

PROTON: PRObes for Termination Or Not (Competition Contribution)

Ravindra Metta[1,2](✉) ⋆, Hrishikesh Karmarkar[1], Kumar Madhukar[3], R. Venkatesh[1], and Supratik Chakraborty[4]

[1] TCS Research, Tata Consultancy Services, Pune, India
{r.metta,hrishi.karmarkar,r.venky}@tcs.com
[2] School of CIT, Technical University of Munich, Munich, Germany
[3] Dept. of Computer Science and Engineering, IIT Delhi, New Delhi, India
madhukar@cse.iitd.ac.in
[4] Dept. of Computer Science and Engineering, IIT Bombay, Bombay, India
supratik@cse.iitb.ac.in

Abstract. PROTON is a tool to check whether a given C program has a non-terminating behaviour or not. It is built around the C Bounded Model Checker (CBMC). CBMC cannot prove non-termination directly, as all non-terminating runs are unbounded. PROTON annotates the loops in a given program with assertions that check for a recurrent program state. Violation of such an assertion shows the existence of a recurrent state and thereby proves non-termination. PROTON also transforms the violating trace returned by CBMC into a non-termination witness for the program.

1 Introduction

Given a program P for which we want to check termination under all inputs, a checker should either provide a witness for non-termination of P, or give a *correct* verdict that P always terminates. For termination checking, PROTON reuses the high confidence, but unsound, technique used in VeriFuzz 1.4 [9]. For proving non-termination, PROTON implements a novel sound technique that attempts to discover recurrent states inside loops. A recurrent state (RS) is a program state at the head of a loop such that (1) RS entails the loop guard; (2) RS is reachable from an initial state in some valid program execution and (3) RS is reachable from itself after the loop body is executed. This notion of an RS is a strengthening of the recurrent set definition proposed in [5].
Consider the example program P in Listing 1.1, adapted from the SV-COMP benchmark `WhileSingle.c`. This program does not terminate for any nondet value ≤ 3. For example, if nondet value on Line 1 is 3, then the if-condition on Line 3 gets evaluated to false and hence the value of i remains unchanged, causing the loop to run infinitely. PROTON works in three main phases, as described below.

⋆ Jury member

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 393–398, 2024.
https://doi.org/10.1007/978-3-031-57256-2_27

Listing 1.1. Program P

```
1 i = __VERIFIER_nondet_int();
2 while (i < 10) {
3   if (i != 3) {
4     i = i+1;
5   }
6 }
```

Listing 1.2. Program P'

```
1 i = __VERIFIER_nondet_int();
2 bool pStored0 = false;
3 while (i < 10) {
4 bool flag = __VERIFIER_nondet_bool();
5 static int oi; if(pStored0)
6 {__CPROVER_assert(!(oi==i),"RSF");}
7 if(flag){oi=i;pStored0=true;}
8 { if (i != 3) { i = i+1; }}}
```

Phase 1 - Program Instrumentation: PROTON instruments each loop in P with a __CPROVER_assert to check for a recurrent state. This is illustrated in Listing 1.2. PROTON first parses P using various Clang/LLVM APIs and collects the set of all program variables visible in the scope of each loop L_k in P. Following this, PROTON instruments each L_k as follows:-

- A boolean variable *pStoredk* is introduced just before the loop-guard of L_k and initialized to *false* (Line 2 of Listing 1.2).
- Another boolean variable *flag* is added inside the loop, immediately past the guard condition, which is nondeterministically initialized (Line 3).
- For each variable i, visible in the scope of L_k, a corresponding static variable oi; i.e. i prefixed with o is added, which tracks the "old" value of i (Line 5).
- An assertion that the "old" state of P never repeats in any later iteration of L_k (lines 5 and 6) is added.
- If *flag* is *true*, then the program state is stored as shown on line 7, and pStoredk is set to *true*.
- Lastly, PROTON emits the loop body as is, but enclosed in braces (Line 8).

