in a way that is both *(i)* low-effort (*i.e.* fits into existing processes), and *(ii)* scalable to the large software systems used in industry. Here our focus is on

software model checking techniques combined with DevOps culture, particularly, continuous integration (CI). On one hand, we have software model checking [4], which has been successfully applied to discover subtle errors but, for larger applications, often suffers from the state-space explosion problem [5]. On the other hand, we have continuous integration, which has been widely adopted by the software development community, but relies on a test suite that typically does not cover significant parts of the state-space [6].

We propose a continuous formal verification (CFV) approach, which aims to automatically detect design errors and integration problems as quickly as possible. First, we concentrate the verification effort to code changes rather than the entire system, thus, we only re-verify the code changes that could potentially break the properties of a system; this verification process should run fast (*e.g.*, in less than 5 minutes) in order to provide quick (and useful) feedback for developers. Second, we select the regression tests related to each code change (*e.g.*, an updated function), generalize these tests, and formally verify the code changes using software model checking. Lastly, we gather all the information from this analysis and report it back to the analytic and development team, who will carry out this process continuously; this step is crucial according to Sadowski et al. since careful developer workflow integration is key for any static analysis tool adoption by engineers [2].

Our **main contributions** are twofold. Firstly, we propose a feasible integration of software model checking into DevOps practices, thus making formal verification techniques accessible to software engineers. Here, our approach will focus on the developer and their feedback; the goal is to increase the adoption of our approach in real-world software projects by integrating our verification tool into the developer workflow. Secondly, we propose to reduce the impact of state-space explosion in development practices using existing regression tests in the verification process, which will provide quick and useful feedback for developers so that they can easily locate and fix bugs.

2 Continuous Formal Verification

The essence of this approach relies on the principle of compositional analysis [1]. Practically, we are inspired by CI practice, a well-known concept in Extreme Programming, proposed by Martin Fowler [6]. CI is particularly relevant when coupled with tools to automatically build and test a project's code base. Since the builds are generated after every incoming code changes (*i.e.*, commits and pull requests), problems can be detected much earlier. We can take advantage of such modularity to apply formal techniques in a continuous environment by model checking a software component only after it is changed, *i.e.*, we place our approach at diff-time. We use the same information (*i.e.*, development history and regression test cases), but in a way to substantially reduce verification complexity and increase coverage in a pull-based development model (*e.g.*, GitHub[1]).

The development cycle initiates with the developer submitting changes to the code base through a software configuration management (SCM) system. For each system build, we thus use the information from the SCM system to identify

[1] More info at https://help.github.com/articles/about-pull-requests/

the components that have actually been modified and focus on them. Importantly, we focus on C projects and each function is considered as a component. Equivalence checking [7] is then performed to identify which changes have an actual impact on the code base. At this point, the regression test suite (containing unit and functional tests) is of paramount importance, since we select the regression tests correspondent to the non-equivalent changed components. To increase coverage, these regression tests are passed to a generalization process. Finally, we check the use of non-equivalent components through the generalized regression tests and collect the reports (*e.g.*, counterexamples) and send it to the analytics team. In order to adopt such an approach, a project must comply with two basic guidelines: *(i)* the development process must be based on a continuous integration environment and *(ii)* it must include a regression test suite. We are currently building tools that can completely automate our CFV process.

The following sections describe the two steps of the process, highlighting key challenges: identifying relevant code changes, and the model checking of generalized test cases. As an illustrative example, we use the a GitHub project called `vec`[2], an ANSI-C type-safe dynamic array implementation. The repository contains 22 test cases, which *intend* to cover all possible execution paths related to the functionalities of the type-safe dynamic array. We focus on the function `vec_insert_`, shown in Fig. 1, to exemplify our proposed approach.

```
1  #define vec_unpack_(v) \
2    (char**)&(v)->data, &(v)->length, &(v)->capacity, sizeof(*(v)->data)
3
4  #define vec_insert(v, idx, val) \
5    ( vec_insert_(vec_unpack_(v), idx) ? -1 : \
6      ((v)->data[idx] = (val), 0), (v)->length++, 0 )
7
8  int vec_insert_(char **data, int *length, int *capacity, int memsz,
9                  int idx) {
10   int err = vec_expand_(data, length, capacity, memsz);
11   if (err) return err;
12   memmove(*data + (idx + 1) * memsz,
13           *data + idx * memsz,
14           (*length - idx) * memsz);
15   return 0;
16 }
```

