

# Dynamic symbolic execution (concolic execution)

Seminar: Understanding of configurable software systems

Bipin Oli

Advisors: Prof. Sven Apel,  
Christian Hecht

Saarland Informatics Campus,  
Saarland University

## 1 Abstract

Concolic execution [7] is a software verification technique that performs symbolic execution together with concrete input values. Concrete values are selected with the help of a constraint solver to guide a program flow in a particular direction. The selection of concrete values helps to scale the verification to a larger program as it makes the symbolic constraints smaller by selecting specific branches in the program. Compared to random execution, this allows us to guide the analysis in a direction likely to have bugs which makes this technique powerful. However, in doing so we sacrifice the completeness of the analysis in favor of the depth of analysis. The sheer number of branches in a large program makes it difficult to perform a complete analysis, so we have to prioritize the branches likely to contribute to finding a bug. There have been a lot of studies to deal with this path explosion problem. In this paper, I have presented state-of-the-art methods to deal with this problem.

## 2 Introduction

- What is it?
- Where did it start?
- How does it work?
- Give an example
- Why is it important?
- It's contributions
- It's limitations
- Example of the use of this technique in finding bugs in configurable system  
eg result of SAGE, EXE, etc.

## 3 Body

### 3.1 2007: Performing dynamic test generation compositionally

[6, paper]

**Gist:** The general idea behind this new search algorithm is to perform dynamic test generation compositionally, by adapting (dualizing) known techniques for interprocedural static analysis to the context of automated dynamic test generation.

**Methodology:** new search algorithm called SMART which stands for *duce a new algorithm, dubbed SMART for Systematic Modular Automated Random Testing*, a more efficient search method than DART without compromising completeness. It tests functions in isolation, collects testing results as function summaries expressed using preconditions on function inputs and postconditions on function outputs, and then re-use those summaries when testing higher-level functions.

A SMART search performs dynamic test generation compositionally, using function summaries as defined previously. Those summaries are dynamically computed in a top-down manner through the call-flow graph *GP*. Starting from the top-level function, one executes the program (initially on some random inputs) until one reaches a first function whose execution terminates on a return or halt statement. One then backtracks inside as much as possible using DART, computing summaries for that function and each of those DART-triggered executions. When this search (backtracking) is over, one then resumes the original execution where it was called, this time treating it essentially as a black-box, i.e., without analyzing it and re-using its previously computed summary instead.

**Result:** SMART can perform dynamic test generation compositionally without any reduction in program path coverage. We also show that, given a bound on the maximum number of feasible paths in individual program functions, the number of program executions explored by SMART is linear in that bound, while the number of program executions explored by DART can be exponential in that bound. SMART = scalable DART

### 3.2 2006: Software Partitioning for Effective Automated Unit Testing

[5]

**Gist:** present an approach that identifies control and data inter-dependencies between software components using static program analysis, and divides the source code into units where highly-intertwined components are grouped together. Those units can then be tested in isolation using automated test generation techniques and tools, such as dynamic software model checkers

**Methodology:** group together functions or components that share interfaces of complexity higher than a particular threshold. Complexity is determined by the popularity and sharing of code. The idea is that if the function is very popular and is being called from a lot of places then it is likely that it is not closely liked

to any component. And, the sharing of code meaning if two functions share many of the same functions then it is likely that the higher level operation they perform is close to each other.

**Configurability part:** evaluated the effectiveness by applying the algorithm to the open source implementation of oSIP protocol (<http://www.gnu.org/software/osip/osip.html>) which is a telephony protocol for call establishment.

**Result:** showing that auto-matic software partitioning can significantly increase testcoverage without generating too many false alarms causedby unrealistic inputs being injected at interfaces betweenunits

### 3.3 2007: Hybrid concolic testing (\*\*)

**Gist:** an algorithm that in-terleaves random testing with concolic execution to obtainboth a deep and a wide exploration of program state space. Our algorithm generates test inputs automatically by inter-leaving random testing until saturation with bounded ex-haustive symbolic exploration of program points. It thuscombines the ability of random search to reachdeeppro-gram states quickly together with the ability of concolic test-ing to explore states in a neighborhood exhaustively.

**Methodlogy:** presenthybrid concolic testing, a simple algorithmthat interleaves the application of random tests with con-colic testing to achieve deep and wide exploration of theprogram state space. From the initial program state, hy-brid concolic testing starts by performing random testingto improve coverage. When random testingsaturates, thatis, does not produce any new coverage points after run-ning some predetermined number of steps, the algorithmautomatically switches to concolic executionfrom the cur-rent program stateto perform an ex-haustive bounded depthsearch for an uncovered coverage point. As soon as one isfound, the algorithm reverts back to concrete mode. Theinterleaving of random testing and concolic execution thususes both the capacity of random testing to inexpensivelygenerate deep program states through long program execu-tions and the capability of concolic testing to exhaustivelyand symbolically search for new paths with a limited looka-head.

The interleaving of random and symbolic techniques isthe crucial insight that distinguishes hybrid concolic testingfrom a naive approach that simply runs random and con-colic tests in parallel on a program. This is because manyprograms show behaviors where the program must reach aparticular statesand then follow a precise sequence of in-put events 'alpha' order to get to a required coverage point. It is often easy to reachusing random testing, but notthen to generate the precise sequence of events 'alpha'. On theother hand, while it is usually easy for concolic testing togenerate 'sigma', concolic testing gets stuck in exploring a hugenumber of program paths before even reaching the states.

