# MUTAGEN: Faster Mutation-Based Random Testing

Agustín Mista
*Chalmers University of Technology*
Gothenburg, Sweden
mista@chalmers.se

***Abstract*—We present MUTAGEN, a fully automated mutation-oriented framework for property-based testing. Our tool uses novel heuristics to improve the performance of the testing loop, and it is capable of finding complex bugs within seconds. We evaluate MUTAGEN by generating random WebAssembly programs that we use to find bugs in a faulty validator.**

***Index Terms*—random testing, mutation, heuristics**

## I. INTRODUCTION

Using randomly generated inputs is a popular and powerful approach when it comes to testing software [1]. While plenty, the tools designed for this purpose can be divided into two main categories: those that use an existing corpus of inputs, and those that generate such inputs from scratch.

On one hand, corpus-based tools create new inputs by combining and mutating seeds from their input corpora using several heuristics [2], [3]. Being mostly black-box, this approach is often limited by the lack of knowledge about the structure of its inputs, although there exist efforts to improve this situation, e.g., by using grammars describing the syntactic structure of the generated data while combining seeds in order to produce syntactically valid test cases [4]–[6].

On the other hand, generational approaches can circumvent this limitation by generating valid data from scratch using specialized random generators [7]–[10]. However, writing good random generators by hand is a demanding task that involves several iterations of trial and error and can take hundreds of hours [11]. While there exist tools that tackle this problem by automatically synthesizing (with varying degrees of complexity) random generators directly from the static information present in the codebase [12]–[15], such approaches are unable to derive suitable generators when the target data involves complex invariants like those required to generate random programs, e.g., well-scopedness and well-typedness. In such cases, generators obtained by automatic tools are extremely unlikely to produce random data with enough structure to penetrate deep layers of our systems before being discarded.

Recently, Lampropoulos et al. introduced FuzzChick [11], a property-based testing framework for the Coq programming language that borrows ideas from the fuzzing community to generate highly structured values while using automatically derived generators. Instead of continuously generating random invalid test cases from scratch, FuzzChick keeps a queue of interesting previously executed test cases that can be mutated using type-preserving transformations in order to produce new ones. The logic behind this is simple, mutating an interesting test case in a small way (at the data constructor level) has a much higher probability of producing a new interesting test case than generating a new one from scratch using a naïve generator. FuzzChick is likely to preserve the semantic structure of the mutated data, as mutations are applied directly at the data type level – the random AST in case of generating code.

In order to work in practice, FuzzChick relies on execution traces to distinguish which executed test cases were interesting and are therefore worth mutating. Mutated test cases are considered interesting for mutation only when they produce new execution traces – any other test case is simply discarded. While powerful, the implementation of FuzzChick is relatively simple, leaving the door open for future extensions.

In this work we present MUTAGEN, a random property-based testing framework implemented in the strongly typed language Haskell that follows the mutational approach behind FuzzChick, extending it with novel mutation heuristics designed to converge to counterexamples in fewer tests. We outline these heuristics in the next section and evaluate them in Section III.

## II. MUTAGEN'S HEURISTICS

This section introduces some of MUTAGEN's novel heuristics.

*a) Exhaustive Uniform Mutations:* Perhaps the biggest distinction between FuzzChick and MUTAGEN lies in how mutations are applied over test cases. Along with random data generators, FuzzChick automatically derives simple type-preserving mutators. These mutators work in a simple recursive manner: with uniform probability, they either mutate a node at the top level of the given value, or apply a mutation recursively to one of its subexpressions. This mechanism favors mutations to happen on the shallower levels of their inputs, while deeper mutations are unlikely to happen due to the multiplicative decline of their probability on each recursive call.

To find complex bugs, we believe that mutations should be able to occur deep inside of the generated values in a reasonable proportion. To allow this, the mutators derived by our tool follow a different approach. *They first traverse the input, collecting the path to each mutable subexpression.* Then, the testing loop schedules mutations targeted over each one of them, ensuring that all of them are mutated in a timely manner.

