VeriAbs: Verification by Abstraction and Test Generation
(Competition Contribution)

Priyanka Darke(✉), Sumanth Prabhu, Bharti Chimdyalwar, Avriti Chauhan, Shrawan Kumar, Animesh Basakchowdhury, R. Venkatesh, Advaita Datar, and Raveendra Kumar Medicherla

Tata Research Development and Design Centre, Pune, India
{priyanka.darke,sumanth.prabhu,bharti.c,avriti.chauhan,shrawan.kumar,
a.basakchowdhury,r.venky,advaita.datar,raveendra.kumar}@tcs.com

Abstract. VeriAbs is a portfolio software verifier for ANSI-C programs. To prove properties with better efficiency and scalability, this version implements output abstraction with k-induction in the presence of resets. VeriAbs now generates post conditions over the abstraction to find invariants by applying Z3's tactics of quantifier elimination. These invariants are then used to generate validation witnesses. To find errors in the absence of known program bounds, VeriAbs searches for property violating inputs by applying random test generation with fuzz testing for a better scalability as compared to bounded model checking.

1 Verification Approach

Background. VeriAbs has implemented abstract acceleration [5] and k-induction techniques to scale Bounded Model Checking (BMC) for programs with loops of large or unknown bounds. VeriAbs abstracts such loops to loops of known small bounds, which can be proved by BMC. This abstraction is achieved by accelerating selected variables processed inside loops. Further, VeriAbs applies incremental k-induction to improve precision. Loops processing arrays of large and unknown sizes are substituted by abstract loops that execute a small non-deterministically chosen sequence of original loop iterations. The idea is based on the concept of *loop shrinkability* [10].

1.1 Tool Enhancements

For SV-COMP 2018, VeriAbs has been supplemented with an efficient implementation of output abstraction to prove properties, random test generation with fuzzing to find errors, and witness generation.

Output Abstraction. The SV-COMP 2017 version of VeriAbs cannot precisely validate programs with loops in which all variables are modified with non-linear

P. Darke—Jury member.

D. Beyer and M. Huisman (Eds.): TACAS 2018, LNCS 10806, pp. 457–462, 2018.
https://doi.org/10.1007/978-3-319-89963-3_32

arithmetic expressions or resets. For such programs, the current version applies an improved output abstraction [13] that simply replaces the corresponding loop with non-deterministic assignments to all the modified variables.

Search for Property Violating Inputs. In order to alleviate the lack of abstraction refinement, VeriAbs adopts an approach to search for a property violating input. To this end, it uses *fuzz testing* to search for the input that reaches the error location. Fuzz testing is a testing technique that aims to uncover run-time errors by executing the target program with a large number of inputs generated automatically and systematically. Grey-box fuzzing [3] is a fuzz testing technique that uses a light weight instrumentation to observe the target program behavior on a test run. It uses this information to generate new test inputs that might exhibit new program behaviors. VeriAbs uses American Fuzzy Lop (AFL-fuzz) [12] as the fuzz testing tool.

Witness Generation. The previous version of VeriAbs used CPAchecker [2] to generate validation witnesses from abstract programs. The SV-COMP 2018 version has implemented techniques for generation of both correctness and error witnesses. If VeriAbs concludes safety of the input program, it generates the correctness witness with loop invariants. These invariants are generated by computing the strongest postcondition equation using methods presented in [8], except for loops where the loop acceleration information is used instead. These invariants can have quantifiers and non-program variables. However, SV-COMP 2017 witness validators recognize only those invariants that are expressed as C expressions in program variables. VeriAbs uses Z3 [6] to eliminate quantifiers and non-program variables from the invariants. These invariants are added to the control flow automaton generated by CPAchecker to generate the validation witness.

The error witness generation technique is decided based on the strategy that was used to falsify the input program. When VeriAbs decides that the input program is unsafe by fuzz testing (i.e., using AFL-fuzz [12]), it generates a violation witness with a valuation of variables at the program points that assign non-deterministic values to program variables. This is achieved by replaying the execution that caused the property violation on an instrumented input program. This instrumented program prints the aforementioned valuation. In order to avoid file latency this instrumented program is only used to replay error execution. The values of variables thus obtained are used to generate error witness. On the other hand, if input program was decided to be unsafe by using BMC, then corresponding error witness is used.

Array Loop Abstraction. We abstract loops that process arrays of large or unknown sizes having quantified property, using the method based on the idea of *loop shrinkability* [10]. We call an array processing loop as *k-shrinkable* when the original program is guaranteed to be correct if execution of every sequence of k iterations of the original loop results in property, which is projected to the chosen sequence, being satisfied. A k-shrinkable loop, is replaced with an abstract loop that executes the non-deterministically chosen sequence of k iterations of the original loop and the property is also translated to be checked

over array elements corresponding to the chosen sequence of iterations only. The k-shrinkability criterion ensures that if the program is incorrect then the translated property will get violated for some sequence of k iterations, in the abstract program.

