# Cooperative Approaches Across Test Generation and Formal Software Verification

Thomas Lemberger, LMU München, Geschwister-Scholl-Platz 1, 80539, München

## Abstract

In the last decade, powerful techniques were developed that either automatically generate tests for software, or automatically verify software with formal methods. In both areas it is common to combine different techniques to leverage their strengths and mitigate their weaknesses. This happens through costly, proprietary reimplementations within a single tool. This thesis [15] contrasts this and provide concepts that enable an inexpensive and fast off-the-shelf cooperation of standalone tools through standardized exchange formats.

## 1 Introduction

There are two major methods for software verification: testing and formal verification. To increase our confidence in software on a large scale, we require tools that apply these methods automatically and reliably. Testing is a widespread engineering technique, and many tools for automatic test generation exist. But testing can never provide full confidence in software—it can show the presence of bugs, but not their absence. In contrast, formal verification can. Unfortunately, even successful formal-verification techniques either scale poorly or rely on heuristics that fail if the used heuristic does not suit the verification task. As workaround, combinations of multiple techniques try to combine their strengths and mitigate their weaknesses. But these combinations are often designed as cohesive, monolithic units. This makes them inflexible and it is costly to replace components.

We propose an alternative approach: If tools support the same input and output formats, we can use these formats and combine existing tools without any adjustments (off-the-shelf).

For formal verification, the International Competition on Software Verification (SV-COMP) [2] has established conventions for both input tasks and result formats. These conventions are supported by more than 40 tools to date. But for test generation, no such conventions exist in the C ecosystem[1].

In the scope of our work, we first work towards input and output conventions for test generation. We then go beyond the conventions of tool competitions and propose a technique to encode partial verification results directly into the program code. Based on this, we introduce new concepts for the off-the-shelf cooperation between test generators and formal verifiers and show their flexibility through different examples.

[1] We focus on verifiers and test generators for the C programming language

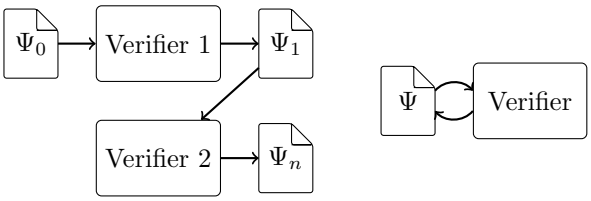


Figure 1: Examples of cooperative verification: Sequential and cyclic combination

## 2 Standardizing Test Generation

SV-COMP [2] is an off-site tool competition that compares formal verifiers on a predefined benchmark set and in a controlled environment. SV-COMP defines how non-deterministic values are introduced into programs, how program errors are signaled, and in which formats formal verifiers have to report alarms and proofs. These rules are partially based on sv-benchmarks, the largest available benchmark set for verification of C programs.

To establish common standards in test generation, we transferred our experience with SV-COMP. An initial comparative study [9] worked towards the establishment of the First International Competition on Software Testing (Test-Comp) [1]. Test-Comp uses the sv-benchmarks as input programs and defines a new, common exchange format for test suites. All participants of Test-Comp must understand input programs according to the sv-benchmarks conventions, and produce test suites in the expected exchange format. These standards open all competition participants to cooperation on the tool level.

We contribute two tools to Test-Comp: the plain random tester PRTest [16] and the test-suite executor TestCov [10]. PRTest samples test inputs from a uniform random distribution. This purely random approach serves as a baseline for tool comparisons. TestCov provides robust test-suite execution and coverage measurement for different coverage criteria. It is used for the coverage-measurement and score computation in Test-Comp.

## 3 Concepts for Cooperative Software Verification

The wide tool support of the SV-COMP and Test-Comp standards enable the off-the-shelf combination of verification tools. But formal verifiers do not have to produce partial results. Because of this, possible combinations are limited to techniques that only consider verification runs that successfully finish (with an alarm or an proof). If a formal verifier's run does not finish successfully, the performed work is simply dis-

carded.

Conditional Model Checking [7] is a technique that makes verifiers exchange partial verification results through *conditions*. This could lead to more cooperation possibilities and information reuse, but there is little tool support. To mitigate this limitation, we propose a reducer-based construction of conditional model checkers [6]: partial verification results are encoded directly into the program code, so that any existing formal verifier can consume them. We apply the same concept to test generators through the concept of *conditional testing* [8].

This construction enables a strong information exchange between verifiers. Figure 1 shows two example compositions: A sequential composition of verifiers, and a cyclic composition of a verifier. Verifiers start with some initial knowledge $\Psi_0$ about the verification task (by default: nothing), compute new knowledge $\Psi_1$, and, if the verification task is not solved yet, communicate their computed knowledge to the next verifier. Note that any such composition is a verifier itself, and compositions can be arbitrarily nested.

To make formal verifiers also usable for test generation, we introduce the concept of *Witness2Test* [4] that transforms alarms reported in the SV-COMP conventions into executable tests.

