Concolic execution
Seminar: Understanding of configurable software systems

Bipin Oli

Advisors: Prof. Sven Apel,
Christian Hechtl

Saarland Informatics Campus,
Saarland University

1 Abstract

Configurable systems with many dials and knobs brings in a big testing challenge. In presence of many possible variants and configuration options it is very important to automate the testing as much as possible. Directed automatic random testing, popularly known as concolic execution is a primary way how it is done. Concolic execution 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 attack this problem in a different way. There have been many studies to deal with this path-explosion problem. In this paper, I have categorically presented them.

2 Introduction

Software is an integral component in many aspects of modern life. Many critical systems are driven by software. Complex relationship between man and machine, and the variety of applications has produced complex software. Much of this complexity is inherent in the problem, however the act of implementation also brings in its own complexity. Furthermore, due to the dynamic nature of human needs the software is required to evolve and change all the time. It is expected to operate under various conditions and use cases. This has given rise to configurable softwares with lot of configurable options and variants. It is important to verify the correctness of software to use it with confidence but this complexity, pace of change and the size of software makes it challenging to do so.

One way to test a software is by providing it with random inputs. It is a black-box testing method where we don't know the inner details of the program, so it is not possible to know if the input has covered all the states of the porgram. With random inputs, it is very unlikley to get the exact condition of the branch. So, we would be running different inputs without progressing any further in the program exploration. For the deeply nested state, the probability of reaching that state is practially zero. This makes it impractical to cover various states of the program. However, in abscence of other alternatives, this method is still used. Specially, a close cousin of random testing called fuzzing [6] has proven its usefulness. Fuzzing is a method of inputing gibbrish bytes of random input to a system to see if that crashes the system. It differs from random testing in a way that we are not inputing random valid values but invalid gibbrish to the program to see if the system handles that gracefully.

An alternative to random testing is symbolic exection. It is a white-box method of testing where we can see the instructions of the program, because of which, we can execute the program with symbolic inputs. We evaluate the expressions in terms of symbolic inputs and the constraints of symbols representing the conditional branches interms of those symbols. We can use constraint solver to find concrete values for the symbols satisfying the constraints. Compared to random testing, this allows us to cover the parts of program which is very hard to reach with random inputs. Unfortunately, symbolic execution doesn't scale. The number of constraints grows exponentially with the size of conditionals which makes it impossible to go deeper into the program. Also, the program can depend on external libraries and environmental factors which cannot be solved by a constraint solver. Constraint solver also struggles to solve for certain constraints such as finding an input that satisfies the output of hash. Furthermore, due to the path selection heuristic of the constraint solver the symbolic execution can get stuck, going through the same path again and again. Due to these things, the analysis gets stuck and struggles to reach a deeper state which makes symbolic exection insufficient in many practical uses.

Directed automated random testing [4, DART], more commonly known as concolic execution, is a technique that combines symbolic execution along with concreate inputs to mitigate the challenges faced by symbolic execution. When the branch is reached, a constraint solver can be used to find a concreate values that satisfies the condition. Then, the condition of the branch is negated and fed to the constraint solver to see if the other branch from the conditional is reachable. Whenever the constraint solver struggles to solve the constraints, a random concreate value can be used to simplify the process. This way concolic execution doesn't get stuck in the middle of execution.

Concolic execution has been widely used in practice to find many security bugs. SAGE [5] tool from microsoft is a prime example of it. SAGE is a whitebox fuzzing tool. It fuzzes the inputs, notably files, under the guidence of concolic execution which helps SAGE to perform fuzzing with good code coverage. Microsoft has been using SAGE since 2007 and it has found many bugs in big applications. Microsoft says [5] that SAGE helped them find one-third of all the security-related bugs in Windows 7 and they have been running SAGE 24/7 since 2008 on 100 plus machine/cores in the Microsoft security testing labs having processing more than billions constraints. Other open-source tools such as KLEE [1] have also been widely used in industry.

