CEUR-WS.org/Vol-3612/QuASoQ_2023_Paper_01.pdf

Coyote C++: An Industrial-Strength Fully Automated Unit Testing Tool

Sanghoon Rho[1], Philipp Martens[1], Seungcheol Shin[1], Yeoneo Kim[1], Hoon Heo[2] and SeungHyun Oh[2]

[1] *CODEMIND Corporation, Seoul, South Korea*
[2] *Hyundai KEFICO Corporation, Gyeonggi-Do, South Korea*

Abstract
Coyote C++ is an automated testing tool that uses a sophisticated concolic-execution-based approach to realize fully automated unit testing for C and C++. While concolic testing has proven effective for languages such as C and Java, tools have struggled to achieve a practical level of automation for C++ due to its many syntactical intricacies and overall complexity. Coyote C++ is the first automated testing tool to breach the barrier and bring automated unit testing for C++ to a practical level suitable for industrial adoption, consistently reaching around 90% code coverage. Notably, this testing process requires no user involvement and performs test harness generation, test case generation and test execution with "one-click" automation. In this paper, we introduce Coyote C++ by outlining its high-level structure and discussing the core design decisions that shaped the implementation of its concolic execution engine. Finally, we demonstrate that Coyote C++ is capable of achieving high coverage results within a reasonable timespan by presenting the results from experiments on both open-source and industrial software.

Keywords
automated unit test, coverage testing, concolic execution, C++, LLVM

1. Introduction

The significance of testing in software engineering is continuously escalating, necessitating thorough validation methods such as white-box testing. However, given the rapid increase in code scale and complexity in the software industry, white-box testing can be time-consuming and resource-intensive[1], often leading to budget constraints. For this reason, there has been a long-standing need for automation in white-box testing.

Lately, efforts to automate white-box unit testing are approaching practical feasibility, with automated testing showing promising results for Java [2, 3], C [4, 5, 6], binary code [7, 8], and a few other programming languages [9, 10, 11]. Conversely, adopting this technology for C++ has proven to be challenging due to the language's unique features and overall complexity [12]. Implicitly invoked copy or move constructors and templates with all their intricacies are just two examples of C++ language features that are especially difficult to handle in automated white-box unit testing.

In this paper, we introduce Coyote C++, an automated unit testing tool designed for C/C++. With a single click, Coyote C++ streamlines the entire testing process, from harness generation and test case generation to test execution. The automated test case generation is based on concolic execution, a modern variant of symbolic execution, and features exquisite harness generation capabilities.

The paper outlines the underlying technologies on which Coyote C++ achieves a practical level of high coverage through test case generation. In order to practically utilize automated unit testing tools in the field, we propose that a testing speed of around 10,000 logical LOC of executable statements per hour with statement coverage above 90% and branch coverage above 80% should be desirable. Currently, Coyote C++ is achieving elevated levels of coverage and performance according to these criteria, and is thus being effectively applied and utilized by our customers in the automotive industry.

The rest of this paper is organized as follows. We first look at research on concolic-execution-based unit testing and then examine design decisions made by existing systems to build efficient concolic execution engines in related works. Next, we provide an overview of the implementation of Coyote C++, and present test results obtained from open-source projects and real-world industrial projects. Finally, we conclude the paper by outlining our plans for further improving Coyote C++.

2. Related works

Symbolic execution [13] is a static program analysis technique that interprets programs with symbolic values

QuASoQ 2023: 11th International Workshop on Quantitative Approaches to Software Quality, December 04, 2023, Seoul, South Korea

rho@codemind.co.kr (S. Rho); philipp.m@codemind.co.kr (P. Martens); shin@codemind.co.kr (S. Shin); yeoneo@codemind.co.kr (Y. Kim); hoon.heo@hyundai-kefico.com (H. Heo); seunghyun.oh@hyundai-kefico.com (S. Oh)

© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

rather than concrete values. Due to scalability issues with symbolic execution, this technique has been extended into concolic execution [5, 6]. The main idea of concolic execution is to compute test inputs from path conditions which are obtained by tracking both concrete values and symbolic values. Concolic execution has been anticipated in the automated testing domain due to its known success in test case generation. However, this research has not yet reached a practical level of test generation for whole programs.