Furthermore, for a given target subexpression, the testing loop produces and tests every possible mutation exhaustively, reducing the reliance on randomness in order to find bugs. This approach is inspired by exhaustive testing tools like SmallCheck [16] or Korat [17]. However, testing mutations exhaustively comes attached to a high testing cost per test

| Bug | QuickCheck | | | MUTAGEN | | | MUTAGEN (no inheritance) | | | MUTAGEN (no scheduling) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time (s) | Passed | Discarded | Time (s) | Passed | Discarded | Time (s) | Passed | Discarded | Time (s) | Passed | Discarded |
| 1 | N/A | 14.8 | 1000000 | 2.16 | 2194 | 11706.7 | 4.06 | 4079 | 15123.3 | 3659.45 | 101536.4 | 134107.1 |
| 2 | N/A | 12.7 | 1000000 | 0.07 | 83.7 | 131.6 | 0.23 | 159.5 | 206.5 | 0.33 | 261.9 | 234 |
| 3 | N/A | 14.9 | 1000000 | 0.03 | 56.7 | 91.8 | 0.04 | 51 | 71.9 | 0.28 | 217.2 | 107.1 |
| 4 | N/A | 13.7 | 1000000 | 0.21 | 157.2 | 314.2 | 0.63 | 427.2 | 903.3 | 10.85 | 6740.6 | 7372.2 |
| 5 | N/A | 14.3 | 1000000 | 6.45 | 6464 | 34856 | 0.82 | 560.1 | 1543.6 | 454.26 | 31577.9 | 43837.5 |

Table 1. Time to first failure, passed and discarded tests accross different bugs for the faulty WebAssembly validator. Mean values computed after 10 executions.

candidate, so the next heuristic is, in part, focused on reducing the time complexity of the testing loop of our tool.

*b) Mutation Inheritance:* The second mutation heuristic focuses on preserving the semantically important subexpressions generated on previous steps. For this purpose, child mutants keep a record of the subexpressions that were already mutated by their ancestors. This allows each mutant to focus on the previously untouched subexpressions, as well as the ones freshly generated by its parent.

As a consequence, this heuristic greatly reduces the available positions where mutations can occur on each new mutant. This, in turn, helps to cope with the cost of running mutations exhaustively, as described above.

*c) Mutants Scheduling:* FuzzChick keeps a queue of mutation candidates obtained by analyzing the execution trace of every executed mutant. If a candidate executes a completely new branch in the code, it is inserted at the end of the queue, and the testing loop will first have to process every candidate ahead of it before it can start mutating this new (and likely more interesting) test case.

To account for this, our tool uses a preemptive (FIFO) schedule with priority for scheduling mutant candidates. In this setting, we capture the depth at which each new execution trace differs from all the previous ones. This depth is later used to give more priority to those candidates that "discover" new parts of the code at earlier stages, favoring a wider traversal of the execution trace space in less time.

## III. Case Study: WebAssembly Validator

This section evaluates how the heuristics implemented in our MUTAGEN affect the testing performance. For this purpose, we use the validator of the WebAssembly programming language [18] as a case study. We took an exiting Haskell implementation of the WebAssembly validator [19] and injected several bugs into it. These bugs let invalid WebAssembly modules pass the validation process. Then, we tested for false positives of the faulty validator by comparing it against the output of the validator from the WebAssembly reference implementation (written in OCaml). This was expressed using the following property:

```
prop_valid :: WasmModule -> Result
prop_valid wasm = validHs wasm ==> validOCaml wasm
```

Using `validHs` as a precondition, we avoid executing the reference validator on invalid modules. Instead, they get automatically discarded and the testing loop continues.

Table 1 shows the performance of our tool against QuickCheck [20] (the reference tool for random testing in Haskell) across five different bugs, using the same automatically derived random generator. To evaluate the effectiveness of our inheritance and scheduling heuristics, we included the results obtained when they are disabled.

