Bubaak: Dynamic Cooperative Verification (Competition Contribution)

Marek Chalupa[1(✉)] and Cedric Richter[2]

[1] Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria
mchalupa@ist.ac.at
[2] Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany
cedric.richter@uol.de

Abstract. Cooperative verification is gaining momentum in recent years. The usual setup in cooperative verification is that a verifier A is run with some pre-defined resources, and if it is not able to verify the program, the verification task is passed to a verifier B together with information learned about the program by verifier A, then the chain can continue to a verifier C, and so on. This scheme is static: tools run one after another in a fixed pre-defined order and fixed parameters and resource limits (the scheme may differ for properties to be analyzed, though).
Bubaak is a program analysis tool that allows to run multiple program verifiers in a dynamically changing combination of parallel and sequential portfolios. Bubaak starts the verification process by invoking an initial set of tasks; every task, when it is done (e.g., because of hitting a time limit or finishing its job), rewrites itself into one or more successor tasks. New tasks can be also spawned upon events generated by other tasks. This all happens dynamically based on the information gathered by finished and running tasks. During their execution, tasks that run in parallel can exchange (partial) verification artifacts, either directly or with Bubaak as an intermediary.

1 Verification Approach

The original idea of Bubaak [7] was to run multiple verifiers in parallel and apply *runtime monitoring on the verifiers* to gather information about their progress and their findings (e.g., found loop invariants). Based on the observed data, Bubaak then would instruct particular tools (at runtime) to stop their search at some points in the program, or to assume an invariant in other points.

The current version of Bubaak[3] generalizes this idea and allows to execute verifiers in a *dynamically changing* combination of parallel and sequential portfolio. What verifiers (or tools in general) are executed next and with what parameters is determined from the information gathered during the verification

M. Chalupa—Jury member and the corresponding author
[3] This paper covers the development of Bubaak since SV-COMP 2023.

A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 212–216, 2025.
https://doi.org/10.1007/978-3-031-90660-2_14

process. The verifiers are instrumented to share information about what they are doing and what they have found (this part amounts to runtime monitoring). They can also be instrumented to receive and use such information from outside: either from Bubaak or directly from other verifiers. An example situation can be that verifier A sends information about reachable states to verifier B, while verifier B informs Bubaak about found invariants and Bubaak distributes the invariants to other verifiers that may benefit from it.

Allowing the information exchange classifies Bubaak as a *cooperative verification* [4,3,2] tool. In cooperative verification, multiple verification tools help each other to increase their strength and verify more than each of the tools can verify alone. Cooperative verification tools usually employ only a static scheme that is fixed before the verification starts (the scheme is usually parametrized by the analyzed property, though). To the extent of our knowledge, Bubaak is the first tool to implement what we call *dynamic cooperative verification*: verifiers are spawned and stopped dynamically during the verification process based on the information discovered by verifiers that are currently running or that have finished running previously. At the same time, verifiers may exchange information. This information exchange includes passing artifacts to tools when they are invoked, but also direct messages (e.g., via sockets) between tools, and messages between Bubaak and tools.

The tool that is closest to Bubaak in the way how it allows tools to cooperate is CoVeriTeam [2] that allows a kind of meta-programming where verifiers are first-class objects in its domain-specific language. However, the language of CoVeriTeam is still very restricted and does not allow, e.g., to immediately react on a message in the standard output of a tool, which is something that Bubaak can do. CoVeriTeam focuses on combining off-the-shelf tools that are treated as black box. While using off-the-shelf tools has its benefits, it means that CoVeriTeam has access only to the output of the tools – standard (error) output and artifacts generated *after* the tool has finished its job. There is also no possibility to control the verifiers while they are running. In Bubaak, on the other hand, we assume the verifiers to be integrated and possibly manually modified to allow their monitoring and interception, which allows us to gain control over their execution. Naturally, off-the-shelf tools can be used by Bubaak too, but without the benefits that are brought by monitoring of their internals.