Nevertheless, concolic execution is known to be remarkably successful in unit test generation, e.g. for Java [2, 3] and C [4, 5, 6]. For C++ however, automated testing has still been far from viable for industrial purposes despite recent research efforts [14, 12].

When implementing concolic execution there are many options for realizing various aspects of the engine [15]. Especially the engine's execution mode, analysis target, handling of the path explosion problem, and its memory model can largely affect the performance of the concolic execution engine in terms of coverage and execution time.

2.1. Online/Offline Mode

Concolic execution can be implemented in online or offline mode. In online mode, the concolic execution engine explores multiple paths in a single run by forking on branch points. The advantage of this method is that there is no need to re-execute the common prefixes of multiple paths. However, it requires a substantial amount of memory to store all the states of multiple paths. Offline mode on the other hand explores only one path in a single run. This method requires less memory than online mode, making it better suited for parallelization. However, since offline mode always starts at the beginning of the program for every path, it spends a considerable amount of time on re-examining common path prefixes. Prominent tools using online mode are KLEE [16], Mayhem [17], and S^{2}E [18], whereas SAGE [7] utilizes offline concolic execution.

2.2. Emulation/Instrumentation

There are two main methods for collecting information about the execution path taken during concrete execution of the program under test. The first method performs symbolic execution at the same time as concrete execution by running the program under test inside of an emulator such as QEMU [19]. The second method instead instruments the program under test with code that handles symbolic execution and the collection of information about the concrete execution of the program. Well-known emulator-based tools are angr [20] and KLEE [16], while QSYM [21] and CREST [22] are instrumentation-based.

2.3. Mitigating Path Explosion

Another important design decision is how to deal with the path explosion problem commonly encountered when performing concolic execution on programs with complex control flow. In such situations, the search space of concolic execution can grow exponentially due to the many possible combinations of branches. To avoid this issue, concolic execution engines use a variety of heuristic search strategies. Notable search strategies include DFS (depth-first search), BFS (breadth-first search), random path selection, coverage-optimized search, and adaptive heuristics [15, 23].

2.4. Memory Model

When modelling the symbolic memory of a concolic execution engine, one can choose between treating memory addresses as symbolic or concrete values. The symbolic approach can theoretically handle all possible paths, but this approach may cause path constraints to become too complex for current SMT solvers. On the other hand, using concrete addresses might not cover all possible paths due to overly simplified path conditions. In practice, a fully symbolic model is used by tools like KLEE [16], and a concrete address model is used by SAGE [7] among others. Additionally, there are tools like Mayhem [17] that use a combination of symbolic and concrete addressing schemes.

3. The Design of Coyote C++

3.1. Overview

In this chapter, we present an overview of Coyote C++ and discuss the core decisions that influenced its design. As shown in the diagram in Fig. 1, the Coyote C++ tool is divided into two main parts. The first part builds executable test files based on harness generation, while the second part handles generating test cases through concolic execution.

In the first phase, Coyote C++ uses a harness generator module to automatically generate test stubs and test drivers for test execution and inserts instrumentation code for concolic execution. This instrumentation is performed on LLVM IR level. Next, the binary generation module compiles the created testbed to executable files used in the second part.