Fig. 1: Implementation of the `vec_insert` function that adds the `val` value in the `idx` index of the `vec` structure. We omit the function `vec_expand_` for simplicity: it reallocates the vector if it needs to be expanded.

2.1 Checking for Relevant Code Changes

We begin from the principle that if a modified version of a component is computationally equivalent to its older version, then it is not necessary to prove that all properties that hold for the old version still hold for the modified one. Thus, we use equivalence checking to check whether the modified components need to

[2] Available at `https://github.com/rxi/vec`

be re-verified. Naturally, proving the equivalence of two functions is in general undecidable [7], and the effort we spend in trying to do so might be wasted. However, such an approach can potentially reduce the immediate verification effort, since proving the equivalence of two function versions can be less expensive than re-verifying the function [7]. In addition, by proving that two versions of a function are computationally equivalent, we eliminate the effort to re-verify any other function that depends on it (unless that function has been changed as well). Therefore, this approach limits the propagation of changes through the system and, consequently, reduces the effort to overall system verification.

The equivalence check will happen in two steps: a (1) fast and imprecise abstract syntax tree (AST) structural equivalence check [8], and a (2) slow and precise formal check e.g. bounded model checking (BMC). In the AST structural equivalence check, easy cases will be caught without the need to formally verify them, e.g., a function is renamed and the call sites are updated, or comments are added to a function body. If the AST is structurally not equivalent, we then encode the old and the new functions, and check if they are equivalent for the same inputs. A time limit is set for the formal check since it is more useful to spend time running the regression tests than checking their equivalence; if the time limit is exhausted, we assume they are not equivalent and start the tests.

In our illustrative project `vec` we find commits that would benefit from our approach. In commit $40d5cc17$[3], the developer changes the name of a macro `vec_absindex` used in an early version of the function shown in Fig. 1, and in commit $7d8588bc$[4], the developer removes the support for negative indexes when accessing arrays. In the former, the ASTs is equivalent neither triggering the next formal check nor starting the tests, while in the latter the formulas are not equivalent by the formal check, triggering the regression tests.

Open Challenges. There are many techniques that could be applied to perform equivalence checking such as SYMDIFF [9] and CORK [10] tools or through directed incremental symbolic execution (DiSE) [11]; in future, we will evaluate their performance in this CFV setting. We will also exploit this module by generating test cases from code changes [12].

2.2 Model Checking Generalized Tests

It is of paramount importance a software project follows two key best-practice principles: *(i)* keep the project as modular as possible and create short functions that focus on one particular objective and *(ii)* provide at least one regression/unit test for every function. Such an approach is key to a successful compositional analysis of the software project, where the combination of the analysis result of its parts represents the analysis result of the whole.

After pruning the unmodified components, we only focus on the existing regression test cases related to the modified ones, in order to reduce the state space to be explored by the model checker. However, we do not generate new concrete values for the test cases with the purpose of maximizing the code coverage. Instead, we combine existing test cases with non-deterministic input values to maximize the coverage of this verification. The use of regression tests also help to reduce the state space by breaking the global model (containing the entire

[3] `https://github.com/rxi/vec/commit/40d5cc17ea41923c662`
[4] `https://github.com/rxi/vec/commit/7d8588bc96c4c7aa68b`

program) into local models (containing only the functions under verification) and generate on demand the reachable states to be visited by the model checker, starting with the state described by the test case. This reduces the number of paths and variables to be considered during model checking.

In our illustrative project `vec`, by measuring the number of linearly independent paths in all functions, i.e., the project's cyclomatic complexity [13], we clearly see the benefit of focusing on the regression tests. In the case of `vec`, the entire system has a cyclomatic complexity of 24; in contrast, its regression tests have an average cyclomatic complexity of 1.