The above instrumentation ensures that the assertion gets checked (due to the if-condition on Line 5), in every iteration after the one in which the state is stored, as pStoredk is set to *true* after this if-statement. So, in the very first iteration in which the program state is stored, the assertion is not invoked. This encoding allows a bounded model checker like CBMC [3,4] to check if the program state stored during a non-deterministically chosen iteration of L_k recurs during any subsequent iteration of L_k, subject to the loop iteration bound used for checking.

Phase 2 - Bounded Model Checking for recurrent states: After instrumenting P, PROTON iteratively invokes CBMC for different unwind bounds until a pre-configured max unwind bound (empirically chosen to be 1000, for SV-COMP 2024) for a pre-configured time limit (set to 2 minutes for SV-COMP 2024). If the recurrent state assertion ever gets violated, it proves the presence of a recurrent state and hence non-termination. When this happens, CBMC generates a corresponding counterexample trace. During Phase 1 described above, PROTON does additional instrumentation (not shown in Listing 1.2 for want of space) to help generate a corresponding non-termination witness in the graphml

format. If the recurrent state assertion does not get violated until the max bound or if it times out, then PROTON moves to Phase 3, described below.

Phase 3 - Value-Bounded Termination Check In this phase, entered only if PROTON could not find a non-termination witness in Phase 2, PROTON invokes the termination check of VeriFuzz 1.4 [9], which is reimplemented in PROTON, for a pre-configured time limit (set to 2 minutes for SV-COMP 2024).

2 Software Architecture

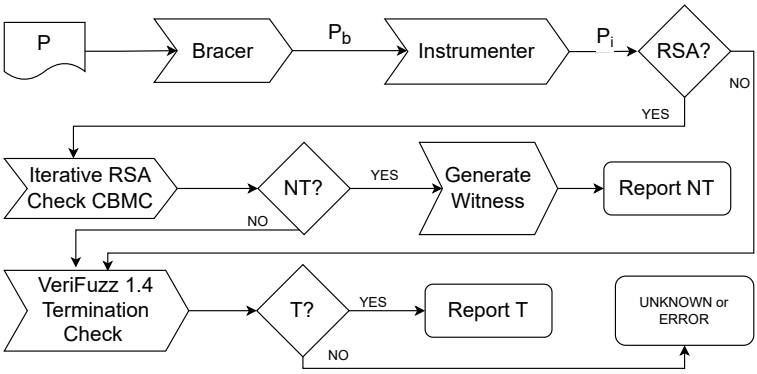

Fig. 1. PROTON architecture

Currently PROTON checks only termination and non-termination of programs. Figure 1 shows the tool flow of PROTON. Given an input program P, PROTON first invokes *Bracer*, which simply adds curly braces around all loop bodies in P to produce P_b. PROTON then invokes *Instrumenter* on P_b, which instruments P_b, as described in Phase 1, to produce P_i. Sometimes, due to internal errors, the Instrumenter may not be able to instrument the program. So, PROTON then checks if P_i has at least one Recurrent State Assertion (RSA). If so, it performs the non-termination check as described in Phase 2 above and generates a corresponding witness if it detects non-termination.

If P_i does not contain any RSA, or if this non-termination check is unsuccessful, PROTON then invokes confidence based termination check on P, mentioned in Phase 3 above. If this termination check concludes that P terminates, PROTON reports P to be terminating. Else, PROTON reports either UNKNOWN (when both checks failed) or ERROR (if there is any internal error).

PROTON is built using CBMC v5.95.0 [3] with Z3 4.12.2 [10] and Glucose Syrup [1] as the backend SMT and SAT solvers respectively. The Bracer and Instrumenter were implemented in C++ using the clang-14 and llvm-14 libraries. The tool flow is implemented in a bash shell script.

3 Strengths and Weaknesses

Here we present our analysis of strengths and weaknesses of PROTON's non-termination check, as that is the main novelty of PROTON.