While running the executable test file in the second phase, the instrumentation code produces trace files containing information about the concrete program execution on the level of LLVM IR instructions. These trace

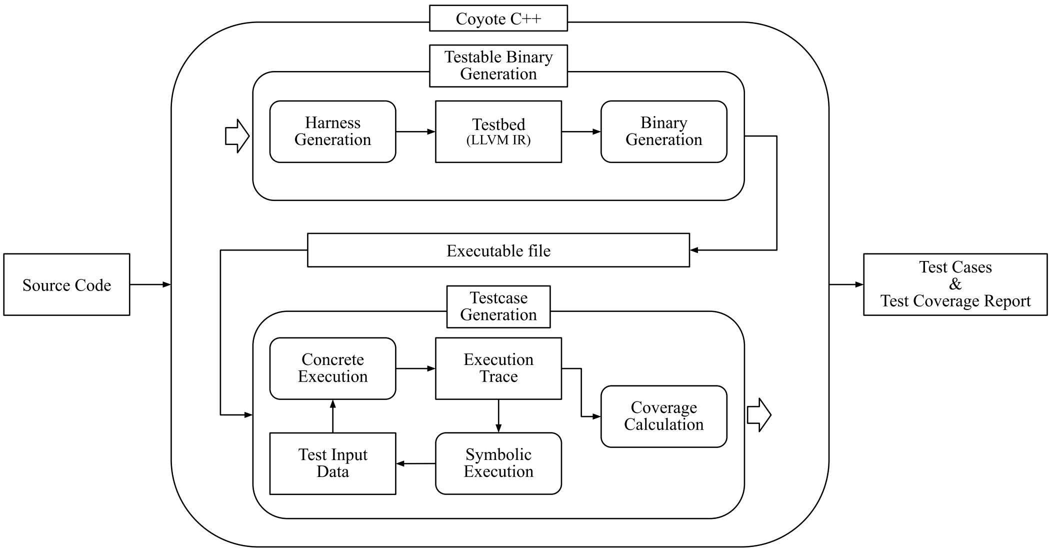

Figure 1: Overview of Coyote C++.

files are then used to reconstruct their respective execution paths, and with this information symbolic execution is performed on the LLVM IR level to generate new test input data. When this concolic execution cycle terminates, the achieved test coverage is calculated based on the generated trace files.

3.2. Design Decisions of Coyote C++

While implementing Coyote C++, many important design decisions had to made. In the modules responsible for the testable binary generation, these decisions were generally made with the goal of enabling a wide range of transformations on intermediate code models while retaining a sufficiently strong connection between these models and the original source code. Most design decisions affecting the testcase generation phase were strongly influenced by the need to find a suitable tradeoff between the achieved code coverage and performance in terms of test time or resource consumption.

A fundamental design decision made in Coyote C++ is using LLVM IR as its symbolic execution target. This allows for more precision than doing source level symbolic execution while retaining more information about the original source code that would be lost when lowering even further to the assembly level. Also, using LLVM as a foundation for Coyote C++ allows for greater freedom in code transformations during harness generation, bypassing syntactic constraints present on the source code level.

We decided to implement offline testing by inserting instrumentation code into the LLVM IR code of the testbed during testable binary generation. The main reason for choosing offline testing over online testing is that it is more suitable for parallelization, which is essential for providing good testing performance. Additionally, offline testing is more advantageous from a memory management standpoint.

A key factor for achieving high code coverage is the search strategy that controls in which order the possible execution paths of a program are explored. During testcase generation, the test files are initially executed with all test inputs set to default values. The trace files generated from this are then analyzed using concolic execution techniques to create new test case inputs for visiting new paths. As our search strategy for exploring of candidate paths, we adopted a hybrid approach that combines CCS (Code Coverage Search) and DFS. CCS focuses on exploring code areas that have not been traversed yet, making it advantageous for quickly reaching high coverage. However, because CCS performs rather aggressive pruning on execution paths, it may produce unsatisfiable path conditions in certain situations. To make up for these issues, we also use the DFS strategy in addition to CCS. DFS is a search strategy that has the potential to cover code areas not covered by CCS, but it comes with the drawback of substantial time consumption. Usually, either of these strategies terminates once every branch it has discovered has been explored. Concolic execution may however also be terminated early if a designated amount of test cases has been generated or if a timeout has been reached.