A major issue while concolic execution is that it is not feasible to test all the execution paths of the program. The number of paths of execution grows exponentially with the number of branches in the program. This is callend a path-explosion problem. As, it is not possible to exhaustively test all the paths in the program, many approaches have been proposed to attack this problem in different ways, such as: by prioritizing the branches likely to lead to a bug, by decomposing the analysis into smaller components, etc. I have categorially presented various approaches.

3 Approaches

3.1 Analysis on compositional parts

Earliest ideas to deal with path-explosion were based on performing analysis compositionally. Chakrabarti et al. (2006) [2] presented an approach for identifying data and control inter-dependencies using the static program analysis. The idea is to group together highly-intertwined components together and separate the program into smaller pieces such that they can be individually tested. They used popularity of the component and the sharing of code to determine the split threshold. If the function is very popular and is being called from a lot of places then it is likely to be not closely linked to any component, whereas 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. They evaluated the effectiveness of the idea by implenting it against the open source implementation of telephony protocol for call establishment called oSIP protocol (http://www.gnu.org/software/osip/osip.html), showing that the automatic program partitioning can be used to test software without generating many false alarams caused by unrealistic inputs being injected at interfaces between the units.

In 2007, Godefroid et al. [3] further proposed a complementary idea [3] to Chakrabarti et al. [2] in the context of performing concolic execution compositionally by adapting the known known techniques of static analysis. They proposed an algorithm called SMART standing for Systematic Modular Automated Random Testing, as a more efficient search method than DART [4] without compromising in the level of completeness provided by DART. SMART works by testing functions in isolation and by collecting the testing results as function summaries which are expressed using the function inputs are preconditions and outputs as post-conditions. The function summaries are re-used to test higher level functions. Starting from the top-level function of the call-flow graph the summaries are calculated in a top-down way dynamically. Using the initial random inputs the summaries are calculated from top-level function until the execution terminates. Using the DART then the analysis is backtracked computing summaries in each of those executions. When the analysis is run again the earlier computed summaries can be re-used without having to perform detailed analysis again. Due to this, SMART improves the exponential program execution of DART to linear, making it much more scalable.

3.2 Switching between random and concolic execution

- 2007: hybrid

3.3 Heuristic based approaches

- 2008: heuristics - 2009: fitness function - 2014: context guided - 2018: automatic heuristics - 2018: towards optimal - 2019: adapting changing heuristics

3.4 Interpolation based approach

- 2013: interpolation

3.5 Template-guided approach

- 2018: template guided

3.6 Optimizatin of a constraint solver

- 2018: QSYM

3.7 Using reinforcement learning to prioritize the path

- 2022: korean paper

4 papers

4.1 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 exhaustive 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 reachsusing 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 limita-tions of symbolic execution based test generation: the dis-covery of uncovered points depends on the scalability andexpressiveness of the constraint solver, and the exhaustivesearch for uncovered points is limited by the number ofpaths to be explored. Therefore, in general, hybrid concolictesting may not achieve 100 percent coverage, although it can im-prove random testing considerably. Further, the algorithmis-not a panacea for all software quality issues. While we pro-vide an automatic mechanism for test input generation, allthe other effort required in testing, for example, test oraclegeneration, assertion based verification, and mock environment creation still have to be performed as with any othertest input generation algorithm. Further, we look for codecoverage, which may or may not be an indicator of codereliability.

Configurability part: compare random, concolic, andhybrid concolic testing on the VIM text editor (150K linesof C code) and on an implementation of the red-black treedata structure. Our experiments indicate that for a fixed test-ing budget, hybrid concolic testing technique outperformsboth random and concolic in terms of branch coverage. of the state space exhaustively. In contrast, hybrid concolictesting switches to inexpensive random test-ing as soon as it identifies-someuncovered point, relying onfast random testing to explore as much of the state space aspossible. In this way, it avoids expensive constraint solv-ing to perform exhaustive search in some part of the statespace. Moreover, if random testing does not hit a new cov-erage point, it can take advantage of the locally exhaustivesearch provided by concolic testing to continue from a newcoverage point

4.2 2008: Heuristics for Scalable Dynamic Test Generation

Gist: several such heuristic search strategies, including anovel strategy guided by the control flow graph of the programunder test.

