Combining Verifiers in Conditional Model Checking via Reducers

Dirk Beyer[1], Marie-Christine Jakobs[2], Thomas Lemberger[3], Heike Wehrheim[4]

Abstract: Software verification received lots of attention in the past two decades. Nonetheless, it remains an extremely difficult problem. Some verification tasks cannot be solved automatically by any of today's verifiers. To still verify such tasks, one can combine the strengths of different verifiers. A promising approach to create combinations is conditional model checking (CMC). In CMC, the first verifier outputs a condition that describes the parts of the program state space that it successfully verified, and the next verifier uses that condition to steer its exploration towards the unverified state space. Despite the benefits of CMC, only few verifiers can handle conditions.

To overcome this problem, we propose an automatic plug-and-play extension for verifiers. Instead of modifying verifiers, we suggest to add a preprocessor: the reducer. The reducer takes the condition and the original program and computes a residual program that encodes the unverified state space in program code. We developed one such reducer and use it to integrate existing verifiers and test-case generators into the CMC process. Our experiments show that we can solve many additional verification tasks with this reducer-based construction.

Keywords: Conditional Model Checking, Testing, Software Verification, Sequential Combination

1 Overview

Automatic software verification received lots of attention in the past years: Many different approaches and verifiers were proposed, and all of them have different strengths, but also weaknesses. Thus, there still exist lots of programs that are in principle verifiable, but that no existing verifier can solve on its own. One solution to this problem is to combine the strength of different verifiers.

Conditional model checking (CMC) [Be12] is one promising combination approach. In CMC, the first verifier outputs a condition which summarizes the verifier's work, i.e., the state space that it successfully verified. If the first verifier fails to verify the complete state space, the condition and the program are passed to a second verifier, a so called conditional verifier. This conditional verifier uses that condition to restrict its verification

[1] LMU Munich, Institute of Informatics, Oettingenstraße 67, 80538 Munich, Germany
[2] LMU Munich, Institute of Informatics, Oettingenstraße 67, 80538 Munich, Germany
[3] LMU Munich, Institute of Informatics, Oettingenstraße 67, 80538 Munich, Germany
[4] Paderborn University, Department of Computer Science, Warburger Straße 100, 33098 Paderborn, Germany

cc by sa

to the unverified state space. Unfortunately, only few conditional verifiers exist and it is difficult to make a verifier understand and use conditions.

Instead of adapting existing verifiers, we propose a reducer-based construction of conditional verifiers [Be18]. Our reducer-based construction uses the template shown in Fig. 1 to easily and automatically build a conditional verifier from an existing off-the-shelf verifier without changing the verifier itself. The idea is to add a preprocessor: the reducer. The reducer takes the condition and the original program to compute a *residual program* that encodes the unverified state space in a format that every verifier understands: program code.

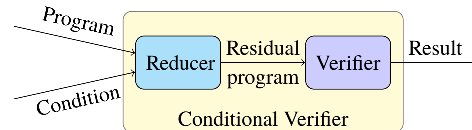

Fig. 1: Reducer-based conditional verifier template

In principle, many different reducers are possible. We implemented a reducer that uses a product construction: The program and the condition are converted to automata and the reduced product automaton is converted back to the residual program. We proved that the residual programs generated by this reducer neither miss any non-verified program path nor add program paths that do not occur in the original program. Hence, the reducer is reliable.

In our experiments, we instantiated the reducer-based construction template with the reducer implemented by us and six off-the-shelf verifiers; the best three verifiers from SV-COMP 2017[5] and three popular test-generation tools. Our evaluation on 5 687 verification tasks from a widely used software-verification benchmark[6] revealed that each of the created CMC combinations (1) solves additional tasks, (2) solves tasks that the corresponding sequential combination cannot solve, (3) solves tasks that were not solvable by any of the considered verifiers, and (4) solves tasks that only this specific CMC combination can solve. Summing up, we showed that reducer-based conditional model checking is effective, and to take full advantage of the approach one requires different conditional verifiers.

To solve difficult verification tasks, one needs to combine the strength of different verifiers, e.g., through CMC. The insight of our experiments is that we need different conditional verifiers for CMC to be effective and it is not worth the effort to modify existing verifiers to become conditional. Instead, our reducer-based construction of conditional verifiers allows us to easily derive k conditional verifiers from k arbitrary existing verifiers, without changing the implementation of the verifiers at all.

References

[Be12] Beyer, D.; Henzinger, T. A.; Keremoglu, M. E.; Wendler, P.: Conditional Model Checking: A Technique to Pass Information Between Verifiers. In: Proc. FSE. ACM, 2012.

[Be18] Beyer, D.; Jakobs, M.-C.; Lemberger, T.; Wehrheim, H.: Reducer-Based Construction of Conditional Verifiers. In: Proc. ICSE. ACM, pp. 1182–1193, 2018.

[5] `https://sv-comp.sosy-lab.org/2017/`

[6] `https://github.com/sosy-lab/sv-benchmarks`