Finally, a significant factor influencing the performance of concolic execution in C++ is the memory model. Sim-

Table 1
Results on Open-Source Projects

Project Info					Coverage		Test Time
Name (C/C++)	*Files*	*Functions*	*Statements*	*Branches*	*Statement*	*Branch*	[m]
nuklear (C)	39	609	9,284	4,309	93.7%	87.1%	55
libsodium (C)	94	887	8,003	1,651	96.5%	89.7%	6
mathc (C)	1	843	4,192	190	99.9%	100.0%	3
aubio (C)	53	520	5,916	1,797	95.7%	92.4%	14
s2n-tls (C)	175	1,621	16,734	15,512	86.7%	81.3%	68
yaml-cpp (C++)	32	367	3,050	1,300	96.9%	95.5%	11
qnite (C++)	48	637	4,294	1,035	95.2%	89.1%	37
json-voorhees (C++)	21	451	2,507	764	92.5%	88.7%	5
QPULib (C++)	24	278	3,561	1,398	87.8%	83.8%	3
jsoncpp (C++)	3	309	2,802	1,148	91.2%	86.3%	11
Total	490	6,522	60,343	29,104	93.6%	89.4%	213

Table 2
Coverage Results from Hyundai KEFICO

Project Info					Coverage		Test Time
Name	*Files*	*Functions*	*Statements*	*Branches*	*Statement*	*Branch*	
Target A	1,855	5,129	129,131	40,718	92.8%	86.8%	N/A
Target B	83	1,774	11,828	3,078	97.4%	90.7%	
Target C	69	375	6,526	2,339	85.5%	79.9%	
Total	2,007	7,278	147,485	46,135	92.9%	86.7%	

ilar to Mayhem, the approach implemented in Coyote C++reads values from memory symbolically but writes values to concrete memory addresses. Utilizing symbolic reads in contrast to reading from concrete addresses leads to a more faithful representation of path constraints, thereby enhancing the potential for generating appropriate test cases. For write operations however, we chose to rely on concrete addresses because symbolic writes are prone to making the process of solving the path constraints overly expensive.

4. Experimental Results

To showcase the performance of Coyote C++, we present experimental results for a set of diverse open-source projects as well as several industrial software projects from one of our customers, Hyundai KEFICO. While our tool allows user to add test cases and write driver functions for achieving higher coverage, all experimental results were obtained through one-click automation without any user intervention.

4.1. Experiment on Open-Source Projects

For the first evaluation, we chose to reuse the test set curated by Shin and Yoo for a survey on white-box automated testing tools [24], as it contains open-source projects written in C and C++ from a wide variety of application domains and was composed specifically for the evaluation of automated testing tools such as Coyote C++. This survey also concluded that currently no other commercial tools truly support automated testing for C++ programs. Among open-source tools for C++, CITRUS [12] is no longer publicly available, and we were not able to successfully apply UTBot [14] to the selected test projects due to its rather limited support for the C++ syntax. Thus, unfortunately there were no suitable candidates to compare Coyote C++ against in terms of coverage and test time.

Table 1 shows the statement[1] and branch coverage results achieved by Coyote C++ on the ten open-source projects in the test set as well as the time needed for conducting the automated test generation and execution for each project. Coyote C++ achieves statement coverages between 86.7% (s2n-tls) and 99.9% (mathc) as well

[1] As statements we consider only executable lines of code. In contrast to physical lines of code, this excludes e.g. whitespace, comments, and type declarations.

as branch coverages between 81.3% (s2n-tls) and 100% (mathc). Summing up the number of overall covered lines/branches and dividing them by the total number of lines and branches in all ten projects yields a remarkable combined statement coverage of 92.5% and branch coverage of 84.9%.

The test times presented in table 1 were attained from an Intel Core i7-13700 system with 64GB of RAM running Ubuntu 20.04. Overall, the test of all ten projects combined only took about three and a half hours, with individual testing times ranging between three minutes (mathc) and just above one hour (s2n-tls). That makes it more than six times faster than the test times reported in the previously mentioned study [24], which we consider a significant improvement despite possible minor differences between test setups. Furthermore, with the exception of the qnite project, the testing speed on all projects surpasses our definition of practicality, with an overall testing speed of roughly 17,000 statements per hour.