Unsurprisingly, QuickCheck fails to find any of the bugs we planted after generating more than a million values. This is because the automatically derived generator is virtually unable to produce valid modules on its own. In comparison, MUTAGEN is able to find all the bugs in a couple of seconds. Guided by the execution traces, the type-preserving mutations that our tool applies allows us to run a much larger number of valid WebAssembly modules through the reference validator, which in turn helps finding bugs faster. As an example of the level of invariants required to find bugs, the following WebAssembly module AST is a counterexample for bug #1:

```
Module {
  types=[FuncType{params=[], results=[]}],
  functions=[
    Function{resultType=0, localTypes=[], body=[
      I32Const 0,
      If{resultType=[], then=[], else=[I32Const 0]}]]},
  tables=[], mems=[], globals=[], elems=[],
  datas=[], start=Nothing, imports=[], exports=[],
}
```

This bug causes false validations by not checking that the type of an **else** branch in an **if** expression matches that of the **then** one. Despite this, several other things need to be in place for this value to be considered valid: the type of the generated function must be declared beforehand, and the function must refer to it by its index in the types list (marked in red). Moreover, the type of its actual outputs must match the one declared by its type (marked in blue). Generating valid modules satisfying such invariants is extremely rare using automatically derived generators.

If we look at the effect of our heuristics, we can observe that mutation inheritance can provide substantial speedups in most cases. In the case of bug #5, however, disabling mutation inheritance seems to help as a shortcut to find counterexamples faster. This suggests that perhaps a hybrid approach would work best in the general case.

Furthermore, our preemptive scheduling heuristic greatly improves the testing performance in most cases, being orders of magnitude faster in the case of bugs #1, #4, and #5.

The heuristics we developed for MUTAGEN offer a substantial improvement over the simple (though still powerful) approach taken by FuzzChick. Our current work focuses on gathering more empirical evidence to strengthen this claim.

## IV. Future Work

As for future work, we are focused on extending MUTAGEN with dependent mutations, where the existence of certain subexpressions would allow or disallow certain mutations to occur. Concretely, this could be useful to improve the performance of our tool when used for testing systems expecting programs as inputs. There, for instance, a mutation that introduces a new identifier would allow subsequent mutations to refer to it, preserving the well-scopedness of the program.

## References

[1] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE transactions on Software Engineering*, no. 4, pp. 438–444, 1984.

[2] M. Zalewski, "American fuzzy lop," 2014.

[3] R. Swiecki, "Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options," *URl: https://github. com/google/honggfuzz (visited on 06/21/2017)*, 2017.

[4] C. Miller, Z. N. Peterson *et al.*, "Analysis of mutation and generation-based fuzzing," in *Independent Security Evaluators, Tech. Rep*, vol. 4, 2007.

[5] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[6] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 445–458.

[7] M. Eddington, "Peach fuzzing platform," *Peach Fuzzer*, vol. 34, 2011.

[8] Mozilla, "Dharma: a generation-based, context-free grammar fuzzer," https://github.com/MozillaSecurity/dharma, 2015.

[9] G. Grieco, M. Ceresa, and P. Buiras, "Quickfuzz: An automatic random fuzzer for common file formats," *ACM SIGPLAN Notices*, vol. 51, no. 12, pp. 13–20, 2016.

[10] G. Grieco, M. Ceresa, A. Mista, and P. Buiras, "Quickfuzz testing for fun and profit," *Journal of Systems and Software*, vol. 134, pp. 340–354, 2017.

[11] L. Lampropoulos, M. Hicks, and B. C. Pierce, "Coverage guided, property based testing," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[12] A. Mista, A. Russo, and J. Hughes, "Branching processes for quickcheck generators," *ACM SIGPLAN Notices*, vol. 53, no. 7, pp. 1–13, 2018.

[13] A. Mista and A. Russo, "Generating random structurally rich algebraic data type values," in *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*. IEEE, 2019, pp. 48–54.

[14] J. Duregård, P. Jansson, and M. Wang, "Feat: functional enumeration of algebraic types," *ACM SIGPLAN Notices*, vol. 47, no. 12, pp. 61–72, 2012.

[15] L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce, "Generating good generators for inductive relations," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–30, 2017.

[16] C. Runciman, M. Naylor, and F. Lindblad, "SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values," in *Proceedings of the first ACM SIGPLAN symposium on Haskell*, 2008, pp. 37–48.

[17] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4, pp. 123–133, 2002.

[18] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 185–200.

[19] Ilya Rezvov, "wasm: WebAssembly Language Toolkit and Interpreter," https://hackage.haskell.org/package/wasm, 2018.

[20] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," *Acm sigplan notices*, vol. 46, no. 4, pp. 53–64, 2011.