Through BMC, we can check for all possible paths in the implementation shown in Fig. 1, by non-deterministically assigning a value for each function parameter (*i.e.*, `pos`, and `val`) assuming a valid initialized structure (*i.e.*, `v`). Rather than modifying the program, we modify the regression tests and replace the concrete input values by non-deterministic choices. Here, we replace the series of function invocations with a non-deterministic one (see lines 5–7 of Fig. 2b). We can try to get full coverage in this particular module because we already pruned the state space by only selecting the modified parts of the system.

```
1  test_section("vec_insert");
2  vec_int_t v;
3  vec_init(&v);
4  int i;
5  for (i = 0; i < 1000; i++)
6    vec_insert(&v, 0, i);
7  test_assert(v.data[0] == 999);
8  test_assert(
9    v.data[v.length - 1] == 0);
10 vec_insert(&v, 10, 123);
11 test_assert(v.data[10] == 123);
12 test_assert(v.length == 1001);
13 vec_insert(&v, v.length - 2, 678);
14 test_assert(v.data[999] == 678);
15 test_assert(
16   vec_insert(&v, 10, 123) == 0);
17 vec_insert(&v, v.length, 789);
18 test_assert(
19   v.data[v.length - 1] == 789);
20 vec_deinit(&v);
```

(a) Original test.

```
1  test_section("vec_insert");
2  vec_int_t v;
3  vec_init(&v);
4  int val = nondet_int();
5  size_t pos = nondet_size_t();
6  vec_insert(&v, pos, val);
7  test_assert(v.data[pos] == val);
8  vec_deinit(&v);
```

(b) Generalized version.

Fig. 2: Generalization of the regression test for the function shown in Fig. 1.

Open Challenges. Our main difficulty here is how to deal with false negatives as the non-deterministic choice of values for program variables may force the exploration of paths that are infeasible in the original program. So, we need to find a balance between coverage and soundness. We also need to increase automation as much as possible. One may combine techniques to automatically generate tests based on counterexamples [14] or source code [15]. We will also increase the power of this analysis by using conditional verifiers [16] or applying different model checking approaches (*i.e.*, explicit-state).

3 Related Work

Fitzgerald and Stol [17] present a holistic overview of the activities related to continuous software engineering, which includes continuous testing and verification. Although they do not propose a new approach, they highlight the importance of continuous (and automatic) testing and verification in the context of DevOps. Interestingly, Beyer and Lemberger [18] perform a comparison between software testers and software model checkers, which shows that model checkers are mature enough to be used in practice (they even outperform testing tools), and the combination of both techniques could lead to even better results. Indeed, there are many reports of successful attempts that use formal techniques in large software systems.

For instance, Klein *et al.* [19] show how to scale formal proofs based on software architecture to real systems at low cost; Godefroid, Levin, and Molnar [20] describe the remarkable impact of SAGE tool, which performs dynamic symbolic execution to hunt for security issues in Microsoft applications; Cordeiro, Fischer, and Marques-Silva [21] as well as Yin and Knight [22] propose approaches to conduct formal verification of large software systems. Furthermore, there are two important studies that tackle the combination of formal techniques with continuous integration, which led to promising results and reflect the need and scientific challenges in the industry to follow this road. First, Chudnov *et al.* [23] describe how Amazon Web Services (AWS) prove the correctness of their Transport Layer Security (TLS) protocol implementation, and how they use CI tools to keep proving the software properties during its lifetime. Similarly, O'Hearn [1] presents Infer, a static analyzer used at Facebook following a continuous reasoning approach. Neither Chudnov *et al.* nor O'Hearn try to handle model checking in a continuous process; the latter states this as an open challenge for the community.

These cases highlight the impact of formal techniques in real software systems; however, they do not present guidelines to generalize these approaches to a wide range of software projects, which could lead to a significant adoption of formal techniques by practitioners. Thus, there is still an open-call for approaches that could potentially popularize formal techniques in software engineering practices.

