BUBAAK: Runtime Monitoring of Program Verifiers[*] (Competition Contribution)

Marek Chalupa[(✉)] [**] and Thomas A. Henzinger

Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria
mchalupa@ista.ac.at

Abstract. The main idea behind BUBAAK is to run multiple program analyses in parallel and use *runtime monitoring* and *enforcement* to observe and control their progress in real time. The analyses send information about (un)explored states of the program and discovered invariants to a *monitor*. The monitor processes the received data and can force an analysis to stop the search of certain program parts (which have already been analyzed by other analyses), or to make it utilize a program invariant found by another analysis.
At SV-COMP 2023, the implementation of data exchange between the monitor and the analyses was not yet completed, which is why BUBAAK only ran several analyses in parallel, without any coordination. Still, BUBAAK won the meta-category *FalsificationOverall* and placed very well in several other (sub)-categories of the competition.

1 Verification Approach

Runtime monitoring (RM) [1] is a lightweight approach to observing the executions of software systems and analyzing their behavior. The system, for simplicity take a single program, is executed and observed to obtain a *trace* of events. The observed events carry information about (a subset of) actions that have been performed by the program like accesses to memory, calls of functions, or writing a text to the standard output. The trace is analyzed by the *monitor* that outputs verdicts, be it verdicts about some correctness property of the program or, e.g., information about resource consumption. *Runtime enforcement* [12] goes a step further and allows the monitor to alter the behavior of the program upon seeing some event or detecting a certain (usually faulty) behavior of the program.

RM is traditionally applied as a complementary method to static analysis to find bugs in computer programs. In BUBAAK, we use RM to do monitoring and enforcement of the *verifiers* instead of the analyzed program itself. The verifiers are manually modified to emit events about their internal actions, for example, that they have reached some part of the analyzed code or that they have discovered an invariant. The monitor gathers and analyzes these events and can decide to command a verifier to stop a search of some parts of a program or to take into account an invariant found by another verifier.

[*] This work was supported by the ERC-2020-AdG 10102009 grant.
[**] Jury member

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13994, pp. 535–540, 2023.
https://doi.org/10.1007/978-3-031-30820-8_32

2 Bubaak at SV-COMP 2023

At SV-COMP 2023 [2], the verifiers that we used are based on *forward* and *backward symbolic execution*.

(Forward) symbolic execution (SE) [14] is well-known for being efficient in searching for bugs. It aims to explore every feasible execution path of the analyzed program by building the so-called *symbolic execution tree*. Such an approach must fail if the SE tree is infinite or very large, in which case we talk about the *path explosion problem*. There are ways how to *prune* the SE tree from paths that are known to exclude buggy behavior, e.g., using interpolation [13].

Backward symbolic execution (BSE) [11] is a form of SE that searches the program backwards from error locations towards the initial locations. It has been shown [11] that BSE is equivalent to *k-induction* [16], another popular but incomplete verification technique. The incompleteness of BSE (k-induction) is caused by the lack of information about reachable states. This deficiency can be tackled by providing (often trivial) invariants that supplement the missing information [5]. These invariants can be computed externally before running BSE, or they can be computed on the fly [5,4,11]. One of the on-the-fly methods is *loop folding* and the resulting technique is called BSELF [11].

SE and BSE(LF) are well suited for analyzing safety properties, but are not suited for analyzing the termination of programs. To analyse this property, we have developed a new algorithm that has not been published yet and that we dubbed *TIIP*: *termination with inductive invariants with progress*. This algorithm runs SE, searching for non-terminating executions by remembering and comparing program states visited at loop headers. At the same time, it tries to incrementally (using a procedure similar to loop folding) compute an *inductive invariant with progress* for each visited loop. This invariant, if found, gives a pre-condition for the loop termination.

At SV-COMP 2023, we run in parallel two SE instances and one BSELF instance when checking properties *unreach-call* and *no-overflow*, SE and TIIP when checking *termination*, and just SE for memory safety properties. Using multiple SE instances at the same time makes sense because we use different verifiers (see Section 3) and their SE implementations support different features.

