GDart: An Ensemble of Tools for Dynamic Symbolic Execution on the Java Virtual Machine (Competition Contribution)*

Malte Mues (✉)[1] and Falk Howar[1,2]

[1] TU Dortmund University, Dortmund, Germany
{malte.mues, falk.howar}@tu-dortmund.de
[2] Fraunhofer ISST, Dortmund Germanny

Abstract. GDart is an ensemble of tools allowing dynamic symbolic execution of JVM programs. The dynamic symbolic execution engine is decomposed into three different components: a symbolic decision engine (DSE), a concolic executor (SPouT), and a SMT solver backend allowing meta-strategy solving of SMT problems (JConstraints). The symbolic decision component is loosely coupled with the executor by a newly introduced communication protocol. At SV-COMP 2022, GDart solved 471 of 586 tasks finding more correct false results (302) than correct true results (169). It scored fourth place.

Keywords: Dynamic Symbolic Execution · Software Verification

1 Verification Approach

This paper presents the GDart ensemble tool, a dynamic symbolic execution engine for the JVM. Dynamic symbolic execution is a well-established technique for software testing (cf. DART [6]) and there have been already two contestants to SV-COMP 2021 using this technique (cf. JDart [7, 9] and COASTAL[3]). It is a search algorithm for systematic exploration of a program's state space for a property violation which either stops after exhausting the resource limits, exploring the complete symbolic state space, or encountering an error. The end of the search is fully configurable in GDart.

In SV-COMP 2022 [3], a dynamic symbolic execution tool (JDart (714 Points)) wins the Java track for the first time beating JBMC (700 Points) [4], a bounded model checker for Java, and Java Ranger (670 Points) [11], a symbolic execution engine extended by veritesting [1] for Java. JDart's result underlines the potential of dynamic symbolic execution for the verification of Java programs in general. The concrete implementation of JDart is closely coupled to the Java PathFinder VM (JPF-VM) [12] running the complete analysis within one virtual machine. The advantage of the JPF-VM is that it runs

* This work has been partially founded by an Amazon Research Award
[3] https://github.com/DeepseaPlatform/coastal

as a guest JVM on top of a host JVM. The analysis might mock parts of the guest JVM and use the host JVM for running side computation required to compute results used in the mock. The downside of the JPF-VM is its research tool status and that it is costly to maintain it given Java's fast pace in releasing new features.

COASTAL demonstrated for the first time what a loosely coupled architecture between the symbolic exploration engine and a concolic execution engine might look like. It instruments the bytecode with ASM[4], a java bytecode manipulation framework, to obtain symbolic traces. This makes the analysis independent of the JPF-VM. The downside is that bytecode manipulation offers less flexibility than hooking directly into the JVM.

2 Software Architecture

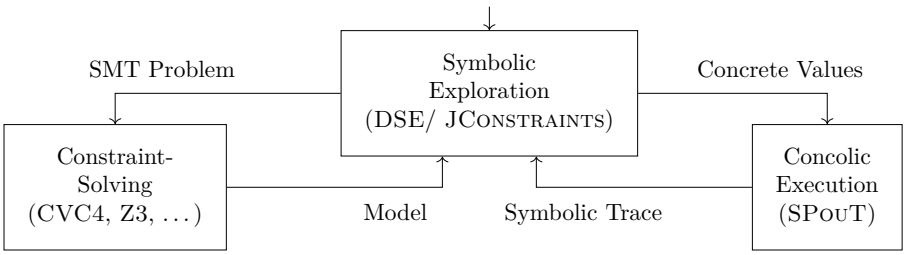

Fig. 1: GDart's ensemble architecture and the interplay between the components.

GDart takes the strengths of JDart's mocking flexibility and combines it with COASTAL's modular design. Figure 1 demonstrates the architecture of the GDart ensemble tool. The main analysis component is the symbolic explorer. It orchestrates the concolic executor and requests solutions for SMT problems from the constraint solvers powering the symbolic exploration.

Symbolic Exploration. We name the symbolic explorer "DSE" component as it does symbolic exploration and starts the concolic executor, the two main steps in applying dynamic symbolic execution. It manages the constraint tree and guides its exploration. Both steps together are the main tasks of a dynamic symbolic execution engine. To explore a path, it computes a set of concrete values that drives the concolic executor down the path of interest and seeds the executor with these values. After the termination of the executor, it parses the obtained symbolic trace and integrates it into the symbolic tree. Next, it constructs from the symbolic tree a SMT problem that describes the next path to explore and starts a constraint solver to get a model suitable to drive the execution down this path or an unsatisfiable verdict implying that the path is unreachable. The

[4] https://asm.ow2.io

search behavior of GDart is configured in the DSE. Once the search terminates, DSE generates a verification witness from the constraint tree.

Concolic Executor. One of the core contributions of GDart is the new concolic executor SPouT implemented as part of the Espresso guest language running on top of the GraalVM [13][5]. The GraalVM is an industrial-grade JVM maintained by Oracle allowing to use most of the architectural benefits the JPF-VM offered apart from state tracking. But concolic execution does not require JPF-VM's state tracing feature. SPouT can be seeded with concrete values to drive down the execution along a concrete path. In addition, it can introduce new symbolic variables for previously unknown inputs. During execution, it records manipulation and constraints checks on symbolic variables and reports a symbolic execution trace together with the concrete execution result on termination of the path exploration. Decisions on the symbolic variables are encoded in the SMT-Lib format. As SPouT maintains the two VM layers, it allows mocking of behavior in the Espresso VM running the analysis and implements a substitute executed on the host GraalVM during concolic execution the same way JDart does for mocking the environment if needed. The feature is also used for intercepting invocations of the string library in Java and encoding them symbolically.