4 Conclusions and Future Work

Model checking of entire systems is usually not feasible for many industrial applications due to the state-space explosion problem, however, one of the scalability challenges can be solved through leveraging changes to the system. Thus, we propose CFV, an approach with the potential to detect software vulnerabilities by combining dynamic and static verification to reduce the state space. This potential propels us to further research this topic: we are currently developing an automated software tool to tackle the key challenges of equivalence checking and test case generalization, so it can be applied to large open-source projects. We are also working in close collaboration with software developers at Samsung with the goal of integrating our automated reasoning tool into their workflow, thus increasing adoption of formal methods in industry.

References

1. O'Hearn, P.W.: Continuous reasoning: Scaling the impact of formal methods. In: Symposium on Logic in Computer Science. (2018) 13–25
2. Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., Jaspan, C.: Lessons from building static analysis tools at google. Communications of the ACM **61**(4) (2018) 58–66
3. Rosen, G.: Security update facebook, inc. `https://newsroom.fb.com/news/2018/09/security-update/` (2018) [Online; accessed August-2019].
4. Clarke, E.M., Henzinger, T.A., Veith, H.: Introduction To Model Checking. In: Handbook Of Model Checking. Springer International Publishing (2018) 1–26
5. Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: An industrial-strength C model checker. In: Automated Software Engineering. (2018) 888–891
6. Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., Vasilescu, B.: The impact of continuous integration on other software development practices: A large-scale empirical study. In: Automated Software Engineering. (2017) 60–71
7. Godlin, B., Strichman, O.: Regression verification: Proving the equivalence of similar programs. Software Testing, Verification and Reliability **23**(3) (2013) 241–258
8. Ramos, D.A., Engler, D.R.: Practical, low-effort equivalence verification of real code. In: Computer Aided Verification. Volume 6806 of LNCS. (2011) 669–685
9. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In: Computer Aided Verification. Volume 7358 of LNCS. (2012) 712–717
10. Lopes, N.P., Monteiro, J.: Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. Software Tools for Technology Transfer **18**(4) (2016) 359–374
11. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. In: Programming Language Design and Implementation. (2011) 504–515
12. Godefroid, P., Lahiri, S.K., Rubio-González, C.: Statically validating must summaries for incremental compositional dynamic test generation. In: Static Analysis Symposium. Volume 6887 of LNCS. (2011) 112–128
13. Bang, L., Aydin, A., Bultan, T.: Automatically computing path complexity of programs. In: ACM Joint European Software Engineering Conference And The Foundations Of Software Engineering. (2015) 61–72
14. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses. In Dubois, C., Wolff, B., eds.: Tests and Proofs. Volume 10889 of LNCS. (2018) 3–23
15. Christakis, M., Emmisberger, P., Godefroid, P., Müller, P.: A general framework for dynamic stub injection. In: International Conference on Software Engineering. (2017) 586–596
16. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: International Conference on Software Engineering. (2018) 1182–1193
17. Fitzgerald, B., Stol, K.J.: Continuous software engineering: A roadmap and agenda. Journal of Systems and Software **123** (2017) 176–189
18. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. In: Hardware and Software: Verification and Testing. Volume 10629 of LNCS. (2017) 99–114
19. Klein, G., Andronick, J., Fernandez, M., Kuz, I., Murray, T., Heiser, G.: Formally verified software in the real world. Communications of the ACM **61**(10) (2018) 68–77
20. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: Whitebox fuzzing for security testing. Queue **10**(1) (2012) 20:20–20:27

21. Cordeiro, L.C., Fischer, B., Marques-Silva, J.: Continuous verification of large embedded software using SMT-based bounded model checking. In: Engineering of Computer Based System. (2010) 160–169
22. Yin, X., Knight, J.: Formal verification of large software systems. In: NASA Formal Methods Symposium. (2010) 192–201
23. Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., Tomb, A., Westbrook, E.: Continuous formal verification of amazon S2n. In: Computer-Aided Verification. Volume 10982 of LNCS. (2018) 430–446