2 Verification Process and Software Architecture

The verification process of VeriAbs is shown in Fig. 1. VeriAbs passes the input C file to a Tata Consultancy Services (TCS) [1] in-house C front end to generate the intermediate representation (IR) of the program. It then analyzes this IR using PRISM, a TCS in-house program analysis framework [9] to perform the abstractions and instrumentation. It uses C Bounded Model Checker (CBMC) [4] version 5.8 with MiniSat [7] to validate the abstraction or the original program of known bounds. VeriAbs generates correctness witnesses by computing loop invariants using strongest-postcondition. It uses Z3 version 4.5.1 to eliminate quantifiers as SV-COMP requires invariants to be expressed as C expressions. These simplified invariants are added to the control flow automaton generated by CPAchecker version 1.6.1 [2]. VeriAbs uses CBMC version 5.8 for generating error witnesses. For fuzz testing, VeriAbs uses AFL-fuzz [12] version 2.35b. It invokes CBMC and AFL-fuzz sequentially, for program falsification.

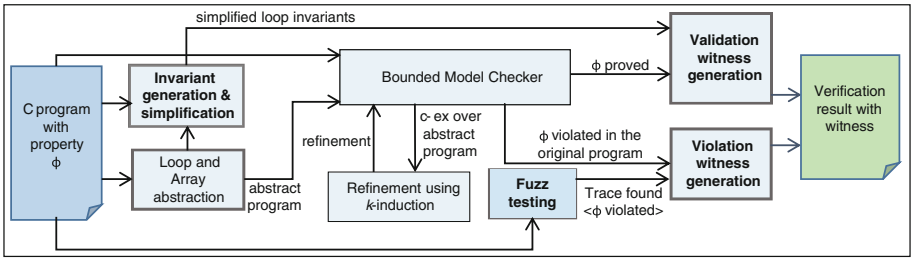

Fig. 1. The verification process of VeriAbs - enhancements are highlighted

The SV-COMP 2018 version of VeriAbs first analyzes every loop to check if it contains some linear modifications to numerical variables so that they can be precisely validated by Loop Abstraction for BMC (LABMC) [5]. If this check passes, it applies a range analysis [11] to identify ranges of those variables. On the other hand, when all variables are non-linearly modified a simpler output abstraction is applied. If the loop reads or modifies arrays, then it applies array loop abstraction as explained in Sect. 1, and then applies BMC to validate the abstraction. To find errors, VeriAbs uses the new program instrumentation for violation witness generation and grey-box fuzzing with AFL to generate witnesses for such programs.

3 Strengths and Weaknesses

The main strength of VeriAbs is that it is sound. All transformations implemented by the tool are over-approximations. In case of CBMC, the tool provides an option (`unwinding-assertions`) which ensures sufficient unwinding for proving the property. Hence if the tool reports that a property holds then it indeed holds. Another key strength is that it transforms all loops in a program to abstract loops with a known finite number of iterations, enabling the use of bounded model checkers for property proving. The main weakness of the tool is that it does not implement a refinement process that is well suited to find errors. But it can find errors using fuzz testing and bounded model checking. VeriAbs is dependent on Z3 for quantifier and non-program variable elimination from correctness witness invariants, and it is dependent on CPAchecker for generating program automata. As compared to the results of SV-COMP 2017 version, VeriAbs performed significantly better in Arrays, Loops, ECA, Sequentialized and Recursive sub categories this year.

4 Tool Setup and Configuration

The VeriAbs SV-COMP 2018 executable is available for download at the URL http://www.cmi.ac.in/~madhukar/veriabs/VeriAbs.zip. To install the tool, download the archive, extract its contents, and then follow the installation instructions in `VeriAbs/INSTALL.txt`. To execute VeriAbs, the user needs to specify the property file of the respective verification category using the `--property-file` option. The witness is generated in the current working directory as `witness.graphml`. A sample command is as follows:

```
VeriAbs/scripts/veriabs --property-file ALL.prp example.c
```

VeriAbs is participating in the ReachSafety category. The BenchExec wrapper script for the tool is `veriabs.py` and `veriabs.xml` is the benchmark description file.

5 Software Project and Contributors

VeriAbs is a verification tool maintained by TCS Research [1], and parts of it have been developed by the authors, Mohammad Afzal and other members of this organization. We would like to thank Charles Babu M and other interns who have contributed to the development of VeriAbs.

References

1. TCS Research. http://www.tcs.com/research/Pages/default.aspx
2. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16

3. Böhme, M., Pham, V.-T., Roychoudhury, A.: Coverage-based greybox fuzzing as Markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1032–1043. ACM (2016)
4. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
5. Darke, P., Chimdyalwar, B., Venkatesh, R., Shrotri, U., Metta, R.: Over-approximating loops to prove properties using bounded model checking. In: DATE (2015)
6. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
7. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
8. Gordon, M., Collavizza, H.: Forward with Hoare. In: Roscoe, A.W., Jones, C.B., Wood, K.R. (eds.) Reflections on the Work of C.A.R. Hoare, pp. 101–121. Springer, London (2010). https://doi.org/10.1007/978-1-84882-912-1_5
9. Khare, S., Saraswat, S., Kumar, S.: Static program analysis of large embedded code base: an experience. In: ISEC, pp. 99–102. ACM (2011)
10. Kumar, S., Sanyal, A., Venkatesh, R., Shah, P.: Property checking array programs using loop shrinking. In: Tools and Algorithms for the Construction and Analysis of Systems (2018)
11. Chimdyalwar, B., Kumar, S., Shrotri, U.: Precise range analysis on large industry code. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 675–678. ACM (2013)
12. Zalewski, M.: American fuzzy lop. http://lcamtuf.coredump.cx/afl/
13. Darke, P., Khanzode, M., Nair, A., Shrotri, U., Venkatesh, R.: Precise analysis of large industry code. In: Asia Pacific Software Engineering Conference, pp. 306–309 (2012)