Methodlogy: We propose a search strategy that is guided by the staticstructure of the program under test, namely the control flowgraph (CFG). In this strategy, we choose branches to negatefor the purpose of test generation based on their distance inthe CFG to currently uncovered branches. We experimentallyshow that this greedy approach to maximizing the branchcoverage helps to improve such coverage faster, and to achievegreater final coverage, than the default depth-first searchstrategy of concolic testing.We further propose two random search strategies. While intraditional random testing a program is run on random inputs,these two strategies test a program along random executionpaths. The second attempts to sample uniformly from the spaceof possible program paths, while the third is a variant we havefound to be more effective in practice

have implemented these search strategies in CREST, anopen-source prototype test generation tool for C

Configurability part: We have implemented these strategies in CREST, ouropen source concolic testing tool for C, and evaluated them on twowidely-used software tools, grep 2.2 (15K lines of code) and Vim5.7 (150K lines). On these benchmarks, the presented heuristicsachieve significantly greater branch coverage on the same testingbudget than concolic testing with a traditional depth-first searchstrategy.

4.3 2009: Fitness-Guided Path Exploration in Dynamic Symbolic Execution

Gist: To address the space-explosion issue in path exploration, we propose a novelapproach called Fitnex, a search strategy that uses state-dependent fitness values (computed through a fitness func-tion) to guide path exploration. The fitness function mea-sures how close an already discovered feasible path is toa particular test target (e.g., covering a not-yet-coveredbranch). Our new fitness-guided search strategy is inte-grated with other strategies that are effective for explorationproblems where the fitness heuristic fails.

Methodlogy: The core of our approach is the Fitnex search strat-egy guided by fitness values computed with a fitness func-tion (Section 4.1). To deal with program branches notamenable to a fitness function, our approach includes integration of the Fitnex strategy with other search strategies(Section 4.2)

A fitness function (Section 4.1.1) gives a measurementon how close an explored path is to achieving a test tar-get (e.g., covering a not-yet-covered branch). We computea fitness value for each already explored path and priori-tize these known paths based on their fitness values (Sec-tion 4.1.2). We compute a fitness gain for each branch in theprogram under test and prioritize branching nodes based ontheir corresponding branches' fitness gains (Section 4.1.3).During path exploration, we give higher priority to flippinga branching node with a better (higher) fitness gain in a pathwith a better (lower) fitness value (Section 4.1.4).

4.4 2013: Boosting Concolic Testing via Interpolation

Gist: propose a new and complementarymethod based oninterpolation, that greatly mitigates path-explosion by subsuming paths that can be guaranteed to nothit a bug.

Methodlogy: first, assume that the program is annotatedwith certain bug conditions of the form "if C then bug",where if the conditionCevaluates to true along a path,the path is buggy. Then, whenever an unsatisfiable pathcondition is fed to the solver, an interpolant is generatedat each program point along the path. The interpolant at agiven program point can be seen as a formula thatsuccinctlycaptures the reason of infeasibility of paths at the programpoint. In other words it succinctly captures the reason whypaths through the program point are not buggy. As a re-sult, if the program point is encountered again through adifferent path such that the interpolant is implied, the newpath can besubsumed, because it can be guaranteed to notbe buggy. The exponential savings are due to the fact that not only is the new path subsumed, but also the paths thatthis new path would spawn by negating its branches.