In the end, hybrid concolic testing has the same limitations of symbolic execution based test generation: the discovery of uncovered points depends on the scalability and expressiveness of the constraint solver, and the exhaustive search for uncovered points is limited by the number of paths to be explored. Therefore, in general, hybrid concolic testing may not achieve 100 percent coverage, although it can improve random testing considerably. Further, the algorithm is not a panacea for all software quality issues. While we provide an automatic mechanism for test input generation, all the other effort required in testing, for example, test oracle generation, assertion based verification, and mock environment creation still have to be performed as with any other test input generation algorithm. Further, we look for code coverage, which may or may not be an indicator of code reliability.

**Configurability part:** compare random, concolic, and hybrid concolic testing on the VIM text editor (150K lines of C code) and on an implementation of the red-black tree data structure. Our experiments indicate that for a fixed testing budget, hybrid concolic testing technique outperforms both random and concolic in terms of branch coverage. of the state space exhaustively. In contrast, hybrid concolic testing switches to inexpensive random test-ing as soon as it identifies some uncovered point, relying on fast random testing to explore as much of the state space as possible. In this way, it avoids expensive constraint solving to perform exhaustive search in some part of the state space. Moreover, if random testing does not hit a new coverage point, it can take advantage of the locally exhaustive search provided by concolic testing to continue from a new coverage point

### 3.4 2008: Heuristics for Scalable Dynamic Test Generation

**Gist:**

**Methodology:**

**Configurability part:**

**Result:**

### 3.5 *paper*

**Gist:**

**Methodology:**

**Configurability part:**

**Result:**

## 4 Different catagorical bodies

- summerize, give an overview of the main points of each source and combine them into a coherent whole
- Analyze and interpret: don't just paraphrase other researchers—add your own interpretations where possible, discussing the significance of findings in relation to the literature as a whole
- Critically evaluate: mention the strengths and weaknesses of your sources
- Write in well-structured paragraphs: use transition words and topic sentences to draw connections, comparisons and contrasts

## 5 Conclusion

## 6 Papers

- 2006: they worked on backtracking algorithms for search heuristics [17]
- 2007: they combined the fuzzing techniques to improve the coverage [11] [6] [8]
- 2008: Heuristic based approach to select the branches [1]
- 2009: they worked on the fitness guided approach to improve the coverage [16]
- 2013: they boosted concolic testing by subsuming paths that are guaranteed to not hit a bug with their interpolation technique [9]
- 2014: they introduced a concept of context guided search strategy [13]
- 2018: automatic selection of suitable heuristic [2]
- 2018: template guided approach [3]
- 2018: based on probability of program paths and the cost of constraint solving [14]
- 2018: they improved the speed of SMT solver by removing the IR layer making it more practical to keep bigger constraints [18]
- 2019: fuzzy search strategy [12]
- 2019: adaptably changing search heuristics [4]
- 2021: Pathcrawler: proposed different strategies to improve the performance of concolic execution on exhaustive branch coverage [15]
- 2022: Dr. Pathfinder [10] combined concolic execution with deep reinforcement learning to prioritize deep paths over shallow ones for hybrid fuzzing

## References

1. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. pp. 443–446 (2008)

2. Cha, S., Hong, S., Lee, J., Oh, H.: Automatically generating search heuristics for concolic testing. In: *Proceedings of the 40th International Conference on Software Engineering*. pp. 1244–1254 (2018)
3. Cha, S., Lee, S., Oh, H.: Template-guided concolic testing via online learning. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. pp. 408–418 (2018)
4. Cha, S., Oh, H.: Concolic testing with adaptively changing search heuristics. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. pp. 235–245 (2019)
5. Chakrabarti, A., Godefroid, P.: Software partitioning for effective automated unit testing. In: *Proceedings of the 6th ACM & IEEE International conference on Embedded software*. pp. 262–271 (2006)
6. Godefroid, P.: Compositional dynamic test generation. In: *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. pp. 47–54 (2007)
7. 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 (2005)
8. Godefroid, P., Levin, M.Y., Molnar, D.: Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft. *Queue* 10(1), 20–27 (2012)
9. Jaffar, J., Murali, V., Navas, J.A.: Boosting concolic testing via interpolation. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. pp. 48–58 (2013)
10. Jeon, S., Moon, J.: Dr. pathfinder: hybrid fuzzing with deep reinforcement concolic execution toward deeper path-first search. *Neural Computing and Applications* 34(13), 10731–10750 (2022)
11. Majumdar, R., Sen, K.: Hybrid concolic testing. In: *29th International Conference on Software Engineering (ICSE’07)*. pp. 416–426. IEEE (2007)
12. Sabbaghi, A., Rashidy Kanan, H., Keyvanpour, M.R.: Fscf: A new fuzzy search strategy in concolic testing. *Information and Software Technology* 107, 137–158 (2019), <https://www.sciencedirect.com/science/article/pii/S0950584918302428>
13. Seo, H., Kim, S.: How we get there: A context-guided search strategy in concolic testing. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. pp. 413–424 (2014)
14. Wang, X., Sun, J., Chen, Z., Zhang, P., Wang, J., Lin, Y.: Towards optimal concolic testing. In: *Proceedings of the 40th International Conference on Software Engineering*. pp. 291–302 (2018)
15. Williams, N.: Towards exhaustive branch coverage with pathcrawler. In: *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. pp. 117–120 (2021)
16. Xie, T., Tillmann, N., De Halleux, J., Schulte, W.: Fitness-guided path exploration in dynamic symbolic execution. In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. pp. 359–368. IEEE (2009)
17. Yan, J., Zhang, J.: Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. In: *30th Annual International Computer Software and Applications Conference (COMPSAC’06)*. vol. 1, pp. 385–394. IEEE (2006)
18. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In: *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD (Aug 2018)