Strengths: Of the 818 Non-termination tasks in SV-COMP 2024 [2], PROTON correctly solved 627, out of which 501 witnesses could be successfully validated. There are 18 tasks, all from *systemc* directory, such as `token_ring.10.cil-1.c` and `transmitter.08.cil.c`, for which PROTON was the only tool in the competition that could identify them as non-terminating. These programs have several function calls and while loops, with around 1000 lines of code. However, none of the corresponding witnesses generated by PROTON could be validated. Further, the total time taken by PROTON for the 818 tasks is 37000 seconds, which is well below other top tools such as ULTIMATE Automizer [6] (correct solved: 548, confirmed: 537, time 100000 seconds) and 2LS [8] (correctly solved: 685, confirmed: 484, time: 52000 seconds). This shows that PROTON's approach of checking for recurrent sets at shallow loop unwinding depths is both effective and efficient.

Weaknesses: As mentioned above in *Phase 2 - Bounded Model Checking for recurrent states*, PROTON checks for a recurrent state only up to an unwind to 1000 in SV-COMP 2024. Therefore, it cannot handle cases where recurrent-states occur beyond this unwind bound, such as in `cohencu1-both-nt.c`, where the first recurrent state occurs after 2^{32} iterations. Another technical limitation of our approach is the inability to handle arrays, as it requires instrumenting each array element, which does not scale for large arrays. So, we currently ignore loops that modify arrays, and hence could not solve cases such as `Arrays02-EquivalentConstantIndices.c`. Also, since our *instrumenter* does not handle recursion currently, PROTON could not identify benchmarks like `RecursiveNonterminating-1.c` as non-terminating. Lastly, due to a bug in the *instrumenter*, one pointer was nottracked by our insutrmenter, leading to PROTON incorrectly reporting the program as non-terminating.

4 Tool Configuration and Setup

PROTON comes with an MIT license, and is available at [7,11]. To install and run the tool, follow the instructions in the file named README.txt. The benchexec tool-info module is PROTON.py and the benchmark definition file is PROTON.xml. A sample run command is: `PROTON --graphml-witness witness.graphml --propertyfile termination.prp --64 example.c.`

PROTON opted to participate only in the Termination category in SV-COMP 2024.

5 Software Project and Contributors

PROTON is developed and maintained by the authors at IIT Delhi, TCS Research, and IIT Bombay. We thank everyone who has contributed to the development of PROTON, Clang and LLVM Infrastructure, CBMC, Glucose Syrup, and Z3.

6 Data-Availability Statement

PROTON is publicly available at https://github.com/kumarmadhukar/term. The SV-COMP 2024 competition version of PROTON is available at Zenodo: https://doi.org/10.5281/zenodo.10185252. For any queries, please contact the authors.

References

1. Audemard, G., Simon, L.: On the glucose SAT solver. Int. J. Artif. Intell. Tools pp. 1840001:1–1840001:25 (2018). https://doi.org/10.1142/S0218213018400018
2. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
3. C Bounded Model Checker. https://github.com/diffblue/cbmc
4. Clarke E., Kroening D., L.F.: A tool for checking ansi-c programs. In: TACAS. pp. 168–176 (2004). https://doi.org/10.1007/978-3-540-24730-2_15
5. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: POPL. pp. 147–158. ACM (2008)
6. Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate Automizer 2024 (competition contribution). In: Proc. TACAS. LNCS , Springer (2024)
7. Karmarkar, H., Madhukar, K., Metta, R.: Proton sv-comp 2024 competition version (Nov 2023). https://doi.org/10.5281/zenodo.10185252, https://doi.org/10.5281/zenodo.10185252
8. Malík, V., Schrammel, P., Vojnar, T., Nečas, F.: 2LS: Arrays and loop unwinding (competition contribution). In: Proc. TACAS (2). pp. 529–534. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_31
9. Metta, R., Yeduru, P., Karmarkar, H., Medicherla, R.K.: Verifuzz 1.4: Checking for (non-)termination (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 594–599. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_42
10. Moura, L.M.d., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
11. Proton github. https://github.com/kumarmadhukar/term (2023), accessed: 22-Dec-2023