Unfortunately, methods such as [12, 14, 11] cannot be useddirectly for concolic testing due to several challenges. First,the soundness of these methods relies on the assumptionthat an interpolant at a node has been computed after ex-ploring the entire "tree" of paths that arise from the node.In concolic testing, this assumption is invalid as the testercan impose an arbitrary search order. For example, concolictesters such as Crest [3] and KLEE [4] use often many heuris-tics that may follow a random walk through the search space,thus making this method unsound. To address this problem,we need to keep track of nodes whose trees have been ex-plored fully (in which case we say the node is annotated withafull-interpolant) or partially (similarly, ahalf-interpolant).Under this new setting, only nodes with full-interpolants arecapable of subsumption in a sound manner. As a result, theamount of subsumption depends on how often nodes get an-notated with full-interpolants from the paths explored by thecon-colic tester. Unfortunately our benchmarks in Section 6showed that the above method by itself results in very fewnodes with full-interpolants, thereby pro-viding poor bene-fit to the concolic tester, because the tester rarely exploresthe entire tree of paths arising from a node. Hence, an im-portant challenge now is to "accelerate" the formation offull-interpolants in order to increase subsumption.

For this,we introduce a novel technique calledgreedy confirmationthat performs limited path exploration (i.e., execution of afew extra paths) by itself, guided by subsumption, with anaim to produce a full-interpolant at nodes currently anno-tated with a half-interpolant. It is worth mentioning thatthis execution of few paths is done without interfering withthe search order of the concolic tester. This technique ul-timately resulted in a significant increase in subsumptionfor our benchmarks, and is vital for the effectiveness of ourmethod.We implemented our method and compared it with a pub-licly available concolic tester, Crest [3]. We found that forthe price of a reasonable overhead to compute interpolants,a large percentage of paths executed by those heuristics canbe subsumed thereby increasing their coverage substantially.

Result: We attacked the path-explosion problem of concolic test-ing by pruning redundant paths using interpolation. Thechallenge for interpolation in concolic testing is the lack ofcontrol of search order. To solve this, we presented the concept of half and full interpolants that makes the use of in-terpolants sound, and greedy confirmation that acceleratesthe formation of full-interpolants thereby increasing the like-lihood of subsuming paths.

4.5 2014: A Context-Guided Search Strategy inConcolic Testing

Gist: While moststrategies focus on coverage information in the branch selection process, we introduce CGS which considers contextinformation, that is, how the execution reaches the branch.Our evaluation results show that CGS outperforms otherstrategies.

Methodlogy: CGS explores branches in the current execution tree. Foreach visited branch, CGSexaminesthe branch and decideswhether toselectthe branch for the next input orskipit. CGS looks athow the execution reaches the current branch by calculat-ingk-contextof the branch from its preceding branches and-dominator information. Then, thek-contextis comparedwith the context of previously selected branches which isstored in thecontext cache. If thek-contextis new, thebranch is selected for the next input. Otherwise, CGS skipsthe branch.

Configurability part: We evaluate CGS on top of two publicly available concolictesting tools, CREST [13] and CarFastTool [29]

4.6 2018: Automatically Generating Search Heuristics for Concolic Testing

Gist: developed a parame-terized search heuristic for concolic testing with an optimizationalgorithm to efficiently search for good parameter values. We hopethat our technique can supplant the laborious and less rewardingtask of manually tuning search heuristics of concolic testing.

Methodlogy: this paper presents a new approachthat automatically generates search heuristics for concolic testing.To this end, we use two key ideas. First, we define aparameterizedsearch heuristic, which creates a large class of search heuristics.The parameterized heuristic reduces the problem of designing agood search heuristic into a problem of finding a good parametervalue. Second, we present a search algorithm specialized to concolictesting. The search space that the parameterized heuristic poses isintractably large. Our algorithm effectively guides the search byiteratively refining the search space based on the feedback fromprevious runs of concolic testing

Configurability part: We have implemented our techniquein CREST [3] and evaluated it on 10 C programs (0.5–150KLoC)

4.7 2018: Template-Guided Concolic Testing via Online Learning

Gist: a template is a partially symbolized inputvector whose job is to reduce the search space. However, choos-ing a right set of templates is nontrivial and significantly affectsthe final performance of our approach. We present an algo-rithmthat automatically learns useful templates online, based on datacollected from previous runs of concolic testing. The experimen-tal results with open-source programs show that our techniqueachieves greater branch coverage and finds bugs more effectivelythan conventional concolic testing

In our approach, concolictesting uses a set of templates to exploit common input patterns that improve coverage effectively, where the templates are automatically generated through online learning algorithm based on thefeedback from past runs of concolic testing.

