NxtUnit: Automated Unit Test Generation for Go

Siwei Wang
siwei.wang@bytedance.com
Bytedance
Mountain View, USA

Xue Mao
maoxue.marissa@bytedance.com
Bytedance
Beijing, China

Ziguang Cao
caoziguang@bytedance.com
Bytedance
Beijing, China

Yujun Gao
gaoyujun@bytedance.com
Bytedance
Beijing, China

Qucheng Shen
qusheng.shen@bytedance.com
Bytedance
Mountan View, USA

Chao Peng
pengchao.x@bytedance.com
Bytedance
Beijing, China

ABSTRACT

Automated test generation has been extensively studied for dynamically compiled or typed programming languages like Java and Python. However, Go, a popular statically compiled and typed programming language for server application development, has received limited support from existing tools. To address this gap, we present NxtUnit, an automatic unit test generation tool for Go that uses random testing and is well-suited for microservice architecture. NxtUnit employs a random approach to generate unit tests quickly, making it ideal for smoke testing and providing quick quality feedback. It comes with three types of interfaces: an integrated development environment (IDE) plugin, a command-line interface (CLI), and a browser-based platform. The plugin and CLI tool allow engineers to write unit tests more efficiently, while the platform provides unit test visualization and asynchronous unit test generation. We evaluated NxtUnit by generating unit tests for 13 open-source repositories and 500 ByteDance in-house repositories, resulting in a code coverage of 20.74% for in-house repositories. We conducted a survey among Bytedance engineers and found that NxtUnit can save them 48% of the time on writing unit tests. We have made the CLI tool available at https://github.com/bytedance/nxt_unit.

KEYWORDS

Go, Automated Test Generation

ACM Reference Format:
Siwei Wang, Xue Mao, Ziguang Cao, Yujun Gao, Qucheng Shen, and Chao Peng. 2023. NxtUnit: Automated Unit Test Generation for Go. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE '23), June 14–16, 2023, Oulu, Finland*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3593434.3593443

1 INTRODUCTION

Automated unit test generation has been extensively studied for decades. Previous research focused on dynamically compiled and dynamically typed programming languages, e.g. EvoSuite[4] and Randoop[8] for Java, Pynguin[7] for Python, etc. Search-based test generation [1] run different test cases thousands of times in a short period of time. Another approach, exemplified by KLEE[2], employs dynamic symbols to execute input parameters for the program under test. However, these methods do not produce well-written, maintainable unit tests explicitly for developers to use.

Go[1] is a rising programming language designed and managed by Google, featuring built-in concurrency and garbage collection that make it ideal for building server applications. In 2021, Go was ranked as the 12th most commonly used programming language[2]. At ByteDance, the majority of our server applications are written in Go.

A dependable and effective method for ensuring the quality of our Go programs, which serve billions of users, is urgently required. We have evaluated several tools, such as final-unit[3] which doesn't support mocking remote procedure calls (RPC)[4]. Another option, EvoMaster[5], generates system-level test cases for web applications, but it does not provide native support for RESTful[6] and Thrift[7] functions, making it incompatible with our microservice architecture[10]. In our environment, an ideal test generate tool should meet 2 requirements: (1) being automated without human intervention and (2) compatible with the microservice architecture. Our initial attempt was to use search-based algorithms, but we encountered two problems. Firstly, Search-based testing requires numerous iterations [6] during test generation, resulting in a slow pace of iteration as a result of having to recompile the code after each modification [5, 9]. Secondly, the computational explosion problem arises when using the search-based feedback mechanism for large and complex programs [3].

To address these limitations, we present NxtUnit, an automated unit test generation tool that employs random testing and alters input and simulates downstream call output. NxtUnit records assertion values as the ground truth, laying the groundwork for future regression testing. NxtUnit creates a mixture of mocked outputs from downstream calls and input parameters for its test scenarios. Successful tests are retained and adapted to address various portions of the source code. NxtUnit offers three main interfaces: a server that generates unit tests and provides a web application for

EASE '23, June 14–16, 2023, Oulu, Finland

ACM ISBN 979-8-4007-0044-6/23/06…$15.00
https://doi.org/10.1145/3593434.3593443

[1] https://go.dev/
[2] https://www.tiobe.com/tiobe-index/
[3] https://github.com/wimspaargaren/final-unit
[4] https://en.wikipedia.org/wiki/Remote_procedure_call
[5] https://github.com/EMResearch/EvoMaster
[6] https://en.wikipedia.org/wiki/Representational_state_transfer
[7] https://en.wikipedia.org/wiki/Apache_Thrift