4.2. Results on Industry Projects

Table 2 presents testing results produced by Coyote C++ on automotive control software projects from our customer Hyundai KEFICO, a member of Hyundai Motors Group. As details about these projects such as their actual names are strictly internal information, we will refer to them as target A, B and C.

The coverage results for these industrial projects are quite similar to the open-source projects, with an average statement coverage of 92.9% and an average branch coverage of 86.7%. At our customer, Coyote C++ is employed not in a controlled test environment but rather in a business setting on multiple machines with varying hardware specifications. Due to these circumstances and the fact that a subset of the test results were produced incrementally over a longer period of time, we presently do not have any meaningful test time measurements available to report for these projects.

While project C individually yields a slightly subpar coverage, our notion of practicality in terms of coverage achieved (statement coverage >90%, branch coverage >80%) is upheld both by projects A and B individually as well as all three projects combined. This again reinforces our claim that Coyote C++ is not simply a research prototype which only works on a limited set of specially curated programs but is rather a mature tool that can also handle more challenging industry software. Also, it should be noted that automated testing with such high coverage results for these projects is only possible because Coyote C++ has explicit handling for some common code patterns in embedded software that would usually make automated testing difficult or plainly impossible, such as the usage of fixed memory addresses in code.

5. Conclusion and Future Work

In this paper, we presented Coyote C++, an industry-grade automated testing tool based on concolic execution. After describing the general tool architecture, we discussed the core design decisions for our implementation of its concolic testing engine. Finally, we evaluated the performance of Coyote C++ in terms of achieved coverage and testing time on both a test set of diverse open-source projects and industry code from one of our corporate customers. We were able to demonstrate that Coyote C++ can achieve high statement/branch coverage of around 90% or higher in a reasonable amount of time for software projects from a wide variety of application domains.

While Coyote C++ is already yielding promising results both on open-source projects and in real industry applications, it is our plan to continuously improve the tool both in terms of reliably achieving high coverage results and broadening its capabilities in the field of automated testing.

One goal for the near future is target testing for embedded software. Our tool currently performs host testing, meaning tests are not executed on the hardware that would run the program under test in a production environment, but rather on a separate computer, e.g., a test engineer's computer or a test server. Especially in the embedded domain however, the discrepancy between embedded hardware in the production environment and the consumer or server hardware in the testing environment may lead to inaccurate test results. Thus, we are planning to implement target testing support so that tests may be run directly on production hardware.

Approaching the goal of increasing automated test coverage from a different perspective, we also strive to provide users of our tool with feedback as to how they should change their code so that Coyote C++ will likely yield better coverage results for it. While we would like to give such guidance on the basis of code metrics, our initial investigations have shown that traditional code metrics such as cyclomatic complexity have little to no correlation with automated test coverage. Thus, we see the need for more thorough research involving the development of new code metrics that can serve as a better estimate for the coverage results produced by automated testing and Coyote C++ in particular.

References

[1] L. Luo, Software testing techniques, Institute for software research international Carnegie mellon university Pittsburgh, PA 15232 (2001) 19.

[2] G. Fraser, A. Arcuri, A large-scale evaluation of automated unit test generation using EvoSuite, ACM

Trans. Softw. Eng. Methodol. 24 (2014). URL: https://doi.org/10.1145/2685612. doi:10.1145/2685612.

[3] K. Sen, G. Agha, Cute and jcute: Concolic unit testing and explicit path model-checking tools, in: T. Ball, R. B. Jones (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 419–423.

[4] Y. Kim, D. Lee, J. Baek, M. Kim, Concolic testing for high test coverage and reduced human effort in automotive industry, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), IEEE, 2019, pp. 151–160.

[5] K. Sen, D. Marinov, G. Agha, CUTE: A concolic unit testing engine for C, ACM SIGSOFT Software Engineering Notes 30 (2005) 263–272.

[6] P. Godefroid, N. Klarlund, K. Sen, DART: Directed automated random testing, in: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005, pp. 213–223.