Methodlogy: we present template-guided concolic testing, a newtechnique for adaptively reducing the search space of concolic test-ing. The key idea is to guide concolic testing with templates, whichrestrict the input space by selectively generating symbolic variables.Unlike conventional concolic testing that tracks all input valuessymbolically, our technique treats a set of selected input valuesas symbolic and fixes unselected inputs with particular concreteinputs, thereby reducing the original search space. A challenge,however, is choosing input values to track symbolically and replac-ing the remaining inputs with appropriate values. To address thischallenge, we develop an algorithm that performs concolic testingwhile automatically generating, using, and refining templates. Thealgorithm is based on two key ideas. First, by using the sequentialpattern mining [9], we generate the candidate templates from a setof effective test-cases, where the test-cases contribute to improvingcode coverage and are collected while conventional concolic test-ing is performed. Second, we use an algorithm that learns effectivetemplates from the candidates during concolic testing. Our algo-rithm iteratively ranks the candidates based on the effectivenessof templates that were evaluated in the previous runs. Our tech-nique is orthogonal to the existing techniques and can be fruitfullycombined with them, in particular with the state-of-the-art searchheuristics

Configurability part: Experimental results show that our approach outperforms con-ventional concolic testing in term of branch coverage and bug-finding. We have implemented our approach in CREST [7] andcompared our technique with conventional concolic testing foropen-source C programs of medium size (up to 165K LOC). For allbenchmarks, our technique achieves significantly higher branchcoverage compared to conventional concolic testing. For example,for vim-5.7, we have performed both techniques for 70 hours, whereour technique exclusively covered 883 branches that conventionalconcolic testing failed to reach. Our technique also succeeded infinding real bugs that can be triggered in the latest versions of threeopen-source C programs: sed-4.4, grep-3.1 and gawk-4.21.

4.8 2018: Towards Optimal Concolic Testing

Gist: show the optimal strategy can be defined based onthe probability of program paths and the cost of constraintsolving. The problem of identifying the optimal strategy isthen reduced to a model checking problem of Markov DecisionProcesses with Costs. Secondly, in view of the complexity inidentifying the optimal strategy, we design a greedy algorithmfor approximating the optimal strategy.

Methodlogy: aim to develop a framework which allowsus to define and compute the optimal concolic testing strate-gy. That is, we aim to systematically answer when to applyconcrete execution, when to apply symbolic execution andwhich program path to apply symbolic execution to. In par-ticular, we make the following technical contributions. Firstly,we show that the optimal concolic testing strategy can bedefined based on a probabilistic abstraction of program behaviors. Secondly, we show that the problem of identifyingthe optimal strategy can be reduced to a model checkingproblem of Markov Decision Processes with Costs. As a re-sult, we can reuse existing tools and algorithms to solve theproblem. Thirdly, we evaluate existing heuristics empiricallyusing a set of simulated experiments and show that theyhave much room to improve. Fourthly, in view of the highcomplexity in computing the optimal strategy, we propose agreedy algorithm which approximates the optimal one. Weempirically evaluate the greedy algorithm based on both sim-ulated experiments and experiments with C programs, andshow that it gains better performance than existing heuristicsin KLEE

4.9 2018: Qsym : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing

Gist: we design a fast concolicexecution engine, calledQSYM, to support hybrid fuzzing.The key idea is to tightly integrate the symbolic emulationwith the native execution using dynamic binary transla-tion, making it possible to implement more fine-grained,so faster, instruction-level symbolic emulation. Additionally,QSYMloosens the strict soundness requirements ofconventional concolic executors for better performance,yet takes advantage of a faster fuzzer for validation, pro-viding unprecedented opportunities for performance op-timizations, e.g., optimistically solving constraints andpruning uninteresting basic blocks

Methodlgy: •Fast concolic execution through efficient emula-tion: We improved the performance of concolicexecution by optimizing emulation speed and reduc-ing emulation usage. Our analysis identified thatsymbol generation emulation was the major perfor-mance bottleneck of concolic execution such that weresolved it with instruction-level selective symbolicexecution, advanced constraints optimization tech-niques, and tied symbolic and concolic executions.