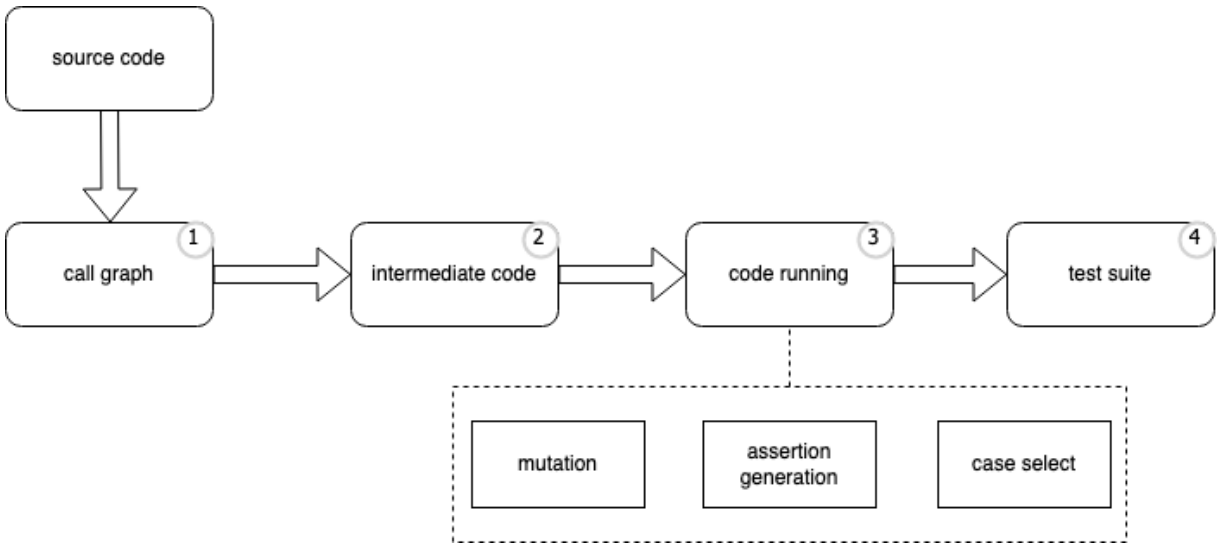

Figure 1: Workflow of NxtUnit

developers to review tests and their results, an IDE plugin, and a CLI for developers. NxtUnit is capable of generating either unit test templates or fully developed unit tests. At ByteDance, NxtUnit supports hundreds of developers each week.

2 IMPLEMENTATION AND USAGE

2.1 Workflow

Figure 1 illustrates NxtUnit's workflow. The process starts with parsing the program's source code and transforming it into static single assignment (SSA) form[8]. The SSA serves as a foundation for building the call graph (① in Figure 1), which links tested functions to their dependent functions. This information is essential for NxtUnit when randomly mocking callees while constructing test cases.

NxtUnit generates intermediate code (② in Figure 1) for code execution. During this stage, NxtUnit populates the template[9] with tested function's parameters and downstream callees. When executing the code (③), NxtUnit runs the code created in ② , focusing on three primary tasks: mutation, assertion generation, and test case selection. The code produces a designated number of candidates, regulated by hyperparameters. During the mutation stage, the inputs and mocked downstream outputs for each candidate exhibit differences at runtime. For instance, if there is no default value established, the int value will be chosen from the range between the minimum and maximum integers. NxtUnit randomly mocks downstream calls and assigns different return values. For instance, if a function has 5 downstream calls, NxtUnit might mock 2 of them while keeping the other 3 callable.

NxtUnit utilizes Go reflect[10], a package in the Go programming language that facilitates inspection and manipulation of variable structures and values during runtime. This package allows for examining types, interface values, and other variable properties, as well as creating and modifying values programmatically. Using Go reflect simplifies mutation implementation compared to the go-types solution[11], as it provides more information and a more accessible interface. For example, Go reflect can accurately obtain a variable's alias and the integrated path of the imported package from the alias, while go-types cannot easily provide this information.

NxtUnit employs suitable strategies to accommodate microservice architectures. For example, it actively searches for instantiated methods for variables within the tested function. To enhance the interpretability of generated test cases, NxtUnit applies a set of anthropomorphic rules for mutating variables. One such rule involves constructing the value of rules based on the variable's name. For instance, IP addresses in the code are mutated to resemble real IP addresses, making the test cases more relatable and understandable in real-world scenarios.

During the assertion generation stage, NxtUnit logs assertion values as the ground truth, serving as the basis for regression testing. In cases where generated assertions are flawed, NxtUnit may fail to produce them. Test case selection is based on code coverage, with NxtUnit recording each test case's path and selecting a set of cases that maximize the test suite's coverage.

In the end, NxtUnit creates the final test suite (④ in Figure 1) by selecting the most comprehensive combination of test cases for each function. NxtUnit might fail to generate test cases for various reasons such as compilation errors and test crashes. In the internal version of NxtUnit, a series of error codes are available to notify engineers of these issues and assist in identifying bugs within the programs.