Constraint Solving. The third component is constraint solving. DSE uses the JConstraints library to model SMT-Lib constraints internally and interact with the solver. GDart is backed by CVC4 [2] and Z3 [5]. We combine these two SMT solvers in a portfolio approach according to the CvcSeqEval strategy presented in our previous work [8].

3 Strengths and Weaknesses

GDart is the fourth place with 640 points behind JDart (714 points), JBMC (700 points), and Java Ranger (670 points). Dynamic symbolic execution tools tend to be stronger in finding property violations than confirming the absence of property violations on the SV-COMP benchmark. This is partially by design as some of the problems (e.g., those problems in the jayhorn-recursive subgroup) aim for testing the handling of tremendously large and hard to explore state spaces. GDart disproves the property in 302 cases and confirms it in 169 cases. In total, GDart answered 471 of 586 tasks correctly and none incorrect. These are 40 more correct false proved tasks than Java Ranger found (262 correct false tasks out of 466 solved tasks). In total GDart solved five more tasks than Java Ranger and 35 less than JBMC.

In direct comparison with GDart, JDart solved 192 (+23) correct true tasks and 330 (+28) correct false tasks. Three factors are contributing to the gap between GDart and JDart: the performance overhead of spinning up one JVM per executor run (We do not have the exact number, but spinning up a JVM

[5] https://www.graalvm.org

costs at least 500 ms per JVM affecting especially tasks with huge exploration trees.), technical maturity of the implementation as JDart is around for more time, and a value tracing heuristic built into JDart for tracking numerical values origin from a serialized string representation not built into GDart. The performance overhead for spinning up multiple JVMs is the only drawback that is influenced by the modular design of GDart and will not go away in the future. JDart's time per task after archiving 600 points is close to five seconds CPU time in the score-based quantile plots for CPU time while GDart's time per task reaches close to 50 seconds CPU time for the same score.

The weakness of dynamic symbolic execution is state space explosion which also affects GDart. Slowing down each executor run by spinning up new VMs is a disadvantage given the resource constraints of SV-COMP. On the bright side, with more relaxed resource limits it is possible to run the execution runs in parallel to the symbolic exploration of the constraints tree as future work for the DSE component allowing parallel breadth-first search on multi-core machines. At the moment all paths are explored sequentially.

4 Tool Setup

GDart is run with various configuration options hard-coded into the SV-COMP run scripts. More precisely, we enabled witness generation, used the described solver strategy in the constraint backend, chose a breadth-first search on the constraint tree, and used the same bounded solving as JDart. The search is configured to terminate on the first hit assertion error.

5 Software Project

The components are currently all developed at TU Dortmund by the group led by Falk Howar. DSE[6] is available under the Apache 2.0 license, JConstraints[7] as well, and SPouT[8] is available under the GPL v2 license. We also provide the run scripts for SV-COMP on GitHub[9].

6 Data Availability Statement

The GDart archive used for SV-COMP 2022 is available at Zenoodo [10].

References

1. Avgerinos, T., Rebert, A., Cha, S.K., Brumley, D.: Enhancing symbolic execution with veritesting. In: Proc. ICSE. pp. 1083–1094 (2014). https://doi.org/10.1145/2568225.2568293

[6] https://github.com/tudo-aqua/dse
[7] https://github.com/tudo-aqua/jconstraints
[8] https://github.com/tudo-aqua/spout
[9] https://github.com/tudo-aqua/gdart-svcomp

2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proc. CAV. pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
3. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). Springer (2022)
4. Cordeiro, L., Kroening, D., Schrammel, P.: JBMC: Bounded model checking for Java bytecode. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Proc. TACAS. pp. 219–223. Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_17
5. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS. pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
6. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 213–223. PLDI '05, ACM (2005). https://doi.org/10.1007/978-3-642-19237-1_4
7. Luckow, K., Dimjaevi, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamari, Z., Raman, V.: JDart: A dynamic symbolic analysis framework. In: TACAS 2016 (2016). https://doi.org/10.1007/978-3-662-49674-9_26
8. Mues, M., Howar, F.: Data-driven design and evaluation of SMT meta-solving strategies: Balancing performance, accuracy, and cost. In: Proc. ASE. pp. 179–190 (2021). https://doi.org/10.1109/ASE51524.2021.9678881
9. Mues, M., Howar, F.: JDart: Portfolio solving, breadth-first search and smt-lib strings. In: Proc. TACAS (2021). https://doi.org/10.1007/978-3-030-72013-1_30
10. Mues, M., Howar, F.: Gdart artifact for sv-comp 2022 (Feb 2022). https://doi.org/10.5281/zenodo.5957294
11. Sharma, V., Hussein, S., Whalen, M.W., McCamant, S., Visser, W.: Java Ranger: Statically summarizing regions for efficient symbolic execution of Java. In: Proc. ESEC/FSE 2020. pp. 123–134 (2020). https://doi.org/10.1145/3368089.3409734
12. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10**(2), 203–232 (Apr 2003). https://doi.org/10.1023/A:1022920129859
13. Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M.: One VM to rule them all. In: Proc. SPLASH. pp. 187–204 (2013)