Because all the algorithms that we use are based on symbolic execution, the enforcement done by the monitor would effectively do a pruning of SE and BSE trees. Unfortunately, we have not managed to sufficiently debug this pruning and therefore it was disabled in the competition. As a result, Bubaak at SV-COMP 2023 only runs analyses in parallel without any coordination.

3 Software Architecture

The high-level scheme of Bubaak for SV-COMP 2023 is shown in Figure 1. Bubaak takes as input C files and the property file. Internally, it compiles and links the input files into a single llvm bitcode file [7] which is also instrumented using UBSan sanitizer [18] if the checked property is *no-overflow*. Then, verifiers are spawned according to the given property. All verifiers run in parallel

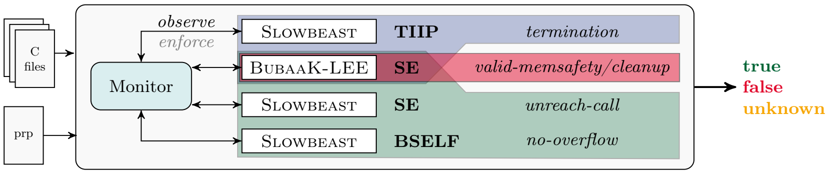

Fig. 1. The setup of BUBAAK at SV-COMP 2023. The colors indicate the properties that were checked by the different tools and algorithms.

(when there is more of them). At SV-COMP 2023, we used SLOWBEAST for SE, BSELF, and TIIP, and BUBAAK-LEE as another instance of SE[1].

SLOWBEAST [17] is a symbolic executor written in Python. It supports checking properties *unreach-call* and *no-verflow* with SE, BSE, and BSELF, and *termination* with TIIP. The tool has no or only a very limited support for properties *no-data-race*, *valid-memsafety*, and *valid-memcleanup*.

BUBAAK-LEE is a fork of symbolic executor KLEE [9] which is implemented in C++ and the current version is a merge of the upstream KLEE and JETKLEE (the fork of KLEE used in the tool SYMBIOTIC [10]) with additional modifications. These modifications mostly concern modeling standard C functions but include also partial support for 128-bit wide integers and support for global variables with external linkage. BUBAAK-LEE implements SE without any SE tree pruning and can check for all SV-COMP properties except for *no-data-race*.

Both symbolic executors use Z3 [15] as the SMT solver. The features they support differ significantly, though. For example, SLOWBEAST supports, apart from BSE(LF) and TIIP, symbolic floating-point computations, threaded programs, and incremental solving, while it does not support symbolic pointers and addresses which are features supported by BUBAAK-LEE.

The monitor is currently a part of the control scripts written in Python and at SV-COMP 2023 it monitors only the standard (error) output of the tools as monitoring anything else is redundant until the implementation of data exchange between verifiers and the monitor is finished. The only enforcement that it does at SV-COMP 2023 is terminating the analysis entirely.

Differences to SYMBIOTIC The tool SYMBIOTIC [10] also uses SLOWBEAST and a fork of KLEE, and therefore a discussion on differences between BUBAAK and SYMBIOTIC is in place. The version of SLOWBEAST used in SYMBIOTIC is outdated while BUBAAK uses the most up-to-date version (at the time of writing the paper) where a substantial part of the code has been rewritten and that contains new features including the implementation of TIIP. The relation between BUBAAK-LEE and JETKLEE is mentioned earlier in this section.

Other differences between BUBAAK and SYMBIOTIC exist: BUBAAK does not use any pre-analyses, slicing, and instrumentation (apart from the instrumenta-

[1] Because these verifiers do not compete at SV-COMP 2023 on their own, this does not make BUBAAK a meta-verifier.

Table 1. Number of benchmarks decided by individual verifiers per property.

Property	Total	BubaaK-LEE	Slowbeast
unreach-call	3263	2952	311
valid-memsafety/cleanup	3401	3401	0
termination	1417	739	678
no-overflow	4716	4399	317

tion by UBSan for the property *no-overflow*, but there Symbiotic uses its own instrumentation), and it runs the verifiers in parallel, while Symbiotic uses a sequential composition [10].