2 Software Architecture

The architecture of Bubaak is centered around *tasks* and their *rewriting*. Internally, a task executes a process, like a compiler or a verification tool, monitors its execution, and acts on events that occurred in the process or its outputs. There are also special tasks that do not execute any processes and, for example, only wait for other tasks to finish and aggregate their results.

Bubaak starts with the execution of a set of initial tasks; upon finishing, each task either yields a result, or rewrites itself into a new task or a set of new tasks. Whenever a task rewrites itself into a set of new tasks, it should also

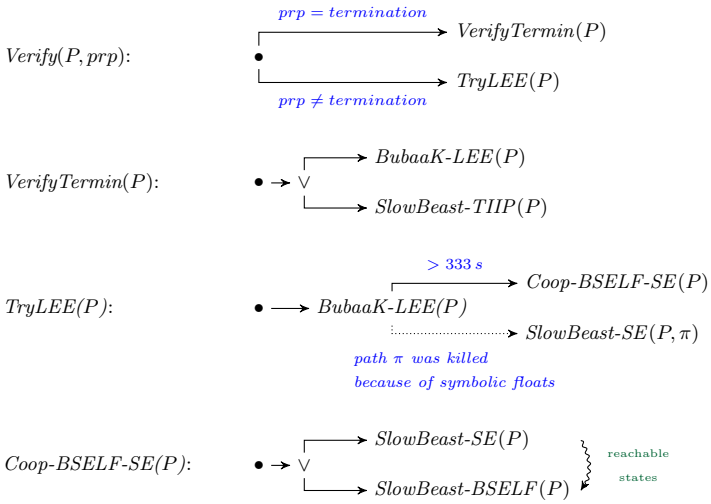

Fig. 1. The workflow of BUBAAK for SV-COMP 2025. For brevity, the scheme does not show errors handling and the result propagation.

specify how the results of the new tasks should be aggregated into a single result. A task is also allowed to spawn new sub-tasks before it has finished. Generating new tasks is not fixed in a static scheme: a task can spawn new tasks or rewrite itself into new tasks based on the context its has at hand and the information that was gathered so far from the finished and running tasks.

What tasks are executed and how they rewrite is defined by a selected *workflow*. The simplified workflow of BUBAAK in SV-COMP 2025 [1] is in Figure 1. The figure hides details like compilation into LLVM [9] (which is implemented also as a task), errors handling and the result propagation. The workflow starts with the task *Verify* that only rewrites itself into the task *VerifyTermin* if the analyzed property is *termination*, and into the task *TryLEE* otherwise.

Task *VerifyTermin* aggregates the results from two other tasks that it invokes; these tasks run *symbolic execution (SE)* provided by BUBAAK-LEE, and SLOWBEAST with *Termination with Inductive Invariants with Progress (TIIP)* [7]. The aggregated result is the one returned by the first tool that gives a conclusive result, or it ends up being *unknown/error*.

Task *TryLEE* runs BUBAAK-LEE for 333 seconds and if no result is reached by that time, the task rewrites itself into *Coop-BSELF-SE*. There may also occur the event that BUBAAK-LEE *killed* the search on a path that hit a computation involving symbolic floating point expressions. BUBAAK detects this event and spawns SLOWBEAST (in parallel to BUBAAK-LEE which keeps running) that replays the killed path and continues search from where BUBAAK-LEE stopped. This way, BUBAAK selectively combines the strengths of more mature and optimized implementation of symbolic execution in BUBAAK-LEE and the support of symbolic floats in SLOWBEAST.

Task *Coop-BSELF-SE* aggregates the results of two new tasks that it spawns. These tasks each run Slowbeast, one runs classical SE and the other runs *backward symbolic execution with loop folding (BSELF)* [8]. The algorithms run in parallel and during runtime, SE sends information about reached states into BSELF via a pipe-based channel. BSELF uses the information about reachable states to quickly filter out invalid candidates for invariants.