[8]https://pkg.go.dev/Go.org/x/tools/go/ssa
[9]A concrete example is available at https://github.com/bytedance/nxt_unit
[10]https://pkg.go.dev/reflect

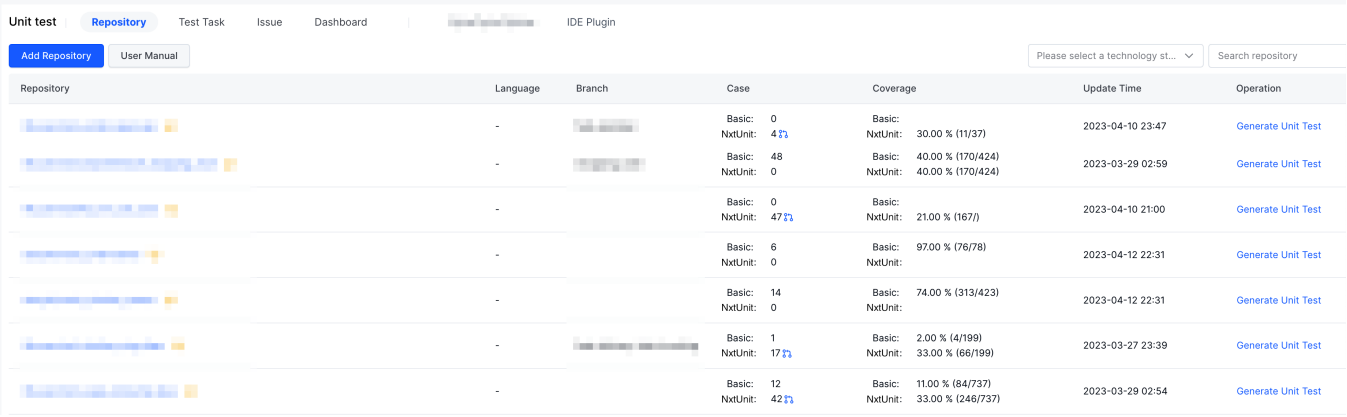

Figure 2: Web-based platform. Due to confidentiality reasons, we have obscured some information.

2.2 Usage

NxtUnit offers a user-friendly solution for generating unit tests in Go projects and comes in three formats: a plugin for popular Integrated Developing Environments (IDEs) including VScode and Goland[12], a Command Line Interface (CLI), and a webpage-based platform. By executing a simple command from CLI, users can easily initiate the unit test generation process. NxtUnit also boasts additional features, such as producing test templates and test suites for entire files. These templates enable users to swiftly write a comprehensive set of tests. Furthermore, users have the flexibility to tailor these templates to their specific requirements by incorporating project-specific data or functions.

The plugin version works as an interface encapsulating the CLI version, enabling users to generate unit tests by simply clicking a button rather than entering commands. The web-platform version is depicted in Figure 2. The "case" column highlights NxtUnit's impact. The "basic" field indicates the number of original test cases in the repository, while "NxtUnit" refers to the number of test cases generated by the application in the backend. The platform version facilitates increased coverage by allowing users to send a merge request containing all generated tests to the code repository.

3 EVALUATION

The evaluation of NxtUnit was conducted on two sets of candidate repositories, one set consisting of 500 Bytedance repositories[13] and the other set consisting of 13 highly-starred repositories from the internet[14]. The rest of the 100 repositories were not included for the following reasons: (1) non-maintained repositories were excluded, (2) repositories with an excessive number of lines of code, such as kubernetes[15], were eliminated, (3) repositories using Go vendor[16] were not included as NxtUnit is not compatible with this outdated package management system, and (4) repositories that could not be built without satisfying prerequisites. These 13 public repositories consist of at least 246 functions, indicating that they are not small projects. No pre-modifications were made to either data set. NxtUnit was executed for a maximum of 24 hours for both sets of repositories, using hyperparameters such as a mutation ratio of 0.2, a maximum struct mutation level of 6, a timeout duration of 40 seconds, and 4 candidates. The mutation ratio represents the probability of a variable undergoing mutation. If a variable is not mutated, it will be assigned its default value based on its data type. For instance, in Go, the default value for a pointer type is nil. The maximum struct mutation level denotes the maximum depth of struct layers that NxtUnit will examine. If the struct layers exceed this limit, NxtUnit will directly use default values to initialize the value. The timeout parameter indicates the maximum runtime during the code execution phase, as the processing time for downstream functions can be lengthy when they are not mocked. The 4 candidates refer to the generation of four potential candidates during each run.