[7] P. Godefroid, M. Y. Levin, D. Molnar, SAGE: whitebox fuzzing for security testing, Communications of the ACM 55 (2012) 40–44.

[8] F. Saudel, J. Salwan, Triton: A dynamic symbolic execution framework, in: Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, 2015, pp. 31–54.

[9] N. Tillmann, J. de Halleux, Pex–white box test generation for .net, in: B. Beckert, R. Hähnle (Eds.), Tests and Proofs, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 134–153.

[10] A. Giantsios, N. Papaspyrou, K. Sagonas, Concolic testing for functional languages, Science of Computer Programming 147 (2017) 109–134. URL: https://www.sciencedirect.com/science/article/pii/S0167642317300837. doi:https://doi.org/10.1016/j.scico.2017.04.008.

[11] K. Sen, S. Kalasapur, T. Brutch, S. Gibbs, Jalangi: A selective record-replay and dynamic analysis framework for javascript, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, Association for Computing Machinery, New York, NY, USA, 2013, p. 488–498. URL: https://doi.org/10.1145/2491411.2491447. doi:10.1145/2491411.2491447.

[12] R. S. Herlim, Y. Kim, M. Kim, CITRUS: Automated unit testing tool for real-world C++ programs, in: 2022 IEEE Conference on Software Testing, Verification and Validation (ICST), 2022, pp. 400–410. doi:10.1109/ICST53961.2022.00046.

[13] J. C. King, A new approach to program testing, ACM Sigplan Notices 10 (1975) 228–233.

[14] D. Ivanov, A. Babushkin, S. Grigoryev, P. Iatchenii, V. Kalugin, E. Kichin, E. Kulikov, A. Misonizhnik, D. Mordvinov, S. Morozov, et al., UnitTestBot: Automated unit test generation for C code in integrated development environments, in: 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), IEEE, 2023, pp. 380–384.

[15] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, I. Finocchi, A survey of symbolic execution techniques, ACM Comput. Surv. 51 (2018). URL: https://doi.org/10.1145/3182657. doi:10.1145/3182657.

[16] C. Cadar, D. Dunbar, D. R. Engler, et al., KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs., in: OSDI, volume 8, 2008, pp. 209–224.

[17] S. K. Cha, T. Avgerinos, A. Rebert, D. Brumley, Unleashing Mayhem on binary code, in: IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA, IEEE Computer Society, 2012, pp. 380–394. URL: http://doi.ieeecomputersociety.org/10.1109/SP.2012.31. doi:10.1109/SP.2012.31.

[18] V. Chipounov, V. Kuznetsov, G. Candea, The S2E platform: Design, implementation, and applications, ACM Transactions on Computer Systems (TOCS) 30 (2012) 1–49.

[19] F. Bellard, QEMU, a fast and portable dynamic translator., in: USENIX annual technical conference, FREENIX Track, volume 41, Califor-nia, USA, 2005, p. 46.

[20] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, G. Vigna, SoK: (state of) the art of war: offensive techniques in binary analysis, in: IEEE Symposium on Security and Privacy, 2016.

[21] I. Yun, S. Lee, M. Xu, Y. Jang, T. Kim, QSYM: A practical concolic execution engine tailored for hybrid fuzzing, in: 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 745–761.

[22] J. Burnim, K. Sen, Heuristics for scalable dynamic test generation, in: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008, pp. 443–446. doi:10.1109/ASE.2008.69.

[23] S. Cha, S. Hong, J. Bak, J. Kim, J. Lee, H. Oh, Enhancing dynamic symbolic execution by automatically learning search heuristics, IEEE Transactions on Software Engineering 48 (2022) 3640–3663. doi:10.1109/TSE.2021.3101870.

[24] K. Shin, Y. Ryu, Performance and functionality evaluation of white-box software testing tools, part 2, https://csrc.kaist.ac.kr/blog/2023/01/25/performance-and-functionality-evaluation-of-white-box-software-testing-tools-part-2/, 2023. Accessed: 2023-10-11.