Workflows are only an abstraction: internally, task execution and rewriting is implemented using an event loop that handles events coming from tasks, task creation and destruction, and the results aggregation.

Note that the workflow of Bubaak in SV-COMP 2025 is substantially different from the workflow in SV-COMP 2023, where Bubaak just ran BubaaK-LEE and Slowbeast in parallel without any cooperation. As a side note, there is also another workflow that competed in SV-COMP 2025 under the name Bubaak-SpLit. This workflow implements *dynamic program splitting*: the input program is split into two „smaller" programs and each of the two *splits* is analyzed by BubaaK-LEE for 16 s. If BubaaK-LEE is unable to finish within 16 s on a split, splitting is applied again on this split and the whole process continues until at most 128 splits are generated, at which point all unverified splits are analyzed by running two instances of Slowbeast in parallel: one that runs SE and the other BSELF (without any information exchange). It is very easy to implement such kind of workflows in Bubaak.

3 Strengths and Weaknesses

The workflow of Bubaak in SV-COMP 2025 builds on a combination of SE, which is very efficient in finding bugs, and BSELF that can prove programs correct. TIIP is in fact based on the very same cornerstones [7].

The dynamic task-based architecture allows Bubaak to implement many known cooperative verification schemes, be it cooperation through residual programs [4] or through dynamic information exchange. This architecture brings a plethora of possibilities to create powerful algorithms. For SV-COMP, however, we do not use the full strength of the architecture as a simple scheme described in Section 2 proved to be working well.

The main disadvantage for Bubaak in SV-COMP is that running multiple verifiers in parallel rapidly consumes CPU time. Also, none of the verifiers we use at the moment can properly deal with concurrency.

Results of Bubaak at SV-COMP 2025 In SV-COMP 2025, Bubaak took the 3rd place in the category *FalsificationOverall*. It has also scored well in some sub-categories. Mainly in the category *SoftwareSystems*, it performed well on the *ASW-C-Common* and *uthash* benchmarks.

Spawning Slowbeast to continue killed paths from BubaaK-LEE took place in 4708 benchmarks and 1681 were successfully decided by the joint forces of the two tools within the time limit 333 s for the task *TryLEE*.

The tool gave 19 wrong answers, mostly because of a restricted support for command-line arguments of the main function (the `argv` argument) and an unhandled failure of the SMT solver in Slowbeast.

Acknowledgments. This work was in part supported by the ERC-2020-AdG 10102009 grant, and in part by the German Research Foundation (DFG) – WE2290/13-2 (Coop2).

Data Availability Statement. The version of Bubaak that competed at SV-COMP 2025 is available at Zenodo [5]. The source code of Bubaak is available at GitLab [6].

References

1. Beyer, D., Strejček, J.: Report on SV-COMP 2025. In: Proc. TACAS. LNCS , Springer (2025)
2. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS 2022, Part I. LNCS, vol. 13243, pp. 561–579. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31
3. Beyer, D., Spiessl, M., Umbricht, S.: Cooperation between automatic and interactive software verifiers. In: Proc. SEFM 2022. LNCS, vol. 13550, pp. 111–128. Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_7
4. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA 2020, Part I. LNCS, vol. 12476, pp. 143–167. Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
5. Bubaak artifact. Zenodo (2024). https://doi.org/10.5281/zenodo.14205712
6. Bubaak repository. `https://gitlab.com/mchalupa/bubaak` (2022)
7. Chalupa, M., Henzinger, T.A.: Bubaak: Runtime monitoring of program verifiers - (competition contribution). In: Proc. TACAS 2023, Part II. LNCS, vol. 13994, pp. 535–540. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_32
8. Chalupa, M., Strejček, J.: Backward symbolic execution with loop folding. In: Proc. SAS 2021. LNCS, vol. 12913, pp. 49–76. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_3
9. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO 2004. pp. 75–88. IEEE Computer Society (2004). https://doi.org/10.1109/CGO.2004.1281665