The proposed concepts can not only improve the effectiveness of verification [6, 8], but can also be used for incremental verification [5]. On the example of counterexample-guided abstraction refinement [11], we show [3] how existing, strongly coupled techniques in software verification can be decomposed into standalone components that cooperate through standardized exchange formats.

## 4 Results

All our work is backed by rigorous implementation of the proposed concepts and thorough experimental evaluations that demonstrate the benefits of our work. We were able to show that cooperative verification improves the effectiveness of verification beyond the prior state of the art [6, 8]. TestCov successfully drives the coverage measurements of Test-Comp since 2019. And the test generator FuSeBMC [14] builds on our concept of conditional testing to achieve a cooperation between the bounded model checker ESBMC [12] and two fuzz testers [13]. This combination won Test-Comp 2022, 2023, and 2024.

The introduced standards and tools also improve the comparability of automated verifiers, enable cooperation between a large array of existing verification tools, and create new opportunities for research in cooperative verification.

## References


[1] D. Beyer. "First International Competition on Software Testing (Test-Comp 2019)". In: *Int. J. Softw. Tools Technol. Transf.* 23.6 (Dec. 2021), pp. 833–846. DOI: 10.1007/s10009-021-00613-3.

[2] D. Beyer. "State of the Art in Software Verification and Witness Validation: SV-COMP 2024". In: *Proc. TACAS (3)*. LNCS 14572. Springer, 2024, pp. 299–329. DOI: 10.1007/978-3-031-57256-2_15.

[3] D. Beyer, J. Haltermann, T. Lemberger, and H. Wehrheim. "Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR". In: *Proc. ICSE*. ACM, 2022, pp. 536–548. DOI: 10.1145/3510003.3510064.

[4] D. Beyer, M. Dangl, T. Lemberger, and M. Tautschnig. "Tests from Witnesses: Execution-Based Validation of Verification Results". In: *Proc. TAP*. LNCS 10889. Springer, 2018, pp. 3–23. DOI: 10.1007/978-3-319-92994-1_1.

[5] D. Beyer, M.-C. Jakobs, and T. Lemberger. "Difference Verification with Conditions". In: *Proc. SEFM*. LNCS 12310. Springer, 2020, pp. 133–154. DOI: 10.1007/978-3-030-58768-0_8.

[6] D. Beyer, M.-C. Jakobs, T. Lemberger, and H. Wehrheim. "Reducer-Based Construction of Conditional Verifiers". In: *Proc. ICSE*. ACM, 2018, pp. 1182–1193. DOI: 10.1145/3180155.3180259.

[7] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. "Conditional Model Checking: A Technique to Pass Information between Verifiers". In: *Proc. FSE*. ACM, 2012. DOI: 10.1145/2393596.2393664.

[8] D. Beyer and T. Lemberger. "Conditional Testing: Off-the-Shelf Combination of Test-Case Generators". In: *Proc. ATVA*. LNCS 11781. Springer, 2019, pp. 189–208. DOI: 10.1007/978-3-030-31784-3_11.

[9] D. Beyer and T. Lemberger. "Software Verification: Testing vs. Model Checking". In: *Proc. HVC*. LNCS 10629. Springer, 2017, pp. 99–114. DOI: 10.1007/978-3-319-70389-3_7.

[10] D. Beyer and T. Lemberger. "TestCov: Robust Test-Suite Execution and Coverage Measurement". In: *Proc. ASE*. IEEE, 2019, pp. 1074–1077. DOI: 10.1109/ASE.2019.00105.

[11] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-guided abstraction refinement for symbolic model checking". In: *J. ACM* 50.5 (2003), pp. 752–794. DOI: 10.1145/876638.876643.

[12] M. Y. Gadelha, H. I. Ismail, and L. C. Cordeiro. "Handling Loops in Bounded Model Checking of C Programs via $k$-induction". In: *Int. J. Softw. Tools Technol. Transf.* 19.1 (Feb. 2017), pp. 97–114. DOI: 10.1007/s10009-015-0407-9.

[13] H. O. Rocha, R. S. Barreto, and L. C. Cordeiro. "Memory Management Test-Case Generation of C Programs Using Bounded Model Checking". In: *Proc. SEFM*. LNCS 9276. Springer, 2015, pp. 251–267. DOI: 10.1007/978-3-319-22969-0_18.

[14] K. Alshmrany, M. Aldughaim, L. Cordeiro, and A. Bhayat. "FuSeBMC v.4: Smart Seed Generation for Hybrid Fuzzing (Competition Contribution)". In: *Proc. FASE*. LNCS 13241. Springer, 2022, pp. 336–340. DOI: 10.1007/978-3-030-99429-7_19.

[15] T. Lemberger. *Towards Cooperative Software Verification with Test Generation and Formal Verification*. PhD Thesis, LMU Munich, Software Systems Lab. 2022. DOI: 10.5282/edoc.32852.

[16] T. Lemberger. "Plain Random Test Generation with PRTest (Competition Contribution)". In: *Int. J. Softw. Tools Technol. Transf.* 23.6 (Dec. 2021), pp. 871–873. DOI: 10.1007/s10009-020-00568-x.