International Journal on Software Tools for Technology Transfer (2021) 23:871–873
https://doi.org/10.1007/s10009-020-00568-x

COMPETITIONS AND CHALLENGES

Special Issue: TestComp 2019

Plain random test generation with PRTest

Thomas Lemberger[1]

Published online: 6 July 2020

Abstract
Automatic test-suite generation tools are often complex and their behavior is not predictable. To provide a minimum baseline that test-suite generators should be able to surpass, we present PRTEST, a random black-box test-suite generator for C programs: To create a test, PRTEST natively executes the program under test and creates a new, random test value whenever an input value is required. After execution, PRTEST checks whether any new program branches were covered and, if this is the case, the created test is added to the test suite. This way, tests are rapidly created either until a crash is found, or until the user aborts the creation. While this naive mechanism is not competitive with more sophisticated, state-of-the-art test-suite generation tools, it is able to provide a good baseline for Test-Comp and a fast alternative for automatic test-suite generation for programs with simple control flow. PRTEST is publicly available and open source.

Keywords Random testing · Software engineering · Software testing · Software verification · Test-Comp

1 Introduction

Automatic test-suite generation is a highly active field of research and many successful tools exist to this date. Unfortunately, most of these tools are based on sophisticated algorithms and thus, both their code and their behavior can be hard to understand for non-experts. In addition, these tools and their improvements are usually only compared to each other, but no naive baseline exists. We present PRTEST, a plain random test-suite generator that provides a solution for both issues. PRTEST is designed to be simple: its full test-suite generation logic consists of 125 lines of code, and it uses no heuristics or sophisticated algorithms. Instead, PRTEST provides random input generation [2]: It repeatedly executes the program under test with random inputs and stores the input values of an execution as a test if the execution increased the overall coverage. Thanks to its pure randomness and native execution of the program under test its behavior is easy to understand and it can be used as a lower baseline for Test-Comp.

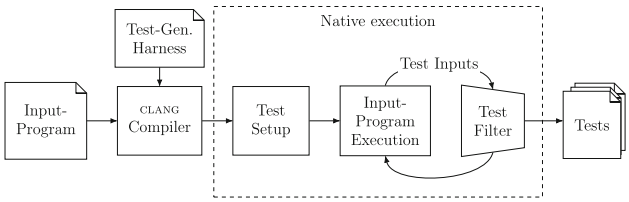

Fig. 1 Workflow of PRTEST

2 Test-suite generation approach

Figure 1 shows the workflow of PRTEST. PRTEST consists of two steps: (1) it compiles the input program against a test harness, and (2) it natively executes the compilation result. This execution consists of the test setup and a test-generation loop.

First, PRTEST uses the CLANG compiler to compile the program under test against a C harness that provides the full test-suite generation logic ('Test Gen. Harness' in Fig. 1). The harness provides: (1) a custom program entry point for test-generation setup, (2) definitions for the Test-Comp-specific input methods `__VERIFIER_nondet_X` (where `X` is any primitive C type; e.g., `__VERIFIER_nondet_int`), (3) a method `input` that creates new test inputs, and (4) CLANG-specific methods that allow PRTEST to track program coverage during runtime.

✉ Thomas Lemberger
thomas.lemberger@sosy.ifi.lmu.de

[1] LMU Munich, Munich, Germany

```
int __VERIFIER_nondet_int() {
  int var;
  input(&var, sizeof(var));
  return var;
}
```

Fig. 2 Test-harness definition of the Test-Comp-specific method `__VERIFIER_nondet_int`

```
for (int i = 0; i < var_size; i++) {
  new_value[var_size - i - 1] = rand() & 255;
}
memcpy(var, new_value, var_size);
```

Fig. 3 Program logic for creating a new input value of `var_size` bytes

When the compilation result is executed, the custom program entry point initializes a random number generator and traps signals that would usually terminate the program (e.g., `SIGINT`) as well as the exit method. This is necessary so that PRTEST is not terminated prematurely if the input program raises a signal or calls the exit method. Then, the test-generation loop starts and calls the original main function of the program under test on clean memory. Whenever a method `__VERIFIER_nondet_X` is called in the program under test, method `input` introduces a new test input of the expected type, records it as the next test input for the current execution, and returns it to the function call in the program under test. When the program under test terminates, PRTEST checks whether the execution covered any new code blocks, and if it did, the test inputs that were recorded for that execution are stored as a new test. If no new code blocks were covered, the test inputs are discarded. We call this mechanism *test filter*. After test filtering, loop starts again by calling the main method of the input program, creating another random test in the process. The test-generation loop stops if a looked-for program bug is found (in case of category Coverage-Error) or if the process is aborted by the user.

The test harness of PRTEST defines input methods `__VERIFIER_nondet_X` so that they declare a new program variable of their respective type `X` and call method `input` to introduce a new test input of the required size. Figure 2 shows this exemplary for method `__VERIFIER_nondet_int`.

Method `input` receives a pointer to input variable `var` that a new value should be assigned to, and the size of the type of `var` in bytes. For each byte, `input` creates a random byte value and stores that in an array that represents the new value of the given size. To create random values, it uses the random number generator `rand()` provided by the C standard library. After a value has been created for each byte, this byte sequence is copied into `var` (Fig. 3). Method `input` considers all types in their binary representation and is thus type-agnostic: it uses a uniform distribution over arbitrary-size binary values and is able to handle both integer and float types.