Table 1 showcases the results of the experiment conducted by NxtUnit on 13 GitHub repositories. "All functions" represents the total number of functions in the repository. Each function has a corresponding test suite, which indicates the successfully generated test suites. "Original test coverage" refers to the coverage before using NxtUnit. "NxtUnit tests without original test" signifies the coverage NxtUnit can provide after excluding the original tests. The "NxtUnit tests with original test" represents the total coverage achieved by combining the original tests with those generated by NxtUnit. The average code coverage for public repositories increased from 44.86% to 50.37%. The coverage improvement results from NxtUnit generating more tests, thus enhancing the overall test suite. However, nsq is an exception, as it had no coverage data due to an unknown program panic [17] when running the tests, but NxtUnit still generated 594 tested functions out of 621. For Bytedance repositories, the average code coverage increased from 3.48% to 20.74%, with an average coverage without the original test of 14.1%. Table 1 showcases the success rate and generation time for single functions, reflecting user experience when using NxtUnit. The success rate for

[11] https://github.com/golang/example/blob/master/gotypes/go-types.md
[12] https://www.jetbrains.com/go/
[13] Data is not publicly available due to confidentiality reasons.
[14] https://evanli.github.io/Github-Ranking/Top100/Go.html
[15] https://github.com/kubernetes/kubernetes
[16] https://go.dev/ref/mod

[17] Panic is an exception arisen at runtime in Go

Table 1: Github Repositories NxtUnit Experiment

Github repo	Generated test suites	All functions	Original test coverage (%)	NxtUnit without original test coverage (%)	NxtUnit plus original test coverage (%)	Generation duration per function(s)
cobra	236	246	87.2	25	87.3	1368
consul	84	9503	30.2	3.1	30.2	2395
dive	287	295	39.8	41.4	67.3	6611
drone	1310	1489	51.1	16.8	51.8	9976
echo	425	425	95.2	11	95.5	2354
esbuild	1337	1513	75.8	17.9	80.3	15095
fiber	1538	2382	39.5	37.3	46.8	16075
fzf	23	525	36.1	4.4	36.5	789
gin	132	435	95.1	16.3	95.1	636
gorm	388	391	26.3	13.1	36.1	8672
kit	709	774	80.6	25.3	85	5950
logrus	205	222	7.8	7.6	35.6	614
nsq	594	621	0	0	0	6430

generating a single function was 50% in Bytedance repositories and 73% in public repositories. The generation time was 26 seconds for Bytedance repositories and 10 seconds for public repositories. The higher function generation rate and faster generation time in public repositories can be attributed to their well-maintained nature and fewer dependencies. However, the coverage in public repositories is lower than in Bytedance repositories due to the specific mutation logic for Bytedance variables.

Additionally, we conducted a survey among 20 of our engineers. They reported that before using NxtUnit, it took them an average of 5 minutes to write a single unit test. Specifically, 4 engineers stated that their unit test writing time fell within the range of 2-4 minutes. 8 engineers claimed it took them 5 minutes, and the remaining 8 engineers indicated that they needed 6-10 minutes[18]. The survey also showed that 9 people claimed that NxtUnit saved them 50% of their time of writing unit tests, 7 people stated that it saved 60% - 90% of their time, and 4 people mentioned that the time saved ranged from 0% to 50%. These results demonstrate that NxtUnit can significantly reduce the time required for writing tests, saving engineers an average of 48% in time.

4 CONCLUSION

Go has become popular due to its high performance. NxtUnit, an automated unit test generation tool for Go, has helped many Bytedance developers write unit tests more efficiently. We explained the concept behind NxtUnit and conducted an empirical analysis of its code generation rate and successful generation rate in both Bytedance and public repositories. The results show that NxtUnit has a strong performance. In the future, NxtUnit aims to make unit tests more readable and improve code coverage, including branch coverage, with further modifications.

REFERENCES

[1] Arthur Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja Vos. 2011. Symbolic search-based testing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 53–62.

[2] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[3] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235.

[4] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[5] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. 32–43.

[6] Xin Kong, Yen-Lun Chen, Wei Xie, and Xinyu Wu. 2012. A novel paddy field algorithm based on pattern search method. In *2012 IEEE international conference on information and automation*. IEEE, 686–690.

[7] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (2022), 168–172.

[8] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84.

[9] Yohei Ueda and Moriyoshi Ohara. 2017. Performance competitiveness of a statically compiled language for server-side Web applications. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 13–22.

[10] Man Zhang, Andrea Arcuri, Yonggang Li, Kaiming Xue, Zhao Wang, Jian Huo, and Weiwei Huang. 2022. Fuzzing Microservices In Industry: Experience of Applying EvoMaster at Meituan. *arXiv preprint arXiv:2208.03988* (2022).

[18] due to space limitations, specific numbers are not listed here