•Efficient repetitive testing and concrete environ-ment.The efficiency ofQSYM-makes re-execution-based repetitive testing and the concrete executionof external environments practical. Because of this,QSYMis free from snapshots incurring significantperformance degradation and incomplete environ-ment models resulting in incorrect symbolic execu-tion due to its non-reusable nature.

•New heuristics for hybrid fuzzing.We proposednew heuristics tailored for hybrid fuzzing to solveunsatisfiable paths optimistically and to prune outcompute-intensive back blocks, thereby makingQSYMproceed.

Configurability part: Our evaluation shows thatQSYMdoes not just outperform state-of-the-art fuzzers (i.e., found 14×morebugs than VUzzer in the LAVA-M dataset, and outper-formed Driller in104binaries out of126), but also found13 previously unknown security bugsineightreal-worldprograms like Dropbox Lepton, ffmpeg, and OpenJPEG,which have already been intensively tested by the state-of-the-art fuzzers, AFL and OSS-Fuzz.

4.10 2019: Concolic testing with adaptively changing search heuristics

Gist: adapting search heuristics on the fly via an algorithm that learns new search heuristics based on the knowledge accumulated during concolic testing

Methodlgy: we present an algorithm that automaticallylearns and switches search heuristics during concolic testing. Thealgorithm maintains a set of search heuristics and continuouslychanges them during the testing process. To do so, we first definethe space of possible search heuristics using the idea of parametricsearch heuristic recently proposed in prior work [5]. A technicalchallenge is how to adaptively switch search heuristics in the pre-defined space. We address this challenge with a new concolic testingalgorithm that (1) accumulates the knowledge about the previouslyevaluated search heuristics, (2) learns the probabilistic distributionsof the effective and ineffective search heuristics from the accumu-lated knowledge, and (3) samples a new set of search heuristics from the distributions. The algorithm iteratively performs thesethree steps until it exhausts a given time budget.

4.11 2022: Dr.PathFinder: hybrid fuzzing with deep reinforcement concolic execution toward deeper path-first search

Gist: propose a concolic execution algorithm that combines deep reinforcement learning with a hybrid fuzzing solution, Dr.PathFinder. When the reinforcement learning agent encounters a branch during concolic execution, it evaluates the state and determines the search path. In this process,"shallow" paths are pruned, and "deep" paths are searched first. This reduces unnecessary exploration, allowing the efficient memory usage and alleviating the state explosion problem.

Methodlogy: We formally define a learning algorithm for a deep reinforcement learning agent that allows concolic execution to first search for a deeper path.

We present a deeper path-first search concolic execution algorithm using a reinforcement learning agent and a hybrid fuzzer called Dr.PathFinder.

Result: In experiments with the CB-multios dataset for deep bug cases, Dr.PathFinder discovered approximately five times more bugs than AFL and two times more than Driller-AFL. In addition to finding more bugs, Dr.PathFinder generated 19 times fewer test cases and used at least 2bugs located in deep paths, Dr.PathFinder had limitation to find bugs located at shallow paths, which we discussed.

4.12 ¡¡paper¿¿

Gist:

Methodlogy:

Configurability part:

Result:

5 Different catagorical bodies

see ralated work section of https://dl.acm.org/doi/pdf/10.1145/2635868.2635872 i.e A Context-Guided Search Strategy inConcolic Testing paper

also of https://dl.acm.org/doi/pdf/10.1145/3180155.3180166 i.e 2018: Automatically Generating Search Heuristics for Concolic Testing paper

- 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

6 Conclusion

References

1. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
2. 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)
3. 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)
4. 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)
5. 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)
6. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of unix utilities. Communications of the ACM 33(12), 32–44 (1990)