4 Strengths and Weaknesses

The combination of SE and BSELF has been previously shown to be promising [11] because SE can quickly analyse many programs and BSELF then solves hard safe instances were SE found no bug or was unable to enumerate all paths. Running TIIP in parallel with pure SE has similar advantages. Still, all of SE, BSELF, and TIIP can be computationally very demanding as the number of executions they must search may be enormous and/or their exploration may involve lots of non-trivial queries to the SMT solver.

Running multiple verifiers in parallel reduces the wall-time while eating CPU time rapidly, which may be a disadvantage in SV-COMP. A remedy for this should be finishing the data exchange support between verifiers, which will allow to avoid burning CPU time on duplicate tasks.

5 Results of Bubaak at SV-COMP 2023

The results of Bubaak were highly influenced by bugs in the implementation. The tool had 41 wrong answers, 31 of these caused by a mistake in parsing of the output of BubaaK-LEE (25 for the property *valid-memcleanup* and 6 for the property *termination*). The rest of wrong answers (10) were caused by miscellaneous bugs. After normalizing scores, these 41 wrong answers resulted in loosing almost 10000 points in the overall score.

Also, BSELF did not decide a single benchmark because of a mistake in command line arguments when invoking it. Therefore, running Slowbeast was useful mainly in the category *Termination* where TIIP was able to solve roughly half of the decided benchmarks (in the rest of cases, BubaaK-LEE successfully enumerated all execution paths). The numbers of decided benchmarks are summarized in Table 1.

Overall, Bubaak won the category *Falsification-Overall* which confirms that SE is very good in finding bugs. The tool also scored silver in the category *SoftwareSystems* where it was also the leading tool in several sub-categories.

Data Availability Statement. The version of Bubaak that competed at SV-COMP 2023 is available at Zenodo [3,6]. The source code of Bubaak is available at github [8].

References

1. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: RV'18, pp. 1–33. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-75632-5_1
2. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). LNCS , Springer (2023)
3. Beyer, D.: Verifiers and validators of the 12th Intl. Competition on Software Verification (SV-COMP 2023). Zenodo (2023). https://doi.org/10.5281/zenodo.7627829
4. Beyer, D., Dangl, M.: Software verification with PDR: an implementation of the state of the art. In: TACAS'20. LNCS, vol. 12078, pp. 3–21. Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_1
5. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: CAV'15. LNCS, vol. 9206, pp. 622–640. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
6. Bubaak artifact. Zenodo (2022). https://doi.org/10.5281/zenodo.7468631
7. llvm. `https://llvm.org`, accessed 2023-02-17
8. Bubaak repository. `https://gitlab.com/mchalupa/bubaak` (2022)
9. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI'08. pp. 209–224. USENIX Association (2008), `http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf`
10. Chalupa, M., Mihalkovič, V., Řechtáčková, A., Zaoral, L., Strejček, J.: Symbiotic 9: String analysis and backward symbolic execution with loop folding - (competition contribution). In: TACAS'22. LNCS, vol. 13244, pp. 462–467. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_32
11. Chalupa, M., Strejček, J.: Backward symbolic execution with loop folding. In: SAS'21. LNCS, vol. 12913, pp. 49–76. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_3
12. Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime failure prevention and reaction. In: Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457, pp. 103–134. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_4
13. Jaffar, J., Navas, J.A., Santosa, A.E.: Unbounded symbolic execution for program verification. In: Runtime Verification, pp. 396–411. Springer (2012). https://doi.org/10.1007/978-3-642-29860-8_32
14. King, J.C.: Symbolic execution and program testing. Communications of ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
15. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS'08. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
16. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: FMCAD'00. LNCS, vol. 1954, pp. 108–125. Springer (2000). https://doi.org/10.1007/3-540-40922-X_8
17. Slowbeast repository. `https://gitlab.com/mchalupa/slowbeast` (2022)
18. UBSan, `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`, accessed 2023-02-17