To measure code coverage of program executions, PRTEST uses the program instrumentation `SanitizerCoverage` that is provided by CLANG. This instrumentation adds a special method call at the beginning of each code block. We define this method so that, whenever a new code block is covered, a Boolean flag is set to indicate that the current test covers new program behavior. This flag is then checked by the test filter to decide whether to keep or discard a test.

The version of PRTEST used in `Test-Comp '19` was implemented as part of TBF [1]. It is written in PYTHON 3 and C, and uses the pseudo-random number generator provided by the C standard library with a uniform distribution. For reproducibility of the Test-Comp results, the seed of the random value generator is set to the arbitrary value 1618033988, derived from the golden ratio. Since version 2.0,[1] PRTEST is a stand-alone application that does not require Python anymore.

3 Strengths and weaknesses

Strengths PRTEST does not interpret or analyze the program under test, but executes it natively with a test-generation harness. Thanks to this, PRTEST is able to handle all existing C constructs and can efficiently handle all numeric types, including floats.

PRTEST is also able to create a vast amount of tests in a very short time: For example, for benchmark task `floats-cdfpl/square_2.i`, PRTEST generated over 400 000 tests per second. This allows very fast generation of a rudimentary test suite that covers the, based on naive input-value probability, most probable program branches. PRTEST is also very simple: The C harness, which is the only necessary component to create tests, is only 125 lines of code. The remaining code exists to determine the input methods for methods outside of Test-Comp, and to transform tests into the Test-Comp test format—functionality that is not required if one wants to apply PRTEST's approach to a specific program with a fixed set of input methods.

Weaknesses The uniform randomness of PRTEST cannot compete with control-flow-aware test generators if programs contain deeply nested branches or branches that are only entered on a small range of inputs or a single input: The probability to generate a random test that reaches the comment 'code block' in the following example is $\frac{1}{2^{32}} \approx 2 * 10^{-10}$:

```
int i = __VERIFIER_nondet_int();
if (i == 1) {
  // code block
}
```

[1] https://gitlab.com/sosy-lab/software/prtest.

If PRTEST produced tests with the same speed as for the task `floats-cdfpl/square_2.i` that was mentioned above, PRTEST would have a chance of about 8 % to create a test to enter this loop within the Test-Comp time limit. To achieve a 90 % probability to produce a test that reaches the code block, PRTEST would have to create almost 10 billion random tests. For task `floats-cdfpl/square_2.i`, this would take PRTEST about 7 hours. The probability to enter a program branch also exponentially decreases with the number of conditions required to enter the branch.

In the literature, random testing is mostly used as a complement to control-flow-aware testing techniques, for example to provide an initial test suite [4] or to avoid other generation techniques from getting stuck [3].

4 Tool setup

Availability PRTEST is developed at Dirk Beyer's Software and Computational Systems Lab (SoSy-Lab) at LMU Munich. It is open source under Apache License, version 2.0, and available online.[2] This work describes the `Test-Comp '19` submission of PRTEST—the newest version of PRTEST is available as a stand-alone program.[3]

Installation and Usage PRTEST requires PYTHON 3.5 or later and CLANG 3.9 or later. It can be installed by following the steps described in file `README.md`. The following command line runs PRTEST in its configuration for `Test-Comp '19`, for coverage-property file `PROP_FILE` and input program `PROGRAM.c`:

```
./bin/tbf -i random --write-xml \
    --svcomp-nondets \
    --spec PROP_FILE PROGRAM.c
```

The created test suite will be located in directory `output/test-suite/`.

Participation PRTEST participated in all categories of `Test-Comp`. In category Cover-Error, PRTEST was not able to get any points in sub-categories ReachSafety-ControlFlow, ReachSafety-ECA and ReachSafety-Sequentialized because of its weakness regarding control flow. In sub-category ReachSafety-Floats, in contrast, PRTEST even reaches the third place due to its ability to natively handle float types. PRTEST also proved useful as a baseline to identify potential weaknesses of other participants: The result tables of `Test-Comp '19` (e.g., for branch coverage[4]) can show scatter plots for the values of chosen table columns. This allows a quick comparison of the coverage achieved per task by the random test suites created by PRTEST and the test suites created by other participants. If a tool achieves significantly worse results for a task than PRTEST, this may hint to a potential weakness in that tool. Such tasks exist for all participants.

Acknowledgements Open Access funding provided by Projekt DEAL.

References

1. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. In: Proc. HVC, LNCS, vol. 10629, pp. 99–114. Springer (2017)
2. Bird, D.L., Munoz, C.U.: Automatic generation of random self-checking test cases. IBM Syst. J. **22**(3), 229–245 (1983)
3. Majumdar, R. Sen, K.: Hybrid concolic testing. In: Proc. ICSE, pp. 416–426. IEEE (2007)
4. Rojas, J.M., Fraser, G., Arcuri, A.: Seeding strategies in search-based unit test generation. Softw. Test. Verif. Reliab. **26**(5), 366–401 (2016)

[2] https://gitlab.com/sosy-lab/test-comp/archives-2019/blob/c991e4/2019/prtest.zip.

[3] https://gitlab.com/sosy-lab/software/prtest.

[4] https://test-comp.sosy-lab.org/2019/results/results-verified/META_Cover-Branches.table.html.